NANYANG TECHNOLOGICAL UNIVERSITY

Enhancing Automotive Embedded Systems with FPGAs

Shreejith Shanker

School of Computer Science and Engineering

A thesis submitted to Nanyang Technological University
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

April 2016

THESIS ABSTRACT

Enhancing Automotive Embedded Systems with FPGAs

by

Shreejith Shanker

Doctor of Philosophy

School of Computer Science and Engineering

Nanyang Technological University, Singapore

Modern vehicles represent a complex distributed cyber-physical system that simultaneously handles critical functions like drive-by-wire systems, non-critical functions like window/door control, and compute intensive multimedia functions. Distributed electronic control units (ECUs), which integrate processing elements and supporting peripherals (network interfaces, memory), implement a variety of functions in software, and information is exchanged between ECUs and sensors/actuators over in-vehicle networks. As the complexity of applications rises with increasing automation, extensive hardware support is required in the form of multicore processors or special purpose hardware accelerators to offer required levels of performance. Additional features also drive an increase in the number of ECUs since new functions are rarely consolidated on existing ECUs. Furthermore, network interfaces implemented as ASICs or dedicated logic must be adapted to handle increased communication loads, consuming power and requiring more infrastructure support in terms of cabling and weight. The increasing number of safety-critical functions further impacts complexity if existing one-to-one redundancy schemes are applied. Rising automation also poses news challenges like security, with researchers showing that internal networks are easily manipulated with catastrophic effects and total loss of control. Our research aims to address these challenges using architectural enhancements that are transparent at the computational and network levels, leveraging the capabilities of reconfigurable hardware. We present advanced ECU architectures with extended network capabilities, apply these in the context of safety critical systems, explore ways of extending these schemes to offer advanced security features, and show how such advanced systems can be validated in hardware. Our work represents an advancement in the state of the art with regard to applying FPGAs in vehicular systems.

Acknowledgements

I take this opportunity to thank my supervisor, Prof. Suhaib A Fahmy, for giving me the opportunity to work on this exciting project, in association with TUM CREATE, Singapore. His enthusiasm and exuberance have helped me to carry out my research with composure and confidence. He has been appreciative of my ideas and encouraged critical thinking and technical writing, which have helped me improve my skills. I am also thankful to Prof. Samarjit Chakraborty for providing the opportunity to work at the Institute for Real-Time Computer Systems, Technical University of Munich as part of the Joint PhD program, and for his guidance and support during my PhD. I am also grateful to Dr. Martin Lukasiewycz, Principal Investigator at TUM CREATE, for his guidance, support, and suggestions. His experience in the automotive industry helped me channel my research into relevant topics and areas of interest.

I would also like to use this opportunity to thank to my colleagues and peers at the Centre for High Performance Embedded Systems (CHiPES), especially Dr. Sharad Sinha, Dr. Vipin Kizheppatt, Rakesh Varier, Dr. Neethu Robinson, Pham Hung Thinh, Dr. Jiang Lianlian, Ronak Bajaj, Abhishek Jain, Abdullah Shamil, Dr. Kavitha Jubin, and Dr. Smitha Sreekumar for their invaluable suggestions and support. Our laboratory executive, Chua Ngee Tat has been enthusiastic in providing me with help and support on issues related to networks and software at CHiPES. I am also grateful to my friends and fellow PhD candidates Rahul Varier and Dr. Amrith Dhananjay for their companionship and support. I am also thankful to my colleagues at TUM CREATE and the Technical University of Munich, especially Dr. Sidharta Andalam, Dr. Sebastian Steinhorst, Philipp Mundhenk, Florian Sagstetter, Martin Becker, and Martin Geier for their suggestions and creative ideas.

I would also like to thank Prof. Ian McLoughlin, University of Kent (formerly NTU), and Prof. Vinod A Prasad in the School of Computer Science and Engineering at NTU for their guidance and encouragement during my coursework and otherwise. I also express my gratitude to Prof. Douglas Maskell, Prof. Nachiket Kapre, Prof. Kyle Rupnow, and Prof. Arvind Easwaran for their guidance and research support.

I also use this opportunity to thank my previous employers Processor Systems India Pvt. Ltd (ProcSys) in Bangalore for giving me the opportunity to work in the then emerging domain of reconfigurable computing systems for high performance

applications. My former mentors Manjusha S., Vinod Narippatta, Hansraj V., and Jaison T. D. were inspiring, and provided me challenging scenarios to stimulate my interest in FPGA design, and their guidance and expertise have proven invaluable. I am also thankful to my previous employer Digital Systems Design Group (DSD), Vikram Sarabhai Space Centre (VSSC), Trivandrum, India, for providing me with the opportunity to work for the Indian Space Research Organisation (ISRO) on cutting edge technology and complex projects. I would like to thank my senior officials at VSSC, A. K. Abdul Samad and Subha Varier, for their guidance on deterministic system design and space technology standards, ideas which have been extremely useful during my research.

Finally, I am indebted to my family and my parents, for their prayers and encouragement. I thank them for their understanding and their efforts to support me in pursuing higher studies. I am also extremely thankful to my wife for her constant support and patience during this period.

Contents

List of Figures

List of Tables

List of Abbreviations

ABS	Anti-lock Braking System
ACC	Adaptive Cruise Control
AEG	Automotive Ethernet Gateway
API	Application Programming Interface
ASAM	Association for Standardisation of Automation and Measuring Systems
ASIC	Application Specific Integrated Circuit
AUTOSAR	AUTomotive Open Systems ARchitecture
AXI	Advanced eXtensible Interface
BD	Bus Driver
BRAM	Block Random Access Memory
C2X	Car to X
CAM	Content Addressable Memory
CAN	Controller Area Network
CAS	Collision Avoidance Symbol
CC	Communication Controller
CFAR	Constant False Alarm Rate
CPS	Cyber-Physical System
CRC	Cyclic Redundancy Check
DCM	Digital Clock Manager

DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
ECU	Electronic Control Unit
FIBEX	Field Bus Exchange Format
FIFO	First In First Out
FF	Flip-Flop
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
F-TDMA	Flexible-Time Division Multiple Access
FTM	Fault Tolerant Mechanisms
HS-CAN	High Speed Controller Area Networks
ICAP	Internal Configuration Access Port
IVC	In-Vehicle Communication
JTAG	Joint Test Architecture Group
LIN	Local Interconnect Network
LS-CAN	Low Speed Controller Area Networks
LUT	Look-Up Table
MCAL	Micro-controller Abstraction Layer
MDM	MicroBlaze Debugger Module
MMCM	Mixed Mode Clock Manager
MOST	Media Oriented Systems Transport
MT	Macro-Tick

NI	Network Interface
NMV	Network Management Vector
OBD	On Board Diagnostics
OFDM	Orthogonal Frequency Division Multiplex
OSEK	Open Systems and their Interfaces for the Electronics in Motor Vehicles
PC	Personal Computer
PCIe	Peripheral Component Interconnect Express
PL	Programmable Logic
PLB	Processor Local Bus
PR	Partial Reconfiguration
PRR	Partially Reconfigurable Region
PS	Processing System
RTE	Run-Time Environment
SMV	Security Management Vector
SUP	Start-Up Pattern
TTCAN	Time Triggered Controller Area Networks
TDMA	Time Division Multiple Access
TMR	Triple Modular Redundancy
TPMS	Tire Pressure Monitoring System
TTE	Time Triggered Ethernet
WUDOP	Wake-Up During Operation Pattern
WUP	Wake-Up Pattern
UART	Universal Asynchronous Receiver/Transmitter
V2V	Vehicle to Vehicle
V2X	Vehicle to X

1

Introduction

Three decades ago, a car was a highly mechanical piece of equipment that incorporated an engine, drive mechanism, and wheels, with a battery, alternator system and some controls for the lights as the only electrical parts. The first computing device to be used in a vehicle was a tiny micro-controller for handling the timing of spark plugs, with a modest control function implemented in a few lines of code [1]. Improving electronics allowed this functionality to be extended to engine timing control, control of the fuel injection system, and diesel engine control, using more powerful micro-controllers which executed code that monitored a set of sensors and controlled a sequence of actuators. The extensive use of micro-controllers for engine related control or management resulted in these small computing devices being called engine control units or engine management units (EMU).

The ability to control functionality through software was appealing and numerous vehicular applications exploited micro-controllers in safety-related systems like Anti-lock Braking Systems (ABS) and airbag control, as well as in comfort functions like window controls, radio/cassette players and others towards the late 1980s. While complex computation was typically not required, reliability became an important factor. When connectivity was required, these embedded units were wired up using point-to-point links.

A modern vehicle today, by comparison, incorporates much more complex computation and dedicated network-based connectivity to handle the complex exchange of information. A high-end vehicle today can execute over 20 million lines of code in real-time, over a distributed array of embedded processing units (called electronic compute units or ECUs), that together control and coordinate both critical and comfort functions [2]. The number of computing units in a vehicle has thus risen across all ranges of cars, with low-end vehicles incorporating 30 to 50 ECUs, while the top of the line models can incorporate over 100 ECUs. These are supported by multiple in-vehicle networking standards that have also evolved to accommodate the reliability and bandwidth requirements of these functions.

The complexity of applications on modern vehicles is significantly higher; many mechanical functions like hydraulic power steering and drive controls (like acceleration and braking) are being replaced by their electronic versions (called x-by-wire systems), which require hard real-time performance. On the other hand, advanced assistance systems makes use of a variety of sensors to present the driver with information about the vehicle and its surroundings, and may even be able to physically assist the driver (like automatic parking). These applications can also be adaptive to conditions, as in the case of adaptive terrain response systems, which can detect the road surface and adapt vehicle settings automatically, increasing their complexity. Such complex applications require major and minor adjustments in near real-time, based on intensive computations performed on data collected from a multitude of sensors. Software offers the benefits of flexibility, easier development cycle, and field upgradability but more advanced special purpose architectures and/or multi-core processor systems are needed to meet requirements. Even with

these enhanced compute platforms, the scalability of systems is bounded by the limited parallelism on offer. Multi-core and application specific platforms can also consume higher power, degrading the energy efficiency of the system.

Another challenge is the integration model followed in vehicular embedded systems that treats each new function as a new ECU. This lack of consolidation depends on multiple factors, but is largely influenced by the capabilities of the computational platforms (or lack thereof). For these new functions to be consolidated onto existing ECUs, the architectures must provide support for isolating different applications to ensure that both of them can operate reliably and are not affected by the presence (or actuation) of other applications. On processor-based architectures, this level of isolation is a non-trivial problem since the processor hardware will be shared by the functions that are being executed on the compute core, resulting in contention that has to be explicitly handled, creating further complexity. This generally results in the use of the "new compute unit for each new function" integration model, along with its associated packaging and network connectivity complexity. It is estimated that a mid-sized vehicle incorporates over 6 km of interconnect cabling contributing over 70 kg to the weight, largely due to this integration model [3].

The increasing proliferation of mobile technology is also pushing newer features into vehicular systems like inter-vehicle communication and connected services. The vehicle-to-vehicle (V2V) communication concept enables vehicles to exchange information about themselves and surroundings, and aims to achieve better road systems in a co-operative manner. The increasing connectivity and automation in vehicles brings with it new challenges like security; a connected vehicle controlled by software provides huge opportunities for a hacker. Computational systems and networks in modern and future vehicles will have to integrate mechanisms to combat security threats at the application layer or at the platform level, further adding to their complexity.

For a mass production industry like automotive, application specific integrated circuits (ASICs) would provide the most cost effective solution. Applications that

demand large scale deployment across ranges of vehicles like ABS can benefit from the highly energy efficient, dedicated approach. However, their complete lack of flexibility to adapt to changing standards and requirements can prevent their widespread adoption in some vehicular systems. For example, automotive manufacturers like *Tesla Inc.* often provide software-based updates to the electronically controlled functions in their electric vehicles, aiming to improve performance and efficiency. Furthermore, the development cycle for software-based functionality is much smaller compared to the development flow of ASICs.

Reconfigurable computing platforms like Field Programmable Gate Arrays (FPGAs) provide an alternative. The configurability of the platform allows designers to build customised circuits which provide many of the performance and energy benefits of ASICs, while retaining the flexibility of software systems. Beyond the obvious performance boost, such systems provide a powerful platform for implementing multiple functions on the same hardware with complete isolation, using hard partitioning, and partial or complete reconfiguration. This further improves cost benefits and power consumption, along with savings in other factors such as size and weight. Moreover, FPGAs enable systems to be built with guaranteed deterministic results. This is particularly important for safety-critical in-vehicle systems, where reliability is of paramount importance.

New generation hybrid reconfigurable platforms further extend the appeal by enabling tight integration of software based control flow with hardware processing, allowing both components to be upgraded after deployment, similar to existing software-based functionality while benefiting from higher performance, lower energy consumption and massive scalability. This would enable a single ECU-on-chip architecture possible, where the computational function is closely integrated with accelerators, sensor/actuator interfaces and network communication interfaces.

Despite these advantages, reconfigurable hardware is not widely used in ECUs. The main hurdle is the difficulty associated with designing and evaluating ECU architectures and the expertise required to design, manage and execute runtime reconfiguration. In the following sections, we discuss the general architecture of

ECUs and in-vehicle networks in modern vehicles and the challenges in adopting new architectures for the in-vehicle infrastructure. This thesis contributes enhanced network architectures, compute systems, and security infrastructure for next generation vehicular computing systems on reconfigurable platforms. It proposes techniques to incorporate such features in a manner that is transparent to the application and develops mechanisms to automatically validate such features.

1.1 Automotive ECUs and In-Vehicle Networks

Early vehicular electronics consisted of simple devices like 8-bit micro-controllers with simple I/O support to connect to sensors [4]. With the introduction of more demanding applications, more powerful 16 and 32-bit processors and domain specific controllers were used to provide improved computational capabilities. Special purpose hardware like Digital Signal Processors (DSPs) are used in modern vehicles to accelerate computations for signal processing applications. The primary advantage of using software-programmable processors is portability and independence from underlying hardware due to abstractions supported by high level languages. This trend was encouraged by standardisation of the requirements and capabilities of underlying operating systems, which enable application developers to design their product independent of the hardware target. These standards, OSEK [5] and its evolution AUTOSAR [6], are widely adhered to in the automotive industry, and determine a standardised platform for automotive applications.

Early automotive systems used simple switches and actuators, and their functionality was achieved using point-to-point wiring. As more complex systems were introduced, point-to-point connections became infeasible due to the complexity of the wiring harness and the resulting additional weight and volume [7]. The early 1980s marked the introduction of vehicle networking as a step to reduce the wiring costs and complexity. Bosch introduced the Controller Area Network (CAN) in the mid-1980s, which gained widespread acceptance in the automotive industry and later became the most widely used networking backbone for in-vehicle systems.

CAN provides flexibility to the user, since it can operated at multiple speeds and thus at varied costs. For instance, low-speed CAN can run at 125 kbps and can cater to all the user oriented electronics in a car, like power windows, electric seats and air conditioning, while high-speed CAN could run at up to 1 Mbps, serving real-time and safety-critical applications like engine management, ABS and others.

Depending on the performance and/or safety requirements, an application in a modern vehicle can be classified into one of the following domains:

1. The *Body Domain* incorporates user comfort features that are not safety-critical and have low quality of service (QoS) requirements.
2. The *Powertrain* handles engine management, power delivery and transmission and has hard real-time requirements.
3. The *Chassis Domain* includes systems which affect vehicle dynamics and controllability like steering, brakes, and suspension. These are safety-critical functions since they affect the behaviour of the vehicle and its response to user inputs.
4. *Telematics and In-Vehicle Infotainment (IVI) Systems* integrate high speed multimedia, driver assistance systems, and the human-machine interface (HMI). This domain typically has relaxed real-time constraints.
5. *Occupant Safety* is primarily concerned with active protection systems for passenger safety and thus has strict real-time requirements.

Each domain requires different levels of service, such as response time, bandwidth, redundancy, and error detection, among others, often referred to as Quality of Service (QoS) levels [8]. These also feature different real-time and performance capabilities at the software level and the associated hardware level.

ECU Architectures

Typically, an ECU is composed of a processing element, a network interface and associated storage, as shown in Figure 1.1. The processing element may be an automotive grade micro-controller, general purpose processor or an application-specific

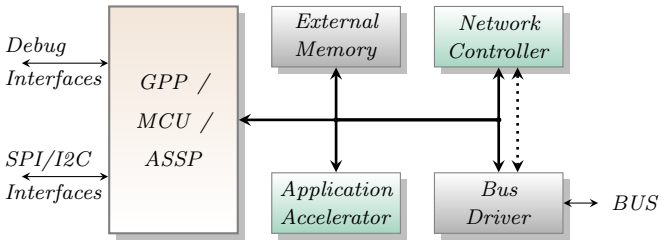


Figure 1.1: Typical ECU architecture.

controller. These execute the software tasks required for the ECU's functionality. The processing element usually integrates common peripherals like timers, I2C/SPI interfaces that may be utilised by the application code. Most applications are executed on top of a standard real-time capable operating system like μC-OS, QNX Automotive, or Windows Embedded (Automotive), which provide the required abstraction for the application tasks. Operating systems may also need to support the abstractions defined by OSEK or AUTOSAR (either on their own or using middleware), which use a set of libraries to provide a standardised interface to the application developer.

The placement constraints for many ECUs that requires their compute blocks and/or sensor/actuator units to be positioned in certain parts of the vehicle led to a distributed computing architecture. To support this distributed model, ECUs integrate automotive standard network interface(s), either as dedicated ASICs, or as co-processing logic integrated into the same die. Multiple network protocol implementations may also be integrated in the case of ECUs which require interfaces to different physical networks. ECUs that handle an extensive amount of data and computation may also incorporate dedicated accelerators (hardware) and memory systems like DRAM. Off-the-shelf components from automotive vendors also incorporate additional features like security extensions, hardware cryptographic blocks, or others.

Depending on the domain the ECU is intended for, vendors also provide customised architectures that are best suited for functionality in that domain. For example, a body domain controller might integrate different network protocols and offer little or no hardware acceleration support, while a telematics controller would integrate high speed interconnect and dedicated accelerator blocks for video processing or

Table 1.1: SAE in-vehicle network classification.

Class	Throughput	Domain	Leading Protocol
Class A	below 10 kbps	Body Domain: Low end	LIN
Class B	10 to 125 kbps	Body Domain: Non-critical and non-diagnostic	Single-wire CAN (SWC) & CAN 2.0
Class C	125 kbps to 1 Mbps	Powertrain: Real-time critical parameters	High speed CAN (HSCAN)
Class D	above 1 Mbps	Powertrain, Chassis: Hard Real-time & Reliable	FlexRay
		Occupant Safety: Real-time & Reliable	Safe-by-wire & Byteflight
		Streaming Media and Entertainment	MOST

radar interfaces. This "right-sizing" enables manufactures to control the cost (development and parts) as well as standardise the software framework for each domain.

In-Vehicle Networks

A variety of in-vehicle networks evolved, driven primarily by cost and performance requirements. CAN proved too expensive and complicated for simple functions like power windows or boot release. Simpler protocols like the Local Interconnect Network (LIN) could offer similar functionality at lower cost per module and power consumption, and thus found widespread adoption for non-critical functions. CAN also proved too slow for high bandwidth applications like multimedia in higher end vehicles resulting in the development of high bandwidth protocols like Media Oriented Systems Transport (MOST) for such applications. Time-triggered CAN (TTCAN) is an evolution of standard CAN, which addresses the lack of determinism by introducing a time-triggered mechanism above the CAN framework. The FlexRay protocol, developed by the FlexRay consortium, offers a combination of time-triggered and event-triggered communication for in-vehicle applications to enhance reliability with higher bandwidth. Time-triggered Ethernet, an extension of standard Ethernet, is also gaining traction as the backbone network for future vehicles. The Society for Automotive Engineers (SAE) classifies in-vehicle

networks based on throughput and domain of operation [7, 8, 9], as shown in Table 1.1

As the number of communicating nodes in a network has increased, CAN has proven incapable of consistently providing deterministic data transfer rates, primarily due to its event-triggered architecture. Emerging safety-critical applications demand higher levels of determinism, which cannot be consistently ensured by event-triggered networks. Future functions like drive-by-wire and brake-by-wire will make time-triggered schemes the de-facto standard in critical domains. However, more widespread adoption of FlexRay or similar time-triggered architectures will be limited by the higher cost per node.

ECU Consolidation

The "new function as a new ECU" integration model allows integration into existing and proven in-vehicle infrastructure with minimal time and cost. However, this approach increases the weight and power consumed by electronic modules in the vehicle and reduces the communication bandwidth. For future electric vehicles, the powertrain, battery management, and control interfaces will be fully computerised, requiring more computational power. The total weight of the on-board computing systems and their power consumption become more critical considerations to maximise performance and range for electric vehicles.

Consolidating multiple functions onto fewer ECUs is important to counter this problem. Traditional approaches using processors running real-time operating systems do not allow for reliable isolated sharing since processor hardware resources are shared by the two (or more) functions. Even in the case of multi-core processor architectures shared caches or registers represent a point of contention. Determinism in such an environment would require more complex real-time support and explicit management in software.

Enhanced Capabilities

Currently, any enhancement to the communication network or computational system are enabled due to software flexibility. For instance, integrating security using

upcoming standards like secure hardware extensions (SHE) requires the application to leverage enhancements to underlying libraries. This creates additional overheads in software execution to utilise these extensions and requires extensive data re-routing which must be handled in software. Furthermore, additional latency is incurred for each data/message, which must be factored into the task/message schedule and re-evaluated to ensure real-time deadlines are not violated. Abstracting such operational details and latencies from the application provides a better mechanism. However, this is not possible with off-the-shelf components and requires extensible, configurable datapath features to be integrated in the hardware architecture.

Evolving automotive systems are adaptive in nature, meaning such systems make intelligent choices to react to operating conditions. Systems like adaptive cruise control and driver assistance systems provide better performance than their corresponding static implementations [10]. On a conventional ECU architecture, adaptation is achieved through sophisticated software routines running on general purpose platforms. More advanced adaptive applications are more computationally complex, often requiring special purpose hardware support and accelerators. Likewise, security policies and algorithms need to adapted during the lifetime of an ECU, which can be 20 years. These trends suggest that flexible hardware will play a key part in next generation ECU architectures, offering benefits like accelerated computation, lower latency, security, and deterministic execution.

FPGAs for Automotive Applications

Reconfigurable hardware platforms like Field Programmable Gate Arrays (FPGAs) offer an alternative for implementing such systems. FPGAs started as simple programmable chips for glue logic at the board level. Modern FPGAs from Xilinx and Altera are capable of implementing large and complex systems [11], and offer a wide range of built in hard macro blocks, such as processors, DSP blocks and block memories (RAMs). FPGAs can allow aggregation of multiple functions and execution in complete functional isolation. They also allow integration of network controllers and ECU functions (either as software running on embedded processors or as custom logic) in a compact footprint. Deterministic behaviour is also

easily factored into systems implemented on reconfigurable hardware. Techniques like dynamic or partial reconfiguration (PR) allow hardware-level adaptation to enable more adaptive next generation automotive applications. PR also provides alternative ways of handling redundancy for advanced safety-critical systems, as well as mechanisms to adapt only segments of the architecture (like cryptographic blocks), without requiring extensive redesign of the entire ECU architecture.

1.2 Motivation

FPGAs are not widely used in the automotive industry partly because they are expensive and also because they require substantial hardware expertise. Exploiting the potential of reconfigurable hardware for automotive applications requires development of architectures which are compact and power-efficient. Validation and certification concerns have also held back adoption, as many of the methodologies applied to software systems do not map well to these architectures. Many vendors provide IP cores for automotive applications but these are mostly generic and platform agnostic, and hence often inefficient. Furthermore, advanced features like partial reconfiguration are rarely exploited. We believe that efficient compute architectures that conform to automotive standards (like AUTOSAR), can offer an abstracted view to the system designer, making reconfigurable hardware more appealing.

Hardware-level adaptation using dynamic reconfiguration currently requires management through low-level hardware access. This is contradictory to the AUTOSAR implementations which require hardware details to be abstracted from application designers. The design process is also extremely complex with significant low-level architecture steps required. This makes integration and management of run-time hardware adaptation an expert feature, further reducing its attractiveness for automotive system and application designers. By providing a standard architecture that integrates run-time reconfigurability and abstracting its

management, we believe that next generation ECU architectures can benefit from hardware-level adaptation that is integrated as a standard feature of the platform.

For enhancing the electrics/electronics (E/E) architectures for next generation vehicular computing systems, enhanced mechanisms are required at the communication/network layer for improving reliability and determinism with high performance and energy efficiency. While current research aims to achieve this through dedicated communication channels (bandwidth or slots) for communicating system status, this requires extensive rescheduling and revalidation of communication and computation each time a new feature is integrated. Such information must also be passed up to the software application layer for processing, which further increases latency. We believe that integrating intelligence into the network layer provides a better alternative, and would help to abstract such low level details from the application designer. This is especially true for applications like in-vehicle network security, which is closely tied to the network properties and can benefit from tight integration with the communication protocol. However, such extensions must be adaptable and configurable so that these can be updated in-field, as with ECU software updates, making reconfigurable hardware the ideal platform for such intelligent network systems. Our research aims to develop architectures for next generation time-triggered network systems that provide extended security and reliability features without affecting the determinism of the protocol, while abstracting such details from the (software) application layer.

Another challenge associated with architecture evaluation for future automotive E/E systems is the difficulty associated with validating the functionality of applications. Presently, a complex hardware-in-the-loop test setup that replicates the E/E architecture in a vehicle is recreated in a laboratory environment, on which the changes to architecture and/or functionality of ECUs are evaluated. A more scalable approach is to use a reconfigurable validation testbed platform that can be easily configured with the required architecture and evaluated in real-time, with bit-level precision.

In the past decade, research on next generation vehicular systems has mainly concentrated on application level enhancements: novel functionality and application-level mechanisms for reliability and security. Most of this work presents approaches from the application designers' perspective, relying on standard ECU architectures and communication systems. Little research work has aimed at improving the underlying hardware architecture and/or evaluating enhancements to the network layer. Though many applications include the use of FPGAs for their performance benefits, they did not consider ECU architectures that exploit the full capabilities of FPGAs across the layers of the hierarchy. While we do acknowledge the hardware expertise required for efficient designs and effective use of reconfigurability, we believe that such details can be abstracted from the application designers and can be embedded within the architecture as a feature. The techniques we propose integrate these extensions at the lower layers of the architecture and the network, over which existing research and applications can be directly ported without any loss in performance or determinism. In this research, we concentrate on ECU systems that rely on FlexRay as the network interface, as time-triggered networks like FlexRay have been established as the architecture of choice for safety-critical applications for future vehicles. The proposed methods can be equally adapted to ECU systems that integrate emerging time-triggered standards like Automotive Ethernet. We validate our approaches on Xilinx devices because of their established dynamic reconfiguration flow, though the same can be ported to the emerging Altera devices and tool flow.

1.3 Objectives

The main objectives of the research in this thesis are to:

1. Demonstrate how extended communication can be integrated into existing infrastructure without sacrificing reliability of the protocol and in a transparent manner.

2. Propose ECU architectures that provide advanced functionality like fault-tolerance and consolidation, leveraging enhanced communication and abstracted from the application layer.

3. Develop techniques to integrate configurable security extensions into the architecture that provide network and application security.

4. Develop a real-time functional validation platform that enables hardware validation of ECU functionality with bit-level accuracy and real-time performance.

1.4 Contributions

The main contributions of this thesis are the architectures and techniques developed for enhancing E/E architectures and communication infrastructure for next generation vehicular systems. The architectures and techniques are designed in a transparent manner such that system designers can port existing automotive functions to the enhanced E/E architecture with minimal modification.

1. We have performed a comprehensive study of in-vehicle networks and architectures with a focus on time-triggered networks like FlexRay. We have also identified features which can be enhanced through extended communication on existing infrastructure.

2. An optimised FlexRay communication controller for reconfigurable architectures has been developed that has configurable extensions that integrate features like timestamps, message identifiers, and reordering logic. The optimised controller offers tight integration with the functional logic on the ECU, while the extensions abstract network-level operations from the application for better system performance and efficiency.

3. An efficient scheme for implementing functional consolidation for non-critical ECUs and hardware-level fault-tolerance for safety-critical ECUs on reconfigurable hardware has been developed. The scheme extends the FlexRay

protocol framework to support system state messages that are handled at the network layer. The network layer also manages the self-healing capability by integrating intelligence (and reconfiguration logic), abstracting such details from the application layer.

4. An efficient Gateway ECU architecture based on a hybrid FPGA platform has been developed that provides deterministic switching performance in a compact and energy efficient footprint. The gateway ECU is developed to support future vehicular systems that utilise Automotive Ethernet as the backbone network, respecting priorities imposed by automotive functions on certain messages while providing gigabit interconnect between different branches.

5. A scheme for integrating zero-latency message encryption is developed as a configurable extension to our enhanced FlexRay interface. This is further extended to provide a network access prevention scheme and cross-layer security for improved entropy of the ciphertext. This enhanced scheme also integrates tamper protection for the software application and hardware bitstream, preventing compromised ECUs from accessing a secure network. The scheme is integrated transparently to the application, allowing standard FlexRay applications to be directly ported to the secure domain.

6. A super-real-time bit-precise hardware evaluation platform has been developed to functionally evaluate an ECU within its network cluster by recreating the cluster on a large FPGA chip. The platform provides numerous capabilities to inject common errors and faults into the system, while a real-time monitor enables the cluster behaviour and network communication to be observed on a standard PC. Furthermore, the entire platform can be automated using simple scripts for long duration and complex test cases. Finally, the entire platform can be accelerated without loss in precision, lowering the validation time.

1.5 Thesis Organisation

The remainder of this thesis is organised as follows. Chapter 2 presents a detailed literature survey on the evolution of embedded systems and networks covering state of the art systems like CAN, FlexRay and Time-triggered Ethernet. We look at other aspects of in-vehicle communication like scheduling, reliability analysis, and extensions like switched FlexRay. We also look at the evolution of reconfigurable hardware, its applicability in ECUs and advanced techniques like PR in this chapter. Chapter 3 describes the implementation of our extensible FlexRay communication controller, the optimisations we have achieved for reconfigurable hardware and the datapath enhancements that allow extended communication using existing messages. The chapter also presents a comparison of our implementation against existing platform agnostic implementations, and the potential of the extensions. In Chapter 4, we describe the enhanced architecture for ECUs on reconfigurable hardware that integrates capabilities of consolidation and fault-tolerance. We also present an architecture for high-performance gateway ECUs for the proposed Ethernet-backbone in-vehicle infrastructure on hybrid reconfigurable platforms. Chapter 5 presents our scheme for integrating transparent network level security and comprehensive system-level security offering system and network level protection without incurring additional latency. Chapter 6 describes the functional validation platform and its extensions for improved observability. Finally, Chapter 7 concludes the work presented in this thesis and outlines future research directions.

1.6 Publications

Most of the work presented in this thesis has been included in a number of published and submitted papers.

1. S. Shreejith, K. Vipin, S. A. Fahmy, M. Lukasiewycz *An Approach for Redundancy in FlexRay Networks Using FPGA Partial Reconfiguration*, in Proceedings of the Design Automation and Test in Europe (DATE) Conference, Grenoble, France, March 2013, pp. 721-724 [12].

2. S. Shreejith, S. A. Fahmy, M. Lukasiewycz *Reconfigurable Computing in Next Generation Automotive Networks*, IEEE Embedded Systems Letters, vol.5, no. 1, pp. 12-15, March 2013 [13].

3. S. Shreejith, S. A. Fahmy, M. Lukasiewycz, *Accelerating Validation of Time-Triggered Automotive Systems on FPGAs*, in Proceedings of the International Conference on Field Programmable Technology (FPT), Kyoto, Japan, December 2013, pp. 4-11 [14].

4. K. Vipin, S. Shreejith, D Gunasekera, S. A. Fahmy, N. Kapre, *System-Level FPGA Device Driver with High-Level Synthesis Support*, in Proceedings of the International Conference on Field Programmable Technology (FPT), Kyoto, Japan, December 2013, pp. 128-135 [15].

5. S. Shreejith, S. A. Fahmy, *Enhancing Communication On Automotive Networks Using Data Layer Extensions*, in Proceedings of the International Conference on Field Programmable Technology (FPT), pp. 470–473, Dec. 2013 [16].

6. S. Shreejith, S. A. Fahmy, *Zero Latency Encryption with FPGAs for Secure Time-Triggered Automotive Networks*, in Proceedings of the International Conference on Field Programmable Technology (FPT), Shanghai, China, December 2014, pp. 256-259 [17].

7. S. Shreejith, S. A. Fahmy, *Extensible FlexRay Communication Controller for FPGA-Based Automotive Systems*, IEEE Transactions on Vehicular Technology, Vol. 64, No. 2, pp. 453–465, Feb. 2015 [18].

8. S. Shreejith, S. A. Fahmy, *Security Aware Network Controllers for Next Generation Automotive Embedded System*, in Proceedings of the Design Automation Conference (DAC), San Francisco, USA, June 2015, 39:1–39:6 [19].

9. S. Shreejith, P. Mundhenk, A. Ettner, S. A. Fahmy, M. Lukasiewycz, S. Chakraborty *AEG: An FPGA-based Gateway Architecture for Ethernet backbone In-vehicle Networks*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems (prepared for submission).

2

Literature Survey

The introduction of electronics for vehicular computation began in the early 1980s with lightweight micro-controllers and point-to-point wired connections for control tasks like fuel injection control, anti-lock braking systems (ABS), and other control functions. The addition of more functions resulted in a large number of point-to-point links, which became complicated and difficult to manage. Moreover, the amount of wiring began contributing noticeably to the overall weight of the car. Network-based connectivity and advanced compute units progressively made their way into vehicles to offer new applications and improve cost, efficiency, and performance. In this chapter, we review the architecture of existing state-of-the-art vehicular networks like CAN, FlexRay and Time-triggered Ethernet (TTE) and proposed extensions for these networks. We review the evolution of ECU architectures and analyse the potential of FPGA-based computing infrastructure for

in-vehicle systems. We also discuss evolution of FPGA architectures and advanced capabilities like partial reconfiguration offered on state-of-the-art FPGAs.

2.1 In Vehicle Computing Systems

The first reported use of an electronic safety-critical control function in vehicles was in 1981. General Motors adopted micro-computer based engine control for their gasoline powered cars which greatly improved efficiency and performance [1]. With the introduction of laws regulating emission control, the use of electronic engine control was required to meet the legal requirements as well as to maintain acceptable efficiency and performance. The ease of implementation and the cost/-efficiency benefits motivated manufacturers to adopt electronic control for engine management and this later spread to other domains.

Modern vehicles incorporate upwards of 30–50 ECUs across all segments. These ECUs employ automotive grade micro-controllers and/or general purpose processors which execute software implementations of control and comfort applications. The number of ECUs in vehicles has been rising at the rate of approximately 1.45× a year, while the application software has been growing at a rate of 4.5 MB per year [20].

Embedded computing in modern vehicles is segmented into different domains mainly differentiated by the criticality of the function executed. In general, each ECU integrates a processing element (single or multi-core processor), memory subsystems (including volatile and non-volatile), optional dedicated accelerators like cryptographic or image processing engines, power supply elements, and the interfaces to the different sensors, actuators, and network. Specific combinations are chosen depending on requirements for each application. For example, in the body electronics domain that handles simple comfort functions like doors, access control, lighting systems, and climate control, an ECU architecture may be composed of an 8- to 32-bit micro-controller, non-volatile code memory, and network interfaces like CAN and LIN. Further, the ECU may have features like ultra-low power

operation to enable it to be constantly active (as in the case of access control systems) [21, 22, 23]. However, in the driver information and multimedia domain, an ECU might integrate a GPS interface for navigation, a display driver interface(s), a 32-bit single or multi-core processor, and a video accelerator, along with an optical network interface like MOST, offering higher compute and communication performance at higher power consumption [24].

Most current generation compute units use CISC/RISC core(s) as the central processing element [21, 22, 23, 24]. These can be custom designed cores or flavours of generic 16/32-bit cores available from multiple vendors. Supporting peripherals interface to the processing element over standard interfaces like Advanced Micro-controller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) or similar high-speed on-chip interconnect. Commonly integrated peripherals include hardened network controllers (CAN, LIN, FlexRay or others), timers, clock generators, sensor interfaces (analog/digital), PWM modules, and others. The choice of peripherals is determined by the application domain and these are often integrated on the same die as the processing element.

Beyond the functional requirements of dependability and reliability, other factors like maintainability, safety and security are important [4]. Compute units are expected to offer a very high mean time between failures (MTBF), ease of fault diagnosis, and ease of maintenance. Also, since drivers are not trained like pilots to use the systems properly, the electronics must be capable of handling varying driver input patterns without degrading safety. Such complex requirements along with the rising computational requirements and economic constraints make the design of automotive embedded systems challenging.

With features becoming more sophisticated and diverse manufacturers have had to address the the rising complexity in design, development, and management of these systems, giving rise to multiple standards: OSEK/VDX, AUTOSAR, and the ISO 26262. These define the way systems are developed, both at the hardware and software levels. The ‘Open Systems and their Interfaces for the Electronics in

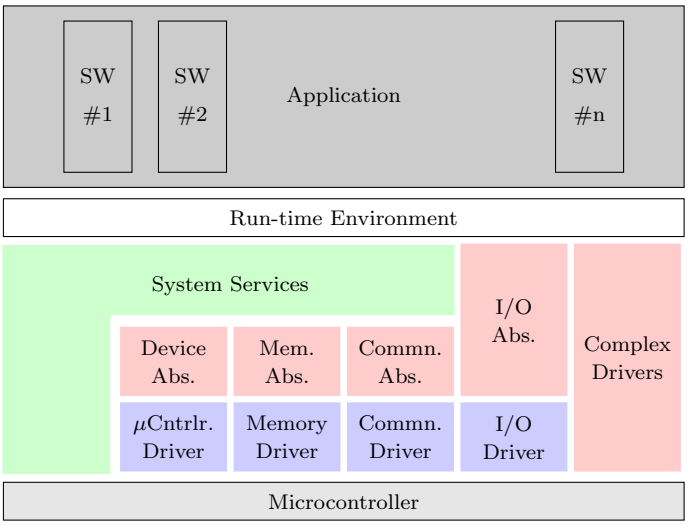


Figure 2.1: The abstraction layers in AUTOSAR.

Motor Vehicles' (OSEK) was founded by a consortium of German automotive companies to standardise the software architecture for different embedded platforms and promote software reuse. It includes specifications for the operating systems, interfaces to the underlying platform, communication, network management, and fault-tolerance mechanisms [5]. This was superseded by the Automotive Open Systems Architecture (AUTOSAR) specification, jointly developed by automotive manufacturers in 2003 [6]. The 'Road Vehicles – Functional Safety' standard (ISO 26262) describes standard procedures for fault detection, fault handling, and fault avoidance in automotive systems to prevent violation of system safety requirements [25, 26]. The entire ECU architecture and software development for automotive applications is governed by these standards.

The main advantage of AUTOSAR compliant system architecture is the decoupling of software and hardware. This enables high-level integration of software functions through abstracted hardware definitions via an API. Thus, a software function that is compliant with AUTOSAR can be executed on any vendor's micro-controller, as long as the device supports the AUTOSAR run-time environment (RTE). The AUTOSAR abstraction model is shown in Figure 2.1 [6].

The bottom layer in Figure 2.1 is the hardware element – the processing system (GPP or MCU) with its peripherals, network interfaces, and memory structure. Above this is the micro-controller abstraction layer (MCAL) which integrates the

basic platform-level drivers for configuring and using the hardware elements. This includes micro-controller drivers, communication and I/O drivers, and memory abstractions, providing a hardware-independent API to the upper layers (or application). The ECU abstraction layer (ECUAL) further abstracts platform details by providing a standardised interface for accessing MCAL features from the operating system. This layer includes complex drivers for specific features of the platform (like accelerators or configurable features of the network interface) as well as abstractions for platform-level drivers. Further, a service layer (SRV) may be added on top for handling system-level and/or device-specific services for the operating system, like network management or watchdog timers. These three layers form the basic software layer (BSW) that is required to support AUTOSAR compliance for the hardware platform. With the BSW in place, AUTOSAR is able to define a collection of software functions for accessing features/capabilities of the hardware platform from an application without explicit use of low-level details.

The AUTOSAR run-time environment (RTE) interfaces the BSW to the application software by enabling communication using a set of signals (sender/receiver) and services (server/client model). The RTE abstracts the basic software components from the application code, allowing generic and portable code development for software-based automotive applications. Further, the RTE enables a component-based structure for application code and handles inter-component communication, promoting scalability and portability of applications across hardware/software platforms. AUTOSAR compliance allows architecture designers and automotive suppliers to deliver scalable software functionality and optimised ECU architectures for the different functional domains.

2.2 In Vehicle Networks

Modern vehicles use different network protocols in different domains, the choice determined by factors such as the functional requirements of the domain, criticality, cost, and others. Among the many protocols, Local Interconnect Networks

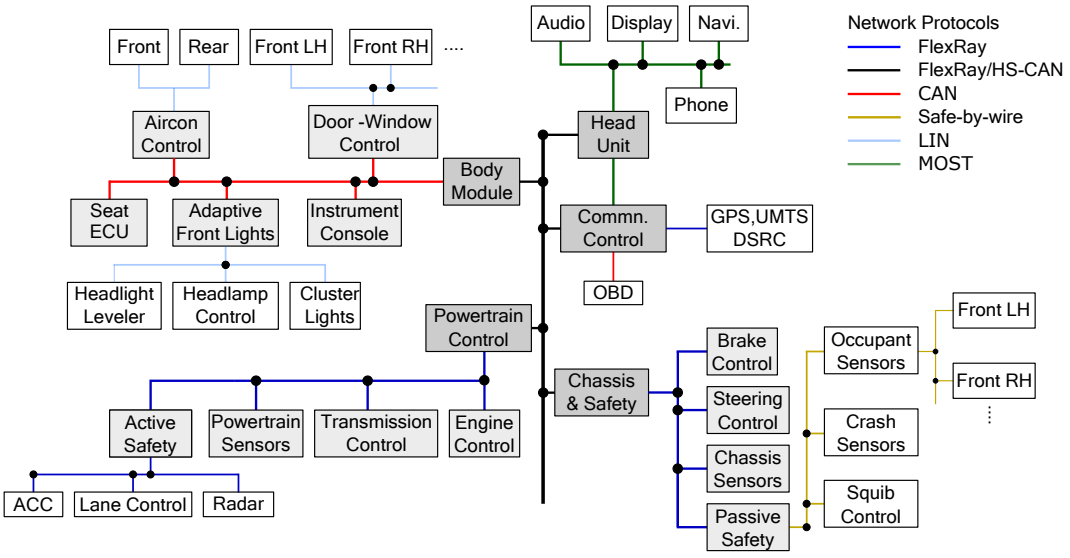


Figure 2.2: Typical in-vehicle network architecture in a modern car.

(LIN), Controller Area Networks (CAN), FlexRay, and Media Oriented Systems Transport (MOST) are the most widely used protocols by the different manufacturers today [9, 27, 28]. Special networks like safe-by-wire may be employed for passenger safety systems like airbags and other active protection systems. A simplified scheme of typical in-vehicle network architecture in a modern vehicle is as shown in Figure 2.2.

As already mentioned, the in-vehicle network is partitioned into different functional domains. A backbone network is used to interconnect the different domains and to transfer data between them. High Speed CAN (HS-CAN) or FlexRay (for newer premium vehicles) are usually chosen as the network backbone because of their higher bandwidth. The different domains connect to the backbone through *gateways*. A gateway may provide interfaces to multiple networks like Low Speed CAN (LS-CAN or plain CAN), HS-CAN, FlexRay, LIN and MOST depending on the the functional requirements of the domain (and the network protocols used to implement them), in addition to interfacing with the backbone network. Gateways use a store-and-forward scheme, where data from one domain is stored, transformed to the new interface format (if necessary), and transmitted over the backbone to the destination domain at appropriate times. The architecture using a gateway as a central active unit also helps provide fault isolation preventing fault propagation across domains.

2.2.1 Controller Area Networks

Despite being three decades old, the CAN protocol is still widely used in production vehicles. The CAN specification was developed by Bosch in the mid-1980s with the aim of reducing the number of point-to-point links used for communication between ECUs. While the introduction of networking resulted in a reduction in the complexity of the wiring harness, it also enabled newer approaches to computing like the use of shared sensors between different functions or applications [8]. CAN gained widespread adoption and soon became the de-facto communication protocol for in-vehicle networks. Features that make CAN attractive for automotive applications are its widespread adoption, its lower cost and complexity (compared to networks like FlexRay or TTE), its robustness, and its bounded delays. CAN is widely used in powertrain communication, the chassis domain (HS-CAN at 500 kb/s) and the body domain (low-speed CAN at 125 kb/s) [27]. CAN for automotive networks became an ISO standard in 1994 [29, 30].

2.2.1.1 Protocol Specification and Scheduling

CAN defines a bus topology using a shared physical medium to which multiple nodes are connected. The number of supported nodes is limited by the electrical properties of the channel. At the physical layer, CAN uses a non-return-to-zero (NRZ) encoding scheme, and transmits/receives data over unshielded twisted pair cables. The physical layer implements *logical AND* functionality i.e., if any node transmits a logical '0', then the bus will remain a state '0' even if other nodes have transmitted logical '1'. This mechanism is also the key to the CAN arbitration scheme for channel access.

The CAN standard uses an event-triggered communication scheme that requires participating nodes to synchronise the start of communication. This is achieved by tracking edges during any transmission. To ensure that there are enough transitions on the bus even when transmitting a continuous pattern of zeros (or ones), CAN uses bit stuffing with the stuffing length fixed at 5 bits.

Messages are exchanged between the different nodes by encapsulating them in CAN frames, which can be transmitted periodically or otherwise. The frame format used for Automotive CAN networks (CAN 2.0A) is as shown in Figure 2.3. The frame has the following sections:

1. The *Start of Frame* is a single dominant bit (logical '0'), which marks the start of a *Data Frame* or *Remote Frame*. A node can only start transmission when it detects that the bus is idle. It is required that all nodes synchronise to the leading edge of the *Start of Frame* bit.

2. The *Arbitration* field consists of an 11-bit node *Identifier* (which also defines its priority) and a remote transmission request (RTR) bit (dominant for *Data Frame* and recessive for *Remote Frame*).

3. The *Control* field consists of 4-bit *Data Length* and 2 reserved bits.

4. The *Data* field can contain up to 8 bytes of information (including the option for zero byte data).

5. The *CRC* field contains a 15-bit cyclic redundancy check (CRC) code followed by a single recessive bit as the CRC delimiter.

6. The *Acknowledgement* field has a 1-bit Ack slot followed by a 1-bit Ack delimiter. Any node that validates the transmitted data will override the recessive bit sent in the Ack Slot by the transmitter to indicate that the frame has been transmitted correctly. However, since CAN is a broadcast network, the node which acknowledges the receipt of data may not be the intended recipient.

7. The *End-of-Frame* field (EOF) is a sequence of seven recessive bits followed by an inter-frame space.

The CAN specification also describes special frame formats like the *Error Frame* and *Overload Frame*. The *Error frame* is composed of two fields, an *error flag* field and a delimiter field. When any listening node detects a transmission error in the

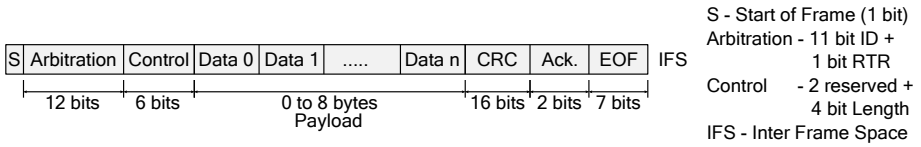


Figure 2.3: Frame format for Data Frames and Remote frames

current frame, it immediately transmits the error frame containing six consecutive dominant (active error flag) or recessive (passive error flag) bits. These violate the bit stuffing scheme of normal frames, thus indicating the error condition to all nodes listening to the bus. The actual frame in transmission is thus corrupted, and may be retransmitted at a later stage. *Overload frames* are used for flow control by the receiver, and have six consecutive dominant bits (*overload flag*) transmitted during the inter frame space or at the start of frame.

Any node may start a transmission if it determines that the bus is idle (recessive or logical high). If two nodes begin to transmit at the same time, the access conflict is resolved using a non-destructive priority based arbitration scheme. The priority is determined by the *arbitration* field within the frame structure. Since the physical layer performs a *logical AND*, if one node transmits a recessive bit while another transmits a dominant bit, the resultant state of the bus is dominant. During arbitration, the transmitting node monitors the bus state and compares it to the logical value transmitted by it. If the node observes a dominant state on the bus when it had transmitted a recessive bit, the node should immediately halt transmission, since another node with higher priority is trying to access the bus. So the node with the lowest numerical value for its identifier has the highest priority. Since it is required that the farthest node must see the state on the bus before deciding to proceed, the bit time must be at least twice as long as the propagation delay. This factor limits the maximum bandwidth of CAN networks as a function of the bus length.

The CAN protocol also specifies mechanisms to detect continuous or permanent failures at any node due to internal (hardware or software) errors and to permanently disable these nodes. The scheme relies on transmit and receive error counters that are incremented and decremented when specific events are detected [31].

However, the drawback of the scheme is that it depends on self-diagnosis performed by the node. An example case where such a scheme would fail is the "babbling idiot" fault: a faulty clock at the node causing it to continuously transmit dominant state on the bus [32, 33]. The CAN specification provides clear definitions for the physical layer, transfer layer, and object layer (together referred to as the data link layer). The physical layer specifications deal with the signal levels and bit representations on the transmission medium. The data link layer specifies the procedures for message handling, error detection and handling, arbitration, and timing. The application layer can decide on the schemes for start-up, periodic transmissions, and data segmentation procedures depending on the specific requirements of an application.

To establish reliable communication between ECUs, they must be integrated using a suitable topology based on the communication requirements. Then a schedule and routing scheme must be determined for each message, ensuring that the end-to-end latencies demanded by the application are met. These design choices are typically made manually by designers in incremental steps resulting in sub-optimal solutions. An approach to automating topology determination and message routing for CAN networks was presented in [34]. The authors formulate the topology selection problem as an Integer Linear Program (ILP), which is solved using a SAT (Satisfiability) solver combining the benefits of a pseudo-Boolean (PB) optimisation and evolutionary algorithms.

Several schemes have been described in the literature for scheduling communication on CAN networks. These include preemptive, non-preemptive, fault tolerant, and real-time schemes. Many existing optimisation strategies use Integer Linear Programming (ILP) or Evolutionary Algorithms (EAs) that have been adapted for the bus (or other) topology used in a particular setup. Given a set of nodes and their communication specifications, these algorithms try to assign priorities to the different communication nodes by framing the communication requirements and the network constraints as a programming problem, and use the above optimisation algorithm(s) to determine the set of feasible solutions. Most approaches follow scheduling based on deadline specification - either a fixed priority approach or a

dynamic priority scheme (using more ID bits). While fixed priority schemes result in lower schedulability, dynamic priority is inefficient since it requires more ID bits (and thus higher overheads) [35]. A non-preemptive scheme provides real-time communication, and uses a mixed traffic scheduler to achieve the best compromise between a fixed priority approach (determinism) and dynamic priority scheme (higher schedulability) [35, 36]. In [37], the authors evaluate a dynamic priority scheme on top of standard CAN networks to overcome the limitations of single priority assignment, by generalising a multi-priority window per message approach, while maintaining backward compatibility with legacy CAN. A similar scheme is discussed in [38] for improved determinism of critical messages, where the windows are defined as the fault-tolerant feasibility windows for transmitting critical messages whose priorities are determined offline. The scheme aims to overcome the weaknesses in native fault-tolerance supported by CAN which treats all messages with equal criticality, resulting in its inability to meet reliability deadlines for critical messages.

2.2.1.2 Implementations and Extensions

CAN has been widely implemented, including standalone controllers and synthesizable IP cores by different automotive vendors. Among many other IP implementations that can be directly used for reconfigurable hardware, FPGA vendors Xilinx and Altera provide optimised CAN controller IP cores for their respective families of devices [39, 40]. Despite the wide popularity and availability of CAN controllers and IP, the bandwidth limitation and reliability of CAN has been a constant challenge, especially for emerging safety-critical functions like drive-by-wire which demand high levels of determinism, real-time support, and increased bandwidth.

An interesting approach to extend the bandwidth of CAN networks is that of CAN+, a backward-compatible CAN protocol that offers 16× higher bandwidth than plain CAN. This is achieved by over-clocking data transmission on the bus during phases where it is certain that only a single device is transmitting. In

instances where multiple devices may transmit, or if a conflict is identified, the CAN+ node defaults to standard CAN to ensure collision free access to the bus. The protocol modifications in CAN+, backwards compatibility, and implementation on FPGAs are discussed in [41]. Similar concepts for improving the bandwidth of CAN using overclocking [42] and a dual-speed approach [43] were attempted, though these suffered from compatibility issues with standard CAN communication on the same network. Flexible Data Rate over CAN (CAN-FD) is a recent enhancement that extends this concept using an enhanced data-link layer protocol [44]. This allows a CAN-FD node to communicate on a standard CAN data bus and utilise its flexible data rate capabilities to exchange larger payloads at higher speeds with other CAN-FD capable nodes on the same network. Extensions to the reserved bits in the CAN frame format are used to distinguish between standard CAN and CAN-FD messages.

Another extension to improve the available bandwidth is to transmit CAN packets encapsulated into an Ethernet frame [45]. The work, *CANoverIP*, discusses the message data and schedule migration from traditional CAN networks, the associated protocol overheads in Ethernet, and advantages of the scheme. However, traditional CAN networks (including CANoverIP) suffer from a lack of predictability (or upper bounds on communication latency) under high communication load. Time-triggered CAN (TTCAN) aims to address this by adapting the CAN standard to the time-triggered domain, enabling simpler scheduling of messages with an upper bound on transmission latency. TTCAN was developed by Bosch as an extension to the existing CAN protocol [46]. TTCAN nodes are backward-compatible with standard CAN nodes, but have the provision to disable the automatic retransmission feature in case of errors. By ensuring that both modes can work with the same controller, they envision the development of a communication system which can support both time-triggered and event-triggered communication.

TTCAN communication is defined by a basic cycle which is a concatenation of one or more time-triggered slots (called *exclusive windows*) and one event-triggered slot (called the *arbitrating window*). Exclusive windows use predefined communication schedules while the arbitrating window uses the standard CAN arbitration

scheme (dynamically allocated based on priority). Unlike other similar protocols (like FlexRay), it is not mandatory that the basic cycle be one and the same. It is possible to configure different basic cycles with different combinations of exclusive and arbitrating windows, ordered in some form and executed as a loop. The ordered list of basic cycles is called a *system matrix*. At the start of each basic cycle, the master node (which maintains timing information) can cause the network to switch to standard CAN (stop functioning in TTCAN mode) and to resume functioning in TTCAN mode by transmitting commands over special slots called *reference messages*. However, TTCAN still suffers from the dependability issues suffered by the underlying CAN framework and thus has not found widespread acceptance like standard CAN [8]. The TTCAN framework and message scheduling schemes for TTCAN are also discussed in [47], among others.

2.2.2 FlexRay

The FlexRay communication protocol was developed by the the consortium of automotive manufacturers and suppliers (later called the FlexRay consortium) that was formed to define the requirements for a future in-vehicle communication system. The main task was to develop a protocol specification that can provide higher communication bandwidth with high accuracy, determinism, and native support for fault-tolerance, specifically for emerging applications like drive-by-wire (or generally x-by-wire). Specific requirements were outlined by the consortium to attain the global requirements for a high bandwidth deterministic protocol: [48]

1. The global goal: The protocol must provide deterministic communication with bounded latency and small latency jitter.

2. The protocol must also support event-triggered (or on-demand) communication at runtime, the bandwidth of which must be configurable (including the option for zero bandwidth).

3. Event-triggered communication must not affect deterministic data transfer.

4. Support for multiple levels of fault-tolerance, with optional non-fault-tolerant modes.

5. Support for multiple transmission rates, with a maximum of 10 Mbps.

6. Host operations for network management, message filtering, interrupts, and monitoring services.

7. The behaviour of the nodes (ECUs) and the network must be predictable in the presence and absence of errors.

8. The communication system should ensure a cluster-wide consistent view of time (called Global time) with known accuracy at all nodes.

The different aspects of the protocol were developed based on these requirements and the set of constraints associated with in-vehicle networks. The consortium resulted in development of FlexRay specification version 2.1 and later version 3.0. The protocol was introduced into production vehicles at the end of 2006 with the *BMW X5*, enabling the use of an advanced adaptive damping system [49, 50, 51]. By 2008–2009, high-end models from *Audi* and *BMW* featured several ECUs interconnected through FlexRay networks, for both critical and non-critical functions [52]. The consortium was disbanded in 2009 after concluding their work on the protocol.

2.2.2.1 Protocol Specification

The FlexRay specification defines a communication scheme that is flexible/configurable with regard to topology and redundancy [8, 53]. A FlexRay network can be configured as a bus network, star network, a multi-star network or as a hybrid combination. The protocol also defines use of an independent physical medium as well as encoding-decoding modules for the two channels, which can be configured independently by the host.

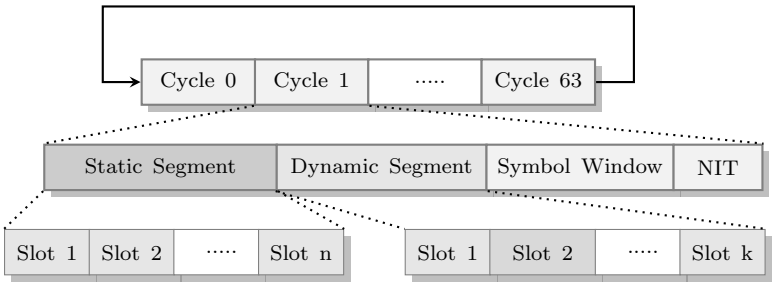


Figure 2.4: FlexRay communication cycle

The fundamental element of the media access scheme in the FlexRay protocol is the communication cycle, which repeats itself over time, as shown in Figure 2.4. Each cycle is comprised of four segments:

1. *Static Segment* which uses a static slot-based access mechanism for critical data transmission in a deterministic manner. Any ECU can send a frame of data in the one (or more) slots that are assigned to it. The slot width is fixed across the whole network.

2. *Dynamic Segment* which uses a dynamic slot-based access scheme enabling communication of event triggered data of arbitrary length. The slot width is dynamic, depending on the amount of data that needs to be transmitted and access to the medium is controlled by priorities assigned to the ECUs.

3. *Symbol Window* which is used to transmit special commands like wake-up which will send a pattern on the bus to wake-up 'sleeping' nodes to initiate communication.

4. *Network Idle Time* (NIT) which is the period when the bus is idle. This period is used by participating nodes to make clock adjustments and to align and correct the global view of time at the individual nodes so they stay synchronous.

At the medium access level, any communication cycle can thus be visualised as a concatenation of a static window (comprising static slots), a dynamic window (comprising the dynamic slots), the symbol window, and network idle time. The

size of each window can be configured at design time (statically). The dynamic window and symbol window can also have a zero size in some configurations.

Two distinct protocols govern the medium access schemes used in the static and dynamic windows. The static window uses a time-triggered slot-based access scheme based on the TDMA medium access protocol. A single node may be configured to have access to several slots in the same cycle, in the same or different segments. Any node having access to the slot *must* transmit data on that slot, which can be either valid data, or null data (all zeros). If the data to be transmitted is less than the slot-size, it must be zero-padded to the payload size of the slot. If the data to be transmitted cannot fit in one slot, it must be segmented across multiple slots.

The dynamic window uses an event-triggered dynamic slot-based access mechanism based on the Flexible TDMA access scheme. The window time is divided into *minislots*. Each node may have access to a configured number of minislots, which may or may not be contiguous. Each node can start the transmission of a frame within its allocated minislot. Once a transmission has started, the minislot counter is not incremented until the transmission has been completed by the node. Thus, each minislot may have a different size depending on data availability at the node. On the contrary, if the node has no data to be transmitted, the slot remains idle and the minislot counter is incremented at the slot boundary.

In the case of dual-channel configurations, the slot counters for the two channels are incremented simultaneously in the static segment, but may be incremented independently according to the data transmission within the dynamic segment. This is because media access on the two communication channels may not necessarily occur simultaneously in the dynamic segment. However, both channels use common grid timing based on minislots.

Any data that is transmitted over the FlexRay bus must be encapsulated into a FlexRay frame with the structure shown in Figure 2.5. The frame comprises three parts : the frame header, payload segment and the trailer segment. The five byte frame header includes the frame identifier, payload length, cycle identifier, header CRC, and flags to indicate special frame types (null, sync, startup). The payload

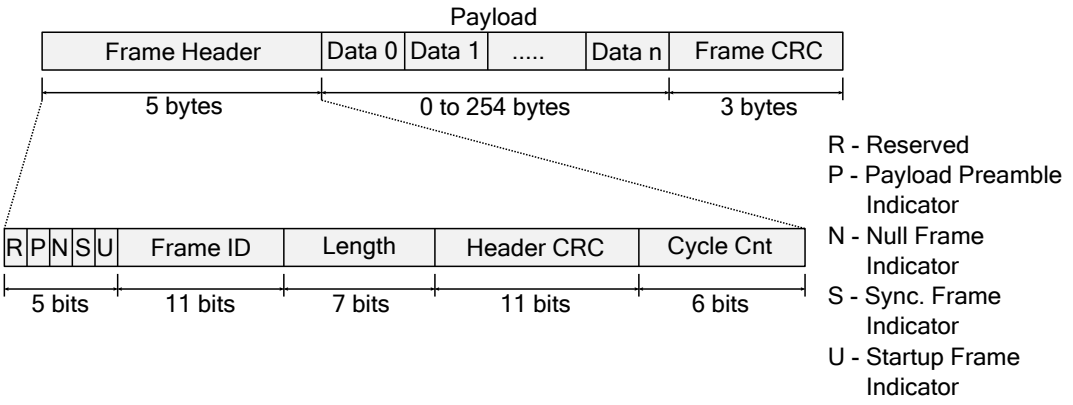


Figure 2.5: FlexRay frame structure

segment can contain 0 to 254 bytes of application data. The payload is indexed starting with 0 for the first byte after the header segment and increasing with subsequent bytes.

For dynamic segment payloads, the first two bytes of the payload segment can be optionally configured as a *message ID* field, which can be used by the receiving nodes to filter data based on the message ID. The status flag *payload preamble indicator* in the header segment indicates the presence of a message ID in the current frame. Similarly, for frames transmitted in the static segment, the first 12 bytes of the payload may be (optionally) used as the *network management vector* (NMV). As in the case with the *message ID*, the presence of a *network management vector* is also indicated by the status flag *payload preamble indicator*. The NMV can be of configurable length (0 to 12 bytes) and is defined statically (as a global parameter) to ensure that it stays uniform across all nodes in the network. The controller does not process the NMV; the host is responsible for writing the NMV like other application data. Data transmission always starts with the first byte of data, with the most significant bit first.

The basic architecture of a FlexRay node is shown in Figure 2.6. The host controller can be an embedded controller within an FPGA, a standalone microcontroller, or a full-fledged processor. The host runs the application software that implements the ECU functionality. The host is responsible for controlling and configuring the FlexRay communication controller (CC), which executes the protocol specification. The CC is connected to bus drivers (BD), which act as the physical

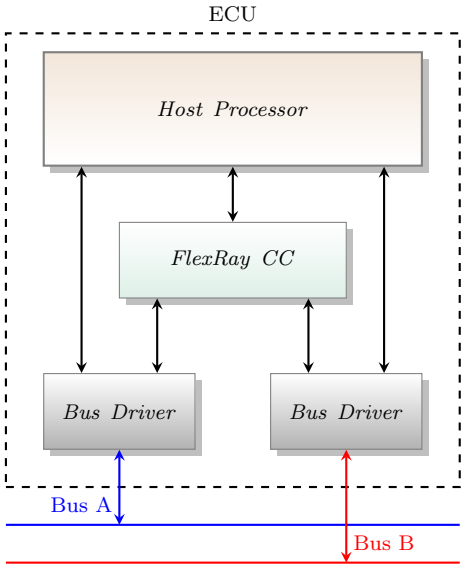


Figure 2.6: A FlexRay node.

and electrical interface to the transmission medium. Independent bus drivers are used if the CC is configured in dual-channel mode. The bus driver is controlled and configured by the host, using a dedicated interface to exchange configuration and status data. The protocol defines an optional hardware module called the bus guardian (BG), which can be attached to each CC. The Bus Guardian is configured with the communication schedule of the attached node and prevents it from transmitting data in slots other than the ones assigned to it. It is not mandatory that each node support dual channels or have a bus guardian, though this is recommended for critical functions such as x-by-wire.

The FlexRay protocol assumes the presence of a steady physical medium for data transfer, with negligible bit-error rates [54]. Hence the standard does not provide a mechanism for retransmission or error correction. The protocol employs bit-level redundancy at the physical layer and majority voting at the receiver for reliable message exchange. Bit errors in the received frame (after majority voting) will result in the frame being discarded, usually resulting in loss of information. Data loss must be managed through the redundancy provided by a dual channel implementation or using higher layer services.

2.2.2.2 Communication, Scheduling, and Implementations

For a time-triggered system to work, every node in the cluster must have approximately the same view of global time. The FlexRay protocol uses a distributed synchronisation scheme whereby each node adjusts its own clock view by observing the timing of synchronisation frames transmitted by the timing sources. To compute the deviation and to apply the correction factor, a fault-tolerant algorithm is described by the protocol.

To initialise communication, all nodes must be awakened and synchronised. The FlexRay protocol requires at least 2 nodes per cluster to act as the timing sources. Special patterns called Wakeup Patterns (WUP) are transmitted on the idle bus to wake up nodes and to initialise communication. Prior to transmission of the WUP, a collision avoidance symbol (CAS) is transmitted to ensure transmission proceeds without collision. For clock startup by a *coldstart* node (which is a timing source), the preconfigured values are used to start the local clock and transmit the synchronisation frames. The non-coldstart nodes use the initialisation values computed from the received synchronisation frames to start a synchronised clock at the node.

FlexRay uses a set of parameters to configure the behaviour of a node on the network. These parameters can be categorised into three different categories

1. Node Parameters - these parameters are defined locally for each node. Examples are the keyslot ID (*pKeySlotId*), clock correction limits (*pOffsetCorrectionOut,pRateCorrectionOut*), and duration of node local clock (*pdMicrotick*).

2. Network Parameters - these parameters configure the protocol behaviour and must have the same value at all nodes within the network. Examples are duration of the static slot (*gdStaticSlot*), the payload length of a static segment frame (*gPayloadLengthStatic*), and the maximum number of minislots in the dynamic segment (*gNumberOfMinislots*).

3. Protocol Constants - these are constant values defined by the protocol and are hard-coded with the same value at all nodes. These include the CRC initialisation polynomial for computing Frame CRC (*cCrcPolynomial*), the maximum payload length in a frame (*cPayloadLengthMax*), and the number of samples taken in determination of a bit value (*cSamplesPerBit*).

The Field Bus Exchange Format (FIBEX) is the most widely adopted standard for representing FlexRay configuration for nodes and clusters [55], as well as for other protocols like CAN and MOST. FIBEX uses an *XML* file to specify nodes, networks (or clusters), and their associated configurations. FIBEX is standardised by the Association for Standardisation of Automation and Measuring Systems (ASAM) consortium.

To ensure that the connected ECUs can communicate over the bus, messages from all ECUs must be routed through the bus satisfying the maximum latency requirements of the respective applications. Periodic messages with fixed size can be allocated to the static segment, while burst mode data is generally handled by the dynamic segment to ensure network efficiency. FlexRay 2.1 does not allow slot-multiplexing in the static segment, and hence the static slot(s) assigned to a node are valid for all cycles. On the contrary, in the dynamic segment, nodes may be assigned slots on a per-cycle basis and hence scheduling communication is more difficult in the dynamic segment.

Much work has been done by the research community on efficient scheduling for FlexRay networks. In the static segment where the slot size is fixed by design, scheduling is usually done by formulation as an integer linear programming problem, which can be solved using ILP solvers. However, in the dynamic segment, scheduling communication is a non-trivial problem. Optimisation of the static and dynamic segment of the FlexRay protocol has been widely addressed in [56, 57] and [58, 59, 60], among many others. [61] is a detailed survey of scheduling algorithms and provides a comparison between optimisation strategies like simulated annealing, genetic, hybrid-genetic, and probabilistic approaches. In the static segment, the aim of scheduling is to efficiently allocate slots to the different nodes in

a cluster, consuming the minimum number of static slots and cycles, in such a way that all the communication requirements at each node are met. The key criterion here is the periodicity of the messages at each node, and the bandwidth requirements of each message. In the dynamic segment, the aim is slightly different. Although the global aim is to provide communication slots (dynamic) to all messages, consuming the minimum number of slots, the optimisation arguments (or requirements) are based on priorities and deadlines associated with each message, along with periodicity and bandwidth requirements. In other words, scheduling for the dynamic segment aims to prioritise messages and assign slots to them based on the criticality of the message, bandwidth, and requirements of other nodes.

Platform agnostic implementations of the FlexRay communication controller and bus guardian modules are available from a number of vendors. The most prominent of them is the Bosch E-Ray [62] IP module, which can be targeted at ASIC as well as FPGA platforms. The E-Ray module is widely used in integrated and standalone FlexRay solutions from vendors such as Infineon and NXP. Freescale Semiconductor uses a proprietary IP module FRCC2100, which is also used in many ASIC solutions, either integrated with an MCU or as discrete standalone units, from Freescale, Philips, and Decomsys [63]

Attempts to develop and demonstrate FlexRay communication controller functionality on reconfigurable hardware are also described in the literature. In [64], the authors highlight challenges like physical-layer design, cycle and schedule design, and selection of termination, sync, and startup nodes which were all simpler design considerations in the case of CAN. They also discuss design considerations for building an SoC-based architecture for reliability and safety-critical networks like FlexRay. In [65], the authors present an implementation of the FlexRay communication controller on a reconfigurable platform. The approach however, only discusses the protocol operations control module, which controls the actions of the core modules of the communication controller. The work does not describe any effort undertaken to efficiently utilise the FPGA fabric or heterogeneous resources. [66, 67] describe implementation of the FlexRay Communication Controller using

the specification and description language (SDL) as the platform and later translation to hardware using Verilog. Their work approaches the protocol from a high level of abstraction and hence does not discuss hardware design details or architectural optimisations. Work has also been done on implementing the FlexRay Bus Guardian module on reconfigurable fabric. A comprehensive outline of the Bus Guardian specification and approaches to implementing it on FPGAs are discussed in [68, 69].

The work in [70] discusses an approach to improve the energy efficiency of a FlexRay controller by allowing it to be controlled by an intelligent communication controller (ICC). The ICC, which takes over the bus from the ECU when the latter goes to sleep, prevents the ECU from being woken by erroneous transmissions allowing the node to achieve higher power efficiency. The concept validation on FPGA and the proposed architecture are also discussed in the paper. However, they use a proprietary implementation of the FlexRay communication controller, which is not available to the research community.

The interface specification described by AUTOSAR, called the FlexRay AUTOSAR Interface Specification Standard [71], is the industry standard for the software specification of the FlexRay nodes, and all implementations must comply with this standard.

Switched FlexRay

In [72], the authors introduced the *FlexRay switch* concept, an architecture similar to the Ethernet switch, which can exploit branch parallelism to improve the effective bandwidth of FlexRay networks. In standard FlexRay, any node can communicate over the bus by transmitting framed data in the slot(s) assigned to it in the static or dynamic segment. Exploiting cycle-level multiplexing of slots in static and dynamic segments, multiple nodes can share the same slot in different cycles, as in the case of odd/even cycle multiplexing where one set of nodes is assigned slots in all odd cycles whereas another set of nodes (which may include some from the first set) is assigned slots in all even cycles. This scheme of cycle-level *slot multiplexing* leads to higher utilisation of system bandwidth. In a

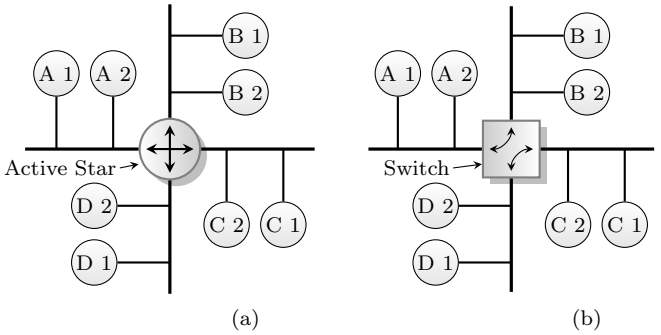


Figure 2.7: A standard FlexRay network topology (a) and a switched network (b)

standard FlexRay configuration, as in Figure 2.7(a), this would mean that node *B2* can send data to node *A1* in slot 1 of cycle 1 while node *A1* can reply in slot 1 of cycle 2. The *active star* is an active repeater that passes information from one branch to all the other branches.

With switched FlexRay, the central switch architecture allows us to exploit *branch parallelism* whereby, the switch will repeat frames only on branches which contain the intended recipient. This allows the same slots to be used simultaneously by nodes in distinct segments and the intelligent FlexRay switch schedules the branch to which information must be relayed [72, 73, 74]. Thus, as shown in Figure 2.7(b), while node *B2* can send data to node *A1* in slot 1 of cycle 1, node *D1* can simultaneously send data to node *C2* and the switch, knowing the schedule, connects the corresponding nodes through the switch fabric. The authors demonstrate a switch architecture developed as an extension of an ECU on reconfigurable hardware, which can then utilise intelligent scheduling to improve bandwidth. With slot multiplexing and branch parallelism, each slot within a cycle may have multiple destinations and thus different switch configurations are selected at run-time.

[73, 74] discuss scheduling for FlexRay switches. As opposed to traditional schemes with an active star gateway, where the communication schedule is a static assignment of data to be repeated across all branches of the active star, the scheduling of switched FlexRay nodes is more complex because of the dynamic nature of message crossovers at the FlexRay switch. Though initial work in this area used genetic algorithms, ILP, and mixed ILP methodologies, they were primarily concerned

with optimising the communication schedule for a cluster, as opposed to a group of networked clusters, which is the case presented by the FlexRay switch. [73] provides a complete problem formulation and detailed solution using polynomial-time decreasing first fit algorithm to generate an initial schedule and a branch and price algorithm (using Dantzig-Wolfe decomposition of the full ILP) to optimise the schedule. [74] provides an ILP formulation to the problem and a simplified heuristic for the solution, which performs as well as the exact ILP solution for realistic cases.

2.2.2.3 Performance, Applications, and Limitations

[75] presents a formal analysis of the timing specifications of distributed systems interconnected by a FlexRay network. The model has a real-time kernel running on each ECU that uses a fixed priority scheduler (FPS) and static cyclic scheduler for scheduling multiple tasks within the ECU. The FPS emulates sensor data processing, while the cyclic scheduler schedules data for transmission, as is the case in most common ECU nodes. The model also uses different approaches for transmitting messages on the static and dynamic segments allocated to the node. Schedulability analysis is performed as a simplified bin-packing problem to demonstrate the overall response times in static and dynamic segments (worst case analysis) for distributed systems. [76] presents a similar analysis of distributed computing units interconnected by FlexRay networks. [77] analyses the performance of a complete ECU on a FlexRay network to determine parameters like end-to-end delays for different message types, buffer space requirements, and bus utilisation. Their work analyses the FlexRay bus in a compositional manner, contrary to the isolated technique used in [75].

[78] demonstrates a protocol analyser that can extract FlexRay bus parameters from the network. The aim is to enable an FPGA based test system for analysing and testing FlexRay based systems for rapid system prototyping. [79] is a measure of the robustness of a FlexRay system by considering the uncertainties encountered within the automotive architectures. The work describes a scheme based

on *info-gap* decision theory to compute the largest point of uncertainty within the performance limits specified for the system. The work also demonstrates a case-study for x-by-wire applications and models different uncertainties (uncertain future bus-loads and overlooked communication) to show the application of robustness analysis and the extensibility of schedules for future applications. In [80], the authors improve on these schemes and present a novel configuration method by considering both the network utilization (bandwidth and end-to-end delays) and the utilisation of the static segment.

In [81], the authors present a constraint-driven design approach for the synthesis of controller parameters and the associated the control-related task and message schedules for a FlexRay-based implementation platform. To achieve this, the problem is formulated as a constraint satisfaction problem (CSP) to synthesise feasible platform parameters for a specific platform (operating system, hardware, and FlexRay communication system). For systems dealing with dynamic real-time conditions and constraints, [82, 83] describe techniques to generate optimum FlexRay parameters for static and dynamic segment communication. [84] discusses new FlexRay-specific metrics that can quantify the extensibility and sustainability (specifically with respect to the protocol) of existing schedules for future application.

Though the FlexRay protocol provides error detection schemes, the error correction is generally handled by dropping the corrupt frame, which usually results in lost data. [85] proposes a scheme to handle this at the application layer, using an acknowledgement-retransmission scheme, where the retransmission slot is allocated on top of the existing schedule. The problem is formulated as a MILP problem, which optimises the fault tolerance metric and subsequently improves the existing schedule by reducing the acknowledgement and retransmission latencies.

A scheme to automate verification of complex FlexRay clusters for network inconsistencies is discussed in [86]. In this work, an FPGA based active star node is used to detect the presence of inconsistent conditions (referred to as Slightly-Off-Specification failure modes) in actual systems. The hardware was used to inject

delays (slot-boundary violations) and corrupt frames on the network to observe responses of the attached nodes. [87] follows a similar approach integrating a host PC with an FPGA based active star node that can monitor the bus and inject faults. It features a few additional schemes like replaying bus data (in-order or out-of-order) and can trigger startup or re-integration of cluster nodes. [88] describes a verification flow to measure and quantify the effectiveness of fault-tolerant mechanisms (FTM) built into the FlexRay protocol. The verification flow helps designers quantify the effectiveness of FTMs and the associated costs in designing a fault tolerant communication scheme (with the required safety and integrity levels) in terms of hardware costs and performance degradation.

[89] describes an analysis of the issues associated with communication of nodes belonging to different clusters, referred to as *Clique problem*. In the analysis, the nodes in one cluster are free to communicate among other members, but not to members of another cluster (such groups are called cliques). The communication breakdown is common with time-triggered systems, usually caused by local faults resulting in loss of synchronisation between the different clusters. The work describes the lack of explicit support within the protocol, studies real FlexRay clusters, and provides useful directions for implementing clique detection and avoidance mechanisms.

As discussed, fault-tolerance within a FlexRay network is not built-into the protocol definitions and depends on application level schemes. A scheme that enables automatic migration of a critical task from a faulty ECU is described in [90], using redundant communication slots in the schedule. The FlexRay bus schedule is extended by adding redundant slots to permit changes in the slot assignment and solutions to migrate tasks using static and/or dynamic segments are presented. However, the scheme uses a dedicated coordinator node to manage the migration, resulting in additional hardware and a possible single point of failure at the coordinator. An improved version of this work was presented in [91], where a distributed coordination scheme is used for task migration, improving the safety and reliability of the solution. The approach also reduces the communication overhead, compared to their initial work. [92] improves further, using a self-organising

distributed coordinator, which uses predetermined information about communication dependencies to determine valid (re)configurations for nodes on a FlexRay network. Here, *reconfiguration* refers to the tasks assigned to each of the static modules defined within the hardware setup. On the other hand, [93] uses dynamic partial reconfiguration (PR) supported on newer FPGAs to reconfigure faulty communication modules, ensuring reliable communication. The work also describes a scheme to integrate a fault-tolerant communication network scheme within an AUTOSAR-compliant environment.

[54] describes the limitations of the FlexRay networks for establishing reliable communication over long distances, like in the case of an aeronautic applications. The work analyses the suitability of different topologies and the signal integrity criteria for FlexRay communication systems when applied to that domain. The work highlights that reliable communication can only be *conditionally* possible over longer distances with sufficient signal integrity, at rates below 5 Mbps.

[94, 95] evaluate the application of FlexRay networks for a steer-by-wire application. The system description, protocol configuration for the sensors and actuators, and support for redundancy are described in [94]. A comparison of similar implementations using other time-triggered schemes (TTCAN) is reviewed in [95], which clearly shows the improved determinism and reliability achieved using FlexRay.

2.2.3 Other Protocols

2.2.3.1 Time-Triggered Ethernet

Time-Triggered Ethernet is an extension of classical Ethernet [96] with additional services for supporting real-time and safety-critical applications. TT Ethernet can support a range of applications, from data acquisition machines to intensive multimedia and demanding hard real-time and fault-tolerant systems [97, 98, 99]. TTE distinguishes two types of traffic :

1. Standard Ethernet packets (ET messages) – stochastic event triggered systems

2. Time Triggered Ethernet packets (TT messages) – special packets with guaranteed deadlines.

TT messages are periodically transmitted with predefined schedules that ensure conflict free transmission. Standard Ethernet packets (ET messages) can be transmitted over the same physical medium without affecting the deterministic properties of TT messages. This makes TTE an attractive scheme for automotive backbone networks, allowing highly stochastic data (like media) to be interspersed with highly temporal data (like x-by-wire) without affecting system reliability or determinism. Moreover, TTE supports data throughputs of over 100 Mbps, much higher those offered by FlexRay.

TTE was designed as a unified communication architecture that can cater to multiple domains of distributed computing, with mechanisms built in to handle the following requirements [100]:

1. Certification – This stringent process requires that it must be possible to define correct operating procedures under all fault and load conditions. This is a mandatory requirement for extending services to safety-critical applications.

2. Compatibility – The determinism (or real-time capabilities) must be built into the system within the framework of the classical Ethernet standard. This allows interoperability between ET messages and TT messages on the same physical layer.

3. Determinism and Predictability for TT messages – quantifiable delays and jitter, within the limits of real-time systems.

4. Global Time View – fault-tolerant scheme to ensure a global view of time.

5. Isolation of faulty systems – schemes to eliminate fault and error propagation caused by one faulty node.

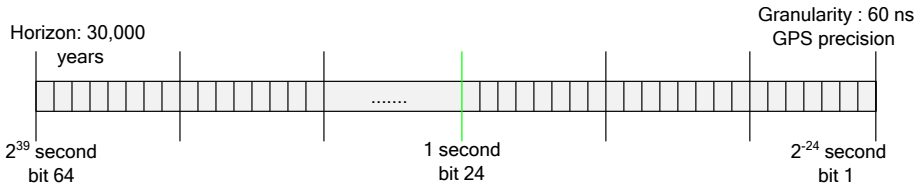


Figure 2.8: 64-bit time format used in TTE.

6. Scalability – no parameter should prevent scalability of the system in terms of bandwidth or number of participating nodes.

To ensure synchronisation among nodes, a common 8 byte time word format is used, with a granularity of 60 nanoseconds and a horizon of 30,000 years, as shown in Figure 2.8. This specification is standardised in the small transducer interface standard, and is closely related to the GPS (Global Positioning System) timing standard, though with a much longer horizon. In the reference implementation in [100], a two byte format is used to configure the periodicity of TT messages, though the full 8 byte format is recommended by the standard.

Two distinct configurations are described in the reference implementation: the first for standard applications using standard Ethernet controllers, TT controllers, and a single switch, and the second for safety-critical application with redundant ports and independent switches [100]. Detailed schemes and arrangements for multi-modular redundancy schemes are discussed in [99]. The key idea of TT Ethernet is pre-emptive scheduling at the switch interface. When a run-time conflict between two messages is observed, the ET message is pre-empted by the switch while TT messages are transmitted with a predefined delay. The pre-empted message is retransmitted once the TT message has been transmitted. The type field for protocol identification is *0x88d7* for TT messages, as defined by the IEEE Ethernet standards authority [101]. The TT message also has an additional header (TT-header) in the payload section of the IEEE Ethernet standard.

The TTE communication controller also handles TT messages differently from the regular ET messages. For each new TT message received (whose period ID is not recorded at the controller), a new buffer space is allocated, which stores the latest version of the message with this period ID. An outgoing message is stored

at the controller until the phase bits of the global time match the phase bits in the period ID. When a match occurs, the controller posts this message on the bus. All messages in the buffer (either received from bus or host) are updated only using write operations.

A subset of nodes act as global time-keepers (up to 5 nodes) for the cluster, initiating start-up and maintaining global time. In addition to the regular TT and ET messages, additional message categories are defined for performing synchronisation, start-up, and fault-tolerant transmission. For non-safety-critical applications, clock synchronisation is performed by the central master algorithm. After power up, the timekeeper node(s) send start-up and synchronisation messages, and enter normal mode, where they transmit periodic resynchronisation messages in addition to ECU data. For a complex setup involving safety-critical nodes, clock synchronisation is achieved through a fault-tolerant distributed clock synchronisation algorithm, with clock deviations and corrections applied using a centralised rate correction scheme [97, 99, 102]. TTE uses a fault-tolerant start-up scheme which has been analysed in great detail in [103]. Following start-up, the timekeeper sends a signed start-up message that includes the start-up information for the guardian. After synchronisation, the guardian checks conformance of the predefined schedule with the transmitted schedule and rejects it if discrepancies are found, disabling switch outputs. [104] describes the architecture requirements of a TTE switch, and compares the performance of commercial off the shelf Ethernet switches for TTE applications. While they note that performance is satisfactory for nodes exchanging only TT messages, in the case of mixed-mode transmission, jitter performance was well below the acceptable thresholds for TTE controllers. In [105], the authors attempt to apply the IEEE 1588 clock synchronisation scheme to standard Ethernet controllers to establish and maintain a global reference time. They also show that the same scheme can be used for TTE controllers that require tight synchronisation.

A comparison between the performance of time-triggered Ethernet and FlexRay is presented in [106]. The work shows that the switched architecture of TTE provides advantages like multicast group communication and destination specific

communication, inherited from the Ethernet standard, while FlexRay provides only broadcast. The comparison also shows that the usable bandwidth (for deterministic communication) of TTE is not affected by end-to-end latency, unlike in the case of FlexRay (when configured using static segments alone). However, the latency of TTE networks, although predictable, is highly dependent on the path from sender to receiver, the switching technology used, and the precision of the clock synchronisation techniques used. Also, at the maximum payload size of FlexRay (254 bytes), the TTE frame is highly under-utilised (only 11%) resulting in larger protocol overhead. Moreover, TTE running at a 10 Mbps traffic rate can only keep up with FlexRay's network utilisation when traffic can be transmitted in parallel (multicast group mode).

Despite the high bandwidth and other potential offered by TTE, it has not found widespread acceptance in the automotive industry. This is primarily because of the expensive physical medium and non-automotive standard interconnect used by Ethernet. While CAN and FlexRay can run efficiently on unshielded twisted pair cables (which are cheap and used heavily in automotive interconnect), the performance of Ethernet over such a physical medium was poor. Broadcom recently announced an Ethernet physical layer (called BroadR-Reach) [107], which demonstrated sustained throughputs of 100 Mbps on unshielded twisted pair cable. It is widely predicted that availability of Broadcom's technology will revive interest in TTE or similar synchronous Ethernet standards (like the upcoming Automotive Ethernet standard) for in-vehicle applications.

2.2.3.2 Media Oriented Systems Transport

Media Oriented Systems Transport (MOST) is currently the de-facto standard for streaming media, infotainment, and multimedia applications in the automotive industry. The technology was designed and developed by the MOST Cooperation for the automotive industry, with the aim of providing a transport mechanism for exchanging audio and video data along with associated control. The advantage of MOST is the simple synchronous nature of the protocol which provides quality

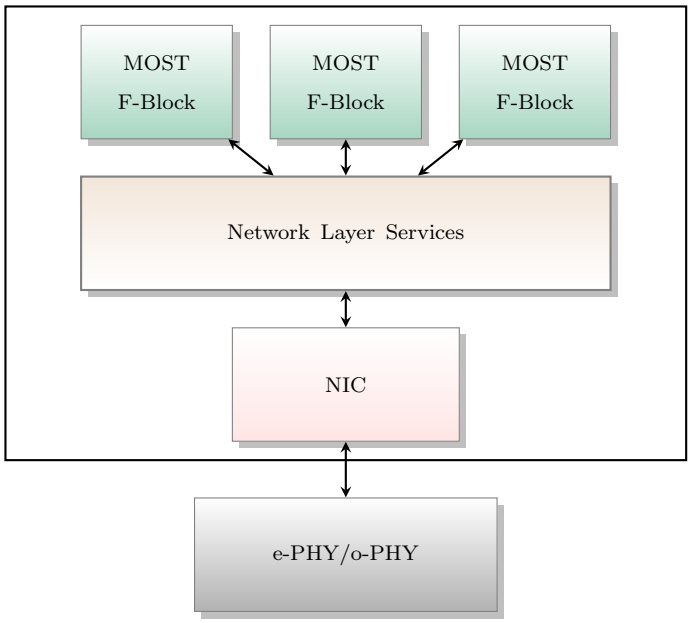


Figure 2.9: MOST Device Model

of service (QoS) guarantees for high bandwidth applications like streaming media. Though originally developed for the automotive industry, the protocol finds applications in many media oriented systems.

A MOST system can contain up to 64 nodes interconnected by electrical (ePhy) or optical (oPhy) physical layers [108]. MOST also supports different speed grades from 25 Mbps to 150 Mbps. Since MOST is a synchronous network, at least one of the MOST network interfaces must be configured as the master node for time synchronisation. All other nodes in the cluster act as timing slaves and derive their slave clocks from bus transactions. The timing master node is responsible for generation and transportation of the system clock, data frames, and blocks.

Any device that can be connected to a MOST network follows the general model described in Figure 2.9. A MOST device may contain multiple functional units, referred to as function blocks (F-Blocks). As an example, a media player incorporating multiple functions like CD playback, radio tuner, and signal processing will have multiple F-Blocks within a single device. In addition to feature specific F-Blocks, a special F-Block called the NetBlock implements the network related functions for the entire device. The MOST Network Interface acts as an intermediate layer between the different F-Blocks and provides an API to simplify

interaction with the hardware.

MOST supports transmission of streaming data and packet data, along with support for dedicated control and synchronisation channels over the same physical layer. The contents of multiple streaming connections and control are interleaved into a basic synchronous data structure for transmission over the network. The Control Channel is used for event-oriented transmissions at low bandwidth and shorter packet lengths, while Data Channels are used for volume data transfer. The data (and control) messages are always addressed to a specific recipient. CRC is used to detect bit errors while an ACK/NACK mechanism with automatic retry (in case of NACK) is supported for fault-tolerance.

A Streaming Data channel is used to transport continuous streams which have high QoS requirements. The Control Channel is used to initiate the streaming data connection between sender and receiver and the connection bandwidths are managed dynamically by the Connection Manager. The bandwidth allocated for the streaming data connections is always available and reserved for the dedicated stream so there are no interruptions, collisions, or delays in the transport of the data stream.

MOST also features a dedicated Packet Data Channel for transmissions requiring high bandwidth in bursts, as in the case of dynamic map loading for navigation. Such transmissions can be asynchronous, and may be addressed to multiple destinations (like internet traffic updates). These are managed on top of the synchronous data stream, completely isolated from the control channel and streaming data to avoid interference. Packet transmissions also feature acknowledge and automatic retry, as in the case of control channels.

MOST uses a ring topology and a different arbitration scheme for channel access depending on the channel type. Packet Data access is handled by a token ring scheme, with nodes having fair access to channel. The Control channel uses a double arbitration scheme to ensure that nodes gain fair access to the channel independent of the communication load of upstream devices and the priority of

the node on the ring, which is determined by position. For streaming data, the bandwidth is allocated dynamically on request by the connection manager.

During system start-up, the node configured as the Network Master initialises the network by performing a system scan. It collects the system configuration by requesting configuration information from all connected nodes (slaves) which is entered into a Central Registry. Once the system states have been identified and verified (marked as OK in the registry), devices can communicate following the arbitration rules. The Connection Manager is a service that may be implemented by any device in the MOST network. All requests for establishing streaming connections are addressed to this device, which dynamically allocates bandwidth. Timing Master functionality is built into the device positioned at the first physical address within the network. MOST also supports a variety of addressing modes from 16-bit direct, indirect, and logical, up to 48-bit MAC addressing. All MOST devices are synchronised to the same sampling clock, either 44.1 kHz or 48 kHz, derived from media sampling rates.

[109] describes the challenges involved in implementing a MOST ring network for an audio application involving three nodes using hardware-software co-design. The work also analyses and verifies the synchronisation and high QoS guarantees claimed by the MOST network specification in a laboratory environment. However, since MOST is used for non-critical communication, it must be isolated from critical communication paths within automotive systems to ensure safe operating environments and to reduce interference. [110] describes an attempt to implement a MOST gateway, that forms a bridge between the high speed media oriented communication and the central bus backbone that runs CAN. They present a scalable architecture with software control which can be used for evolving applications in the human-machine-interface (HMI) domain and body domain.

As discussed, MOST is designed to be used as the communication backbone for media-centric applications. Although error detection and retransmission schemes are built into MOST, it is not intended to be used for critical applications, unlike

FlexRay or TTE, primarily because of the lack of deterministic communication support, non-real-time nature and event-driven architecture.

2.3 Field Programmable Gate Arrays

Field Programmable Gate Arrays started as simple devices for glue logic implementations, primarily used for extending interfaces of a core computing unit (a processor or ASIC) to other supporting devices. Modern FPGAs feature diverse computational resources such as dedicated DSP blocks (including multipliers), block & distributed memory, and embedded processors, along with a wide range of I/O capabilities (high speed serial I/O), making them powerful programmable devices capable of implementing complex systems. The major advantage of FPGAs over other programmable devices like processors is hardware-level programmability; designers can describe custom computational architectures with tailored wordlength datapaths, targeted at a specific application, resulting in significant acceleration and improved computational efficiency.

FPGA resources can be categorised into two classes:

1. *Logic modules*, which are resources that implement the given digital function. These include Look-up Tables (LUTs), Flip-Flops, Multiplier blocks (or DSP blocks), and storage (memories).

2. *Routing resources*, which are used to pass signals between those small modules to build large and complex circuits. These include the interconnect network, the switch boxes, and the input-output blocks.

Look-up tables are the basic building blocks of combinatorial logic. As the name suggests, the LUT stores an output value corresponding to each possible input combination, and output a result for any input value. Modern FPGAs from Xilinx offer 6-input LUTs, while Altera devices offer 8-input fracturable LUTs, supporting a wide range of combinational logic implementations. LUTs are combined

with Flip-Flops for synchronous logic. These resources are hierarchically clustered into Slices, and Configurable Logic Blocks (CLBs). FPGAs also feature built-in hard blocks like Block RAMs and DSP blocks that are built in silicon to offer improved performance, area, and energy consumption for regularly used structures. The flexible routing fabric provides configurable connectivity between the different building blocks of the FPGA. Configurable input/output (I/O) blocks provide an interface to the outside world, with built-in support for a wide range of I/O standards.

A designer describes any digital circuit using hardware description languages like *VHDL* or *Verilog*. Vendor tools *synthesise* this description to generate a circuit netlist described in terms of low level resources interconnected to build the larger circuit. The tools then map the function into device specific resources, assigning these to the different locations and connecting them (placement and routing) to generate a device specific configuration stored in a *bitstream*. During this process, the designer has control over the choice of specific hardware blocks to be used, the interconnect width and other design parameters. Modern tools like Vivado HLS accept user designs specified in high-level languages like C, reducing design effort for complex systems.

Custom hardware offers the opportunity to design architectures tailored towards specific applications. FPGAs implement computation spatially offering a high degree of parallelism that can result in significant acceleration for many algorithms. Deep pipelining and wordlength optimisation allow various design metrics to be traded off against each other. High performance can be offered at lower power consumption and clock frequencies than computations implemented on general purpose processors.

In abstract terms, FPGAs can be considered as being composed of two planes: the configuration memory and the logic plane [111], as shown in Figure 2.10. The configuration memory holds the description of the designer's specification of the system, in terms of the interconnection of the logic blocks in the logic plane, generated by vendor tools. The logic plane contains the building blocks of the

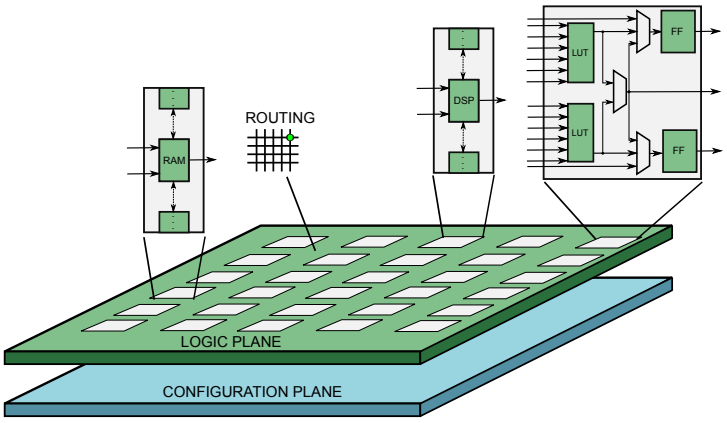


Figure 2.10: An abstract visualisation of FPGA Architecture

FPGAs; the LUTs, flip-flops, block memory (Block RAMs), and DSP blocks, along with the routing resources.

The configuration plane stores the design description in binary, referred to as the bitstream. It contains information such as the logic values stored in LUTs, initial status of flip-flops, initialisation values for memories, standards for I/O blocks, and routing information. For the majority of FPGAs from Xilinx and Altera, the configuration memory is SRAM based and hence volatile. Flash-based non-volatile configuration memory is offered in some devices produced by Actel and others [112]. In order to change the circuitry implemented in an FPGA, the contents of the configuration memory are modified by loading a new bitstream. Bitstream loading can be performed externally using interfaces such as JTAG, or SelectMap [113], or internally using specialised interfaces such as the Internal Configuration Access Port (ICAP) [114]. FPGAs achieve their hardware-level reconfigurability due to this architecture.

2.3.1 Reusing Resources with Dynamic Reconfiguration

Though FPGAs have grown in terms of capacity and features over time, applications and algorithms have become more complex and demanding, pushing the need to reuse limited hardware. FPGAs are field-reconfigurable but complete reconfiguration is a time consuming process and destroys the current state of the system, unless it is stored elsewhere. Dynamic or partial reconfiguration (PR)

allows us to create virtual hardware with indefinite resources and reduce reconfiguration time. Here, a selected region of the FPGA is reconfigured to implement a new circuit while other parts continue to operate, and thus can maintain the state of the system.

Modern FPGAs use a single configuration memory organised in frames [115], a concept first introduced by Xilinx in their Virtex-II and Virtex-II Pro series of FPGAs [116, 117]. A frame marks the smallest unit which can be reconfigured, the size of which is device family dependent. Generally, a frame is one bit wide, with a depth equal to the row height of the device. For earlier devices like the Virtex-II, this was the entire height of the device, while in Virtex 4 (and later) devices a fixed frame size is used with a fixed row height. This organising allows a group of frames to be modified without affecting others, resulting in only portions of the logic being modified, while others continue to work. Frames are organised into tiles that each correspond to a specific resource type. Figure 2.11 shows a simplified representation of a Virtex 6 logic plane, and the relation between frames and logic tiles [118]. This partial change to the configuration memory can be applied externally using the SelectMap or JTAG interfaces. Xilinx introduced an internal configuration port, the ICAP, in their Virtex series, which can be used by logic within the design to access the configuration plane offering "self-reconfiguration".

Presently Xilinx is the only vendor to provide extensive support for dynamic partial reconfiguration, through their software tools PlanAhead [119] and Vivado Design Suite [120]. The process involves manually floor planning to define the different partially reconfigurable regions (PRRs), and assigning different designs for these regions. The tool then generates the complete and partial bitstreams corresponding to different configurations. On the hardware, different modules are activated by loading the corresponding partial bitstream(s) though any of the configuration interfaces. Availability of high speed dynamic (partial) reconfiguration makes FPGAs ideal for implementing real-time adaptive systems and safety-critical systems with built in fault-tolerance.

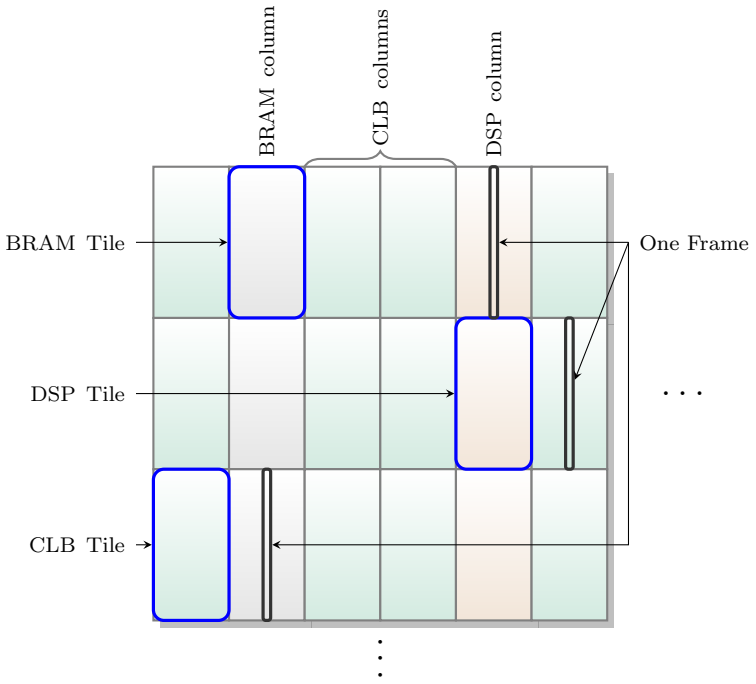


Figure 2.11: A section from Virtex-6 FPGA architecture

2.3.2 FPGAs for In-vehicle Systems

In-vehicle applications can be categorised into two distinct classes: safety-critical functions and non safety-critical functions. Safety-critical functions control and coordinate critical functions like engine timing, controllability (steering, brakes, acceleration), and passive safety systems like airbags. In other words, these functions directly relate to the safety of the vehicle and passengers. Hence, such systems must provide reliable outputs at deterministic time instants. In this context, reliability can be related to time-invariance, meaning the system provides the same output for the same input conditions at any instant in time, while determinism can be related to the response time. This hard real-time nature enforces strict requirements on the determinism and reliability of the safety-critical embedded control units.

In most cases, implementations of safety-critical systems support fail-safe or fault-tolerant operation. A fail-safe system ensures that a critical failure does not cause a catastrophic failure of other systems. A fail-safe node maintains a very basic level of functionality to ensure proper operating conditions for other compute units on

the cluster, without requiring a halt due to fatal errors [121]. Fail-safe operation requires simpler hardware and is hence often chosen in production for non-critical systems.

A fault-tolerant system is more robust and can adapt and recover from faulty situations without severely degrading system performance. Fault-tolerance is achieved by building redundancy for critical operations and intelligence to switch to redundant logic, while a primary unit recovers from erroneous conditions, for example [92]. Fault-tolerance can be achieved at the individual node level or at a global level for the entire function. For safety-critical systems like drive-by-wire, and occupant safety systems, fault-tolerance is often insisted at the cost of additional hardware and complex software.

On the contrary, non-safety-critical systems are applications that augment the user experience in the vehicle. These could be comfort-related functions like seat settings, multimedia, and air conditioning, or assistance functions like cruise control and remote diagnostics.

2.3.2.1 FPGAs as Compute Units (ECUs)

Non-safety-critical ECUs: FPGA based systems for real world in-vehicle applications are mostly limited to non-critical driver assistance and multimedia functions that demand complex processing. Modern driver assistance systems utilise multiple sensors like RADAR, LIDAR and stereo vision to enable simultaneous detection of multiple objects of interest and to keep the driver aware of his surroundings. Complex ECUs like radar signal processing for driver assistance, motion estimation algorithms for pedestrian/vehicle detection and motion characterisation, or multimedia applications that require dynamic compression for streaming audio/video content for in-vehicle entertainment systems involve processing large volumes of data with QoS and deadline requirements. Executing such complex processing in real-time requires efficient compute architectures that can exploit parallelism for acceleration, an area where FPGAs excel.

The literature describes efficient implementations of complex signal processing systems on FPGAs for the automotive domain. [122] presents an implementation of a reactive cruise control system on a single FPGA that interfaces with radar inputs, performs signal processing operations, and produces control outputs. The authors use hardware software co-design and leverage pre-built IP cores for several functions. Meanwhile in [123], the authors design custom hardware and apply FPGA specific optimisations to a CFAR detection algorithm implementation on a Virtex II device, discussing the impact of their optimisation on system performance. An enhanced scheme for next generation driver assistance systems using vision based techniques is described in [124]. The authors present a discussion on the state-of-the-art and development of their proposed system using stereo vision mapped on FPGAs.

The use of partial reconfiguration enables implementation of adaptive applications that time multiplex mutually exclusive functions. An architecture that leverages dynamic partial reconfiguration (PR) for a multi-target tracking scheme for advanced driver assistance is presented in [125]. Here, the model dynamically adapts to driving conditions by using PR to alter the filter characteristics, resulting in deterministic performance in many different driving conditions. In [10], the authors apply dynamic reconfiguration to time-multiplex functionality in a smart node for driver assistance systems. In this case, PR enables the use of a smaller FPGA with reduced power consumption for multiple functions.

In many similar applications, FPGAs have been used to enable more complex applications than would be possible using general purpose processors. FPGA vendors Xilinx and Altera provide a host of IP blocks to support implementation of high performance systems for in-vehicle applications. Both vendors provide platforms for integrating extensive infotainment functionality in a low-power, small-footprint solution, and a host of IP functions ranging from system interfaces to optimised functional blocks.

FPGAs are also a platform of choice for evolving V2X (or C2X) systems. The

compute complexity of the algorithms involved as well as the requirement to incorporate security and privacy means parallelism, isolation, and deterministic performance are important. Multiple researchers have explored the possibilities for reconfigurable hardware in V2X systems [126, 127, 128, 129]. Novel ECU architectures [126] and efficient implementations of applications like collision detection [127] that utilise V2X communication on reconfigurable hardware have been presented. They show that FPGA-based implementation enables better computational performance as well as integration of extended features like security and authentication, that are a key requirements in V2X systems. Researchers have also explored efficient implementation of communication protocol PHY and MAC layers to enable V2X communication [128, 129].

Safety-critical ECUs: FPGA-based safety-critical applications in the literature make use of the capabilities of the architecture like partitioning, determinism, and run-time reconfigurability to provide fail-safe or fault-tolerant features. Partitioning enables segments of designs to operate in complete functional isolation, as required for redundancy schemes. Dynamic reconfigurability allows FPGA-based designs to recover from errors at run-time by either partial or complete reconfiguration.

The literature describes FPGA-based designs that instantiate multiple instances of identical ECUs within the same device to aid redundancy while providing better determinism [121, 130]. In [130], the authors detail a flexible ECU description on reconfigurable hardware. The authors also describe methods to achieve complete compatibility with AUTOSAR standards, while taking advantage of FPGA specific features like partial reconfiguration. Safety-critical schemes that leverage FPGA reconfiguration are also described in the literature. An architecture for implementing fail-safe safety-critical ECU systems on FPGAs, leveraging dynamic reconfiguration (complete reconfiguration), is described in [121]. Their architecture uses FPGA logic as a fail-safe back-up, which is completely reconfigured with a back-up mode when errors are detected.

Triple modular redundancy (TMR) is a commonly used technique to mitigate transient and persistent errors in a design. TMR uses three independent instances of the same function that interact with the same input parameters or signals. The computed outputs of the three modules are majority voted to disregard any outlier, and hence, there is no single point of failure. TMR is widely employed in aerospace applications and space research, where SRAM based FPGAs are prone to radiation induced single event upsets (SEU). TMR on FPGAs and voter insertion schemes are widely researched in industry and academia, but mostly related to space induced effects and SEUs. In [131], the authors describe techniques to analyse the effectiveness of complete TMR for commercial SRAM-based FPGAs. In [132] and other related literature, the effect of SEUs on routing resources, throughput oriented circuits, state machines, and configuration memory are analysed. The authors describe that though complete protection many not be guaranteed, manual floor-planning that isolates the different redundant modules improves the efficiency of function-level TMR. However, TMR at the function-level introduces a considerable overhead in terms of area and power, since each function is triplicated along with associated voters. Partial TMR is an alternative approach which sacrifices some reliability for a lower area penalty [133, 134]. Here TMR is applied to only critical parts of the design and continuous sections that are identified by the automated tool. To improve reliability, they also apply TMR to components (or modules) that when affected by an SEU event cause a design failure that can be corrected only using a system reset. TMR is less suitable for vehicular systems where cost is a more important factor.

An interesting approach is to combine partial TMR with PR to achieve a better area trade-off [135, 136]. Here, PR is used to selectively reconfigure a region with persistent faults, hence requiring fewer critical regions under complete TMR. PR has been applied in conjunction with partial (fine-grained) and function-level TMR implementations. In [137], the authors compare TMR to existing alternatives like quad-mode redundancy, state machine encoding, and temporal redundancy. They show that competing techniques do not outperform TMR with regard to fault-tolerance but consume more area and resources in several cases.

2.3.2.2 FPGAs for Prototyping and Validation

FPGAs have also been employed as a rapid prototyping and simulation platform to verify automotive applications. FPGA based test platforms can be used to monitor and verify the performance of ECU systems on CAN and FlexRay buses. Such platforms offer a wide range of customisable configurations that can be employed to test the systems in normal and edge-case scenarios.

In [78], the authors demonstrate a FlexRay protocol analyser capable of extracting FlexRay bus parameters from network transactions. They aim to enable rapid system prototyping using an FPGA based test system for analysing and testing FlexRay based systems and automating the configuration generation using the protocol analyser. In [86], the authors describe a scheme to automate verification of complex FlexRay clusters for network inconsistencies. They present an FPGA based active star node that is used to detect the presence of inconsistent conditions (referred to as Slightly-Off-Specification failure modes) in actual networks. Specialised hardware was used to create test conditions like delayed transactions (for slot-boundary violations) and to corrupt frames on the network. Similarly, an FPGA based star configuration for monitoring clusters is described in [87], where the star node is connected to a host PC running debugging tools. The software enables a few additional schemes like replaying bus data (in-order or out-of-order) and can trigger startup or re-integration of cluster nodes.

Testbeds and prototype implementations on FPGAs are also discussed in the literature for functional validation of ECUs. In [138], the authors detail the use of an FPGA-based hardware-in-the-loop validation framework for evaluating fault-detection schemes in electronically controlled suspension systems. In [139], the authors use FPGAs to prototype a traffic sign recognition system and use the platform to evaluate their approach and algorithms. An FPGA-based prototype is used in [140] for the validation of fuzzy-network and neural network based approaches (soft-computing) for implementing automotive control applications like lane following and parallel parking. FPGAs have also been used for prototyping

alternate network protocols like *SpaceWire* for active-safety applications in automotive systems [141]. Similar uses of FPGAs for prototyping and functional validation are discussed in [142, 143, 144], among many others.

FPGAs have also been explored for upcoming vehicle-to-vehicle (V2V) and vehicle-to-X (V2X) communication systems. In [145], the authors make use of a V2X multi-antenna test bed using FPGAs for evaluating the performance of the IEEE 802.11p standard for vehicle to infrastructure communication. The test-bed enables a compact implementation for real-world measurement with the bit-accurate, real-time analysis and the modelling of complex conditions like receiver diversity. In [146], the authors present a prototype implementation of their car-to-X (C2X) communication infrastructure on an FPGA platform and evaluate its performance using generated traffic. FPGAs also enable emulation of complex real-world conditions around the vehicle, which can be used to evaluate the network protocols proposed for C2X applications. One such application is described in [147], where the authors evaluate the reliability of the 802.11p PHY layer by modelling real-time channel behaviour using FPGAs.

2.3.2.3 FPGAs in In-vehicle Networks

Implementations of in-vehicle network protocols like CAN on FPGAs are available commercially and in the research community. FPGA vendors Xilinx and Altera provide optimised implementations for their respective platforms [39, 40]. Extensions to CAN like CAN+ have also been implemented on FPGAs [41]. Similarly, FlexRay implementations on FPGA have also been discussed in the literature, though these were designed from a high level model without making optimal use of the FPGA architecture [66, 67]. FlexRay implementations are also available commercially, like the *eRay* IP core from Bosch [62] and the FRC2100 from Freescale semiconductors [63]. However, these are platform agnostic, and aimed for generic implementation, suffering from sub-optimal performance on an FPGA.

Network controllers that have enhanced capabilities have also been implemented on reconfigurable hardware. A fault-tolerant communication controller scheme for

safety-critical ECUs is described in [93]. Here the authors make use of partial reconfiguration to reconfigure the faulty communication controller of the safety-critical ECU. The authors also describe methods to integrate their approach into the AUTOSAR run time environment for complete compliance with automotive standards. In [70], the authors present an energy-efficient FlexRay controller implemented on an FPGA, closely integrating it with a monitoring extension that improves its energy efficiency. Similarly, in [148], the authors present a FlexRay controller implemented on an FPGA with integrated extensions for aiding fault-detection during a verification run.

FPGAs have also been employed for gateway ECUs in in-vehicle networks. In [149], the authors define a modular approach for an automotive gateway architecture implemented on an FPGA, while in [150] they describe a mechanism to achieve accelerated routing of messages between ports using architecture-level enhancements. A scalable gateway architecture based on a programmable system-on-chip architecture in described in [151]. It closely couples the integrated Power PC hard block on a Virtex-4 FPGA with interface logic on the fabric and an external daughter card (featuring a Spartan-3E device). However, the switching is performed in software running on the Power PC, which creates additional overheads as the number of ports is scaled.

Despite the numerous advantages discussed, FPGAs have not found widespread adoption in automotive computation or networking systems, for numerous reasons. Cost can be a factor, though modern FPGAs are now much cheaper than in previous decades, with their cost comparable to mid to high-end automotive grade processors. For example, the shelf cost (per unit) of an automotive grade Xilinx Spartan-6 device (LX45) is comparable to that of a mid-range processor used in the power-train domain (Freescale SPC5764, both around S$ 130), while a newer generation Xilinx Artix-7 device (automotive grade) is 1.75× more expensive, but offers 2.3× more resources, better energy efficiency and native support for partial reconfiguration. Note that the costs at volume are subject to other factors that are beyond the scope of this discussion. Certification is another concern as

many established standards are defined specifically for processors that run software. However, the pressures of next-generation applications mean that FPGAs are under consideration once more.

2.4 System-level Challenges : Security and Functional Validation

Automotive networks were originally closed, with only the mandatory on-board diagnostics (OBD) port providing an interface to the outside world. Since these OBD ports provide direct and/or bridged access to both critical and non-critical networks, they represent an ideal point for a hacker to gain access via an (infected) OBD dongle. Malicious software or hardware provides another pathway, mostly introduced through non-approved after-market upgrades. These can be used to launch internal attacks (observations or manipulations) on messages or other ECUs since the network provides implicit full bus access to all components. As wireless communication becomes increasingly common in modern vehicles, whether in wireless sensor systems, internet-enabled services, or (future) vehicle-to-vehicle (V2V) communication, new pathways become available to potential attackers without even requiring physical access. As an example, the use of short-range wireless communication integrated into tyre pressure sensors provides access to pressure values which would otherwise be impossible to monitor/measure during runtime, but also offers a wireless pathway for a hacker [152].

Many high-end cars now feature remote diagnostics systems which allow the manufacturer to monitor and alert users about system issues and performance. This is achieved by monitoring the internal networks remotely, using secure links, as in the case with GM's proprietary diagnostic scheme called *OnStar*. However, this scheme can be easily misused to collect user information and travel patterns, breaching user privacy. Moreover, if exploited, these interfaces could proved a way to transmit spurious and corrupt data onto internal networks, causing ECU

systems to fail. This would be catastrophic for vehicles that depend on x-by-wire systems.

In [152], the authors describe experiments performed to estimate the effort required to hack into in-vehicle networks and control ECUs. They exploit the wireless interface used by the tire pressure monitoring system (TPMS). In their experiments, the IDs of TPMS transmitter(s) are captured by eavesdropping on the broadcast transmission. These IDs are then reused to transmit spurious data using a higher power transmitter. Since wireless receivers are sensitive to signal power, the actual data is discarded, while the tampered data is accepted by the central ECU. By flooding the system with corrupt frames, they showed that the TPMS ECU could be halted with very little effort. Similar efforts also show more serious exploits with hackers being able to control, corrupt and subsequently stall engine and chassis ECUs. Such security vulnerabilities in a modern vehicle and the analysis of the protocols from a security perspective are discussed in [153, 154, 155, 156], among others. The literature also describes mechanisms like trusted communication groups [156], access control lists, and information tracking schemes [157], but the scope of applying such systems is limited to certain domains and specific applications.

V2V allows vehicles to communicate relevant data to provide advanced driver assistance, collision avoidance, and remote assistance. Communication is achieved over multi-hop self-organising wireless networks, comprising the mobile units (vehicles) and fixed infrastructure [158]. Each vehicle broadcasts relevant information about itself, which can be received by other vehicles (and infrastructure) in the vicinity. Beyond the ad-hoc network nature and the requirements of authentication and security, the scheme adds further challenges like ensuring timeliness of data, since delayed, fraudulent or tampered data can result in misguiding commands to the user [159, 160]. Moreover, the lack of suitable security features at gateways can be exploited, resulting in loss of controllability and damage to property and life. This requires novel electrics/electronics (E/E) architectures and advanced techniques to process and forward data at gateways that can detect intrusions [161].

Researchers have proposed the use of FPGAs for V2X communication systems, aiming to exploit their computational capabilities and run-time reconfigurability. The EVITA project describes the use of hardware security modules (HSMs) that aim to provide security for vehicular on-board compute and communication systems. The project describes multiple flavours of HSMs, ideally implemented on FPGAs, to meet performance requirements for different contexts [162]. For example, the EVITA HSM *Full Version* meets the performance and security requirements for V2X communications, while the *Medium Version* is aimed at meeting requirements for in-vehicle networks, implemented on powerful ECUs such as gateways. The PRESERVE project also utilises FPGA implementations to accelerate cryptographic primitives in their hardware security modules dedicated to the Vehicular Communication Security Subsystem (VSS). Other projects in this area include SeVeCom and PRECIOSA that also demand the use of reconfigurable hardware to meet performance requirements. An architecture for implementing an intelligent cryptographic engine without significant loss in performance for the IEEE 1609.2 (WAVE) standard is described in great detail in [163]. The authors highlight the computational capabilities of FPGAs and explore custom designs with high performance, even for compute intensive algorithms like AES encryption, SHA256 hashing, and elliptic curve cryptography for verifying digital signatures (ECDSA). However, in all cases, the use of HSMs for in-vehicle network security does involve some level of application awareness and execution/communication latency.

Verification of these HSM extensions and ensuring standards compliance as well as application deadlines for the proposed security enhancements presents a larger challenge. Presently, verification of such complex systems is done (and proposed) using model based simulations or with prototype implementations. However, with safety-critical systems like x-by-wire, security extensions for in-vehicle and V2X systems must be certified before use in commercial vehicles since a failure after the integration stage can be very expensive. Novel techniques and associated hardware and software for validating and certifying these systems is needed. These platforms

should be capable of mimicking real in-vehicle environments, with real latencies and delays, used to benchmark and verify such systems for compliance.

2.5 Summary

In-vehicle networks have significantly evolved from simple network schemes to sophisticated time-triggered networks and other high bandwidth interconnects. Complex intensive applications have also made their way into vehicular systems, pushing the computational requirements of embedded computing units. As more and more functions are introduced into newer vehicles, E/E architectures and networks have to be enhanced to support them, while satisfying the reliability and deterministic constraints imposed by the safety-critical applications. This is further complicated by emerging requirements like security of embedded systems and networks, putting additional strain on the ECUs and the network interfaces.

We believe that modern FPGAs are capable of handling such complex tasks, through the design of custom hardware that can extend standard functionality. However, designing efficient hardware on FPGAs and integrating advanced capabilities like dynamic reconfiguration requires high levels of expertise. Further, custom designs should also ensure that application developers have a reliable and standardised interface to the hardware, which ensures compatibility and simplifies integration and functional validation. We believe that this can be tackled through intelligent design of E/E architectures in a manner that offers tight integration between the network interface and the computational entity. Extended capabilities like dynamic reconfigurability and security can be presented as features of the platform, accessible through simple function calls. This transparent integration offers minimal overhead and low latency, allowing designers to design context-aware computing systems, explore deterministic consolidation and novel fault-tolerant schemes. Such abstraction will help in widespread adoption of reconfigurable hardware for automotive applications and research.

3

Extensible Network Interfaces

3.1 Introduction

We have seen how electronically controlled/assisted safety-critical functions are increasingly gaining widespread acceptance in modern vehicles. Many such functions demand higher communication bandwidth and quality of service (reliability) which exceeds the capabilities of the event-triggered Controller Area Network (CAN) protocol, that has been pervasive in automotive systems until now. Furthermore, for next generation electric vehicles with high levels of automation, the required levels of determinism cannot be facilitated with event-triggered networks, leading to emerging widespread adoption of time-triggered communication schemes and protocols like FlexRay and, more recently, time-triggered Automotive Ethernet. FlexRay has been widely accepted as a de-facto communication standard

for safety-critical functions like drive-by-wire, cruise control, and adaptive braking systems, while also facilitating communication for non-critical ECUs.

The communication standards used in automotive systems, including FlexRay and CAN, aim at providing a deterministic and reliable mechanism for exchanging messages between ECUs. Any further enhancements, like a message source identification or timestamping are to be defined by the designer and must be handled explicitly in the application layer. Neither the protocol nor its implementations provide any hardware-level support for implementing such extensions, which the software can rely on using function calls, analogous to dedicated hardware accelerators to which computations can be offloaded by software using standard function calls. Moreover, handling functions like timestamping is non-trivial; while the underlying time-triggered networks have built-in mechanisms to ensure synchronised operation, the software tasks themselves are often not synchronised across different ECUs. This is because many functions are activated only by specific events which mandate time-bound response, and such tasks cannot wait for synchronisation to occur before producing the required response. These conflicting requirements make straightforward implementation and management of such functionality non-trivial at the software level.

Another aspect of managing timestamping in software is its inaccuracy. A task in the ECU generates control data to be communicated to another ECU which is passed to the protocol implementation for transmission over the shared bus. However, since transmission occurs only at predefined points in time (assigned slots in case of time-triggered schemes, non-deterministic priority in the case of event-triggered schemes), the time at which the message is transmitted on the bus may be different from the time it was generated. For a message timestamped in software with the time it was generated, this delay in transmission results in inaccuracy at the receiving ECU. The inaccuracy is further compounded when the delay in the data fetch mechanism from the network interface at the target ECU is also considered, as this is also typically an unsynchronised operation. In addition, there is the overhead in adding/processing such extra information,

which is undesirable in hard-real-time safety-critical systems that run on standard micro-controller platforms.

Ideally, such network-level functions should be implemented close to the network layer with support from the underlying mechanisms of the protocol. However, as mentioned, none of the automotive standards incorporate such features as standard or provide mechanisms which can be directly leveraged in the upper layers. Having such data-layer extensions provide unique mechanisms to reliably implement features like stale data rejection, replay attack prevention, among other enhancements.

To explore such possibilities of enhancing automotive ECUs and to demonstrate such extensions in a certifiable environment, a prototype implementation of an extensible network interface is necessary, ideally in reconfigurable logic. While implementations of protocols like FlexRay are available as IP and discrete ASICs, they are not open to the research community. An extensible FlexRay controller designed to take advantage of the capabilities of FPGAs opens up exciting possibilities – a closely coupled interface to the ECU and extensions to predefined communication framework can enable advanced and intelligent ECUs with built-in mechanisms to ensure deterministic fail-safe operation. Furthermore, the capabilities of the hardware platform can be explored to merge the controller with multiple applications on the same device, while ensuring sufficient isolation between them, and partial reconfiguration can be exploited to further improve the energy efficiency of non-critical non-concurrent systems. Such ideas cannot be explored using off-the-shelf controllers or platform agnostic designs – a flexible modular implementation of the network interface is essential, ideally on FPGAs.

To enable our further research on enhancing future automotive architectures and networks, we have designed and developed an Extensible FlexRay communication controller (CC) for FPGA-based ECUs. However, beyond a standard prototype implementation, the controller features architecture-level optimisations which enable it to be more efficient than current implementations. Our extensible CC also features configurable extensions like programmable width message time-stamping,

network-level data filtering, data-layer header-insertion, and processing features that augment the capabilities of the CC beyond those defined by the FlexRay standard. Furthermore, the operation of these extensions is managed within the controller, abstracting these details away from the ECU processor.

The optimisations at architecture-level along with the host of extensions makes an energy efficient smart ECU possible, while abstracting such details from the applications allows standard functions to be directly ported to such enhanced architectures for improved efficiency. The extensions also enable advanced features for functional ECUs and gateways, without affecting protocol-defined determinism or reliability, making them ideal for compute intensive and safety-critical applications in next generation vehicular systems. Also, the proposed extensions to the data layer are not tied to the FlexRay standard, and similar extensions can be applied to emerging time-triggered standards such as Automotive Ethernet.

The work presented in this chapter has also been discussed in:

1. S. Shreejith and S. A. Fahmy, *Extensible FlexRay Communication Controller for FPGA-Based Automotive Systems*, IEEE Transactions on Vehicular Technology, Vol. 64, No. 2, pp. 453–465, Feb. 2015 [18].

2. S. Shreejith and S. A. Fahmy, *Enhancing Communication On Automotive Networks Using Data Layer Extensions*, Demo Paper in Proceedings of the International Conference on Field Programmable Technology (FPT), pp. 470–473, Dec. 2013 [16].

3.2 Related Work

The migration to time-triggered networking standards in vehicular systems aims at providing required levels of determinism and bandwidth for safety-critical and compute intensive features. Though event-triggered networks like CAN are widely prevalent in existing vehicles, they fail to provide such determinism especially when operating at near full capacity. Moreover, CAN networks cannot support

increasing bandwidth requirements, and the time-triggered extensions of CAN (TT-CAN) that enforces a slot-based structure over standard CAN to enhance determinism have not gained widespread adoption. Hardware extensions to CAN network controllers were also proposed by some researchers to overcome these limitations [164].

FlexRay has emerged as the choice of network for safety-critical systems in the automotive domain. And during work on this thesis, time-triggered Automotive Ethernet has emerged as another possible candidate for backbone infrastructure to address the bandwidth limitations in FlexRay, though standard communication protocols are still under development. We now look at some of the implementation approaches and network-level enhancements to the FlexRay protocol, that are discussed in literature.

Implementations of the FlexRay controller that can be compiled to a wide range of platforms are available from Bosch and Freescale [62, 63]. These are largely platform independent, suitable for implementation on ASICs or FPGAs. However, they are not optimal for implementation on reconfigurable hardware, since they do not fully utilise the heterogeneous resources in the underlying fabric. For instance, the E-Ray IP core from Bosch, which is a dual channel controller, does not directly instantiate FPGA primitives like DSP Blocks or Block RAMs but uses general purpose logic to implement these functions. Moreover, such implementations do not enable research into extensibility of the communication infrastructure as they are available only as encrypted netlists, preventing the opportunity to optimise them or explore possibilities to enhance them.

Other implementations in the research literature include [65] that discusses implementation of a FlexRay communication controller. It discusses in detail one of the sub modules, the protocol operations control module, that coordinates the actions of the core protocol segments. However, no details are presented about any of these core protocol segments or their implementation aspects and architecture-level optimisations. Also the implementation aims to purely mirror the existing specification, with no new features. [66, 67] also describe implementation of the

FlexRay Communication Controller using the specification and description language (SDL) as the platform and later translation to hardware using Verilog. Their work approaches the protocol from a high level of abstraction and hence does not discuss hardware design details or architectural optimisations.

Beyond these generic approaches, other researchers have aimed at improving/optimising certain aspects of the FlexRay controller. The work in [70] discusses an approach to improve the energy efficiency of a FlexRay controller by allowing it to be controlled by an intelligent communication controller (ICC). The ICC, which takes over bus operations from the ECU when the latter goes to sleep, prevents the ECU from being woken by erroneous transmissions allowing the node to achieve higher power efficiency. The concept validation on FPGA and the proposed architecture are also discussed in the paper. However, they use a proprietary implementation of the FlexRay communication controller that is not available to the research community. Similarly, the work in [148] describes implementation of a FlexRay controller on an FPGA with add-on features to aid functional verification. These features are primarily to enable a verification framework and do not point in the direction of optimisations or enhancements for improving the node/network functionality beyond standard implementations.

3.3 Contributions

We present a resource optimised implementation of a dual-channel FlexRay communication controller, that features a configurable set of data-layer extensions. The data-layer extensions are implemented as parallel extensions to the main controller datapath, thus ensuring compliance with FlexRay protocol requirements. The design is extensively optimised by effectively utilising the hardware primitives, which results in an area-efficient implementation that leaves aside a considerable portion of FPGA logic for implementing functional components of the ECU. Though this approach results in limited portability between platforms, it provides superior utilisation and power efficiency during operation, especially in

case of FPGA-based ECUs. Also, portability between platforms is becoming a minor problem as FPGA vendors have standardised hardware primitives across all their device families in a given generation. Furthermore, we show that our architecture targeted for a Xilinx device can be easily ported to an Altera platform, with similar efficiency in area and power.

This work enables a number of investigations in the space of FlexRay on reconfigurable hardware and more generally for enhanced ECUs on FPGAs. The extensions, when closely coupled with functional logic on the same hardware, offer enhanced communication between ECUs without consuming excessive bandwidth or power. The case studies show that such enhancements can be employed in multiple automotive applications from individual ECUs to gateways, to improve the overall efficiency of the system.

3.4 Architecture Design

A node on the FlexRay network consists of a Communication Controller (CC), an application running on a host ECU, and multiple bus drivers to independently support 2 communication channels. The Host ECU is the computational implementation of an algorithm like adaptive cruise control or engine management, and it may communicate with other ECUs or sensor nodes over the network. The Communication Controller ensures conformance with the FlexRay specification when transmitting or receiving data on the communication channel. The Bus Driver handles the bit stream at the physical level and provides the physical level interface to the communication channel. The Host ECU monitors the status of the Communication Controller and Bus Driver independently and configures them appropriately during startup or runtime.

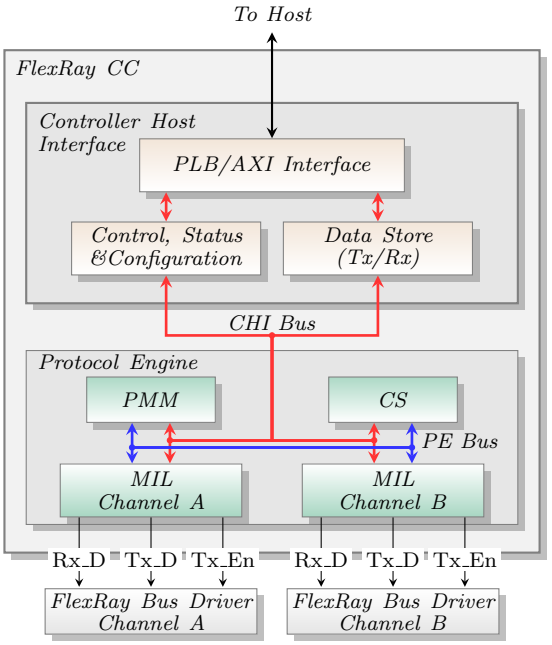


Figure 3.1: Architecture of custom FlexRay communication controller.

3.4.1 Communication Controller

The FlexRay CC switches between different operating states, based on network conditions and/or host commands, ensuring conditions defined by the FlexRay protocol are met at all times. The CC architecture, as shown in Figure 3.1, comprises the Protocol Engine (PE) which implements the protocol behaviour and the Controller Host Interface (CHI) which interfaces to the host ECU.

The CHI module communicates with the host and handles commands and configuration parameters for the FlexRay node. These parameters are defined for the particular cluster the node is operating on, and are initialised during the node's configuration phase. The CHI feeds the current state and operational status to the host for corrective action if necessary. There are transmit and receive buffers and status registers for the datapath, to isolate control and data flow. The CHI may also incorporate clock domain crossing circuitry to enable the different interfaces to work in distinct clock domains.

The Clock Synchronisation (CS) and Medium Interface Layers (MIL) submodules of the Protocol Engine implement specific functions of the protocol, which are controlled and coordinated by the Protocol Management Module (PMM). These

sub-modules support multiple modes of operation and can alter their current operating mode in response to changes in any of the parameters, error conditions, or host commands. The PMM ensures mode changes are done in a way that complies with the FlexRay specifications. The Medium Interface Layer handles the transmission and reception of data over the shared bus. It encodes and decodes data, controls medium access and processes decoded data to ensure adherence to protocol specifications. The CS module generates the local node-clock, synchronised to the global view of time. It measures deviation in the node clock on a per-cycle basis so that it stays synchronous with other nodes in the cluster.

Timing in a FlexRay node is defined in *macroticks* and *microticks*. Microticks measure the granularity of the node's local internal time and are derived from the internal clock of a node. A macrotick is composed of an integer number of microticks. The duration of each local macrotick should be equal within all nodes in the cluster. The FlexRay protocol uses a distributed clock correction mechanism, whereby each node individually adjusts its view of time by observing the timing information transmitted by other nodes. The adjustment value is computed using a fault-tolerant midpoint algorithm. A combination of rate (frequency) and offset (phase) correction mechanisms are used to synchronise the global time view of different nodes. These corrections must be applied in the same way at all nodes and must follow the following conditions:

1. Rate correction is continuously applied over the entire cycle
2. Offset correction is applied only during NIT in an odd cycle and must finish before the start of the next communication cycle.
3. Rate correction is computed once per double cycle, following the static segment in an odd cycle. The calculation is based on values measured in an even-odd double cycle.
4. Calculation of offset correction takes place every cycle, but is applied only at the end of an odd cycle.

Rate correction indicates the number of microticks that need to be added to the configured number of microticks per cycle and may be negative, indicating that

the cycles should be shorter. Offset correction indicates the number of microticks that need to be added to the offset segment of the network idle time and may also be negative, indicating that the duration has to be shortened.

The FlexRay bus supports two independent channels for data transmission and reception. The transmission rate can be set at 2.5 Mbps, 5 Mbps or 10 Mbps. The protocol also defines multiple bus access mechanisms, in the form of *static slots* for synchronous time-triggered communication and *dynamic slots* for burst mode event-triggered (priority-based) data transfer. Special symbols can be transmitted within the *symbol window*, like wake-up during operation (WUDOP) and collision avoidance symbols (CAS). During the *network interval time*, all nodes synchronise their clock view with the global clock view so that they stay synchronous. Each transmitted bit is represented using 8 bit-times to ensure protection from interference. At the receiving end, these are sampled and majority voted to generate a voted bit. Transmission and reception must be confined to slot-boundaries and transmission (or reception) across slot-boundaries is marked as a violation. The node should transmit only on slots that are assigned to it (either in the static or dynamic segments). Each node is assigned a *keyslot*, which it uses to transmit startup or synchronisation frames (along-with data).

3.4.2 Implementation and Optimisations of Custom CC

The state of the PMM module, at any instant, reflects the current operating mode of the CC. The PMM module triggers synchronised changes in the sub-modules CC and MIL, and describes the different operating modes of the node, as depicted in Figure 3.2. These mode changes can be triggered by host commands or by internal and/or network conditions encountered by the node. Table 3.1 describes the different commands issued by the host and how the operation of the CC is modified in response to specific commands. As can be seen, certain commands demand an immediate response from the controller, while others are to be applied at specific points within the communication cycle. This distinction makes the

Table 3.1: Commands from Host that affect CC operating modes.

Host Command	Affected States	Final State	Processed at
ALL SLOTS	Active, Passive	no state change	End of cycle
ALLOW COLDSTART	All states except Def. Config, Config and Stop	no state change	Immediate
CONFIG	Def. Config, Init Wait	Config	Immediate
CONFIG COMPLETE	Config	Init Wait	Immediate
DEFAULT CONFIG	Stop	Def. Config	Immediate
FREEZE	All States	Stop	Immediate
HALT	Active, Passive	Stop	End of cycle
READY	All states except Def. Config, Config, Init Wait and Stop	Init Wait	Immediate
RUN	Init Wait	Start-Up	Immediate
WAKEUP	Init Wait	Wake-Up	Immediate

control flow more complex than the case of a straight-forward finite state machine (FSM).

The FlexRay protocol allows a cluster and its associated nodes to switch to sleep mode to conserve power. When any node needs to start communication on the network, a wake-up sequence is triggered by the host by putting the CC into Wake-up state. In the Wake-up state, the node tries to awaken a sleeping network by transmitting a wake-up-pattern (WUP) on one channel. Sleeping nodes decode this pattern and trigger a node wakeup. Nodes which have dual channel capability then trigger a wake-up on the other channel to complete a cluster-wide wakeup. The node cannot, however, verify the wakeup trigger at all connected nodes, since WUP has no mechanism to communicate the ID of the nodes that have responded. The nodes then follow the startup procedure to initialise communication on the cluster. The startup operation also caters for re-integration of a node onto an active network. To do so, the node must start its local clock so that it is synchronised with the network time.

Within the Start-up state, the clock synchronisation startup (CSS) logic in the

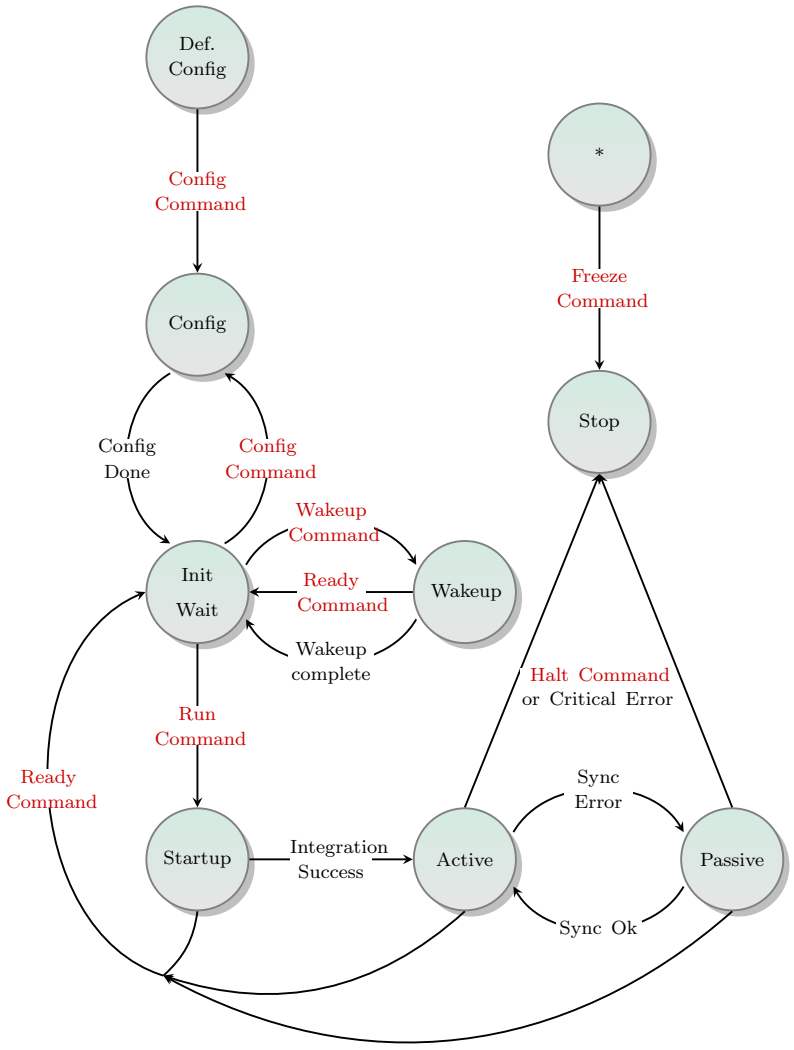


Figure 3.2: FlexRay CC Modes of Operation.

Clock Sync module is initialised, which extracts timing information from a pair of synchronisation frames received from the bus and starts the macrotick generator (MTG) in alignment with them. Over the next few cycles, it monitors the deviation of its clock from the actual arrival time of sync frames on the bus, and if these are within limits, the process is signalled as successful. If at any point, the observed deviation is beyond the configured range, the integration attempt is aborted and the node restarts the process. Once it integrates, the node moves to *Active* mode, with a clock that is synchronised with the network. Once the node successfully joins the network, the PMM normally follows a cyclic behaviour switching between active and passive states, in response to network-node conditions, causing synchronised changes in all modules.

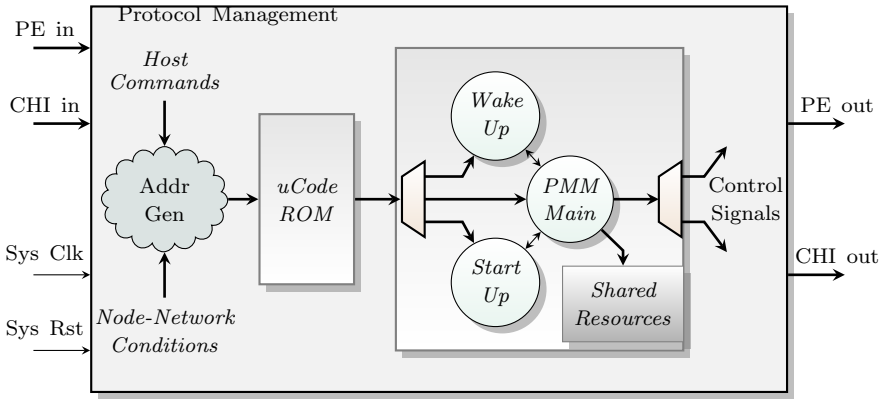


Figure 3.3: Protocol Management Module (PMM) architecture.

In our design, the PMM module also encapsulates Wake-Up and Start-Up. Combining the operations of WUP and SUP with the operations at each state of PMM results in a hierarchical structure, as in Figure 3.3, with the combined state encodings stored in the microcoded ROM. Combining the two functions into the same module also allow us to share resources between the two operations, which are not required concurrently, using simplified control flow. Since the CS and MIL modules are also controlled by WUP and SUP for the associated wake-up and start-up operations, integrating them with the PMM results in centralised control for all operating conditions, simplifying interfaces to the submodules. The response to different conditions or stimuli is now reduced to the process of generating appropriate addresses for the ROM, similar to the program counter implementation on a standard processor. The ROM is efficiently implemented using distributed memory (LUTs) because of its small size.

Figure 3.4 shows a simplified architecture of the CS module in our design. The CS module generates the clock, computes the deviations of the generated clock from the distributed timing information and applies the corrections in the specified manner. The CS module is comprised of 2 concurrent operations (or sub-modules): firstly, the MTG process which controls the cycle counter and the macrotick counters and applies the rate/frequency and offset/phase correction values; and secondly, the Clock Synchronisation Process (CSP) that performs the initialisation at cycle start, the measurement and storage of deviation values during the cycle

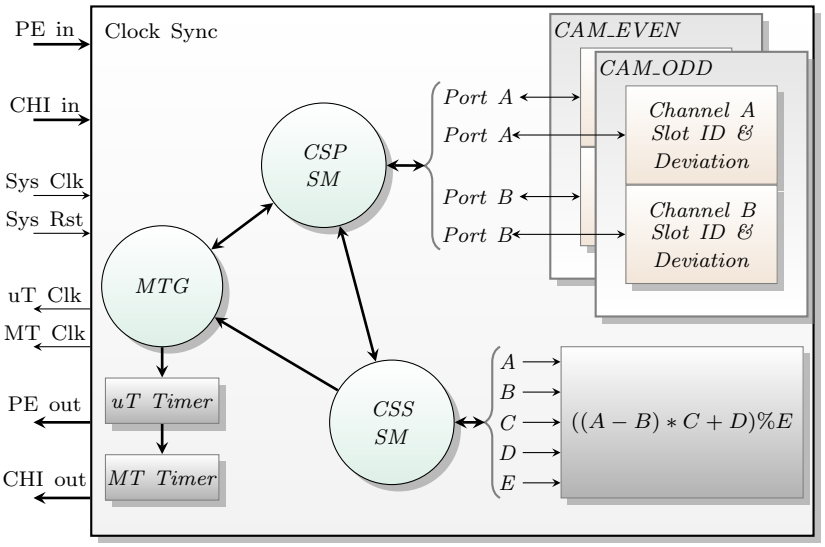


Figure 3.4: Clock Sync (CS) module architecture.

and computes the offset and rate correction values. In addition, the CSS module is responsible for starting a synchronous clock when the CC tries to integrate into either an active network or initiate communication on an idle network. The CSP state machine controls and co-ordinates the operations of the CS module by interacting with the CSS and MTG sub-modules.

During Start-Up, the CSS process monitors the arrival time of the even synchronisation frames and generates the global reference time by computing the initial Macrotick value as

$$\begin{aligned} Macrotick = (&pMacroInitialOffset + gdStaticSlot \\ &\times (ID - 1)) \bmod gMacroPerCycle \end{aligned} \tag{3.1}$$

where *pMacroInitialOffset*, *gMacroPerCycle* and *gdStaticSlot* are FlexRay parameters. The expression is implemented using cascaded DSP48A1 slices, whose inputs are multiplexed between channels A and B to handle startup requests from either channel. If a subsequent odd frame arrives within the predefined window, the integration attempt is flagged as successful by the CSS module and the CSP commands the MTG state machine to start the Macrotick clock (MTClk) using the computed Macrotick value for this channel. The MTG then generates the

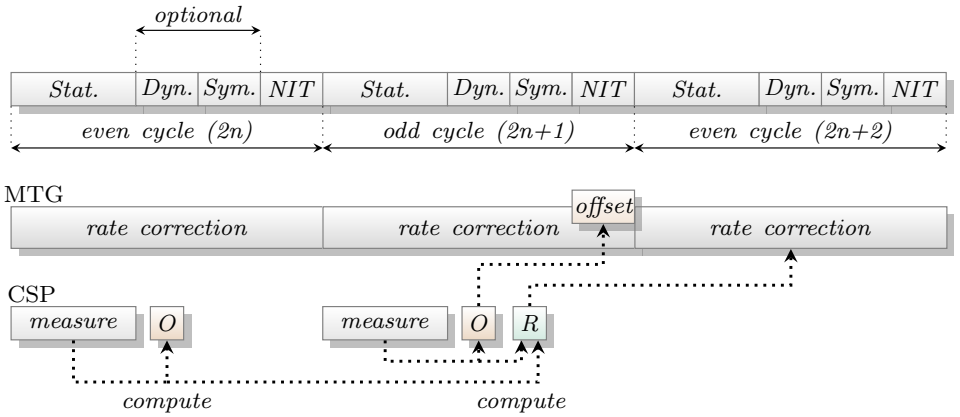


Figure 3.5: Rate and offset computation by MTG and CSP.

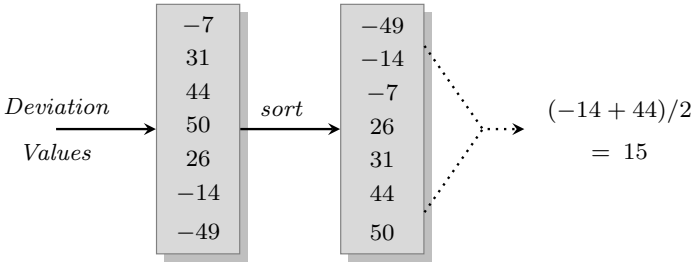


Figure 3.6: Fault tolerant midpoint illustration for seven deviation values.

Macrotick clock from the Microtick clock (uTClk) using the configured parameter values.

Figure 3.5 shows the clock deviation computation for each cycle, once the CC successfully integrates onto the network. The measuring cycle refers to the duration of the static segment, where sync-nodes transmit synchronisation frames which are used to compute rate and offset corrections. During each measurement cycle, the node measures the deviation of time of arrival registered at the node from the calculated time of arrival of the synchronisation frame, which is stored in memory. At the end of measurement phase, the node computes the offset and rate correction factors from the stored values using a fault-tolerant midpoint algorithm. The operation is depicted in Figure 3.6, for a cycle that recorded seven deviation values. The real challenge here is that a network may be configured without dynamic and symbol window segments. Hence the offset and rate computations have to be completed consuming a minimum number of cycles to ensure that correction values are available to be applied at the network interval time segment.

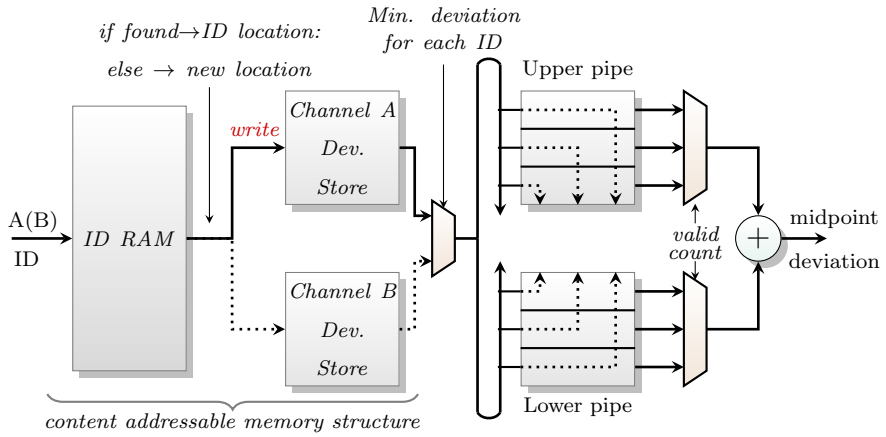


Figure 3.7: CAM organisation and fault tolerant mid-point computation for offset correction.

Figure 3.7 shows our solution to the mid-point computation mechanism, expanded from the slotID and deviation store in Figure 3.4, for an even cycle. The fault-tolerant midpoint algorithm computes the rate and offset corrections that are to be applied to the MacroTick clock. During normal operation, the CSP module handles the computation and storage of individual deviation values and the computation of mid-point correction values. As a frame is received, its ID is used to address the slotID RAM, the output of which is used as the address for the deviation store, mimicking a content-addressable memory. The deviation from the expected arrival time of sync frames to their actual arrival time is stored in the deviation store. The upper and lower pipes perform dynamic sorting (descending and ascending) as and when the deviation values are replayed from the store, at the end of the cycle. Dynamic sorting is implemented using a FIFO structure and multiple comparators. Hierarchical comparison is performed from top to bottom (bottom to top) in the upper (lower) pipe. At any level, if the input value is greater (less) than the existing values at that level, the input value is pushed into the FIFO at that level.

For each ID, the multiplexer chooses the minimum deviation among the two channels, in the case of offset computation, and the difference between the corresponding channels in a pair of cycles, in the case of rate computation. The mid-point deviation is the average deviation over the corresponding stages of the upper and lower pipes, the stage chosen depending on the number of valid deviation values

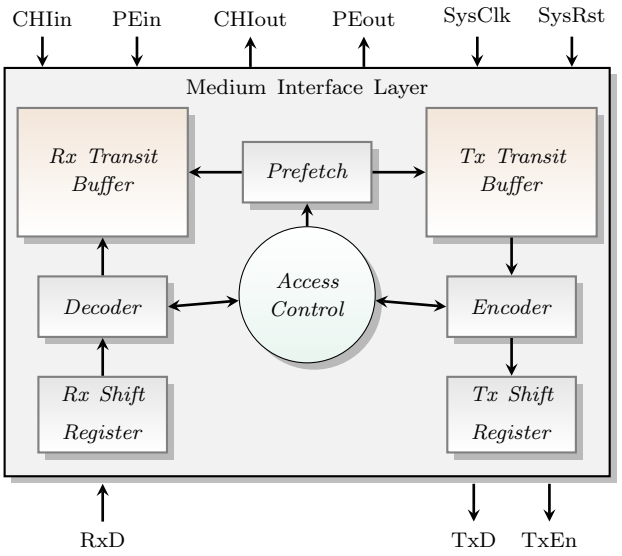


Figure 3.8: Medium Interface Layer architecture.

stored. The MacroTick Generation module uses the computed mid-point deviation values to make corrections to the node's view of time. Utilising the tagging established by the content-addressable memory and the pipelined architecture, the mid-point computation can be efficiently implemented at system clock rate to meet protocol requirements. A more conventional architecture would require a higher clock rate for this computation. Architectural optimisation also enables us to utilise fewer resources while maximizing performance.

The MIL module instantiates independent transmit and receive transit buffers to manage temporary storage of a frame, as shown in Figure 3.8. The MIL ensures that medium-access occurs only at slots assigned to the node. The access control state machine handles the bus access, depending on the current slot counter value and slot segment. The access control logic generates and maintains the slot counter and the slot segment, which are used by other modules in the CC. Within each slot, the logic generates control signals called action points, which mark points at which transmission can start (in static and dynamic slots) or end (in dynamic slots).

The signals trigger the encoding logic to start the frame transmission sequence, provided the current slot is allocated to this node. The data to be transmitted is moved to the transmit buffer over a 32 bit data bus. If no data is available for

transmission, the node transmits a null frame. The module also handles encoding and serial transmission of data (at the oversampled rate) in the current slot. The decoder functionality is also integrated into this module, which performs bit-strobing, majority-voting, byte-packing and validation of received data at the end of the slot.

The transmit interface is implemented using shift registers with gated clocks. This allows us to provide multiple functions with the same set of registers: encode and transmit data bytes, control signals and symbols. The shift register reads each byte from the transmit buffer, encodes it within the shift register and pushes it to the transmit line at the transmit clock, along with the transmit control signals. At the receiving interface, sampling, bit-strobing and edge synchronisation are implemented using a sequence of shift-register modules: one set samples the data and produces a majority voted bit every cycle, and the second set performs byte-packing of the data. This system offers the advantage of simpler control and higher throughput. The byte-packed data is written into the receive transit buffers. As and when protocol errors or violations are detected (like reception crossing boundary points), appropriate flags are set locally, which are used to validate the data at the slot boundary. At the end of the current slot, the flags are checked to signal valid data, which can then be written into the receive data memory in the host-interface.

The control modules are efficiently implemented as multiple state machines at different levels to ensure parallel and independent operation. The transmit buffers prefetch data from the transmit data store in the CHI at the start of each slot to minimize latency. Similarly, the data location for each received frame is pre-computed to enable the complete data to be written to the receive data store in the CHI before the start of the next frame, minimising latency from the time of frame reception to the time of intimation to the host. Also, the data available flag and interrupts (if enabled) are set, as soon as the first D-word is written into the receive buffer in CHI. The data store and the associated control and status store in the datapath mimic a content-addressable architecture in Block RAMs to enable

prefetching and addressing using the slot-cycle-channel complex, as required by the protocol.

Two such MIL modules are instantiated within the controller to support independent dual channel operation. These modules may transmit and receive data at the same slots, as configured by the host. To facilitate this, we have implemented a configurable scheduler, which can be configured for priority access (Channel A over B or vice versa) or first-come-first-serve mode. High word-length interconnects are used between the data store in host interface and transit buffers within the MIL module to ensure low-latency prefetch and write-back for both channels. Using such an architecture, the prefetching can be handled at system clock rates, without high latency. The physical layer can be configured to support multiple bit-rates of 2.5 Mbps, 5 Mbps or 10 Mbps. The shift register-based encoder/decoder module simplifies the logic requirements for handling multiple bit-rates.

The interface standard to host processor is designed to be compatible with Processor Local Bus (PLB) interface and the AMBA Advanced eXtensible Interface 4 (AMBA AXI4) standard from ARM, two of the widely used high-performance low-latency peripheral interconnects for system-on-a-chip (SoC) designs. The host interface supports parameterised widths and a wide range of system and interrupt configurations to provide a rich interface to the host processor (or logic). The control path comprising the command, status and configuration registers are isolated from the datapath and implemented as a register stack. Data corresponding to each cycle, slot and channel is addressed using an indirect addressing technique. The data pointer is stored at an address determined by the cycle-slot-channel complex. This allows us to use true dual-port Block RAM modules and simpler address generation as opposed to the complex FIFO-based schemes used by existing controllers. Another advantage is that the memory can be configured as a cyclic buffer resulting in an indefinite memory space, as opposed to the limited memory space available in a FIFO-based scheme. The memory space is dynamically allocated at the end of each slot that is configured as a receive slot, only if valid data has been received, thus optimising memory usage.

Asynchronous FIFOs are instantiated between the host interface and the control/-data stores, which enables the host interface to run at a clock speed independent of the PE. Using such a low-level design paradigm, we are able to leverage FPGA resources within the modules of the FlexRay Controller, thereby saving the remaining area for system implementation.

3.4.3 Controller Datapath Extensions

Traditional controllers depend on the host processor to read the received data and determine its usefulness. The controller issues a data interrupt, to which the processor responds with a status register read followed by a data read request, subsequently receiving the data. The associated overheads are wasted in the case of frames with irrelevant data (like obsolete or un-timely data) or multi-cycle data frames where the processor cannot process the received fragment until more data is available. In the case of critical data frames like error state that require immediate attention, the latency introduced by the traditional scheme limits the performance of safety-critical systems which rely on host-triggered recovery. With custom extensions, such exceptions can be handled at the controller, which processes the information and informs the host processor (using interrupts). The host retains absolute control, but is not involved in the low-level processing, which is handled instead by the configurable extensions. Figure 3.9 describes the functioning of such extensions on the receive-path of our controller.

On the receive path, the extensions can monitor the received data for matching FlexRay *message ID*, application-based *custom headers* or *timestamp* information, contained in the data segment of the FlexRay frame. The FlexRay message ID can be used for application/user defined communication in *dynamic segment* data frames. An interesting use-case is to embed the error status of the ECU into the message ID, which can trigger a fault-recovery procedure in safety-critical units. Application specific headers may be embedded into the data segment in any frame. Such headers convey information about the data contained in the frame, like sequence number and length, and are particularly useful in the case

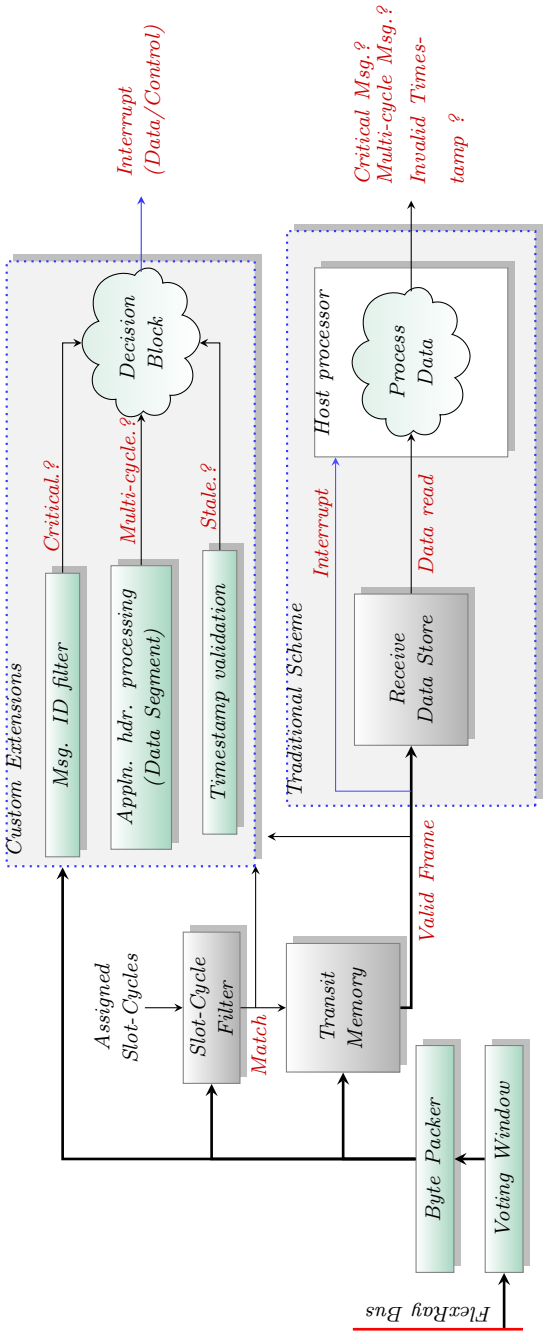


Figure 3.9: Receive path extensions on custom CC versus traditional schemes.

of large data transfers which are accomplished as multi-cycle transactions on the FlexRay bus. Information in the headers is used by the controller to re-pack the multi-cycle data. The header processing extensions on the receive path can look for such information and re-organise the segmented data and present it as a single transaction to the host.

Similarly, the timestamp validation extension can be configured to reject frames which are obsolete or untimely. On the transmit path, these extensions can insert relevant headers and timestamp information, as configured. Timestamp resolution is configurable, with a finest resolution of one macrotick and maximum length of four bytes. The header is entirely user configurable, and can be matched at the receiver by programming the corresponding registers.

Such extensions on the controller can help extend the functionality and overcome the inherent limitations of the FlexRay network, and are impossible to achieve on discrete controllers. Our pipelined architecture in the transmit and receive paths allows us to add this functionality with no additional latency. Standardising such extensions, automotive networks like FlexRay can be enhanced to implement a data-layer segment that provides security against replay attacks (using timestamps) and a standard methodology to communicate the health state of ECUs (using headers). Though such enhancements can be handled by the application in software, this would incur additional processing latency and unwanted complexity at the software level (like timing synchronisation).

3.4.4 Timestamp Synchronisation Mechanism

The timestamp synchronisation mechanism is derived from the distributed clock synchronisation mechanism used in FlexRay networks, and its startup procedure is integrated into the startup procedure of FlexRay communication. During startup, the leading coldstart node initialises traffic on the network by transmitting sync frames, which are also marked as null frames (i.e., no data content). The first non-null frame is transmitted once a second coldstart node starts transmitting sync frames in acknowledgement of the startup procedure. As soon as the leading coldstart node shifts to the coldstart resolution state (waiting for acknowledgement from other coldstart nodes after 4 cycles of sync frames), the timestamp timer is started locally at this node. In the first non-null frame, the current value of the timestamp of the leading coldstart node is transmitted along with the message, which is received by other nodes on the network. The nodes lock on to the ID of

the node (slotID) and use this later as the reference node for synchronisation. Like in the case with clock synchronisation, the nodes compute their current value of received timestamp by adding the known decoding delay and transmission latency (known from bit-rate and payload length parameters of FlexRay) and initialise their internal counters based on the computed value.

Once in operation, the nodes synchronise on every odd cycle, like the rate adaptation mechanism in FlexRay. The deviation of their internal timestamps, with respect to the timestamp in the sync frame from the leading coldstart node, is monitored in every odd-even cycle pair. At the end of the odd cycle, the clock correction is applied based on the average deviation recorded. This allows the timestamp timer to be synchronised at the MacroTick level. In the event of a fault with the leading coldstart node (i.e., the leading coldstart node does not transmit over an odd-even pair), the nodes will reset the stored ID and uses the first sync frame in the next cycle as the reference, and saves this ID as the reference ID for timestamp synchronisation.

3.5 Implementation Results

To validate our design and to measure actual performance on hardware, we have implemented it in a low power Xilinx Spartan 6 XC6SLX45 FPGA with a host module described using a state machine, modelling a complete ECU. We choose the Spartan 6 as it is a low cost, low power device, that would be a likely choice for an automotive implementation. To test the network aspects, we emulate a FlexRay bus within the FPGA, using captured raw bus transactions from a real FlexRay network (using Bosch E-Ray controllers) communicating using a predefined FlexRay schedule; these are stored in on-board memory. The information is replayed to create a cycle accurate replica of transactions on the bus. Our CC is plugged into this FlexRay bus, and configured with the same FlexRay parameters. Table 3.2 shows a specific set of parameters which was used for our experiment.

Table 3.2: FlexRay node parameters.

Parameter	Value
Number of Cycles	64
Cycle Duration	5 ms
Number of Static Slots	62
Static Slot duration	65 (macroticks)
Payload Length (Static)	21 words
Number of Dynamic Slots	10 (max)
Symbol Window duration	139 (macroticks)
NIT duration	208 (macroticks)
Sample Clock	12.5 ns
Keyslot ID Assigned	Slot 7
Transmission slots	Slot 7 in cycles 32 and 62

Table 3.3 details the resource utilisation of the individual modules of the controller and the power estimates generated by the Xilinx XPower Analyser tool, using activity information from simulation. We have configured the core to support all extensions on the transmit and receive path; a two-byte data header and a four-byte timestamp. The maximum operating frequency in this configuration was reported to be 88 MHz in the *balanced* compilation setting on Xilinx ISE. The core is initialised with parameters using a logic-based host-model over a PLB/AXI interface. The actual power measured using a power supply probe during operation in hardware is also shown.

Table 3.4 compares the resource utilisation of our implementation against the platform agnostic E-Ray IP core on the same Altera Stratix-II device, with DSP inference enabled and the extensions disabled. For the purpose of comparison, the consolidated utilisation on a Xilinx Spartan 6 is also shown in the same table, with both extensions disabled and enabled. It can be observed that the hardware centric approach results in much better utilisation of the heterogeneous resources, leading to a compact implementation. The design can also be easily ported to other Xilinx and Altera devices, and to other platforms with a little more effort. The resource utilisation and optimisations that we have achieved in comparison

Table 3.3: CC Implementation on hardware.

Usage	PM Module	CS Module	MIL Module
Registers	222	1864	732
LUTs	537	3579	1050
BlockRAMs	0	2	2
DSP48A1s	0	3	0
Est. Pow. (mW)	45	66	54
Actual Power	121 mW (at 80 MHz system frequency)		

Table 3.4: Comparison of implementations.

	E-Ray[62]	Our CC		
Utilisation	Altera Stratix II		Xilinx Spartan 6	
	No Extensions		Extn Disabled	Extn. Enabled
Registers	7754	4966	4910	5612
LUTs	12780	7856	7978	8767
BRAMs	23×M4K	33×M4K	5×9k + 12×18k	5×9k + 13×18k
DSPs	-	12×9-bit	3×DSP48A1	

with the platform agnostic E-Ray core is significant enough to justify the somewhat reduced portability. With the DSP inference disabled, our implementation consumed 8282 LUTs and 5248 Registers (on the Stratix-II), which is still below the E-Ray core. Another advantage is that the power consumption at full operation on a Spartan 6 device is below the power consumed by typical stand-alone controller chips like the Infineon CIC-310, which uses the E-Ray IP module [165], and consumes about 150 mW in normal operating mode.

A key advantage of implementing the communication controller in the FPGA fabric is the ability to compose more intelligent ECU nodes with enhanced communication capabilities on a single device. As an example, we have integrated a fully functional ECU node that combines this controller with a MicroBlaze softcore processor on a Xilinx Spartan 6 XC6SLX45 device, as in Figure 3.10. The ECU functions as a front-end processing node for radar-based cruise control and is built

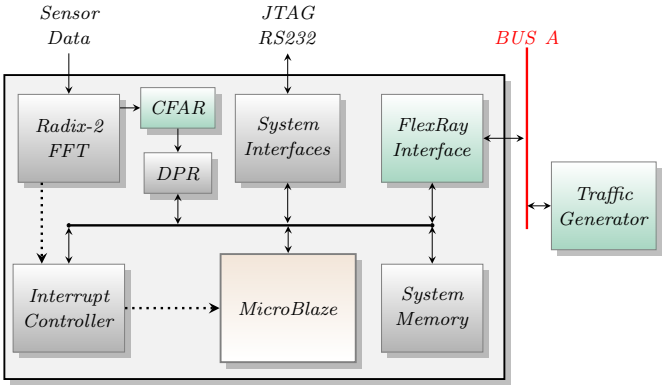


Figure 3.10: Integrated ECU function on Spartan-6 FPGA.

using Xilinx FFT IP cores and pipelined logic which performs target detection using a constant false alarm rate (CFAR) scheme [166]. The application is based on a frequency modulated continuous wave (FMCW) technique with a triangular modulating wave, which can simultaneously determine distance and range-rate of the preceding vehicle. The test data generates 1024 data points every 30 ms, which are transformed to the frequency domain by the FFT module. The CFAR module performs detection on the frequency domain data using multi-stage pipelined logic and writes results into the dual-port RAM. The processor is then interrupted, and it consolidates the data over a configurable number of cycles. The controller is configured with parameters defined in Table 3.2. Thus at cycles 32 and 62, consolidated results are sent on the FlexRay bus. Table 3.5 details the resource utilisation and power consumption measured during operation in hardware. Such an application would otherwise require specialised DSP processors, since the latency cannot be met by software implementation on a general purpose processor [166]. Similar performance can be obtained by interfacing high performance DSP devices like the Analog Devices ADSP-TS202S [167] with a standalone FlexRay controller like the Infineon CIC-310 or Freescale S12XF [23], but the node would consume much higher power overall than the integrated FPGA implementation. The key advantage here is that integrating ECU functionality and the network interface on the same device only increases power consumption marginally, and this interface can be shared between multiple functions on the same FPGA.

Table 3.5: Spartan-6 implementation of ECU on Chip.

Solution	Metrics	Proposed CC	Hardware Accelerator	Full ECU
Proposed Scheme	Registers	4922	4216	11778
	LUTs	7969	3221	13566
	BlockRAMs	13	11	60
	DSP48s	3	44	48
	Power Consumption	291 mW		
Discrete Solution	ADSP TS202	596 mW @ 100 MHz clock		
	Discrete CC	150 mW [165]		
	Total Power	746 mW		

MicroBlaze offers a low power, low throughput processing option for sensor applications. Alternatively, hybrid platforms like the Xilinx Zynq can be used for more compute intensive and real-time applications since they offer a more powerful ARM processor. By using AXI-4 for communication between the CC and the host, our design can be used with the ARM in the Zynq (consuming 5612 Registers, 8685 LUTs and 2 DSP48E1s) or with a MicroBlaze soft processor, or a custom hardware ECU.

3.6 Case Studies

We now present three distinct case studies that showcase the effectiveness of the custom extensions in the context of existing or proposed automotive applications. In each use case, we observe that the application can leverage the intelligence built into the controller, leading to smarter and more efficient systems when compared to standard implementations.

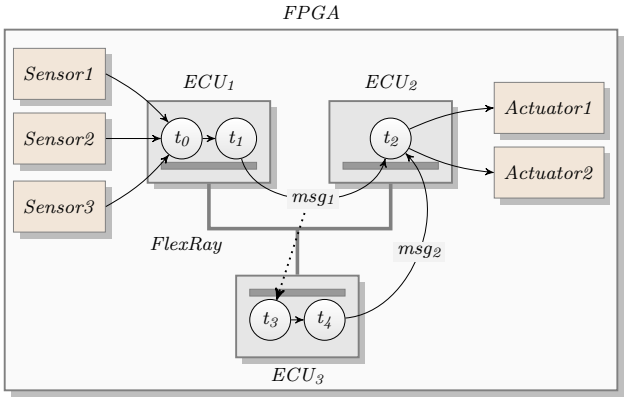


Figure 3.11: Test setup for brake-by-wire system.

3.6.1 Deterministic Decoding of System-State Messages in Safety-Critical ECUs

Safety-critical systems employ redundant or fall-back modes, which enable minimum guaranteed functionality, even in the presence of hardware/software faults. One of the critical parameters in such a system is the time to switch to fall-back mode once a fault has been identified. For this experiment, we model a brake-by-wire system comprising two MicroBlaze ECUs on the FlexRay network; the brake sensor ECU, which interfaces to the sensor modules, and the actuator ECU, which issues commands to the braking system. Each ECU incorporates fall-back logic which is triggered when a fault-status message is received. These status messages are generated by a centralised fault detection ECU that monitors bus transactions for unsafe commands/data. The sensors and actuators are modelled using memories: sensor data is generated from a *Sensor BRAM*, and commands are pushes to the *Actuator BRAM*. The sensor ECU combines inputs from the different sensor interfaces periodically and passes it over the FlexRay bus to the processing ECU. The processing ECU uses this data to compute commands, issues them to the actuators, and acknowledges the receipt of commands from the sensor ECU. Both ECUs run software routines on the popular *FreeRTOS* platform. A simplified model of the test setup is shown in Figure 3.11.

To mimic the behaviour of off-the-shelf controllers, we disable the custom extensions on the CC. A fault-status message is triggered on the sensor ECU system

by configuring invalid data in the *Sensor BRAM*, causing incorrect sensor-data to be issued to actuator ECU over the FlexRay bus. The fault-detector logic detects the error and transmits the error code in the next slot assigned to it. A normal controller decodes this message, passes it to the MicroBlaze processor, where the data is processed to trigger fall-back mode. The latency from the transmission of the error message to the triggering of fall-back mode is largely determined by the interrupt-based data passing mechanism used in off-the-shelf controllers. Even for an RTOS-based (real-time) system, this latency can be significant, and was measured at an average of 9.05 ms for our implementation, as illustrated in Figure 3.12.

By moving such critical data processing to the controller, it becomes possible to significantly reduce this delay and enhance the determinism of the system. To quantify this, a processing extension that detects packets on a user-configured slot with a user-specified data header is enabled on the CC. On detecting this combination, the controller can either process the remaining data for specific patterns, or trigger an interrupt. In this particular experiment, it is configured to process the critical error flags and the consecutive error numbers to decide whether to trigger fall-back mode. This generates a direct interrupt to the MicroBlaze processor and enables fall-back mode, resulting in a faster and more consistent turnaround time (average 50× faster than RTOS), as shown in Figure 3.12.

We have also repeated the experiment using the Xilinx standalone (SA) OS, the lightweight minimalistic OS for MicroBlaze. It can be observed from Figure 3.12 that though the simplified standalone OS results in lower average interrupt latencies than the RTOS, it results in a larger spread of latencies.

3.6.2 Extended Communication using Data-layer Extensions

While the pattern matching extensions on the controller can be configured to decode selected control messages for initiating action, this requires alterations in

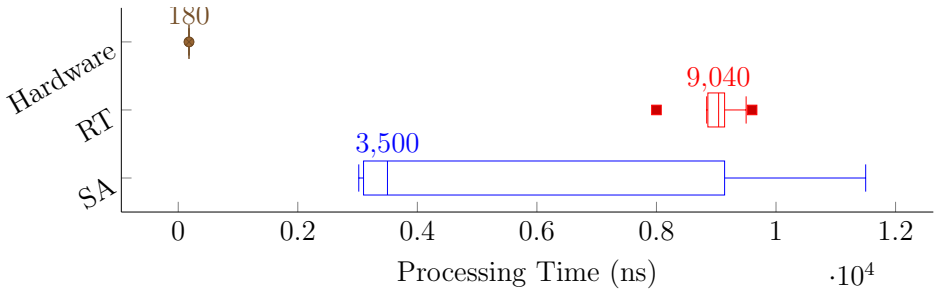


Figure 3.12: Latency distribution for interrupt-based critical data processing.

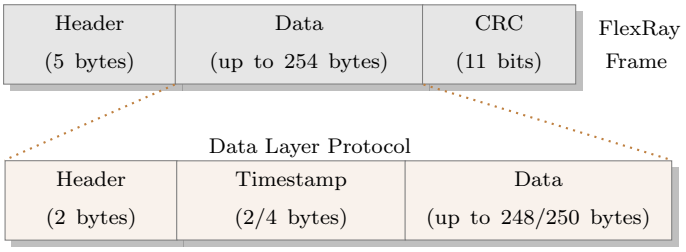


Figure 3.13: Encapsulating additional information into existing messages as data-layer headers.

the communication schedule to incorporate additional communication to handle such changes. A smarter approach would be to integrate the health-status and acknowledgement into the data-segment as message headers, creating a data-layer extension to the messages that are already being exchanged, as shown in Figure 3.13. This prevents the need for regenerating the communication schedule and to validate the performance of the system in the new setup before evaluating the efficiency of the fault-tolerance features. The controller is also capable of handling such scenario's using the different configuration options available on the pattern matching extension.

To evaluate this scheme, we use the same 3 ECU brake-by-wire system described earlier. MicroBlaze processors form the ECU units, with applications (tasks) loaded as software. The ECUs are networked using the FlexRay protocol.

The ECUs exchange messages similar to the earlier setup, but with the messages incorporating additional information, as shown in Figure 3.14. The definition of the different messages are as below:

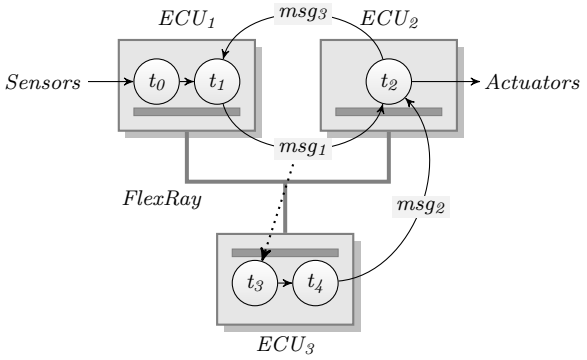


Figure 3.14: Messages exchanged between the ECUs

- ECU_1 (sensor ECU) reads the sensor data and generates sensor commands (msg_1).
- ECU_2 (actuator ECU) receives sensor commands and health messages (msg_2) from ECU_3. If health status is normal, it acknowledges sensor commands (msg_3) and produces actuator commands; else it flags a fault state via (msg_3).
- ECU_3 (monitor ECU) monitors msg_1 and generates health status for ECU_1 (msg_2).

In this test, we utilise health flags on msg_3 to indicate a fault in ECU_1. From a pre-determined point, the ECU_1 messages are corrupted at the bus by injecting faults, which triggers the monitor ECU (ECU_3) to generate the health message with fault-flag's set. This would further set the fault-flag in the acknowledgement message from ECU_2 to ECU_1, forcing the fault-tolerant mode to be activated in ECU_1. Once the fault is removed (after the configured time window), the fault injection is disabled, and the monitor indicates this using a change in health state of ECU_1. This is reciprocated in the acknowledgement message of ECU_2, by setting the flags to normal.

In the default setup, such enhancements are added into the software that interfaces with the communication tasks. The same functionality can be enabled in the hardware by enabling the custom extensions in our FlexRay CC. With hardware extensions enabled, we can observe faster and more deterministic response rates for the above test instances, compared to the software processing of such messages.

Table 3.6: Software and hardware performance comparison.

Enhancement	Processing	Software	Hardware
Timestamp accuracy (Tx)		64 µs (2 slots min)	100 ns
Health monitoring	Msg processing	3550 ns	180 ns
	Turnaround (standby)	3650 ns	280 ns

We also observe that the hardware-based timestamps enable much higher accuracy than the software-based scheme, where even the best-case scenario produces an inaccuracy corresponding to two FlexRay slots. The key idea here is that, such extensions can be enabled without requiring additional communication slots, by embedding such information within the existing message exchange. The only software overhead is to periodically report the system health state to the FlexRay CC, which is then incorporated into the outgoing messages transparently.

3.6.3 Time-Awareness for Messages

A major security risk in time-triggered systems like FlexRay is the lack of time-awareness for messages. By monitoring bus transactions, an external attacker can easily employ simple replay attacks, flooding the bus with stale data, as described in [152]. The FlexRay protocol leaves this vulnerability to the higher layer applications to tackle. In our controller, the transmit path allows messages to be optionally time-stamped to make the message time-aware, at the cost of increased payload size. By inserting the header and timestamp within the data segment of the FlexRay frame, it is transparent to other FlexRay controllers present on the network, ensuring interoperability with off-the-shelf controllers. With timestamps enabled, the receive path can be configured to automatically drop frames which are outside an allowed time window. This creates a basic security layer at each ECU, which can be augmented further by incorporating encryption/decryption logic in the datapath.

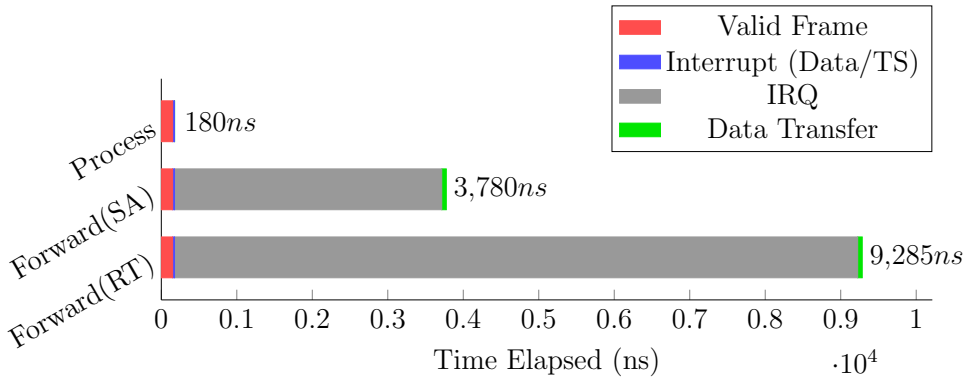


Figure 3.15: Timestamp processing at interface.

An interesting use-case is in high-performance gateways that move data between network clusters. With traditional interfaces, messages arriving from each interface will be forwarded to the switch logic, which decides whether to forward the data to its destination or drop it because it has expired. By building intelligence into the controller, the validity of data can be determined before it is forwarded to the switching logic. We modify the experimental setup in Section 3.11 to model a gateway configured to discard untimely data, either at the processing logic (MicroBlaze), mimicking off-the-shelf interfaces, or at the interface using our enhanced controller extensions. Our tests show that the interface can process the timestamp and discard the message within 180 ns of frame reception. A standard approach consumes a further 3.6 and 9.1 µs on average, for standalone (SA) and RTOS (RT) respectively, as shown in Figure 3.15, since the data must be processed by the host.

3.6.4 Handling Volume Data at Interfaces

Applications like radar-based cruise control utilise volume data gathered by the radar-sensors to compute distance and relative velocity of other vehicles in the vicinity. A complete dataset from a sweep is required by the processing logic to determine these parameters, and this data is received over many data slots. The processing ECU must reassemble these fragments before the data can be processed. By moving this packing/re-packing to the controller interface, the processing logic can overlap the computation with data reception, enabling it to run at lower frequencies and hence consume lower power.

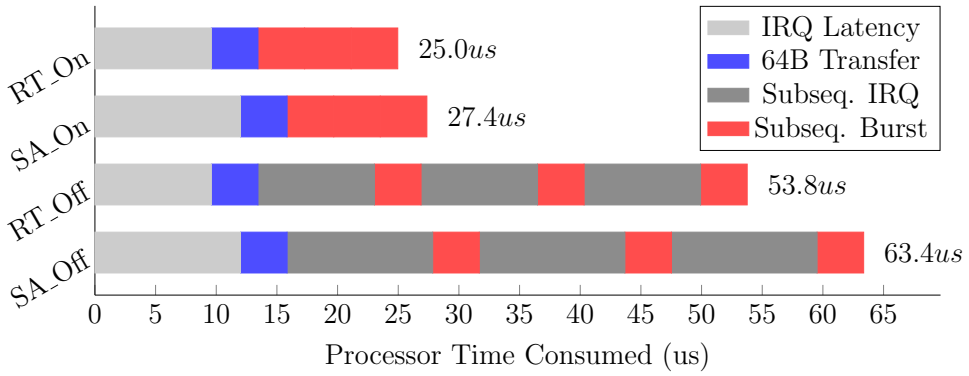


Figure 3.16: Data re-packing for multi-cycle data transfers.

To demonstrate this, we use the experimental setup for the radar-based cruise control ECU, described earlier in Section 3.5. The data from the radar sensor is received over the FlexRay bus in bursts of 256 bytes, the maximum payload size defined by FlexRay standard. The MicroBlaze processor runs the standalone (SA) OS from Xilinx. In a normal design (referred to as SA_Off), the processor is interrupted each time a block of data is received. The processor responds with the first data read request 12 ms (worst-case) after receiving the interrupt, with the burst read consuming a further 3.84 ms. This is repeated over four cycles to complete the data transfer, cumulatively consuming 63.36 ms.

We then test the same application with an extension that allows the controller to intelligently buffer the entire frame, only interrupting the processor at the end of the transaction (referred to as SA_On). This enables the processor to issue back-to-back reads from the controller completing the entire data movement in 27.36 ms from the reception of the interrupt. To provide a balance between multi-cycle and single-cycle data, the design has been constrained to handle up to four data cycles at full payload size. To support larger data sizes, larger buffer memories must be added to the controller, resulting in higher device utilisation, however, this may be a tolerable cost for some ECUs, and the CC architecture supports it.

The experiment was also repeated using the *FreeRTOS*-based (RT) software (denoted by RT_Off for normal design and RT_On for CC with extensions enabled), which provided better determinism than the standalone OS, resulting in a lower worst-case interrupt latency, as shown in Figure 3.16.

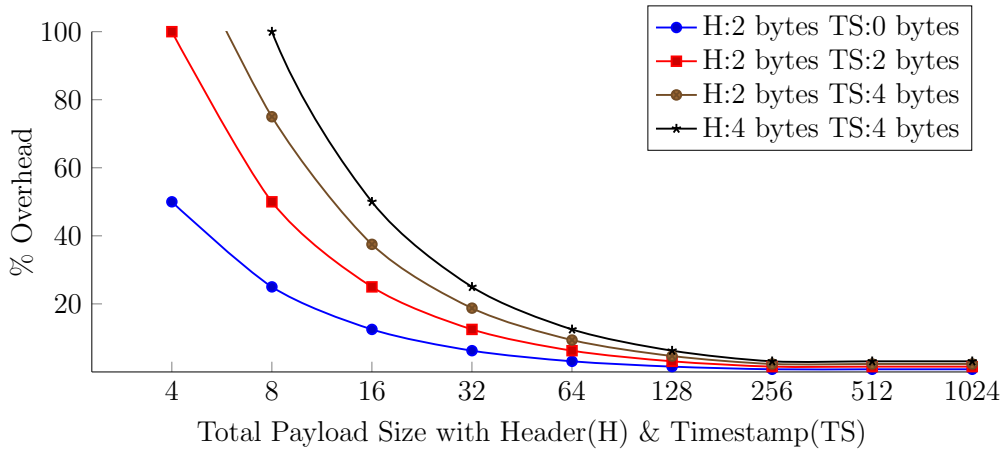


Figure 3.17: Overheads for including headers and timestamps.

3.7 Discussion

The FlexRay protocol does not define the usage of headers within the data segment, which is left to user implementation. While the use of headers and timestamps within data provides the aforementioned advantages, it does result in significant payload overheads for small data sizes, while also limiting the payload capacity of a FlexRay frame. Figure 3.17 compares the overheads associated with different configurable values for the application header and timestamp, as a function of the payload size. As can be observed, at lower payload sizes, the inclusion of a timestamp and application header results in large overheads, but for large payload sizes, the penalty paid is very small. Beyond the maximum payload size of 256 bytes, additional data has to be handled as multi-cycle transactions, causing the curve to flatten out for higher payload sizes. Since the application header and timestamp data is inserted within the data segment of the FlexRay frame, it is transparent to other FlexRay controllers on the network, ensuring interoperability with standard controllers.

3.8 Summary

In this chapter, we have given an overview of the FlexRay protocol and the generic architecture of the communication controller, as defined by the specification. By

identifying and extracting operations which are mutually exclusive or natively parallel, we have designed a custom controller which takes advantage of the heterogeneous resources on modern FPGAs, resulting in reduced logic footprint, and low power consumption, while providing a host of features beyond those described by the standard. A flexible implementation of the network interface opens up research opportunities to enhance automotive communication and systems architecture. This architecture-driven design approach also improves power consumption compared to the use of discrete controllers. The datapath extensions enable capabilities like deterministic communication of system state, enhanced data movement for multi-cycle data and synchronised timestamps for messages, without affecting the reliability of the protocol. We later see how these can be leveraged to help secure FlexRay networks in Chapter 5. Such extensions can also enable advanced computational capabilities like fault-tolerance and function consolidation to be built into nodes that integrate complex ECU functions with advanced communication controllers. We explore these approaches in Chapter 4. The availability of a protocol compliant interface provides an opportunity to explore such architecture-level enhancements and to demonstrate the concepts within a realistic, certifiable environment.

4

Enhanced ECU Architectures

4.1 Introduction

We have seen that the more complex applications in modern vehicles are leading to a rapid rise in the number of ECUs, resulting increased complexity and added weight for the computational infrastructure in vehicles. A key challenge in future vehicles is consolidation of multiple functions onto fewer ECUs with fault-tolerant behaviour to ensure reliability.

However, traditional ECU architectures based on automotive grade general purpose processors or micro-controllers (MCUs) do not support aggregation in a way that ensures isolation between tasks. The different functions running on an MCU must share its resources, resulting in unwanted contention unless specific steps

are taken to manage it. Such a shared architecture also suffers from a lack of determinism which can be problematic in safety-critical systems.

One alternative being explored is the use of multi-core MCUs with isolated processing units for safety-critical applications like drive-by-wire. However, multi-core platforms are expensive and require complex software to manage interactions and ensure necessary isolation to provide contention-free access to resources. And though multi-core operation can offer some scalability due to an increase in raw compute power, performance is still limited by the sequential software execution paradigm and the limitations like task switching latency.

We have seen that reconfigurable hardware offers a promising platform for smart nodes that closely integrate the communication interface and functional logic. Multiple functions can be aggregated onto a single FPGA with complete functional isolation, as different hardware regions can be used. Advanced techniques like partial reconfiguration (PR) can provide extensive functionality and run-time hardware level adaptability to implement complex functionality. Leveraging PR, such smart nodes can also be adaptive and self-healing. We consider their suitability as an alternative to expensive multi-core platforms for next generation safety critical systems. Furthermore, the natively parallel nature of hardware architectures allows for accelerated computations that are isolated in such a way that they do not violate the deadlines of other aggregated tasks.

In chap. 3, we saw that integrating the extensible network interface with the CC on a single chip provides advantages in power consumption and performance. We now look at some of the architecture level enhancements that benefit from tight integration of the CC and computational units on reconfigurable hardware. We show that the extended communication framework, combined with the capabilities of FPGAs opens up opportunities for architecturally superior ECUs with built-in support for consolidation, isolation, determinism, and scalability, while including the benefits of superior performance and energy efficiency.

The work presented in this chapter has also been discussed in:

1. S. Shreejith, K. Vipin, S. A. Fahmy, M. Lukasiewycz *An Approach for Redundancy in FlexRay Networks Using FPGA Partial Reconfiguration*, in Proceedings of the Design Automation and Test in Europe (DATE) Conference, Grenoble, France, March 2013, pp. 721–724 [12].

2. S. Shreejith, S. A. Fahmy, M. Lukasiewycz *Reconfigurable Computing in Next Generation Automotive Networks*, IEEE Embedded Systems Letters, vol.5, no. 1, pp. 12–15, March 2013 [13].

3. S. Shreejith, P. Mundhenk, A. Ettner, S. A. Fahmy, M. Lukasiewycz, S. Chakraborty *AEG: An FPGA-based Gateway Architecture for Ethernet backbone In-vehicle Networks*, Transactions on Very Large Scale Integration (VLSI) Systems (prepared for submission).

4.2 Related Work

FPGAs have been proven in a wide range of domains to offer significant benefits in accelerating algorithms and offering high computational efficiency. Advanced techniques like dynamic partial reconfiguration (PR), allow parts of the FPGA to be used differently at different times, enabling non-concurrent functions to use the same hardware. PR is supported in most recent FPGAs; selective alteration of portions of the FPGA configuration memory mean parts of the circuit can be changed while other parts continue to function.

FPGA-based ECUs have been proposed in the automotive domain for compute intensive non-critical functions like driver assistance. One of the mechanisms to ensure AUTOSAR compliance for FPGA-based ECUs is discussed in [130], using soft-core processors as an MCU replacement. Here, the AUTOSAR run-time environment is mapped to a register interface on the FPGA, providing the same functionality as standard AUTOSAR compliant MCUs. The combination of reconfigurable extensions and customisations allow such ECUs to provide superior computational throughput over off-the-shelf systems. Furthermore, the hardware

execution enables their performance to be predictable, making them suitable even for safety-critical applications.

FPGA-based ECU architectures that exploit reconfigurability and parallelism have also been explored by researchers. In [121], the authors describe an architecture for implementing fail-safe safety critical ECU systems on FPGAs leveraging dynamic reconfiguration (complete reconfiguration). The described architecture uses FPGA logic as a fail-safe back-up, which is completely reconfigured with one of the back-up modes when errors are detected. Partial reconfiguration has been used in non-safety-critical automotive applications such as driver assistance systems [10]. In such cases, using PR can allow a reduction in the required target FPGA size by time-multiplexing different functionality. An ECU with fault-tolerant communication controller was also demonstrated using PR to dynamically reconfigure a faulty controller [93]. The authors propose the use of partial reconfiguration to swap in a new communication interface (different protocol) in case of a fault with the current communication controller, ensuring a minimal operative mode can be supported over a secondary network.

Partial reconfiguration is usually performed using a special built-in hardware macro called the internal configuration access port (ICAP). Traditionally, the reconfiguration operation is controlled by a processor, through a vendor-provided controller such as the HWICAP (AXI or OPB), connected as a slave device to the processor bus [168]. Using these vendor-provided controllers gives low throughput in the region of 4.66–10.1 MB/s [169, 170]. This should not be the case, as the ICAP hard macro itself supports much higher speeds of 400 MB/s. Low speed ICAP controllers are not suitable for time-critical systems, where slow reconfiguration may lead to system failure. To overcome this, we make use of a custom ICAP controller that is capable of performing reliable reconfiguration at nearly the theoretical maximum limit of the ICAP macro [171].

More complex adaptive systems are now being integrated into vehicles, as in the case of Adaptive Cruise Control (ACC) systems. Simpler implementations of such time- and safety-critical applications have been part of vehicles for more

than a decade [172, 173]. With increasing complexity, software implementations on processor-based platforms are insufficient to satisfy timing criticality and new approaches have to be explored. This provides a promising approach for FPGAs, which have been widely employed in several time-critical applications in aerospace [174, 175], telecommunications [176], and the medical domain [177].

FPGAs have also been widely employed in line-rate switching systems for more general high performance networks like Ethernet [178, 179, 180], offering low latency switching. Different switching modes can also be supported as described in [178], where high performance and low power modes are supported with varying latency. Customisable datapaths allow FPGA-based switches to analyse traffic during switching [181], and this can be extended to incorporate security features like intrusion detection [182, 183]. For vehicular systems that integrate multiple networking protocols, message exchange between the different segments is enabled by such switching mechanisms, commonly referred to as gateways. FPGAs are ideally suited for such mixed criticality data exchange where custom designs can analyse traffic criticality and prioritise switching operations in real-time. FPGA-based network gateways have been proposed in the literature [149, 150, 151], providing deterministic message routing between current automotive networks like LIN, CAN and FlexRay. We explore the possibilities of utilising FPGAs for mixed criticality message exchange in Ethernet backbone networks in future vehicles.

4.3 Contributions

We present a system-on-chip ECU architecture for safety-critical and non-safety critical ECUs that benefits from the underlying parallelism and reconfigurability of FPGAs. The architecture extends the communication framework to integrate mode-switch commands or fault-diagnosis messages which are then handled by extensions on the network interface for deterministic decoding and predictable behaviour. We demonstrate that vendor provided mechanisms for partial reconfiguration offer sub-optimal performance that impacts turn-around time constraints for

critical systems, limiting the use of such approaches for non-safety-critical ECUs. Using PR in this manner also requires ECU software to manage the low level operations, requiring knowledge of FPGA features. We overcome these limitations by extending the functionality of the network interface further by integrating a custom reconfiguration controller. This allows the entire functional module to be reconfigured without the application being aware of or managing the process. Our approach also enables low-latency activation for the secondary logic, and high speed recovery of the faulty functions, even from a hardware-level fault.

Finally, we evaluate the case for using hybrid platforms as the core element for next generation vehicular gateway systems, by providing a deterministic and scalable interconnection scheme between existing networks like FlexRay and CAN, with future networks like Automotive Ethernet.

The work in this chapter consists of novel architectures and methods that have been implemented and validated in FPGA hardware with realistic experiments.

4.4 Consolidation Methods using PR

Non-safety-critical systems include user-oriented features like multimedia, telematics, remote diagnostics, and future systems like Vehicle-to-Vehicle (V2V) communication. Such systems are characterised by the high volume of data handled, high throughput requirements and complex computation, an area where FPGAs represent an ideal implementation platform. Indeed the computational power of custom hardware on FPGAs enables applications that would otherwise be infeasible on low-power processors. Since FPGAs implement computation spatially, we can split the available resources among multiple functions, maintaining the predictability of each while ensuring complete isolation between them.

Furthermore, we can time multiplex applications that are not needed concurrently by using dynamic Partial Reconfiguration (PR) [130]; mutually exclusive functions are mapped to the same dynamic region on the device, which can be reconfigured

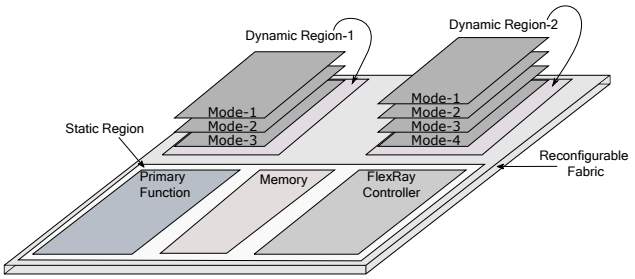


Figure 4.1: Visualisation of adaptive ECUs on an FPGA using PR.

at run time. Primary ECU functionality can be defined in the static, non-changing region of the device, while each dynamic region would have specific functions or accelerators for the current operating mode, as illustrated in Figure 4.1. The illustration described in Table 4.1 shows the different functions that can be integrated on a smart-node and the distinct interfaces or modules required to implement them. The FPGA design is partitioned to incorporate concurrent modules in different dynamic regions, considering the computational and bandwidth requirements of each. The dynamic regions can then be reconfigured, when needed, to integrate multiple non-concurrent functions on the same node. This architecture can be extended to integrate complex adaptive systems for current and future in-vehicle applications, on a much smaller device.

Future automotive systems significantly increase the amount of data that is gathered for processing and use algorithms that are significantly more complex. Examples are driver assistance systems like pedestrian detection or blind spot warning. Software implementations of such algorithms require specialised processors or multicore systems while hardware implementations can provide more performance at lower power. The computational capabilities of FPGAs can be exploited to provide an efficient and flexible solution that also integrates the communication

Table 4.1: Modes of Operation for Consolidated ECUs.

Functionality	Modules in Dynamic Region 1	Modules in Dynamic Region 2
Park Assist	Custom Logic	Sensor Interfaces
Application Acceleration	Softcore Processor	Custom Logic Interfaces
Cruise Control	Adaptive Logic	Sensor Interfaces
Safety-critical ECU	ECU Function	Redundant ECU

interfaces to the various sensors and in-vehicle networks on the same hardware, saving significantly on infrastructure. Furthermore, PR can be employed to re-use the dynamic areas of an FPGA to offload computations from another ECU, while these resources are not being used by the initial application.

4.4.1 Evaluating Consolidation using Vendor-based PR

Consolidating multiple non-concurrent ECUs on a single device reduces the number of ECU modules, bus drivers and the associated wiring, all contributing towards a better in-vehicle ecosystem. Traditionally, each ECU uses discrete (or integrated) controllers to access the bus. In an FPGA-based node, PR can be utilised to consolidate multiple functions at much lower power consumption, while sharing the interface between multiple functions can be managed in hardware in a predictable, fair manner.

To demonstrate the efficiency of this approach, we integrate multiple non-concurrent functions on the same hardware and show that reliable activation can be achieved when the respective modes are triggered. For this, we reuse the radar-based cruise control ECU described in Chapter 3 with an intelligent parking solution [184] on the same hardware platform. We make use of a Xilinx ML605 development board containing a Virtex-6 FPGA (XC6VLX240T), since it natively supports PR. The parking algorithm design uses fuzzy logic to evaluate the present conditions based on sensor inputs and generates appropriate control signals for the steering system. Since these applications are mutually exclusive, we can utilise PR to create an adaptive node, which modifies its functionality based on current requirements.

To manage PR operations, we make use of the PR framework provided by Xilinx through their software package, PlanAhead [119]. This is achieved by instantiating a separate MicroBlaze processor to which the Xilinx hardware ICAP module (XPS_HW_ICAP) is integrated to manage reconfiguration when a mode change

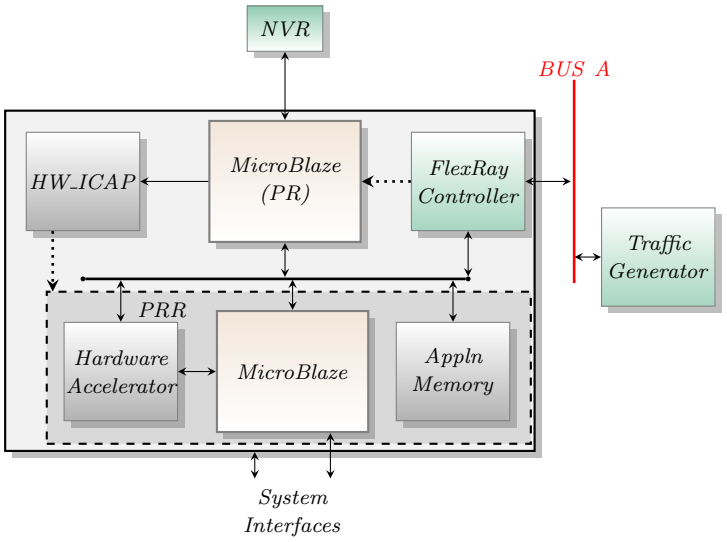


Figure 4.2: Consolidating non-concurrent ECUs on FPGA.

is required. This controller function is completely isolated from the system functionality (Park assist, cruise control), allowing the two different modules to be incorporated when required.

Both applications make use of software running on a MicroBlaze soft core processor along with dedicated hardware implementations for accelerating the computations. The system functions are implemented within the region on the FPGA that has been marked as partial reconfiguration enabled during design time, referred to as a partially reconfigurable region (PRR). Each ECU, when active, interfaces with the shared network bus over our extended FlexRay CC, that is implemented in the static region along with the reconfiguration controller logic. The various components are connected using a Processor Local Bus (PLB) interface, as shown in Figure 4.2.

One of the interesting things about PR is that when no modes are required to be active (e.g., when the driver is using the vehicle normally), power savings can be achieved by disabling the PRR. For this, the PRR is filled with a blank bitstream, constituting the *IDLE* mode. When a mode switch is required, which is usually triggered by a driver action (like enabling cruise control or switching on park assist), a mode switch command is generated by the driver interface ECU and transmitted over the FlexRay bus. The pattern detection extension within the

custom FlexRay CC processes the command and issues a high priority interrupt to the MicroBlaze processor that is responsible for reconfiguration management, triggering the PR operation. The interrupt service routine (ISR) corresponding to this interrupt reads the partial bitstream from the flash interface and sends it to the internal configuration access port (ICAP) to load the requested module.

Since both functional ECUs make use of software control, the MicroBlaze processor used for handling the reconfiguration control may be assigned as the functional controller for both the ECUs. This approach requires only the hardware acceleration logic to be reconfigured, reducing the reconfiguration overhead. However, this approach does not provide clear isolation between the functions, since the code/-data memory of the processor is shared between the two applications. Furthermore, the software would also have to handle the reconfiguration request, which may offset its time-bound execution, since it requires the processor to explicitly handle the operation. We also evaluate this case to show the potential benefit of sharing the processor.

Table 4.2 shows the implementation metrics for this design, including resources consumed by these modules (in the partial region), and the dynamic power consumption while operating in these modes. Since for each scenario (completely isolated functions with independent MicroBlaze or shared MicroBlaze), the PRR that implements the two functions (park assist or cruise control) is of the same size, the partial bitstream size for each case is the same, resulting in identical time to switch between the two in any scenario. The results also show that the

Table 4.2: Virtex-6 Implementation of Adaptive Smart ECU.

Mode		Utilisation		Switching Time	Dynamic Power
Design	Operation	Registers	LUTs		
PR	Idle	12223	13695	-	240 mW
	Isolated Functions	6699	5490	2336.4 ms	365 mW
	Shared MicroBlaze	4215	2130	936.4 ms	340 mW
Static	All Modes	32484	31558	NA	640 mW

adaptive node has a definite advantage in terms of power consumption and utilisation compared to a purely static implementation, integrated as two isolated functions on the same device. However, the reconfiguration speed is low since the maximum throughput obtained using the vendor provided PR management system is around 10 MB/s. This results in an activation time of 2.3 seconds for the completely isolated implementation and 936 ms for the the shared system.

The use of custom high speed reconfiguration controllers [171] enables a much faster turnaround time, making FPGA-based fault-tolerant nodes more viable for safety-critical applications. The custom reconfiguration controller also does away with the additional processing system that is integrated purely for management of PR, which results in higher power consumption in static mode. We explore the possibilities of tightly coupling a custom reconfiguration controller with the network interface to make effective use of the datapath extensions discussed in Chapter 3.

4.5 Redundancy for Safety-Critical ECUs

Safety-critical systems like drive-by-wire, ABS, or occupant safety systems are hard real-time systems requiring high levels of determinism and isolation. They may have to interface with multiple in-vehicle networks to control and coordinate operations of critical systems like the drive-train. Safety-critical systems are often implemented to support fail-safe or fault-tolerant operation. As discussed in Section 2.3, a fail-safe system maintains a basic set of tasks even in presence of fatal errors, while fault-tolerant systems can adapt and recover from critical errors without severely degrading system performance, often achieved by redundancy.

FPGA-based designs can instantiate multiple instances of identical ECUs within the same device to aid redundancy while providing better determinism [130, 121]. FPGA-based designs that incorporate PR provide alternative solutions for redundancy since PR allows us to reconfigure only necessary logic rather than the whole FPGA. In a fault-tolerant scheme, an error causes the logic to switch to

a redundant mode which operates with lower specifications. PR enables us to reconfigure the faulty region alone, without affecting current operations, resulting in faster turnaround times. The fault detection logic on the FPGA triggers the switch to the redundant mode of operation and the subsequent reconfiguration of appropriate dynamic region(s), when a critical error is detected. Also, multiple implementations of an application with differing levels of error tolerance can be swapped in on the fly to deal with changing conditions. In addition, a single region of programmable fabric can be assigned as the redundant region for multiple functions, rather than the need for distinct circuitry for all systems to be present at the same time.

Deterministic behaviour is easily factored into systems implemented on reconfigurable hardware. FPGA-based designs are synchronous, event-triggered systems and hence respond to events in a deterministic manner. Hardware-level parallelism can be exploited by designs to ensure that multiple simultaneous events can be handled independently without contention. Specific events like Single Event Upsets (SEUs) can be mitigated in logic, using either fail-safe or fault-tolerant design methods. Incorporating PR, the erroneous ECU (or function) alone can be reconfigured without affecting other regions, providing higher determinism.

4.5.1 Extending FlexRay Communication Cycle

As we have mentioned, the fundamental element of the FlexRay protocol is the cycle that is composed of the static and dynamic communication segments, as illustrated in Figure 4.3. Though the *Static segment* can ensure guaranteed determinism for our purpose, it is heavily used for high priority real-time communication in most real implementations. Our focus is on extending the flexible *dynamic segment*, which is mainly used for event-triggered communication on a priority basis. Dynamic segments can have dynamic slot-width, depending on the amount of data that needs to be transferred, which is of interest in our case. Our choice of using the dynamic segment also allows us to build extended communication infrastructure, without disturbing current systems. The scheduling of such communication

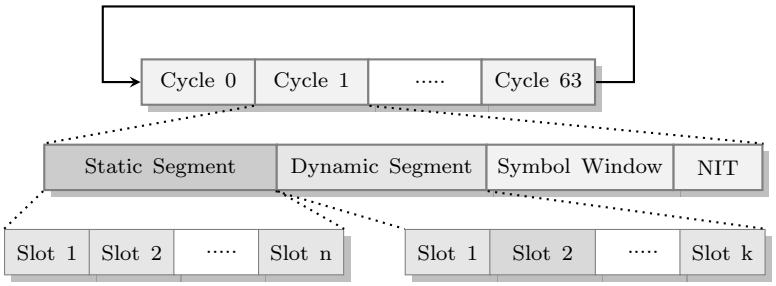


Figure 4.3: Specification of the FlexRay cycle.

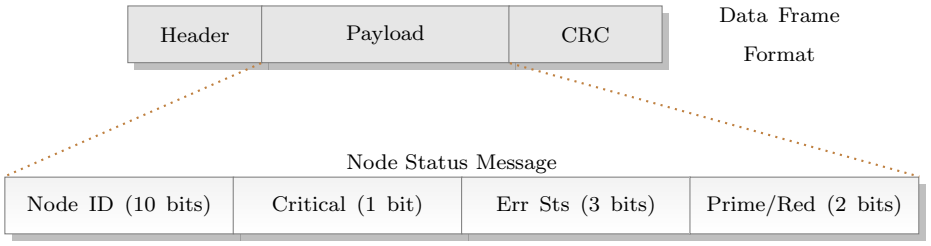


Figure 4.4: Dynamic segment payload with Message ID.

on the dynamic segment for deterministic communication can be solved by using standard scheduling algorithms. Here we explore a simple mechanism, which incorporates the two possible extreme cases – use the first *n slots* in each cycle or use one slot in every cycle, which is reused for *n nodes* across different cycles. In either case, the slot in the dynamic segment communicates a fixed length system status messages for safety-critical nodes, at varying periodicity depending on the scheme chosen. These messages are used to trigger recovery in case of errors.

For this application, we propose to make use of an optional *message ID* feature that is described by the protocol specification [53]. *Message ID's* are used in the dynamic segment as an optional two-byte data element, whose use is not restricted by the protocol. Using this inbuilt mechanism allows us to make use of the existing protocol filtering scheme, allowing this approach to be applied even with standard off-the-shelf controllers. Alternatively, the datapath extensions on our custom CC can be configured to achieve the same functionality with lower latency, since our extensions process the messages closer to the lower protocol layers compared to the upper layer operation defined in the protocol.

We propose to utilise bits of the *message ID* to indicate the critical status of the device, and whether its functions need to be taken over by a redundant unit. The

proposed frame structure for communicating this status is shown in Figure 4.4. Node-ID defines the unique ID assigned to the different safety-critical nodes within the network. The critical error status flag describes a critical error condition, and demands immediate attention. The error status flag indicate the consecutive number of non-critical errors encountered by the node, which are tolerated up to a predefined threshold. The prime/redundant flag is used to distinguish the prime unit and the redundant unit, which use the same node-ID. Within our architecture, each system may have localised fault detection logic or depend on distributed fault detection modules or on a combination of the two.

In each cycle, the nodes fill their dynamic slot with current status (encoded into the *message ID*) from internal fault-detection logic or received fault status from centralised fault-detection logic. The node indicates (or receives) critical conditions or reports of high error rate using these flags. Since each node transmits only a fixed amount of payload data (2 bytes) in the highest priority slots, determinism in data transfer (or reception, as the case may be) is ensured by this scheme. The lower priority dynamic slots may still be used by other nodes for volume data transfer.

4.5.2 Proposed Approach to Reconfiguration

Figure 4.5 is a high level block diagram of an ECU system on reconfigurable hardware. The *primary function* is the computational implementation of the safety-critical function, which uses custom hardware accelerators for compute-intensive calculations. The ECU may also communicate with other ECUs or sensors over the FlexRay bus through the FlexRay communication controller (CC), which is an instantiation of our enhanced CC described in Chapter 3. The functional unit, comprising the primary function (MicroBlaze soft processor), accelerator(s), and associated memory are implemented in a partially reconfigurable region (PRR), so that the functionality can be switched or reconfigured in case of an error. There can be multiple PRRs in a single FPGA. We designate this one PRR-1. The FlexRay CC and the PR controller are implemented in the static region SR-1.

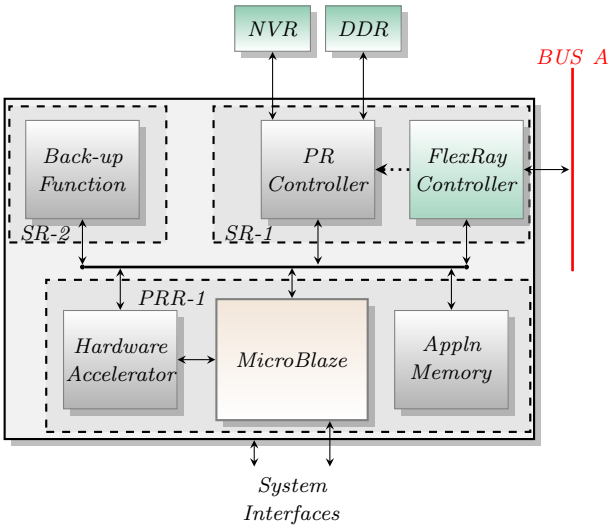


Figure 4.5: Fail-safe ECU architecture on FPGA.

The second static region is designated to implement redundant logic. Under normal operation, the node is healthy and SR-2 is configured with the redundant logic, but is clock-gated to conserve power. If a local fault is detected, the status is indicated to other nodes over the FlexRay bus by setting the critical error flag in the status data and the redundant logic is enabled. If the node depends on remote fault-detection, a FlexRay frame with the critical error detected flag set in the status data triggers the switch to redundant mode. Disabling clock-gating takes only one clock cycle, resulting in fast turnaround to the redundant mode. Subsequently the partial reconfiguration controller is triggered to reconfigure PRR-1, to enable recovery from the error. PRR-1 is isolated from the system bus by the PR controller, and hence the reconfiguration proceeds without disturbing the functionality of the node. Once PRR-1 is reconfigured, it is enabled and the FlexRay CC can be *optionally* brought to the reset state and re-initialised with the network parameters, so that the node can perform a complete restart. After the node integrates onto the network, SR-2 is disabled to conserve power.

As mentioned, use of message ID allows the standard filtering scheme with off-the-shelf CCs to use this scheme; however, they may not be able to match the pattern of the content to figure out if a reconfiguration is required or not. This would have to be added to the application and processed in software. With the datapath extensions on the custom CC, we can configure the extensions to filter dynamic

segment messages with a *message ID* matching its own ID (in case of remote fault detection) and to transmit messages with its own *message ID* and status, in the priority slot assigned to it (in case of local fault detection). This approach incurs lower latency, and is transparent to the software application. Hence we only explore this scheme, for which we tightly integrate the configuration management circuitry with our extended datapath on the network interface.

Whenever an error state is received or transmitted, the trigger is passed to the partial reconfiguration controller by the pattern matching extension on the CC. This allows reconfiguration commands to be acted on immediately by the PR engine, instead of routing them through the normal data path via the Host processor and application software. In a normal design, using a fixed standard controller, the FlexRay messages, or local error state would need to be processed by the processor, resulting in noticeable latency before the redundant logic is activated. In our approach, the turn-around time from the point of receiving a reconfiguration command to switching to the back-up mode can be made negligibly small.

4.5.3 High-Speed Reconfiguration Management

Though traditional FPGAs like the Xilinx Virtex-6 support PR out of the box, we have already seen that the vendor provided PR management incurs the additional overhead of adding a processing system to manage reconfiguration (see Section 4.4). The reconfiguration latency of such schemes is also high, making them unusable for critical systems.

To overcome these limitations, we make use of a custom designed ICAP interface and integrate it closely to the FlexRay CC. The custom ICAP that we are using accelerates reconfiguration by operating close to the theoretical maximum performance of ICAP [171]. Figure 4.6 shows the design of the partial reconfiguration controller, expanded from the *PR Controller* block in Figure 4.5. The FPGA is connected to an external non-volatile memory, which stores the partial bitstreams.

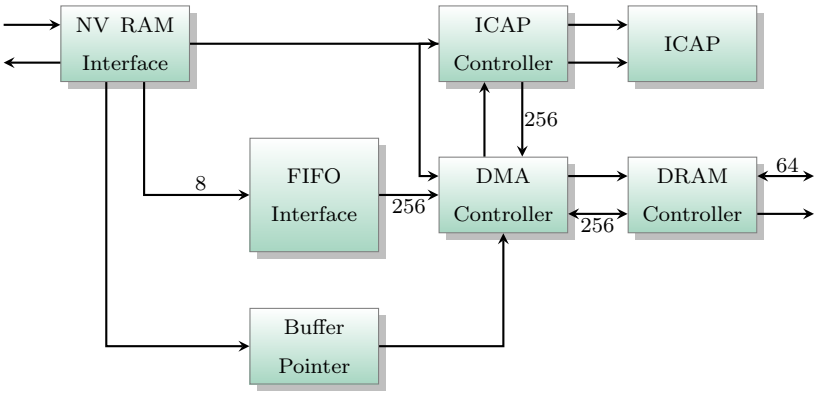


Figure 4.6: DMA based PR controller.

During system initialisation, these partial bitstreams are cached into the DRAM memory using block write operations issued by the DMA controller.

The higher reconfiguration throughput is achieved by the combination of a DMA controller and high-speed DRAM interface. Once a reconfiguration operation is triggered, the DMA controller configures the memory controller to buffer the partial-bitstream from the cached location, using back-to-back requests. The configuration pointer holds the size of partial bitstream and its physical address, which is used by the control state machine to configure the DMA engine. The DRAM controller logic interfaces with the external DRAM generating the required control signals for the interface.

The custom ICAP controller also instantiates an asynchronous FIFO for buffering, the ICAP clock generator, ICAP control state machine, and the ICAP hard-macro. The asynchronous FIFO allows the logic to operate at a different clock speed to achieve maximum throughput (read port to DRAM clock and write port to ICAP clock). The ICAP control state machine handles the ICAP hard-macro and is responsible for feeding required control information to the macro for enabling reconfiguration.

4.5.4 Distributed Redundancy

While highly critical nodes demand point-wise redundancy, a distributed architecture may be employed for other nodes which are less prone to errors and failures.

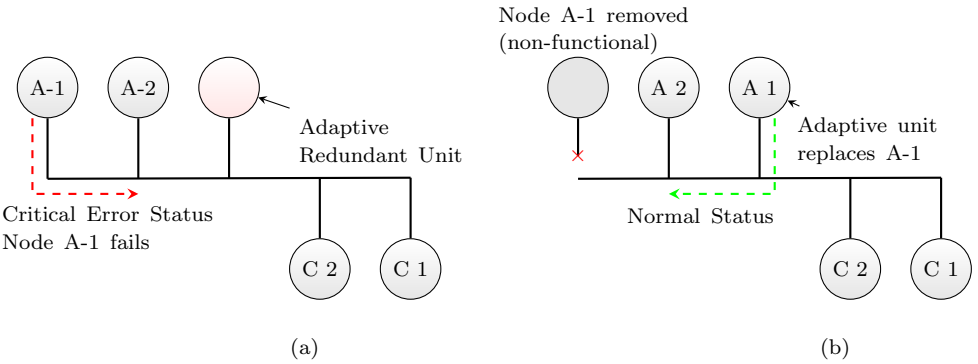


Figure 4.7: A Distributed redundancy scheme for non-critical ECUs.

In such an environment, we can assign a single backup region as the redundant unit for multiple nodes. Each node in the subset has a virtual redundant unit, which is enabled only when the original node is disabled by errors. The architecture of the redundant node is the same as in Figure 4.5, where virtualisation is enabled by multiple bitstreams that emulate the behaviour of the subset nodes. SR-2 in Figure 4.5 of the adaptive node can be programmed as centralised fault-monitoring logic, which monitors the fault status of each of the subset nodes in the respective slots. On detecting (or receiving) a fault condition, the active node forces itself to turn off, while the redundant unit assumes its position and identity on the bus. Such a virtual redundant scheme is more efficient in terms of power, space and weight, than a point-wise discrete redundancy scheme. PRR-1 in Figure 4.5 of the adaptive node is initially blank, consuming zero dynamic power. The region is programmed with the functionality of the erroneous ECU as soon as the fault is identified and takes over the function, including the bus identity and configuration of the failed ECU, as shown in the transition from Figure 4.7 (a) to Figure 4.7 (b).

It is also possible to integrate multiple redundant units onto a large enough FPGA, which can operate in complete isolation. Resources can be effectively partitioned into two or more distinct regions, which are large enough to incorporate the functional units, running in parallel with complete isolation. Such redundant units can be distributed throughout the network or associated with clusters of critical ECUs. This improves the number of simultaneous failures that can be handled by the system at the expense of slightly higher cost and complexity. The datapath extensions in our CC enable such functionality to be implemented in hardware,

without the need to process these messages in application software. In this way, PR can offer numerous possibilities for robustness and consolidation in in-car systems.

4.5.5 Validating PR-based Functional Fault Tolerance

To validate the architecture and investigate turnaround times, we have extended our radar-based cruise control front end application, based on FMCW radar and target detection using Constant False Alarm Rate (CFAR) algorithm. The system was developed using Xilinx EDK 13.3 and hardware validated on a Xilinx ML605 development board containing a Virtex-6 FPGA (XC6VLX240T). The *primary function* from our node architecture is a MicroBlaze based system, as in Figure 4.8, which executes software routines to estimate parameters from the frequency domain data that is generated by the radix-2 FFT module. To make the computation simpler, we use a 1 millisecond triangular modulator with a radar cycle of 32 milliseconds. The estimates are then passed over the FlexRay bus to the central node, which also serves as the monitor to identify critical or erroneous conditions on the node. The error logic sets the critical error flag when a critical error is identified and increments the error status bits in the presence of non-critical errors. The primary ECU functionality is contained within PRR-1, while the redundant unit is contained within static region, SR-2. The FlexRay controller, reconfiguration controller, and test logic are designated as static region SR-1. The logic utilisation of the design is as shown in Table 4.3. PRR-1 has a partial bitstream size of 262 KB.

The FlexRay bus configuration is as shown in Table 4.4. Since we are using a single ramp FMCW for this experiment, the node has valid data only once every 32 ms, and hence, is assigned static slot 7 in cycles 32 and 64 for data transfer. The node is configured to monitor the first dynamic slot (slot 24) which relays node status using our proposed scheme. The node is configured as a non-cold-start node to measure the worst case delay to integrate back on to the network, after successful reconfiguration of the primary ECU functionality. The performance of

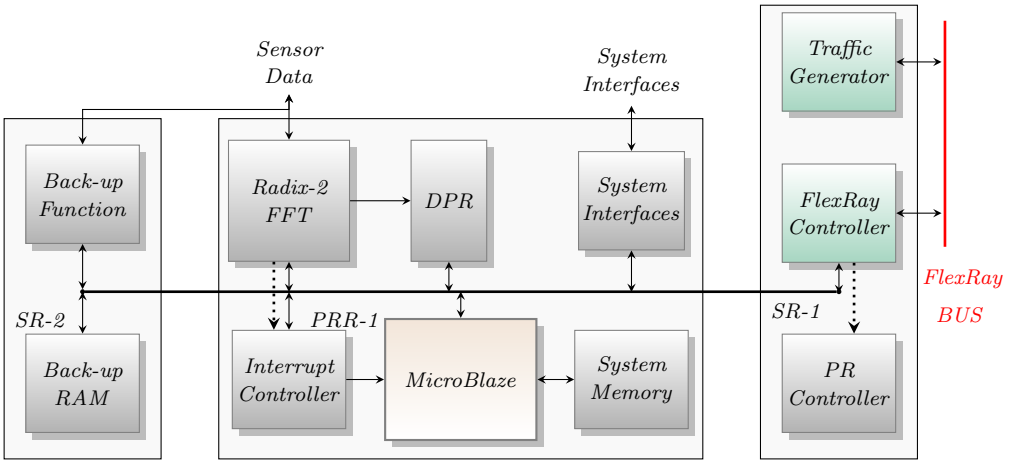


Figure 4.8: Prototype for fail-safe Radar Signal Processing node.

Table 4.3: Resource Utilisation of Radar ECU.

Resources	PRR-1	SR-2	SR-1		
			ICAP Cntrlr	FlexRay I/f	Test Logics
Registers	3632	200	672	4607	268
LUTs	3473	138	586	7021	577
BRAMs	24	2	8	8	6
DSPs	6	-	-	3	0
Total Power	1.8 Watts				

Table 4.4: FlexRay Parameters used for evaluating Radar ECU.

Number of Cycles	64
Cycle Duration	1 ms
Number of Static Slots	23
Static Slot duration	35 (microticks)
Number of Dynamic Slots	35 (max)
Slot IDs Assigned	Slot 7 in Cycles 32 and 64
Slot IDs Monitored	Slot 24 (Dynamic)
CHANNEL ID Used	1023 (hex 3FF)

the framework in the different test cases is measured in terms of turnaround time and node recovery time. Turnaround time defines the time to switch to redundant logic from reception (or detection) of an error, while recovery time is the time taken to recover from an error and resume normal operation. The turnaround time to switch to the redundant mode of operation is shown in Table 4.5.

Table 4.5: Turnaround time to the Redundant mode for different Fault-Tolerant Architectures.

Interrupt Processor	Xilinx MicroBlaze interrupt controller	Custom Logic
FlexRay CC Latency	20 ns	20 ns
Interrupt Latency	12 µs to 420 µs	-
Back-up Active (turnaround)	12 µs to 420 µs	30 ns

The failure command was issued via a bus traffic generator, emulating error detection by a centralised monitor node. In our first experiment, the data interrupt is processed by the MicroBlaze processor, which checks for the critical error flag or a consecutive error count greater than the threshold and issues a reconfiguration command to the ICAP controller. However, the MicroBlaze also processes FFT data using a high priority interrupt resulting in a worst case interrupt latency of 41038 clock cycles, depending on when the error status flag was received. Thus the turnaround time to switch to redundant mode can vary from 12 µs in the best case to 420 µs in the worst case.

For our second experiment, the reconfiguration interrupt was instead processed within the CC using the pattern matching extension that generates a reconfiguration interrupt when it detects an error code in slot 24. This can then be routed directly to the ICAP controller, resulting in a reduced latency of 20 ns. Hence, the redundant logic can be enabled in a deterministic short turnaround time of 30 ns, which is independent of the state of the functional logic in PRR-1.

To measure the overall recovery time, tests were undertaken with different PR controllers, with the results shown in Table 4.6. To measure the worst-case recovery time, we also consider the *optional* step of re-initialising the FlexRay interface, which consumes 6 FlexRay cycles.

In the first experiment, the ICAP controller used is the Xilinx XPS hardware ICAP controller. The design took 26,240 µs to reconfigure the functional block in PRR-1. The node re-integration consumes further 6 FlexRay cycles, taking the total recovery time to 32.25 ms.

Table 4.6: Recovery time for the different Fault-Tolerant Architectures.

ICAP Controller	Xilinx ICAP controller	Custom Controller (DMA based ICAP)
FlexRay CC Latency	20 ns	20 ns
PRR - 1 Active	26,240 µs	656 µs
Recovery time (node restart)	32.25 ms	6.66 ms

In the second experiment, we replace the standard controller with our custom PR controller. Its high throughput means PRR-1 reconfiguration is completed in just 656 µs, providing a 40× improvement over the Xilinx controller. Considering the re-integration time, the node completed recovery and re-integrated onto the network in 6.656 ms.

The results show that the extensions in the CC allow us to achieve short and predictable turnaround time (to redundant logic) of 30ns, compared to a traditional interrupt-based processing (up to 420 µs) that is used with discrete controllers and processor based designs. This is because we can process the messages within the controller hardware using datapath extensions, rather than the a long round trip to a processor.

The use of clock gating results in a power efficient architecture, when compared with other alternatives where redundant logic must be active constantly. It can also be inferred that the worst case latency of a processor based architecture can be significant and non-deterministic depending on the instructions being executed and can be overcome by using custom logic and extensions. Using a custom ICAP controller for PR, the recovery time of a safety-critical ECU on reconfigurable hardware can be reduced to support higher levels of fault-tolerance. The percentage utilisation of the device is also less, permitting aggregation of more functions onto the same hardware, which can run in complete isolation. Furthermore, a single piece of hardware can be designated as a redundant unit for multiple nodes and the framework can be extended to support multiple redundant modules on the same device, thus saving power and space.

PR can also be combined with other techniques like artificial neural networks to enable fault tolerance in sensor nodes, where physical redundancy is infeasible, like in the case of the air-flow path within combustion engines [185]. In this case, PR is used to replace a fault-detection model with a sensor replacement model in the cast of a fault. PR enables such dynamic adaptation at the hardware level, resulting in predictable performance and improved energy efficiency.

4.6 Gateway ECUs for In-Vehicle Ethernet Backbones

For evolving applications like advanced driver assistance systems (ADAS) that integrate a large number of sensors and actuators, existing network protocols are making way for higher bandwidth interconnect. Recent developments point to Ethernet as a likely candidate, with Broadcom's BroadR-Reach physical layer chips shown to support 100 Mbps bandwidth reliably over unshielded cables [107]. However, vehicular systems would still depend on CAN and FlexRay for some existing functionality. The Ethernet backbone infrastructure illustrated in Figure 4.9 has been proposed as a viable solution to allow existing systems to operate without modification, while providing high performance interconnect for services that rely on information from these existing systems as well as volume data sensors, which are connected through an Automotive Ethernet Gateway (AEG). The AEG would be another ECU on the network, with software-based control mechanisms allowing each branch of the network to be independently controlled and disconnected (if needed), to meet the reliability requirements of critical systems. Moreover, the interconnect must offer low-latency switching with priority-based routing to support exchange of mixed criticality messages across domains.

Since the gateway architecture must be adapted across various vehicle models (and ranges), an ASIC implementation would not be ideal. Performance and scalability requirements suggest FPGAs are an ideal implementation platform. However,

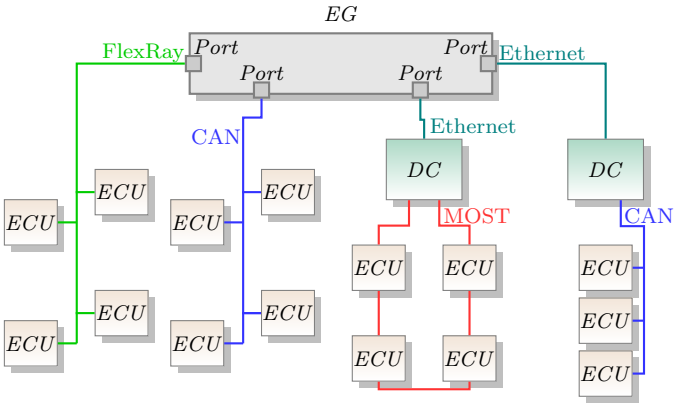


Figure 4.9: Proposed Vehicular Network Architecture using an AEG. Non-critical functions on legacy networks use the Ethernet backbone via the Domain Controllers (DC), while critical functions are directly interfaced to the AEG over corresponding networks.

standard FPGA architectures offer accelerated digital computation through custom architecture implementations. Software control can be added using simple soft processors instantiated in the logic, like the MicroBlaze we have shown in our experiments so far. These soft processors, however, do not offer sufficient computational capability to implement complex algorithms with time-bound performance.

Alternatively, the FPGA fabric may be connected as an extension of a standard automotive MCU as an accelerator. However, this approach only provides a loose coupling between the function on the MCU and the accelerator, impacting overall latency, especially when there is frequent data movement between the two processing elements (MCU and accelerator). As we have shown earlier with our extensible network interface, tightly coupling the computational logic with the network interface allows us to extend the functionality of the network as well as enhancing the application's capabilities.

Such coupling is offered architecturally in new hybrid FPGAs recently introduced by both Xilinx and Altera. Our AEG architecture exploits this coupling to provide a scalable switching architecture with software control, with switching branches designed in a modular fashion to allow adaptability to different in-vehicle network designs.

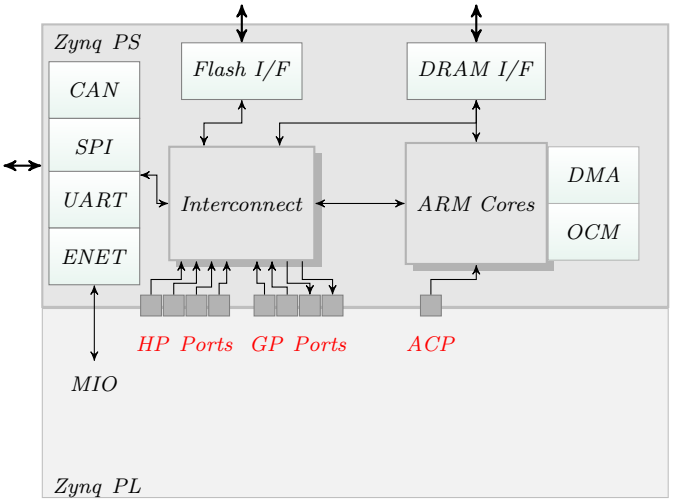


Figure 4.10: Zynq Architecture showing the Processor Subsystem (PS) and Programmable Logic (PL).

4.6.1 Zynq Hybrid FPGAs

The new Zynq family from Xilinx are hybrid reconfigurable devices that offer tight integration of a capable processing system (PS) with configurable programmable logic (PL) on the same die, as shown in Figure 4.10 [186]. The PS is a hardened region of the die that combines a dual-core ARM Cortex-A9 processor along with several memory and connectivity interfaces. The Cortex-A9 along with its memory subsystem is capable of hosting a fully-fledged operating system like Linux and can operate as a standalone device without any support from the PL, providing a familiar environment for embedded software developers. For extended functionality, the ARM cores are connected to DRAM interfaces, Flash memory controllers, CAN, Ethernet, USB, and UART controllers. The connectivity to these peripheral blocks is established through ARM's AMBA eXtensible Interface(AXI) interconnect. For interfacing to real-world signals, Zynq devices also incorporate two multi-channel (17-channel) high resolution (12-bit) analog-to-digital converters (ADCs). This wide array of interfaces in the PS makes it ideally suited as a hardware platform for a full fledged embedded system.

The functionality of the PS can be further extended with custom logic in the PL region. Zynq offers high bandwidth interconnect between the PS and PL. Furthermore, dedicated direct memory access (DMA) blocks enable high-speed data

movement between PL and interfaces managed by the PS, like DRAM memory or the Ethernet interface. The PL is based on the Xilinx 7-series architecture, combining flexibility features like partial reconfiguration, higher computational capabilities (like advanced DSP48E1 blocks) and lower power consumption. The hybrid architecure enables scalable and parallel implementations of complex processing blocks in the PL, while retaining software-based control through the tightly coupled ARM cores [187]. Our AEG further explores the case for hybrid FPGAs in next generation vehicular networks.

4.6.2 AEG Architecture

The AEG offers multiple physical switching ports (4 in our use case, numbered 1 to 4), each capable of providing up to 1 Gbps data throughput. The physical interfaces can be Gigabit Ethernet, FlexRay, or CAN, and for our experiments we use Gigabit Ethernet and FlexRay interfaces. The number of branches and interface types are defined using top-level parameters, which can be altered for different configurations. The physical interface logic is responsible for implementing protocol related functions like CRC checks and header insertion/processing. Independent transmit and receive paths handle connections from the interfaces to the switch fabric, which implements cross-connectivity. The FIFOs embedded within these paths helps to decouple the physical interface from the switch fabric. These FIFOs provide a byte-wide data interface to the Medium Access Controller (MAC) logic in the physical interface and are of sufficient depth to prevent data loss due to overflow. All forwarding decisions are based on the Ethernet layer-2 headers, with each non-Ethernet ECU having a virtual mapping in the MAC address space.

4.6.2.1 Receive Path

The receive path buffers incoming frames and makes the forwarding decision based on the Ethernet MAC header. It employs three modules that operate on-the-fly

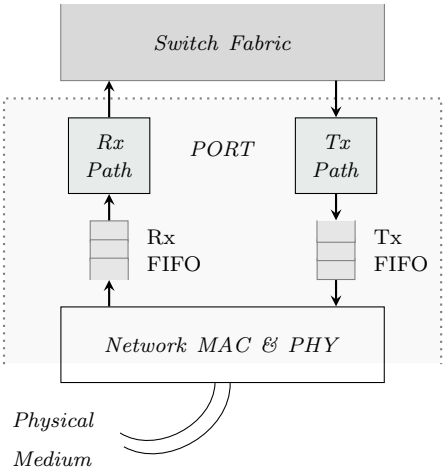


Figure 4.11: High-level block diagram of a Port.

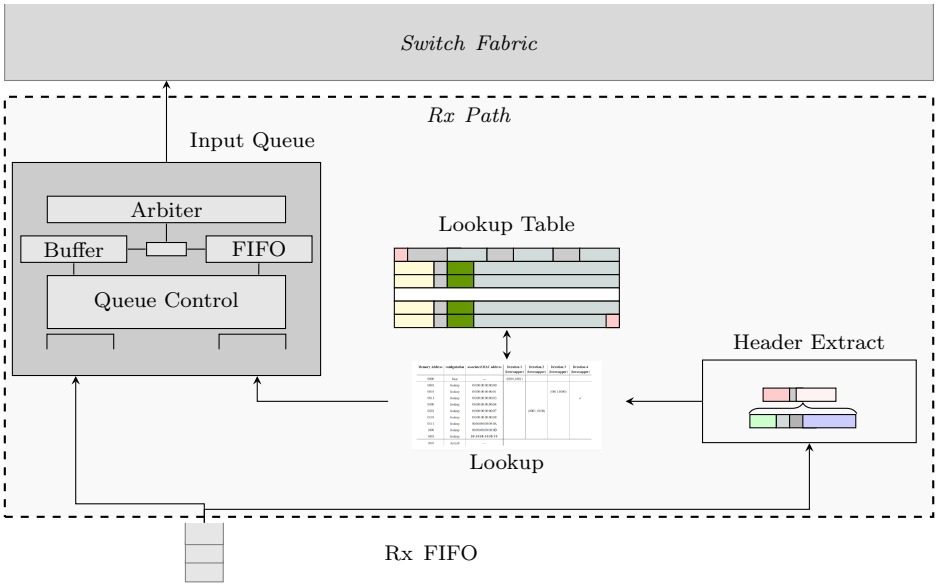


Figure 4.12: Receive path of a Port with their functional sub-modules.

on received frames as they are passed up to the switching infrastructure. The *header extraction* logic determines the header segment, classifies the frame and passes the information to the *lookup* module which determines the output port for the frame in the form of a binary vector (called *port vector*). The *input queue* acts as the temporary buffer for incoming packets, before they are forwarded to the switch logic. The operations of these modules are scheduled in such a way that the destination port for a packet can be determined before the complete frame has been received, even for the smallest allowed payload size. The high-level block representation of the receive path is as shown in Figure 4.12.

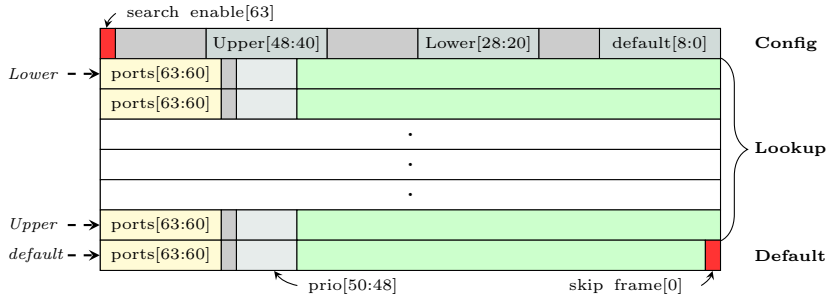


Figure 4.13: Lookup table structure in the Receive Path.

The **header extraction** module determines the start of each frame and extracts the destination MAC address, VLAN tag, and frame priority from the frame header as they are received. With the first data byte, it also records the 64-bit arrival time stamp for the packet, with a resolution of 8 ns (125 MHz). This timestamp is used by later stages of the logic to ensure latency-based routing and also by the management interface at higher layers to determine performance and worst case delays. The extracted header information is passed to the **lookup** module to determine the destination port mapped to the destination MAC address.

The **lookup** module implements a binary search on the sorted list of MAC address values to determine the destination port for a given address. Since all possible destinations are predefined in an automotive system, a sorted table structure presents a more efficient scheme for look up than specialised associative mechanisms like a content-addressable memory. Use of binary search allows the 1000 MAC entries in the table to be searched within a maximum of $ld(1000) \approx 10$ clock cycles. The lookup memory contains three types of memory entries; the *configuration entry*, the *lookup entries*, and the *default entry*, as shown in Figure 4.13.

The *configuration entry* is the first in the lookup table and is read by default before each search is initiated. It provides information about the lookup memory organisation; the *lower* field indicates the lowest MAC address, the *upper* field indicates the highest address in memory and the *default* field specifies the default destination port in case no entries are matched. The *search_enable* bit indicates the status of the lookup table and if set to zero, indicates that the lookup should not be

performed, forcing the use of the *default* setting (which could be to drop the frame). This bit can also be used to isolate a branch, in case of persistent faults, allowing communication from other ports to be handled without introducing errors.

If enabled, the search algorithm operates on the *lookup entries*, the sorted array of MAC addresses and their destinations in MAC address order. Each entry has a 48-bit destination mac field, the associated output port vector (1-bit for each port) and its priority. During the lookup, if a match is found between the incoming MAC address and the location, then the corresponding port vector is used. The priority field indicates the priority that will be allocated to the frame if it is not VLAN-tagged, else this field is ignored. The *default* configuration also uses the same structure as the *lookup entries*, but uses a *skip_frame* bit, which when set, forces the input path to drop the frame instead of forwarding it.

The **input queue** module temporarily buffers incoming frames until they can be transmitted to the output port, and is composed of five sub-modules: control logic, queue buffer, scheduler FIFO, overflow control logic and the arbitration module. The high level organisation of the different sub-modules is shown in Figure 4.14.

The *queue control* module interfaces the lookup module with the Rx FIFO. It initiates a read from the Rx FIFO as and when the a frame becomes available, and a write into the *queue buffer*, saving the location of the first write saved as the *mem_ptr* for that frame. Locations are dynamically allocated (next free space) in the *queue buffer* which is configured as a ring buffer. Further, the *length* of the frame and the error indications from the MAC (*mac_error*) are also saved when an entire frame is received. Once the lookup information for the corresponding frame is available, and if there are no errors (mac_error and skip_frame are both 0), the *mem_ptr* and *length*, along with the lookup information (*port vector*, *prio*) and timestamp are written into one of the *scheduler FIFO* blocks, depending on the frame priority (i.e., highest priority frame written to the highest priority FIFO and so on). In the present system, we have only two possible priority settings defined ((real-time traffic and non-real-time traffic) and thus only two *scheduler FIFOs* are used.

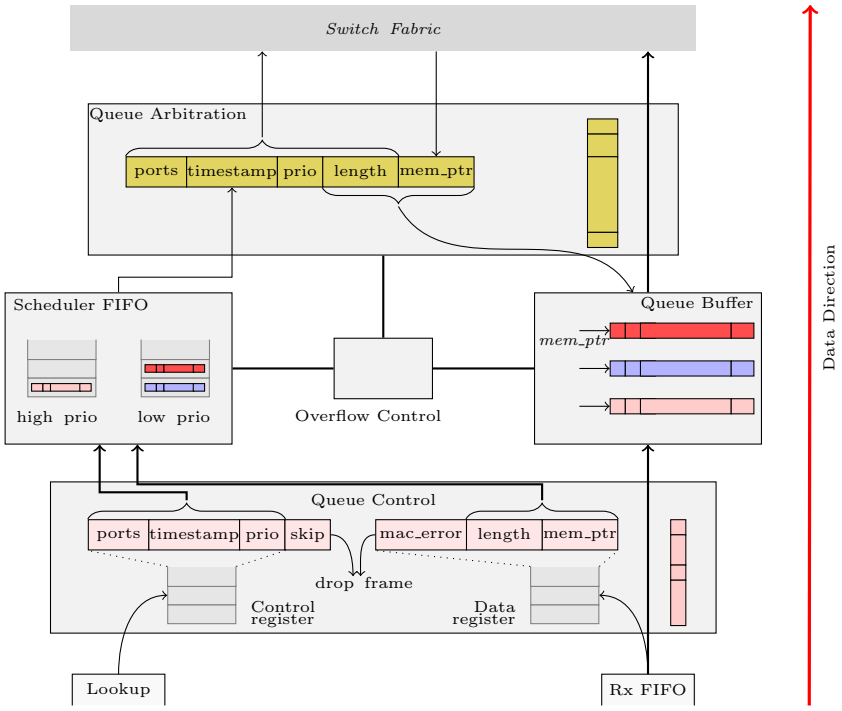


Figure 4.14: Input queue block diagram: each frame in the memory has a corresponding entry in one of the FIFOs.

The *queue arbitration* logic forms the interface to the switch fabric and constantly monitors the *scheduler FIFOs* for new entries. If an entry is available in any of them, a read is issued to the highest priority *scheduler FIFO* among the ones which have an entry. The arbitration logic requests access to the switch fabric for the destination ports, along with the priority and timestamp information. The switch fabric acknowledges the request allowing the data to be forwarded. In the case of multi-port forwarding, the switch fabric can issue a partial acknowledge for only a subset of the requested ports. For example, if the request was 0111 (i.e, access to ports 3, 2, and 1), the switch may approve only ports 3 and 1.

If at least one port is acknowledged, and no higher-priority frames have arrived at the *scheduler FIFO*, the arbiter initiates a read from the *queue buffer*. At the end of transmission, the arbitration logic updates its port request vector by setting the acknowledged port to '0' (i.e., in the above case, the vector now is reset to 0010), indicating that the frame still needs attention. If no higher priority frames are available, the updated port vector is presented to the switch to request

transmission to port 2. Once all ports have been acknowledged and transmission is complete, the vector is updated to 0000 and the arbiter waits for the next frame. However, if a higher priority frame becomes available in memory, the updated port vector (0010) and its corresponding *timestamp*, *prio*, *length*, and *mem_ptr* signals are buffered, and the higher priority frame is serviced first. Once all higher priority frames are served, the arbiter serves the buffered frame.

The *overflow control* logic monitors the *scheduler FIFOs* and the circular *queue buffer* for overflows. Since a frame's transmission could be deferred in case of higher priority traffic, it is possible for incoming frames to overwrite this data. The overflow control logic tracks such issues and drops stale frames, by removing buffered entries in the arbitration logic (or its corresponding entries in the FIFO), in favour of incoming data. To ensure that such cases are minimised in normal operation, the *queue buffer* is designed with sufficient depth to handle multiple outstanding frames while wider channels (4-bytes) are used at the switch interface compared to the byte-wide interface to the Rx FIFO.

4.6.2.2 Configurable Switch Interconnect

The central element of the gateway architecture is the configurable crossbar switching interconnect, allowing multiple ports to be active simultaneously. The switching infrastructure uses multiple priority schemes to handle the requirements of different automotive applications interconnected via the gateway. To achieve this, fabric arbitration modules are associated with each transmit port to determine connectivity based on the configuration and received data.

Figure 4.15 shows the architecture of our configurable switching interconnect. Each receive interface can be connected to any transmit path except itself. As soon as a frame is received at a receive interface, access to the corresponding transmit path is requested and the related arbitration module determines if the connection can be established based on the current state of the transmit path. If the transmit path can hold another frame, and no higher priority interfaces/data have requested access to this path, the arbitration module enables the connection

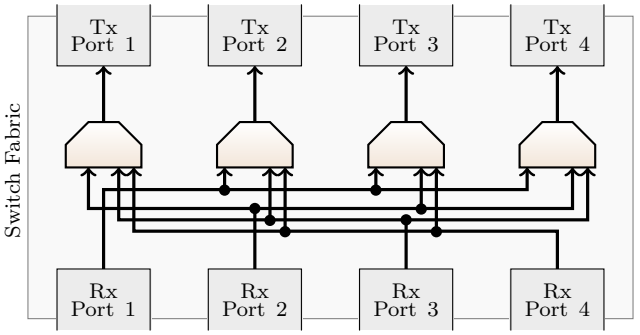


Figure 4.15: Crossbar matrix implementation of the switch fabric.

and initiates transfer of data, along with the status information of the frame, like *priority*, *timestamp* and *length*. Once the frame has been transferred, the arbitration module releases the connection and waits for the next request. To guarantee latencies we use a **strict latency** arbiter that selects the next port based on the priority value (*prio*) of the frame (range 0 to 7, 7 being highest) to be transmitted.

Integrating the fabric arbitration module into the switch logic, rather than the interfaces allows the design to be scaled more easily as the necessary paths can be scaled for any number of connections. For example, upgrading to a 6-port switch requires minor changes to the switch fabric and replication of transmit and receive interfaces.

4.6.2.3 Transmit Path

The transmit path receives frames from the switch fabric, buffers them and schedules the output messages according to their priority. The functionality is implemented in multiple sub-modules of the output *queue control* logic, which have similar functions to those in the receive path. These sub-modules *queue memory monitor*, *scheduler FIFOs*, *queue buffer*, and *queue arbitration* logic and their operations are shown in Figure 4.16

When a new frame is to be received into the transmit path from the switch logic, the *memory monitor* logic examines the output *queue buffer*, and acknowledges the transfer if the entire frame can be buffered. Depending on the priority of the

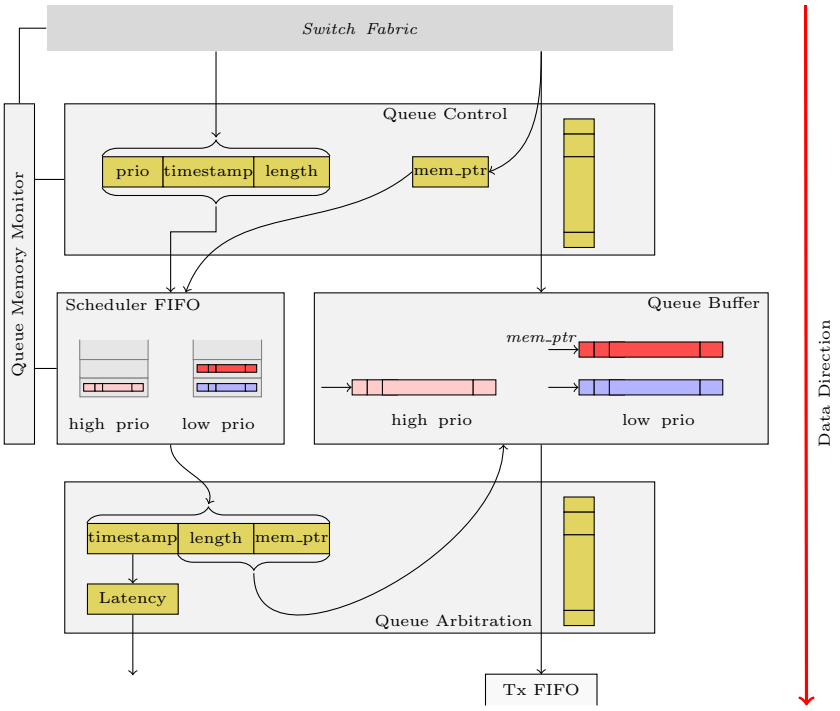


Figure 4.16: Block diagram of the output queue module and its sub-modules.

arriving frame (*prio*), the frame data and its control information are directed to the appropriate priority *queue buffer* and *scheduler FIFO*. The memory pointer corresponding to the first data word is appended to the received control information and stored in the corresponding priority FIFO, once a complete frame is received.

The output *queue arbitration* logic forms the interface to the physical transmit path and constantly monitors the *scheduler FIFOs* for available frames. When the Tx path MAC is ready to accept a frame and a frame is ready for transmission, the arbitration logic fetches the highest priority control information and the data corresponding to that entry from the corresponding output *queue buffer*. With the last data word, the *queue arbitration* logic signals an end-of-frame to the MAC using the *last_word* signal, and starts arbitration for the next frame. The MAC then performs the required protocol operations and pushes the data to the PHY to drive the physical network.

The *queue arbitration* logic takes a timestamp corresponding to the first data write

to the Tx FIFO (MAC), which is the egress timestamp. The difference between this and the corresponding ingress timestamp (which is part of the frame control information) is the path latency, which is passed to the AEG monitoring module for software-level monitoring.

4.6.2.4 Translation for FlexRay/CAN Systems

Unlike Ethernet, which uses explicit addresses to identify destination and source nodes, automotive networks are based on a broadcast scheme with no explicit identification for sources/sinks. The priority of messages in CAN and assigned transmission slots in FlexRay are statically defined parameters that can implicitly identify a source node. Further, a translation scheme is required to determine the destination port on the switch interface and to manage message mapping due to different payload sizes and priorities.

The *tblock* handles this message translation from the switch to CAN/FlexRay networks; it includes

- An address mapping scheme that relates the implicit identifiers on CAN or FlexRay to destination ports/addresses on the AEG.
- A data-packing scheme that respects the deadlines and payload sizes on the CAN/FlexRay networks for mapping messages.
- A stream-based data interface to the Tx/Rx path of the AEG from the CAN/FlexRay Communication Controllers.

We present the approach for FlexRay, which can be modified for application to CAN networks.

Mapping Logic: For the FlexRay static segment, a purely time-triggered scheme is used where messages are assigned fixed slot(s) in every FlexRay cycle. This means that a receiving ECU can subscribe to a set of slots on which messages are scheduled, creating an implicit addressing scheme.

Policy-based scheduling [188] provides a mechanism to translate this implicit addressing scheme, and is employed in our gateway. It allows packing of event-triggered messages into time-triggered FlexRay slots, with consideration for their real-time deadlines.

As with standard FlexRay, messages are analysed at design time when generating the message schedule for the FlexRay network. This schedule is used by all participating nodes on the FlexRay network, including the AEG's FlexRay interface. The static schedule assigns transmission slots to all nodes but allows normal FlexRay interfaces to subscribe to pre-determined slots to receive messages from the network.

However, unlike standard FlexRay nodes, interfaces which rely on policy-based scheduling subscribe to all slots assigned to the gateway. This allows priority-based packing of messages, whereby any received message can be transmitted in the next available transmit slot assigned to the transmitter (not limited to the slot reserved for the message by the static schedule). Additional information like transmitter/receiver identifiers, packet sizes, and sequence numbers are incorporated in the data-header of the message. Nodes participating in policy-based scheduling filter received messages based on this data-header to determine if they are the intended recipients. Since the data-header is inserted within the payload segment and communication is performed over statically determined schedules, policy-based nodes can coexist with conventional FlexRay nodes on the same network. The policy-based schedule is enabled only at those nodes which require communication to cross to another domain via the AEG. These nodes are assigned a virtual address at the AEG, allowing them to be addressed from devices on other network domains.

The gateway's FlexRay interface integrates onto the FlexRay network for that branch, and is assigned transmission slots like all other interfaces. When a message in the switch is to be transmitted to the FlexRay network, it is scheduled in the next slot assigned to the gateway. A data-header is added to the message, allowing it to be identified at the destination nodes, either in software or using hardware

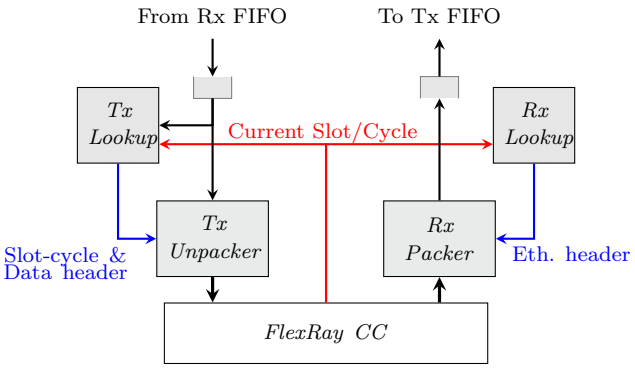


Figure 4.17: Architecture of tblock and its integration with the FlexRay communication controller (CC).

extensions. The receiver identifiers in the data-header are determined at design time and preloaded into a lookup memory in the *tblock*, while the packet size and sequence numbers are generated dynamically based on message received from the switch.

When a message from the FlexRay network is to be forwarded to a node on the Ethernet link, the FlexRay interface of the gateway receives the message from the slot and decodes its data-header. The *tblock* maps the message to an Ethernet frame, with the source address as the virtual address of the transmitting ECU and destination address determined by the identifier in the data-header.

Architecture of *tblock*: The *tblock* comprises of *tx lookup* and *rx lookup* lookup memories, an *rx_packer* module to handle FlexRay → Ethernet messages, and a *tx_unpacker* module to handle Ethernet → FlexRay messages, as shown in Fig. 4.17. The virtual MAC addresses of FlexRay nodes are linked to the FlexRay slot-cycle identifier(s), that are statically defined by the schedule. The FlexRay parameter *KeySlotID*, which is a unique slot assigned to each node by the schedule, is used as the transmit/receive identifier for the data-headers. These mappings are preloaded into the tx and rx lookup memories of the *tblock*.

When a frame is received from the switch for forwarding on the FlexRay network, the *tblock* buffers the frame and strips the Ethernet headers. The *tblock* performs a lookup to determine the corresponding data-header and appends it to the frame data. The list of transmit slots assigned to the AEG's FlexRay interface is then

sorted based on the current slot-cycle in progress on the network. The frame along with the sorted list is forwarded to the *tx_unpacker* module, which then segments the data (if needed), adds FlexRay frame headers and writes to the FlexRay interface's frame buffers.

On the receive side, the messages received from the FlexRay network are handled by the *rx_packer* module. At every slot/cycle boundary, the *tblock* performs a prefetch to determine if there are any Ethernet addresses mapped to the slot-cycle combination that has just ended. Since the FlexRay protocol requires the frame to be completely received before validating it, the prefetch operation enables *rx_packer* to prepare the Ethernet container before the FlexRay frame is completely received from the network. The FlexRay payload is directly filled into the Ethernet container and presented to the Rx Path of the port. The *rx_packer* also appends the receive packet with '0's if the FlexRay frame does not fit the minimum payload size for Ethernet.

We have integrated our extensible CC architecture as the FlexRay interface of the AEG. The host interface of the FlexRay CC has been altered to a streaming interface to the *tblock*, based on the AMBA Advanced eXtensible Interface (AXI) streaming interface standard. FIFOs are instantiated at the interface to decouple the *tblock* from the CC's clock and data rates. The configuration of the protocol parameters is now integrated using a state machine that interfaces over a separate AXI bus. The isolation of the configuration and datapath interfaces enable a generic *tblock* architecture that can be directly reused for other automotive protocols like CAN.

4.6.2.5 Run-time Management of Interface Configurations

The configuration space of the AEG is mapped as an addressable location from the ARM core on the Zynq device. The mapping allows configuration of each individual port to be altered in isolation or to update all tables in a single write operation. This includes routing paths, message priorities as well as the behaviour of the switching system. The lookup memory within the *tblock* is also mapped

to the address space, allowing changes to be made in the FlexRay/CAN message routing. In addition, the software on Zynq's ARM core can also monitor the latency of packets on the individual interfaces to further fine-tune the routing performance and to isolate paths in real-time.

4.6.3 Evaluating the AEG on Zynq

To evaluate the switching performance of our AEG, we have implemented the architecture on both the ZC702 development board and the ZC706 board featuring two different Zynq devices. The ZC702 board features an smaller XC7020 device that incorporates an Artix-7 grade fabric, while the ZC706 board features a XC7045 device that incorporates a superior ARM core and a Kintex-7 grade fabric. For the evaluation, we have chosen a design that incorporates 3 Ethernet ports and 1 FlexRay port. The I/Os of the Ethernet ports are directed to the FPGA Mezzanine Card (FMC) interface, on which we have used two Gigabit Ethernet FMC cards (FMCL-GLAN-B) from *Inrevium Inc.* to provide physical connectivity. The resource utilisation on the ZC7020 device is as shown in Table 4.7. As can be observed, the AEG consumes 55% of the resources on a small device, and can easily be extended to support more ports. On the larger ZC7045 device, the maximum resources consumed were under 15% (BRAMs) for the same combination. The superior PL fabric on the ZC7045 offers higher performance and is thus a better implementation platform for supporting more interfaces (ports).

To observe the end-to-end latencies of the AEG under cross-traffic and isolated traffic conditions, we use a test setup with 3 Ethernet links and one FlexRay link on the larger XC7045 device (ZC706). The Ethernet ports are connected to independent and isolated traffic sources and sinks which are implemented on the AC701 and VC707 development boards. Each link is capable of handling traffic at 1 Gbps. For FlexRay, we have integrated a small cluster of nodes on the ZC706 board that

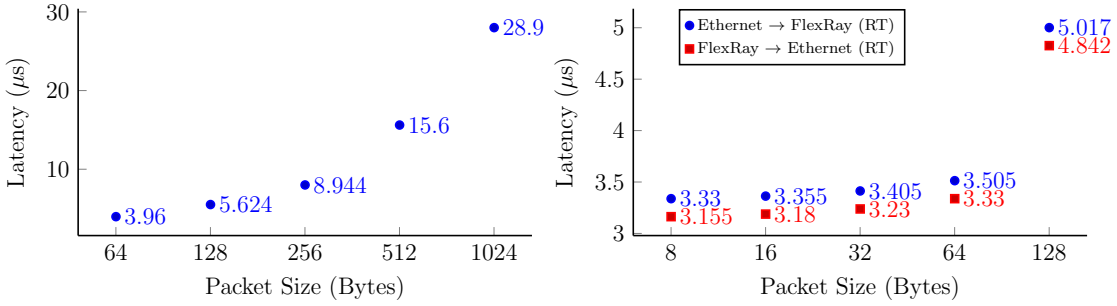
Table 4.7: AEG: Resource Consumption on Zynq XC7020.

Function	Submodule	FFs	LUTs	BRAMs	DSPs
FlexRay Port	CC	5572	9768	20	2
	Tblock	2227	1849	13	0
	Rx_Path	662	492	4	0
	Tx_Path	160	121	5	0
	Total	8576	12230	42	2
Ethernet Ports ×3	MAC	2619	1851	1	0
	Rx_Path	661	476	4	0
	Tx_Path	160	121	5	0
	Total	3549	2559	10	0
Switch	-	204	856	0	0
Total (%)		19585 (18.4)	21095 (39.7)	72 (54.75)	2 (0)

also includes the FlexRay interface of the AEG. This allows us to measure end-to-end latencies in all possible combinations: Real-time network↔Non-real-time network, Non-real-time↔Non-real-time and Real-time (FlexRay/Ethernet)→Real-time (Ethernet)←Non-real-time cross traffic.

Figure 4.18 shows the average latency incurred by the AEG for different data sizes in the absence of cross traffic for non-real time Ethernet↔Ethernet traffic (Figure 4.18(a)) and real-time FlexRay↔Ethernet traffic (Figure 4.18(b)). The latency is computed end-to-end; from the start of frame transmission at the transmitter to the frame header reception at the receiver. For larger data sizes, we see an increase in latency due to the increased data movement within the switch, the transmit and receive interfaces of the gateway and at the source and sink interfaces. In the absence of cross traffic, we observe that the maximum variation in latency is about 40 ns and is insignificant compared to the end-to-end latency values.

For Ethernet→FlexRay transfers (Figure 4.18(b)), the measurement is terminated at the FlexRay interface, since the transmission of the message on the FlexRay network is guaranteed by the policy-based schedule. It can be observed that



(a) Ethernet ↔ Ethernet. (b) Ethernet ↔ FlexRay.

Figure 4.18: Switching bandwidth of AEG for different data-sizes in absence of cross traffic: Plot (a) corresponds to non-real-time Ethernet↔Ethernet traffic, while plot (b) corresponds to real-time Ethernet↔FlexRay traffic.

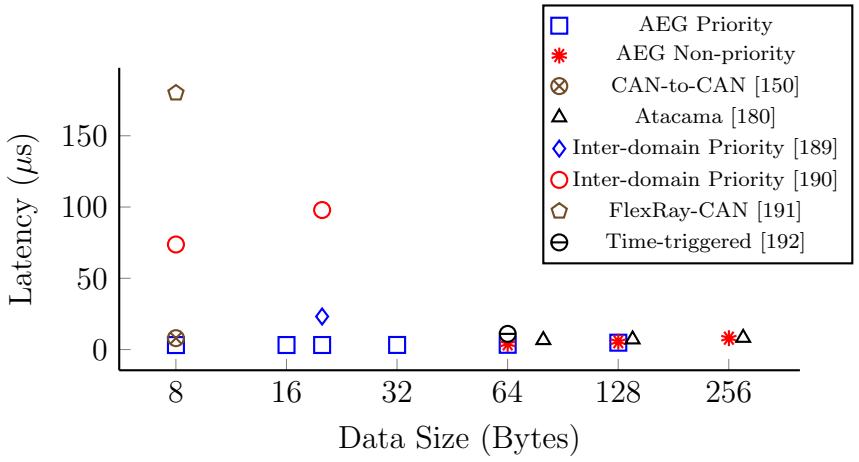


Figure 4.19: Comparison of end-to-end latencies of our AEG with other implementations from literature.

there is no appreciable variation in latency below the 64-byte frame size, since the *tblock* pads smaller frames to meet the minimal Ethernet frame size requirement. For 128-byte data, we observe a slightly increased latency due to the larger data size. For data packets forwarded from FlexRay to an Ethernet link, the prefetch mechanism helps to reduce lookup latencies compared to the Ethernet→FlexRay transfers.

Figure 4.19 shows worst case AEG latency compared with existing work in the literature, in the absence of cross traffic. As observed, our architecture outperforms gateway structures based on software-based approaches (Lim *et al.* [189] {simulation}, Kim *et al.* [190], Yang *et al.* [191]), and and Müller *et al.* [192]) and

FPGA-based gateways for traditional networks (Sander *et al.* [150]). Our architecture also outperforms FPGA-based Ethernet switching infrastructure Atacama (Carvajal *et al.* [180]), though the margin is small (1.3× lower latency at 128-byte priority data).

We also evaluate the performance of the AEG in the presence of cross-traffic. For this evaluation, priority and non-priority traffic were directed to the same destination at an aggregate bandwidth that nearly saturates the AEG. The setup generates non-priority traffic at 600 Mbits/s with a 1 KB payload size and variable rate priority frame (10-200 Mbits/s) with a 64-byte payload. Fig. 4.20 shows the variation in latency in the presence of cross traffic, measured over long duration. It can be observed that additional (and varying) latency is incurred in the case of priority frames compared the fixed deterministic latency in the absence of cross-traffic. This is due to the non-preemptive nature of the switch fabric that blocks the priority frame once a non-priority frame has entered the switch, resulting in a maximum end-to-end latency of 21.3 µs for priority data. In the case of non-priority traffic, the smaller size of the priority frame causes only a minor increase in end-to-end latency as the blocking period (due to the priority frame) is much less than the transmission latency of the non-priority frames. When the rate of the priority frame reaches 200 Mbits/s, we observe that the non-priority frames start accumulating within the Rx port buffers, which eventually leads to dropped frames. However, in the same conditions, the priority frames were routed without any data loss, ensuring that critical data is always delivered to the destination. In comparison, the Atacama switch achieves better performance in cross-traffic conditions due to its dedicated routing structure for priority traffic, at the expense of increased resource consumption and poor scalability.

The AEG, with the dedicated hardware paths for switching and software-based monitoring and control, offers the low latency message switching required for next-generation high-performance vehicular interconnect. The modular implementation also allows parametric modification for different network architectures, without sacrificing performance.

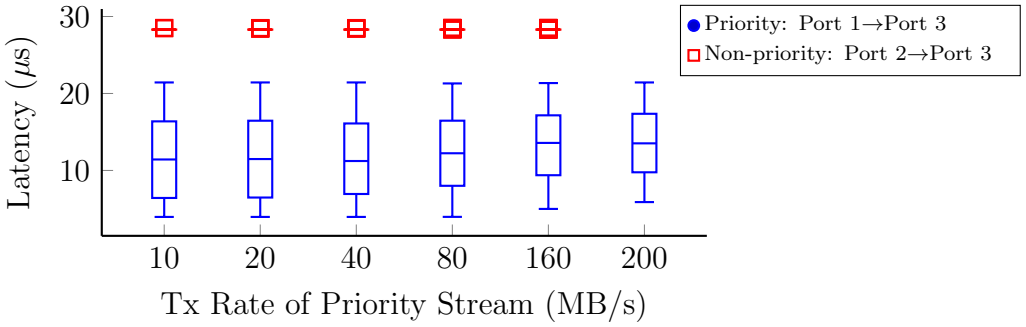


Figure 4.20: End-to-end latency measurements of our AEG in the presence of cross-traffic.

4.7 Summary

In this chapter, we have described enhanced architecture models for next-generation in-vehicle embedded computing units based on reconfigurable hardware, which can inherently support function-level consolidation as well as fault-tolerance mechanisms. These capabilities are enabled by the tight coupling between computing units and the extended network interface, and are enhanced with high-speed reconfiguration capabilities. Controller data-path extensions provide unique mechanisms to integrate fault-monitoring and mode-switch commands in existing traffic. Abstracting the reconfigurability and extended communication from the application layer allows seamless consolidation of functions, and integration of fault-tolerant modes without adding software complexity.

We have also described a gateway architecture for evolving automotive Ethernet standards that is modular, scalable, and customisable, while providing software-based flow control, monitoring, and priority-based message routing, all without a significant impact on message latency. Integration of these extended functionalities at the ECU is possible due to the capabilities of reconfigurable hardware is not possible using existing MCU-based ECU systems. With FPGAs becoming more popular for in-vehicle compute-intensive tasks, enhanced ECU architectures on FPGAs will find more adoption in future in-vehicle infrastructure.

5

Securing Vehicular Networks

5.1 Introduction

Beyond the challenges posed by the increasing number of ECUs and the complexity of the distributed architecture in modern vehicles, another major concern is the potential drawbacks of increased automation. As mechanical systems are increasingly being replacing by electronic equivalents, and the focus on reliability and performance drives adoption of established standards, this leaves systems susceptible to possible harm against which those networks were never protected. With the emergence of external wireless access to vehicle systems, security is becoming a major concern due to the potential harm that can be caused through malicious attacks.

Researchers have demonstrated the susceptibility of present automotive networks to remote attacks [153, 193, 155, 194]. With high levels of automation in modern vehicles, an attacker can gain control over critical systems that control vehicle dynamics using multiple attack vectors. These attack vectors can range from simple techniques like *network fuzzing* and *replay attacks* to more complex ones which could alter the software code on an ECU at run-time. Fuzzing attacks involve injection of commands/data on the network by an attacker ECU, either with prior knowledge of these command/data combinations or as a brute-force mechanism. The attack aims to force one or more functional ECUs into certain vulnerable modes (such as to enabling flashing), to produce a specific response (like activating the brakes) or in the easiest case, to cause them to halt due to unexpected command-data combinations. A replay attack is another simple mechanism that involves capturing network messages from an active scenario and reusing them at a later time to spoof commands to an ECU.

Early vehicular networks were designed to be closed systems, with each ECU connected to its required network through a network interface. On such vehicles, attackers implemented compromises by tampering with an existing ECU, supplying infected after-market devices, or through the on-board diagnostics (OBD) port. OBD ports are designed to allow service personnel to debug system faults through a single access point, and thus provide direct and/or bridged access to both critical and non-critical networks, representing an ideal point for a hacker to gain access via an (infected) OBD dongle. Malicious software or hardware provides another pathway, mostly introduced through unapproved after-market upgrades, and can be used to launch internal attacks (observations or manipulations) on messages or other ECUs since the network provides implicit full bus access to all components. Once access to the network is gained, it might be possible to install defective software on safety-critical ECUs over the network, compromising their functionality, as was demonstrated in [153, 193, 195].

An example case was demonstrated in the remote hack of the Jeep Cherokee, where hackers used wireless access to install infected software on a non-critical ECU on the CAN network [195]. The infected ECU enabled them to remotely control

the braking, acceleration, steering, and engine performance while the vehicle was being driven manually. These attacks are possible because of the lack of standard mechanisms in current automotive network standards to verify the authenticity and timing of a message, the broadcast bus nature, and lack of authorisation schemes for ECUs.

Connectivity within the car and to the outside world has increased over the last decade offering new pathways to the attacker. Wireless access for entertainment and other core functions is common in many modern high-end vehicles. With technologies like Vehicle to Vehicle (V2V) communication on the horizon, a single infected vehicle might jeopardise the safety of other vehicles on the road by presenting malicious information to them. Also, attackers may exploit such links to spread compromised data/software to multiple vehicles during such data exchange. Network standards, particularly in-vehicle standards, do not support mechanisms for resisting these attack vectors. Though techniques have been described to limit such attacks at the application layer, either in software or using hardware security modules, these are still inefficient because of the of the logical separation between computation in the ECU, and physical bus access in the network controller. Since communication on the network is managed entirely by the network controller, it would be beneficial to implement security measures at the network interface. This scheme allows properties of the network protocol to be exploited in adding security features. Building such extensions in higher layers is challenging since this involves increased software complexity as well as the requirement for the application to be aware of low-level network details.

We have already shown that extended communication can be made possible using datapath extensions within a network interface. We also saw that such extensions can enable unique ways to enhance the overall capabilities of an ECU, on a supportive architecture. We now look at mechanisms to incorporate network and system-level security within the network interface, by further extending the datapath. This presents numerous advantages, most important of which is to effectively hide the latency incurred by the security mechanism from the network and application, creating effectively zero-latency overhead. We further extended

the scheme for limiting network access for authorised devices through obfuscating network-level information in messages. Finally, we present an architecture that prevents compromised ECUs from accessing the network.

The work presented in this chapter has also been discussed in:

1. S. Shreejith, S. A. Fahmy, *Zero Latency Encryption with FPGAs for Secure Time-Triggered Automotive Networks*, in Proceedings of the International Conference on Field Programmable Technology (FPT), Shanghai, China, December 2014, pp. 256-259 [17].

2. S. Shreejith, S. A. Fahmy, *Security Aware Network Controllers for Next Generation Automotive Embedded System*, in Proceedings of the Design Automation Conference (DAC), San Francisco, USA, June 2015, pp. 39:1–39:6 [19]

5.2 Related Work

As we have seen, modern vehicles employ multiple network protocols which support the requirements for the different functions. Vehicular networks can hence be physically separated into high and low performance networks, based on factors like criticality of functions, latency requirements, and communication bandwidth, and are typically bridged via a central gateway. High performance networks link together safety-critical ECUs and sensors, like engine control and drive-by-wire systems, while low-performance networks connect non-critical ECUs like window controls and door locks. Typically, protocols like high-speed CAN (HS-CAN) and FlexRay are employed for high performance networks and low-speed CAN (LS-CAN) and local interconnect network (LIN) are used for low performance networks, each providing the required bandwidth and reliability guarantees necessary for the respective applications. However, telematics and driver assistance systems often require information from both networks, and are often connected directly to both network classes, creating unwanted bridging [153, 193].

The security of in-vehicle communication (IVC) has recently been subject to significant investigation. In [153, 193], the authors analyse the security of computation and communication systems by exploring different attack vectors on production vehicles. The authors present some of the common weaknesses in vehicular architecture, that enable attackers to gain access into systems with relative ease. Though automotive standards are in place that place restrictions on architecture and software management, their analysis and experiments on vehicles show that such restrictions are often not followed for performance and practicality reasons. As an example, the standards forbid unwanted bridging between high and low performance buses, and thus any ECU that requires access to both networks must achieve this through a gateway. However, in many cases, telematics systems are directly connected to the high and low performance networks, creating a bridge. They show that with this bridge in place, they could alter commands on the high performance networks using an infected device on the low performance network. They also highlighted another issue whereby an ECU on the higher priority network could be forced to accept a configuration request from an ECU on the low priority network. This presented them with the opportunity to alter the software on the critical ECU and to infect it (and the associated network), through an infected device on the low priority network. Such security flaws enable attackers to potentially gain access to critical functions while the vehicle is in operation, using simple attack vectors and an infected after-market device.

While the experimental approach exposed many practical attack possibilities, analytical evaluation of threats and effects has also been explored [155, 194, 196]. In [155], the authors present an analysis of the different threat models and their criticality. They use multiple factors like severity of the threat, success probability of the attack and others to classify threat models. Their results also show that many threats critical to the safety of the occupants can be achieved with relatively low effort and high success rate. In [194], the authors classify ECUs based on safety effect levels, based on similar analysis of the threat models. In [196], the authors extend threat models to incorporate wireless interfaces which are widely

popular in modern vehicles. They show that availability of wireless interfaces further enhances the attack planes, with potential attackers not requiring physical access to the ECUs or networks. In [197], the authors present an approach to analyse and detect system-level security flaws using probabilistic model checking. The analysis is performed by evaluating confidentiality, integrity and availability of different architecture variants to determine the least vulnerable variant.

The *EVITA* project focuses on evaluating the security of vehicular systems, focusing on future V2V and the enabled-vehicles concept. The security research under *EVITA* evaluates techniques like formal verification for detecting and evaluating threat models for over-the-air services like software updates and connected services [198]. Another research direction under the same project explores the possibilities of utilising efficient hardware-software co-design for defending against attacks in connected-vehicles communication systems like Car to X (C2X) [199].

Techniques have also been proposed to address these challenges to some extent. Standard methods like cryptography and anomaly detection were proposed in [196] to provide data security. The proposed mechanism incorporates an application layer approach: however, this incurs considerable latency and the network is still susceptible to replay attacks. A scheme based on trust and access control lists to verify message authenticity on CAN-based ECU systems was proposed in [156]. This is also an application layer approach where filters enable messages to be marked as trusted for processing by the application. This method provides a higher level of security, but does not prevent an unauthorised device (either newly plugged in or compromised) from accessing network traffic. Software based automotive security solutions are also proposed in [200]. Though functional, the software-based approach incurs a significant latency overhead for implementing even simple cryptographic functions. Moreover, mechanisms must be in place to protect this software against tampering and manipulation from invasive and internal attacks.

Alternatively, automotive hardware-based security modules (HSMs) and secure hardware extensions (SHEs) have been proposed, providing high levels of tamper protection for V2V and IVC [201]. Such HSMs are attached as co-processors to

the standard MCUs in the ECU and since these are dedicated hardware blocks, they do not need to be protected from manipulations. HSMs allow acceleration of cryptographic functionality and take the computational load away from the software and application. However, the application must still be aware of the HSM – the datapath from the network to the application must be explicitly handled by the software. Furthermore, since security is invoked at a higher level (i.e., after the messages have been received), the latency incurred in the operation is not transparent to the application or to the network. Large latency changes can require extensive rescheduling of the communication network and the application tasks to ensure that the deadlines are met.

Our method aims to integrate security and network protection closer to the network and in a transparent manner. Since security standards need to be adapted multiple times over the life of an ECU (typically more than 10 years), reconfigurable hardware is ideally suited for the task.

5.3 Contributions

We present an approach for embedding cryptographic functions at the network layer, in a manner that is transparent to the ECU, applications, the network. This creates a layer of security for messages that are otherwise exchanged over the network in plaintext. To achieve this, we extend our enhanced FlexRay CC's datapath further, and add a layer of cryptographic functions that is completely managed within the CC. Integrating a cipher block within the network interface provides an opportunity to enhance the cryptographic properties by exploiting the time-triggered features of the network. Further, the operation of the cipher is completely overlapped with the data movement and transmission/reception, effectively creating a zero-latency scheme as far as the network and the application are concerned. The difference between our proposed mechanism and the HSMs/SHE-based schemes are the zero-latency aspect and complete transparency

to the application. The cipher operations are self-organising and leverage existing extensions in our CC to offer higher entropy in the ciphertext.

We further extend this message security in a system-level scheme that ensures authenticity of the application running on the ECU before allowing network communication and allows only authorised devices to integrate and communicate over the network. These features are achieved in a manner that is transparent to the application and software layers. The proposed method does not impact the real-time characteristics of the network nor require re-scheduling of communication, as in case of HSMs.

We demonstrate our security approach on a Xilinx Zynq platform, while it is also applicable to our enhanced ECU architectures discussed earlier. The proposed method establishes a configurable hardware security layer, which can be built upon to provide adaptable security schemes.

5.4 Security-Enhanced Network Interfaces: Enabling Zero Latency Message Ciphers

The key weakness of existing automotive network protocols (and hence controllers) is that no standard mechanism is available to verify the authenticity and timing of a message. However, as seen in Chapter 3, such functionality can be integrated and verified at the network layer, without intervention of the processing function. Time-triggered networks, that are standard for safety-critical systems already offer a synchronised view of time at different nodes in the network, and this can be leveraged to secure communication. If messages can be augmented with a timestamp that identifies when the message was sent, this could allow stale data to be rejected. Adding cryptographic elements over this would allow an ECU to be certain of the source of the message.

Our approach enhances the communication controller (CC) by integrating a data cipher into the architecture, as shown in Figure 5.1, thus altering the normal datapath (marked **1**), creating the enhanced datapath (marked **2**). Reformatting a data stream to include a timestamp introduces additional entropy to otherwise static data, as proposed in [202] for TCP frames. These extensions are overlapped with standard functions within the CC by extending the datapath using double buffering, prefetching, pipelining, and wide interconnect. Though such extensions could be implemented in software running on the ECU processor, this would result in significant overheads in computation and synchronisation, as well as large variations between the actual time of transmission and the timestamp encoded in the messages.

For the cryptographic functions, we use a low-latency PRESENT cipher [203] that can meet the real-time and power/computational constraints imposed by automotive ECUs. The PRESENT cipher offers one-round-per-clock operation and has 2.5× lower resource requirements than complex ciphers like Advanced Encryption Standard (AES), allowing it to be efficiently implemented for each channel in isolation, as required in case of dual-channel networks like FlexRay. PRESENT is also an internationally accepted standard for lightweight cryptography by the International Standards Organisation and the International Electrotechnical Commission (ISO/IEC). Though our experiments use PRESENT over a FlexRay network, the same principles can be used with other low-latency ciphers and/or time-triggered networks.

5.4.1 Security Extensions in the Enhanced FlexRay Communication Controller

Figure 5.2 shows the datapath extensions that implement timestamping and timestamp synchronisation at the interface, integrated into the MIL module of our secure CC. The normal dataflow path in a standard controller design is marked as **NP** in Figure 5.2, where the data from the processing logic flows up/down from/to the bitwise decoding/encoding blocks. With our integrated timestamp block, the

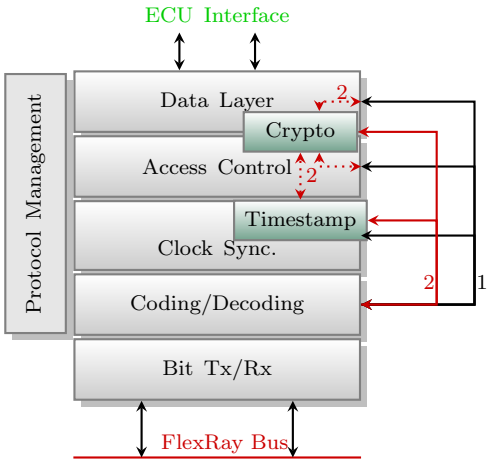


Figure 5.1: Time-triggered Network enhanced with an Intermediate Timestamp and Data-ciphers.

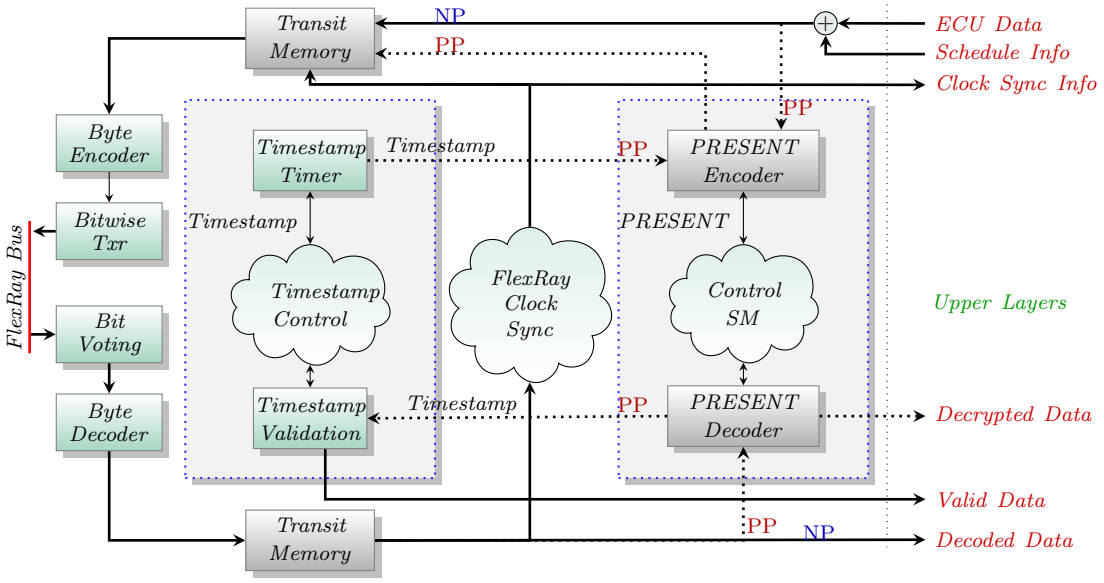


Figure 5.2: Datapath Extensions: Timestamp synchronisation, Processing and Cipher logic.

communication controller establishes a common time-base using techniques defined by the FlexRay protocol during the initialisation phase of the network. A leading coldstart node initialises the start-up procedure by transmitting a start-up frame and waiting for other coldstart nodes to join in. Once more than two coldstart nodes integrate, non-coldstart nodes (i.e., the ones which cannot trigger the start-up procedure) adopt this schedule and integrate onto the network completing the start-up procedure. The timestamp start-up procedure closely follows the

FlexRay clock synchronisation scheme (see Section 3.4), thus providing a common time-base to all participating nodes.

On the transmit path, the timestamp insertion logic appends timestamp information into the outgoing data, just before it is encoded into the byte-packed format for transmission, all within the CC. Alternatively, if the function is disabled, the ECU processor would instead read the timestamp information and append it to the sensor data before passing it on to the CC for transmission. Similarly, on the receive path, the timestamp processing unit extracts the timestamp information from the decoded data and can be configured to reject incoming data if the timestamp is not current. Alternatively, it can forward the entire packet to the ECU (as in the normal case), where the ECU would compare its timestamp to the current time (from the CC) and decide on its validity.

The timestamped data is encrypted by the cryptographic block using a pre-shared key, then encoded and transmitted bit-by-bit, as per the FlexRay standard. In the receive path, the bits from the physical medium are voted and decoded as per the FlexRay protocol. The cryptographic block deciphers the received data and extracts the timestamp, which is then validated, with the plain text data forwarded upstream. This *cipher enhanced datapath* is shown in Figure 5.2 marked as **PP**.

The PRESENT module uses multiple iterations (or rounds) of substitution and permutation operations to convert an incoming data block into ciphertext, with each round is controlled by a *round key*. The cipher operates on a block of 64 bits and can be configured to have 80 or 128 bit keys and can have up to 65536 rounds per cycle. These options can be changed by the ECU during operation.

The PRESENT module instantiates one buffer each in the transmit and receive direction to store the (prefetched) data from/to the upper layers. A 64-bit interface is used to ensure that a block is transferred in every cycle. Once a block of data is available in the buffer, it is read out in one cycle and further encrypted in n cycles, where n is a configurable number of rounds. A key store manages the round keys for each round, and handles the commands to alter cipher properties (no. of rounds, keys) by regenerating the new set of keys in the background using a double

buffer. The pipelined operation enables the encryption/decryption to overlap with transmission/reception of previous/subsequent data blocks (or headers), effectively hiding the latency in the buffering latency that is present even with no encryption.

5.4.2 Encryption in Software

When a powerful computational core (or co-processor) is available in the ECU, data encryption can be carried out within the ECU instead. In such a case, the current time value is read from the CC registers and is used to encode the data by using round-keys generated in the same manner. In the receive path, the decoded timestamp value from the header segment is read into the ECU to decode the data. However, we observe that the timestamps are inaccurate as there is no synchronisation between the communication schedule and task schedule on the ECU.

5.4.3 Evaluation of Security-Enhanced Network Interface

Table 5.1: Comparison of CC resources on XC6VLX240T.

Function	Submodule	FFs	LUTs	BRAMs	DSPs
normal CC	PMM	223	542	0	0
	CS	1862	3551	2	1
	MIL	1038	1610	2	0
	CHI	1456	1788	10	1
	Total	5617	9068	16	2
secure CC	PMM	222	515	0	0
	CS	1846	3430	2	1
	PRESENT	2548	2274	4	0
	MIL	1275	1463	3	0
	CHI	1629	2136	10	1
	Total	11224	13386	26	2
Overhead (%)		5607 (99.8%)	4318 (47.62%)	10 (62.5%)	0 (0%)

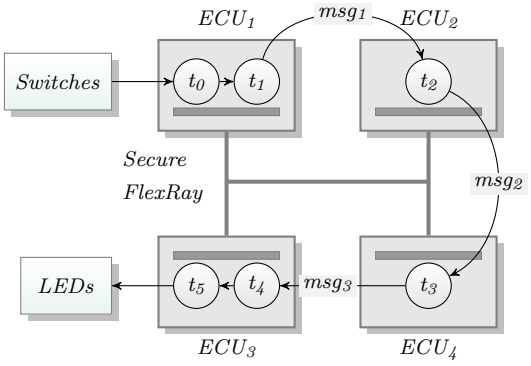


Figure 5.3: Test setup with 4 ECUs on ML-605 development board.

To evaluate the proposed scheme, we have integrated four ECUs on a single Xilinx Virtex-6 LX240T FPGA (Xilinx ML-605 development board), connected together using a FlexRay bus channel which is completely contained within the FPGA, replicating an emulated cluster of ECUs. Each ECU in Figure 5.3 is served by a local clock and comprises a MicroBlaze softcore processor (each slightly differing in memory configuration) integrated with our secure FlexRay CC. These run a set of tasks that generate incremental data patterns every 100 milliseconds and exchange data in every cycle of duration 10 milliseconds. The enhanced CC consumes up to 86% of the resources in each ECU, with the test setup consuming 44% of the Virtex-6 device, and it can be clocked up to 124 MHz, more than the 80 MHz required by FlexRay. Table 5.1 describes the utilisation of the secure CC in more detail, comparing it against a standard implementation without enhancements. It can be observed that including the PRESENT logic within the controller results in higher resource consumption and marginally higher power consumption than the standard core.

We also evaluate the cost of performing encryption in software as opposed to the integrated approach within the CC. For this, a C implementation of PRESENT is used on the MicroBlaze ECUs. We also evaluated the same on an ARM-based ECU using the Xilinx Zynq platform. The results are shown in Table 5.2. We observe that the integrated approach (in CC) does not incur any additional latency in the transmit or receive direction for handling sensor data encryption,

Table 5.2: Latency of operation per 64 bit data block.

Impli. (Rounds, Clock)	Encryption(μs)	Decryption(μs)
HW (up to 470, 80 MHz)	w/in Txn boundary	w/in Rxn boundary
SW MB (32, 100 MHz)	7204	7450
SW ARM (32, 667 MHz)	40.9	42.1

since encryption/decryption are performed during buffering. Meanwhile the software versions require 41 µs and 7 ms on the ARM and MicroBlaze respectively for every 64 bits of data to be encoded or decoded. This delay could be reduced by offloading encryption to a dedicated co-processor like custom IP on the MicroBlaze or the Neon engine on the Zynq. The software implementations also rely on additional data exchange (timestamp) between the ECU and CC which results in some inaccuracy, with a worst case slack of a complete network cycle.

5.5 Security-Aware Network Interfaces: Integrating System-level Security

Beyond data security, the ECU should also be protected from attacks that aim to alter the software or application, either permanently or temporarily. Also, though data is secured with the zero-latency approach, it doesn't prevent an unauthorised device from accessing the network or tampering with the messages. Prolonged access to bus data would allow hackers to crack the encryption scheme or to determine the cipher behaviour for employing replay attacks. Thus it is essential to ensure that an unauthorised commercial off-the-shelf device cannot integrate and observe the communication on a secure network.

It is also required to ensure that an ECU authorised to access the network executes only an authentic application. The network authorisation mechanism would fail to serve the purpose if an attacker can successfully inject malicious code into an authorised ECU. A combined mechanism is thus required to support both network authorisation and authentication of application code.

In present secure ECUs, an HSM unit attached to the ECU as a processor extension monitors the authenticity of the application code and network data. However, HSMs incur some latency when handling network tasks which must be accounted for in the message schedule. The broadcast bus structure also makes the HSMs ineffective in handling network level access control and authorisation. Our method proposes integration of both (application) authentication and (network) authorisation within the enhanced network interface (NI) that tightly integrates security within and around the communication controller (CC).

To authenticate the application, an agent external to the ECU must be involved, like the HSM unit which is attached as a co-processor. We integrate this functionality within the custom NI. It reads the contents of the boot-ROM and verifies the authenticity of the contents using a one-way hash function whose expected value is embedded in the hardware at production. If the authenticity is verified, the authorisation to use this hardware by the application is verified using a unique identifier. This hardware identifier is generated at run-time by a circuit that maps intrinsic properties of the device to its unique identifier. Since intrinsic properties cannot be controlled during manufacturing and are difficult to predict, two identical devices will generate different identifiers using such circuits, providing a unique signature. Only after this combined authentication and authorisation step will the CC (also within the NI) be enabled, thus preventing a compromised ECU from accessing the network.

Once an ECU has been authorised by its NI, it can start normal communication over the shared bus. However, since the bus is of a broadcast nature, malicious hardware on the bus could still decode the communication and integrate onto the bus. To circumvent this, we utilise synchronised timestamping and cross-layer encryption using light weight ciphers within the CC, providing configurable data security and obfuscating protocol header information. The timestamps prevent replay attacks since stale data is rejected at the CCs of the receiving ECUs. The obfuscated headers mean that an unauthorised device cannot decode the protocol parameters by observing them. Hence, only devices authorised with a pre-shared

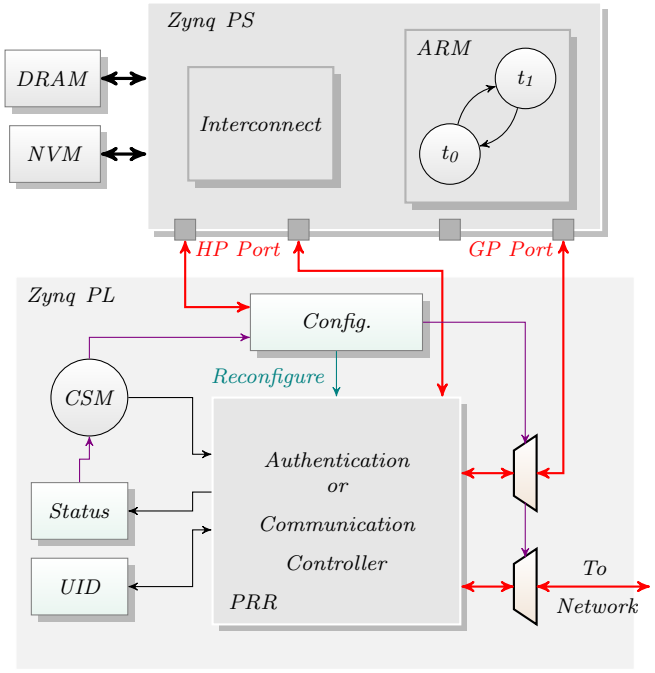


Figure 5.4: High-level System Architecture on Xilinx Zynq.

key (from the OEM) can integrate and communicate over the network. We also incorporate a method to update the keys at run-time using symmetric cryptography, which can be further improved using lightweight asymmetric schemes.

5.5.1 Architecture

In this section, we describe the system architecture of our case study for a FlexRay-based ECU on the Xilinx Zynq platform. We use partial reconfiguration (PR) to dynamically invoke the authentication and CC modules as and when required, thus optimising area and power consumption. We integrate an SHA-1 hashing function and the PRESENT lightweight cipher [203] for software authentication and data ciphering respectively. The proposed concept can be used with other time-triggered network architectures which may eventually replace FlexRay and using other standard hashing/cipher functions.

Hardware Architecture: A high level architecture of the proposed system is shown in Figure 5.4. The Zynq processing system (PS) integrates highly capable ARM cores and a number of peripheral devices like the DRAM Memory Interface,

Non-volatile memory interface (over SPI), Ethernet and others. The PS is tightly coupled with programmable logic (PL), which can be used to implement custom functions and/or accelerators, communicating over High Performance (HP) or General Purpose (GP) ports. The application code, boot-loader and the PL bitstream(s) are stored in non-volatile memory (NVM), as is common for automotive ECUs.

Within the PL, we instantiate either the software authentication mode or the FlexRay CC in a partially reconfigurable region (PRR). During startup, the PRR instantiates the authentication logic by default. The authentication logic comprises an SHA-1 one-way hashing function and a Ring Oscillator-based Physically Unclonable Function (RO-PUF), along with the supporting logic in the static region to support communication with the PS. The *UID* register is a configurable width register (8-bit to 128-bit) that stores the unique identifier for the hardware-software combination on this ECU and is later used by the FlexRay network as the identifier of the specific ECU. The *Status* register holds the status of the software authentication process and is used to enable the interface multiplexers that connect the PRR to the FlexRay bus and to the PS (for configuring the FlexRay CC). The control state machine (*CSM*) is responsible for initialising data movement between the PS and PL (for authorisation) and for initialising reconfiguration of the PRR to load the FlexRay interface once ECU software is authorised. The *Config.* module manages reconfiguration of the PRR.

Authentication Function: During system initialisation, after the Zynq PS has completed the boot sequence and programmed the PL with the default bitstream, the *sys_init()* function initialises the interfaces to the PL logic. The CSM in the PL logic then initialises a DMA read from the non-volatile memory to compute the hash value of the memory content, including the bootloader, the default bitstream for the PL, and the application software (called the *boot image*). The DMA reads are directed into the hashing function and are double buffered to improve the performance. The SHA-1 core is custom designed and operates on 16 32-bit blocks of data (the size of a DMA burst) to iteratively compute the SHA hash for the entire *boot image*.

In parallel, the physically unclonable function (PUF) module generates a 128-bit hardware identifier (HID), which is combined with the SHA hash to authorise the software on this hardware. We use a configurable RO-PUF function with 128 instances of a configurable ring oscillator (RO), each instance contained within a single logic block (CLB), based on the design in [204]. This design allows for accurate reproduction of the hardware signature, since the routing within each RO is completely constrained to the CLB and its associated switch box (interconnect). Once the SHA-1 hash and HID are generated, the software hash is authenticated against the hard-coded *SAR* register value, while the HID and software hash are combined and hashed to determine if the software is authorised to run on this specific ECU by matching it against the hard-coded *SHAR* register value. Once the software and hardware are authenticated and authorised (as valid or invalid), the status register is updated and a reconfiguration may be triggered depending on its value. If authorised, the CSM triggers the *Config. Controller* to read the (cached) bitstream data corresponding to the FlexRay CC and enables the interfaces to the bus, which are otherwise disabled to prevent even a forced reconfiguration from the PS. The architecture of the authentication function, expanded out from the PRR block is as shown in Figure 5.5.

Once authenticated, corrupt software could still be forced onto the PS at run-time (via JTAG for example); however, such programming triggers the PS reset, which is also wired to the hardware logic. This run-time reset disables the interface multiplexers (from CC to PL and CC to FlexRay bus) which stay disabled until the hardware is power cycled, forcing the system to boot from the NVM. This prevents invasive attacks from employing non-persistent run-time manipulation of application code. Any persistent alteration (requiring the code to be changed in the NVM) will be detected as a violation by the hashing function, preventing the system from authenticating the software and loading the CC.

Enhancing the Security Extensions on the FlexRay CC: As mentioned earlier, to provide data security for the messages exchanged on the broadcast network, we have integrated a timestamp module and a lightweight PRESENT cipher into the datapath of the encoding/decoding blocks of our enhanced FlexRay CC

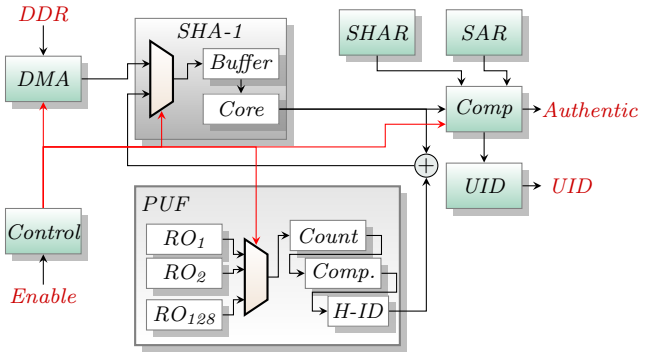


Figure 5.5: The Hardware-Software Authentication logic

as in the previous section. Even though introduction of timestamps increases the entropy of data being exchanged making attacks difficult, it does not prevent a newly plugged-in (or compromised) unauthorised device from accessing communication on the network. To achieve this, we must prevent unauthorised devices from integrating onto the network. The protocol headers encompass information about the communication schedule and the configuration of the network in plain text, which can be observed by an unauthorised device to recover protocol parameters. Knowledge of protocol parameters and the communication schedule can then be used by an unauthorised device to generate a valid configuration for itself, which would allow it to integrate onto the network and manipulate messages to attack other ECUs.

To circumvent this, we extend the controller so that the protocol headers are also obfuscated by the cipher logic. The first byte of the protocol header containing the flag bits is left untouched, but the following 4 bytes that comprise the slot number, cycle number and payload length (along with header CRC) are combined with the 4 byte timestamp (Tx_TS) to form the first 64-bit block. This is then encrypted with the pre-shared key, over 8 cycles (configurable, up to 32), while the first 64 bits of ECU data are prefetched into the buffer. To increase entropy, the data can be (optionally) padded with the timestamp (2 or 4-bytes), sacrificing bandwidth. This data is encrypted with a *timestamp-based key* which is a combination of the header timestamp and the pre-shared key. This approach ensures that there is significant entropy even with identical data (and no timestamp padding) with further increased entropy in the case of data padded with the timestamp. This is

possible in time-triggered networks since all nodes are synchronised. The data may be encrypted with a larger number of cycles (up to 482 cycles) without affecting the latency of the system, since this can be hidden within the encoding/transmission delay of the preceding 64-bits.

At the receiving end, the byte-decoded data is read directly into the PRESENT decoding buffer. The first byte of the frame header is passed through untouched, allowing the protocol defined startup and synchronisation logic to work without changes. The remaining 4-bytes along with the timestamp are decrypted with the same pre-shared key to determine the actual protocol information (slot, cycle, and payload size), which is used by nodes to integrate onto the network. Since only authorised devices are preconfigured with the pre-shared key, this scheme ensures that unauthorised devices cannot integrate on to the network, since they cannot determine network parameters from observing the bus.

The remaining received data is decrypted in a similar manner, but by using the *timestamp-based key* regenerated from the decrypted timestamp (Rx_TS) and the pre-shared key to recover the original data. Since the encryption/decryption latencies are deterministic, the timestamp validation system can add this deterministic offset to time-validate the received message and discard it if older than a configured threshold, protecting the ECU from replay attacks. The altered datapath for the timestamp-based header obfuscation and data encryption is shown in Figure 5.6.

5.5.2 Run-time Alteration of Cipher Parameters

As mentioned, a pre-shared key and default cipher configuration is loaded into the CC of the critical ECUs by the vendor to ensure that only authorised parts are used. This key is used to obfuscate the schedule and data during the network integration phase. To further enhance security, it should be possible modify encryption parameters periodically at runtime. This provides an added layer of protection, since this offers a mechanism to alter the cipher properties and configuration in case a threat has been detected on the network. However, we want

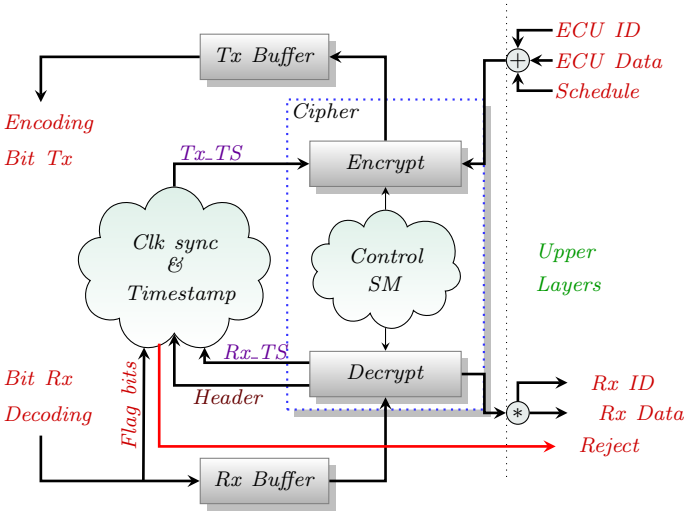


Figure 5.6: Enhanced Datapath at the Cipher logic for incorporating Header Obfuscation.

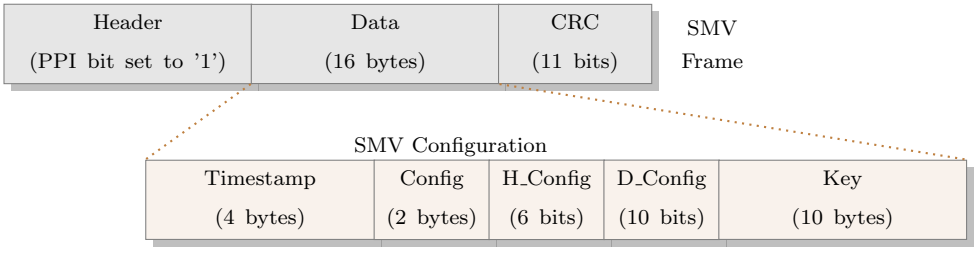


Figure 5.7: Frame format of the Security Management Vector (SMV).

such functionality to be integrated into the existing communication mechanism and abstracted from the application. For this, we make use of a scheme similar to the network management feature available on FlexRay.

Once integrated, the secure gateway can trigger a new cipher configuration using an (encrypted) security management vector (SMV) message, that is received by all CCs. The SMV is transmitted as a network management vector (NMV) feature that is optionally supported by FlexRay ECUs. The frame format for the SMV is as shown in Figure 5.7 and comprises 12 bytes of cipher management data with 4 bytes of timestamp information. The *H_Config* bits represent the round configuration for the header obfuscation feature, while the *D_Config* bits describe the round configuration for regular ECU data. Finally, the new pre-shared key (which can be generated using standard algorithms) is contained in the *Key* segment. The *Config* bits help differentiate the SMV from the normal NMV that is used to adapt parameters at the network interface and these are used by the pattern detection

logic in the CC to detect an SMV. The *Config* bits indicate the validity of the sub-segments (specific bits set to '1' if valid information is present in *H_Config, D_Config, Key* sub-segments) and may additionally be used to determine when the adaptation should take effect (next cycle, next n^{th} cycle).

Once the NMV is identified as an SMV, the security extensions extract the new set of parameters for the cipher logic from the decrypted data. To ensure that all CCs migrate to the new configuration at the same time, the adoption is synchronised to the FlexRay timing cycle (either the following, or after a specific number of cycles defined in the *Config* bits). We have chosen the start of cycle as the synchronisation point, which allows the CC's to pre-compute the round-keys for the new configurations at the previous end-of-cycle marker. Thus all CCs in the network simultaneously migrate to the new configuration at the start of the next cycle. Handling the alteration of cipher properties within the CC abstracts these details from the application and does not provide a path for software hacks to access this information.

5.5.3 System Evaluation

The proposed system is evaluated in two steps. First the software tamper protection is evaluated on a Zedboard featuring a Xilinx Zynq XC7Z020 to measure the boot time, determinism of the hardware identifiers, overall latency and resource overheads. Next, the run-time network access control, data encryption, and enhanced frame entropy are evaluated on a Xilinx AC701 board by integrating multiple ECUs with authentic software (authentication block is removed due to area constraints).

Table 5.3 shows a comparison of resource overheads of the proposed system, compared to a standard implementation of the FlexRay NI (standard CC without any extensions) on the Zynq. Incorporating the cipher and datapath extensions to achieve network access control in both channels results in a 115% overhead in flip-flops (FFs) and 59% in Logic (LUTs) at the NI alone. However, the overall

implementation (NI including PR support) still utilises under 33% of the resources on the low capacity XC7Z020 device with only a marginal increase in power over the standard NI (38 mW increase). PR allows the non-concurrent authorisation and NI blocks to be located in the same physical region, reducing overall resource requirements.

To evaluate the hardware-software authentication, the implementation was tested on multiple Zedboards and the Xilinx ZC702 development board featuring the same XC7Z020 device. It was observed that the HID generated by the RO-PUF differed by 34 bits on average (16 bits minimum) between the different boards for the same configured challenge value, whereas the generated HID had no appreciable variation on the same board for over a few thousand runs at room temperature. Tampered software on the other hand resulted in large variations in the hash value, with a single line edit resulting in more than 80 bits difference. With the *boot image* read directly from the NVM, the authentication function took 118.3 ms to authorise the software while the reconfiguration operation took 62.5 ms to load the secure CC (once authenticated). Alternatively, the *sys_init()* routine can buffer the NVM contents (including CC bitstream) in DRAM memory, and overlap data movements allowing authentication to be completed in 66.5 ms, and reconfiguration in 4.4 ms (to load the secure CC). However, this could be prone to tampering with DRAM contents (via software or otherwise). The default scheme buffers the CC bitstream only, which provides high reconfiguration speeds (4.4 ms) with reasonable authentication latency (118 ms) without compromising security.

We evaluate the network security and data encryption scheme by integrating 4 ECUs (3 authorised and one attacker, all based on MicroBlaze soft cores and a FlexRay CC), each running tasks that trigger data exchange over the network in every cycle, on a single FPGA device (XC7A200T on the AC701 board). For this evaluation, we have assumed that the ECUs are running authentic software, since the single chip does not offer sufficient resources to manage reconfiguration for multiple ECUs. The FlexRay schedule uses a 10 ms cycle at the full 10 Mbps bitrate, with the CCs operating at 80 MHz and the MicroBlaze at 100 MHz.

Table 5.3: Comparison of Resources on XC7Z020.

Function	Submodule	FFs	LUTs	BRAMs	DSPs
standard CC	PMM	225	550	0	0
	CSS	1861	3508	2	1
	MIL(x2)	851	1393	2	0
	CHI	1676	2060	10	1
	Total	5491	8939	16	2
secure CC	PMM	225	555	0	0
	CSS	1861	3525	2	1
	Cipher(x2)	2548	2379	4	0
	MIL(x2)	1380	1635	3	0
	CHI	1676	2086	10	1
	Total	11618	14194	26	2
S/W Auth.	PUF/SHA	4665	6574	1	0
	Static	3895	3619	1	0
	Total	8560	10193	2	0
Overhead (%)	CC	6127	5255	10	0
		115%	58.8%	62.5%	0%

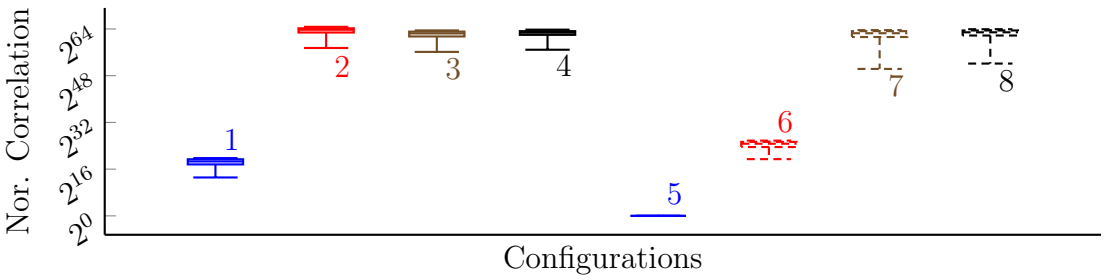


Figure 5.8: Entropy of FlexRay frame with the obfuscation scheme 1: Normal Header, 2-4: Header via different rounds, 5: Static Data by Static key, 6: Timestamped Static Data (unencrypted), 7 : Static data with time-varying key 8: Timestamped Data with time-varying key

Figure 5.8 shows the normalised entropy (correlation) of the transmitted frames (with 1 ms timestamp accuracy), when the authenticated secure CCs integrate and start communicating over the network. The timestamped header, when encrypted over 5,8 and 16 rounds with the same pre-shared key (cases 2 to 4) results in much higher entropy than the standard header from the same device (single slot, all

cycles). The frame headers appear as noise to any unauthorised device, preventing it from integrating onto the network, since the clock synchronisation parameters cannot be extracted from a pair of frames (adjacent cycles) without decrypting them, as observed in our experiment with the attacker ECU. In fact, the attacker is forced to quit the integration process after exceeding the number of failed attempts, as required by the protocol. Figure 5.8 also shows the improvement over entropy of static data (case 5) achieved by timestamping alone (case 6), encrypting with time-varying key without and with timestamped data (case 7 and 8 respectively). The non-timestamped data when encrypted with the time-varying key produces nearly the same impact as encrypting the timestamped data, without consuming additional bandwidth required to incorporate the timestamp in the data segment.

We observe that header obfuscation causes a delay of 125 ns (for the default 8 round decryption of header) from the arrival of header bytes to the decoding of protocol parameters like the slot number, cycle number, and payload length, compared to the standard CC. However, these parameters are only used by the upper layers of the protocol (like medium access) and need only be validated after receiving a complete frame (including the frame CRC), unlike the flag bytes. Hence this delay does not impact protocol behaviour.

Finally, we triggered a run-time cipher adaptation by forcing one of the ECUs to send an SMV message changing the key and number of rounds in the next FlexRay cycle. We observed that the SMV was decoded by the security extensions on the CC within 5 clock cycles of reception of the entire frame. At the end of cycle marker (which corresponds to the slot end marker of the last dynamic slot of the cycle), the round-key generation for the new configuration was initiated, as required by the SMV. We also observed that the round-key generation for the new configuration could be performed even in the current cycle making use of our double key buffering mechanism, enabling the new round keys to be generated without affecting the current encryption parameters (at the expense of increased utilisation). The round-key generation consumes n clock cycles (for n rounds), and can easily be computed in the symbol duration (or network idle period). This enabled instant migration to the new cipher parameters at the end of the current

cycle, with all authorised ECUs moving to the new parameters synchronously at the cycle boundary.

Stronger encryption techniques like AES could be employed in place of the light-weight PRESENT cipher. However, the multi-layer security approach presented here goes beyond just the message encryption to address the requirements of network access control, improved entropy, zero-latency operation, and system-level security. These are important considerations since prolonged access to the broadcast network would help an attacker to easily decipher the security scheme using advanced cryptanalysis techniques. Also, stronger encryption schemes are much more computationally complex, and would result in increased latency, higher power, and resource consumption.

5.6 Summary

In this chapter, we have described schemes to integrate message-level and system-level security for in-vehicle networks by integrating them around and within the communication interface. The schemes present unique benefits over proposed security mechanisms for vehicular systems like HSMs or the upcoming secure hardware extension standard. Firstly, the integration of cryptographic elements close to the network improves the quality of the cipher by making use of cross-layer approaches that leverage features of the communication infrastructure like the synchronised view of time. Secondly, such integration enables complete transparency during operation, with neither the application nor the network aware of the presence of a data cipher. The scheme also incurs zero latency in either direction, by efficiently overlapping cipher operations with transmission and data-movement. The proposed scheme allows an existing application to be ported to the secure domain without requiring rescheduling at the task level or network level, which are critical adjustments required with HSM/SHE-based approaches.

We further extended this data-level security to incorporate two major aspects of system-level security: to verify the authenticity of applications requesting access

to the network, and to integrate a network access management scheme that allows only authorised devices to integrate and communicate on the broadcast bus. The system-level security architecture nullifies the impact of tampering with an ECU's boot-ROM contents or its run-time code memory by preventing such an ECU from accessing the network. This authentication is tightly coupled with a hardware signature generated on the device, preventing attackers from recreating the combined software-hardware signature. For an authorised set of ECUs, the network access is controlled using packet-level obfuscation, that prevents attackers from deducing protocol parameters using off-the-shelf devices. Further, a mechanism for network-wide adaptation of cipher configuration allows the system to react dynamically to threats. Finally, the system-level security architecture is also abstracted from software/application layer and does not compromise the real-time guarantees of the underlying protocol.

The proposed architecture can be further enhanced by incorporating asymmetric ciphers for improved security. Techniques like lightweight authentication utilising asymmetric ciphers can be employed at the gateway to trigger adaptation of the system, offering tighter security [205].

6

Functional Validation Platform

6.1 Introduction

In Chapter 1, we discussed the increasing complexity of automotive computing as new capabilities are integrated in vehicles. This trend is expected to continue with the introduction of networked vehicles. This rising complexity presents a significant challenge in validation and design iteration, as alterations to the network or architecture should not deteriorate the performance or safety of existing systems. This is particularly important in the case of safety-critical systems.

Traditionally, functional validation is carried out by recreating the entire network in a laboratory environment. This test setup, called a hardware-in-the-loop (HIL) setup, enable designers to profile system performance in the presence of actual

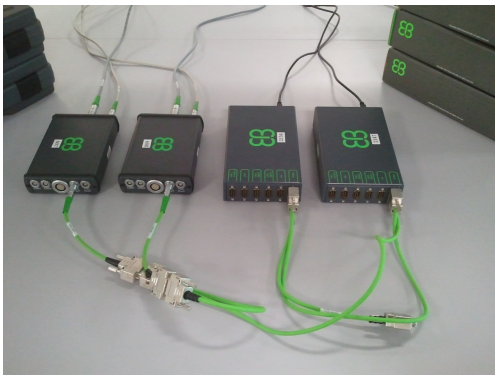


Figure 6.1: Replicated test bed comprising 4 ECUs.

physical conditions that the system may encounter. Real world conditions are modelled in a controlled environment, including multiple real ECUs and their network subsystems integrated with the actual cable lengths used in the vehicle, as shown in Fig. 6.1 for a small 4 ECU system. This is further supplemented by sensors and actuators to interact with the physical environment, enabling the technicians to evaluate the quality of the ECUs and to certify their performance and safety.

New functions, which are developed as stand-alone ECU(s) and later integrated (as new ECUs) onto the existing network, are evaluated in this HIL setup before integration in the actual vehicle. Conducting such evaluations is also important when critical segments of the automotive systems are tweaked, like the network parameters or safety-critical sub-systems, to ensure that the tweaks do not affect the reliability or performance of other subsystems and network guarantees in general. The HIL facility also enables the design team to evaluate optimisation strategies by tuning the network structure (topology), message/task schedules, and other parameters, while staying within the performance/safety bounds. Such extensive test systems use real hardware since high-level simulations often trade-off precision for simulation speed, missing low-level details and errors that could significantly affect the overall system performance.

However, with increasing complexity, the size of the recreated cluster also rises, in addition to the interconnect for the myriad sensors and actuators that must be integrated. For mid-size to high-end models, the HIL environment can be very

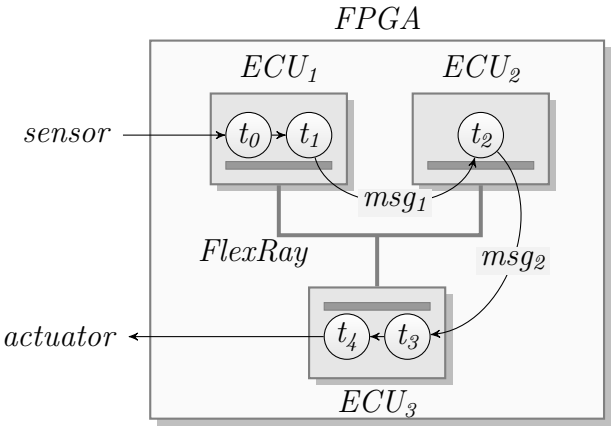


Figure 6.2: Hardware-in-the-loop test setup.

complex and determining the state of the system during a validation test can be cumbersome. Also, with a large amount of safety-critical equipment, as in the case of hybrid and fully electric vehicles (EVs), such an HIL setup does not scale well to handle the complexity and automation of the validation process. For example, a small modification in the network structure may require a significant amount of rewiring, as well as reconfiguration of all nodes' network parameters, which must then be verified before the new function can be evaluated. Furthermore, many of these modifications cannot be automated and require manual effort, further limiting the scope for design space exploration, optimisation, and automated validation.

We propose to model these ECU systems on reconfigurable hardware, with their actual implementations, communication interfaces, interconnect topologies, along with real-world interfaces to sensors and actuators. A schematic representation is shown in Figure 6.2, where the actual sensors and actuators are integrated over a generic interface to complete the HIL system. This allows multiple configurations and subsystems to be validated using the same hardware setup, where changes can be effected through reconfiguration of the FPGA, resulting in deterministic test setups, and faster adaptation and turnaround times for experiments. The same FPGA setup can be adapted to different vehicle models that could be using different network and/or system architectures, either via programmable interconnect (for small topology changes) or through complete reconfiguration.

In this chapter, we show that such a platform can integrate a sizeable automotive cluster on a large enough FPGA, with actual implementations of the ECUs and network interfaces. This allows functional validation to be performed at line rate with bit-level precision. The platform can be configured and controlled remotely from software running on a host computer via a python API. The provided API also allow designers to remotely control debug features (like error injection) and observation (data capture). The FPGA infrastructure and the python bindings together comprise our validation platform, which can aid in automating validation of complex distributed systems and can also accelerate the validation process with super-real-time execution, without compromising bit-level precision.

The work presented in this chapter has also been discussed in:

1. S. Shreejith, S. A. Fahmy, M. Lukasiewycz, *Accelerating Validation of Time-Triggered Automotive Systems on FPGAs*, in Proceedings of the International Conference on Field Programmable Technology (FPT), Kyoto, Japan, December 2013, pp. 4-11 [14].

2. K. Vipin, S. Shreejith, D Gunasekera, S. A. Fahmy, N. Kapre, *System-Level FPGA Device Driver with High-Level Synthesis Support*, in Proceedings of the International Conference on Field Programmable Technology (FPT), Kyoto, Japan, December 2013, pp. 128-135 [15].

6.2 Related Work

Software simulation and model-based verification are widely employed in validating complex systems or software using abstract models [206, 207, 208]. These high-level models capture the generic behaviour of the system but abstract away the finer details about the implementation platform and communication. Many methods also concentrate on modelling the distributed systems as states and their transitions, by assuming lower level details and communication can be considered

deterministic [209]. Such abstractions enable software simulations to run in reasonable time, and can also be easily adapted to different scenarios with minor changes in code. While such abstractions are acceptable in many domains, critical applications must be validated against all possible scenarios. The lack of fine-level details and bit-level communication mean plausible corner cases can be overlooked while validating safety-critical systems. Hence, accurate models are required for reliable simulation. Generating these highly accurate models is cumbersome compared to the abstracted or partial-behavioural models [207], and the use of precise representations make them less adaptable compared to generic models. They also increase simulation run-time even on powerful platforms. Furthermore, the assumption of determinism in the underlying hardware and/or communication is not a guarantee in harsh environments like vehicular systems, where electrical disturbances, electromagnetic interference (EMI), and the operating environment contribute to multiple possible errors in the network and computing infrastructure.

FPGAs have long been used for rapid prototyping and validation of complex applications. Their ability to implement models with cycle- and bit-accurate behaviour is in addition to the possibility of running orders of magnitude faster than software simulations. When the systems being modelled are clocked at rates within the bounds of the selected FPGA, it is also possible to emulate the system in real-time. This makes FPGA-based emulation an efficient method for validating complex processor architectures, application specific integrated circuits (ASICs), systems-on-chips (SoC), and other complex computing systems [210, 211, 212, 213]. ASIC emulation and co-verification of embedded systems is an area which makes extensive use of FPGA-based emulation, primarily because of the design costs involved in ASIC manufacturing and the challenges involved in verifying the software and hardware aspects in complex embedded systems. In [214, 215], the authors present detailed architectures for efficient FPGA-based emulation systems for ASIC validation and complex embedded systems evaluation. FPGA-based emulation is also widely employed in validation of high-throughput communication networks [141]. Here, the use of FPGAs to model the system provides near real-time performance for the network, without sacrificing bit-level accuracy.

While modern FPGAs provide computational capabilities for emulation, the limited number of I/O pins available for user logic can be a challenge when emulating complex systems. This is significant in case of larger designs require partitioning across multiple FPGAs with parallel evaluation. These interconnected FPGAs tax the limited user I/O, further complicating the emulation setup. Early work highlighted this challenge and proposed a solution using a *virtual wires* concept [216]. While FPGA logic capacity grows at a rapid pace, the number of I/Os grows at a much slower rate making large emulation platforms challenging to implement because of a lack of I/O bandwidth. Virtual wires time-multiplex the available I/O that then runs at much higher speed compared to the logic emulator effectively achieving the required interconnect bandwidth.

Modern FPGAs feature high speed serial I/O pins that can achieve sustained throughputs of 6 Gbps or more, and the virtual wires concept can be used to time-multiplex such high speed I/O to achieve the required interconnect bandwidth. Also, interfaces like PCIe can now be used as the backbone interconnect to provide deterministic communication between partitions of a multi-FPGA setup.

Within the automotive context, FPGA-based validation of novel functions has been proposed through mapping computational units and interfaces with special test capabilities on them [148]. Here, the network interface is enhanced with features that enable injection of errors close to the network, opening up possibilities for extended validation. However, the test ECU with enhanced interfaces must be integrated into an HIL setup to evaluate its performance in real world conditions. FPGA-based prototypes have also been proposed and used to validate emerging technologies like vehicle-to-vehicle communication [217]. Here the logic on the FPGA can mimic the wireless communication channel and its properties such as multipath fading, which can be used to evaluate the performance of the proposed communication scheme and transceivers.

6.3 Contributions

We present a framework that integrates a hardware-based functional evaluation platform with software-based control on a host PC. The proposed functional validation infrastructure models multiple computational nodes and their interconnect (a cluster of ECUs) on a large FPGA, mapping actual implementations of computational units (ECUs) and communication interfaces onto the FPGA. The different clusters are mapped onto multiple FPGA boards, which can then be integrated to replicate the entire vehicular system. Since actual implementations of the network and communication infrastructure are utilised, the test setup can perform extensive HIL validation, and alterations can be made through a fast reconfiguration operation. In many cases, it is also possible to achieve speed-ups during functional validation by using optimisations in the network infrastructure. Designers can also utilise the reconfigurability of the platform to explore complex design variations and deliver optimisations which are validated in the actual hardware.

We have developed a validation platform using a single FPGA board, that integrates up to 6 complete ECUs along with their network infrastructure on the ML-605 development board featuring a Xilinx Virtex-6 device (10 ECUs on the VC-707 board that uses a Xilinx Virtex-7 device). For our experiments, we have used the FlexRay network protocol, by integrating the optimised interface described in Chapter 3 with the computational units. We have also built a software API that allows users to configure and control the platform over the UART, JTAG, and Ethernet interfaces to observe ECU communication in real-time from a host machine. An optimisation framework can then be integrated on top of this infrastructure to explore different design combinations and evaluate them on the actual hardware to determine the optimal architecture and communication schedule for a given set of functions.

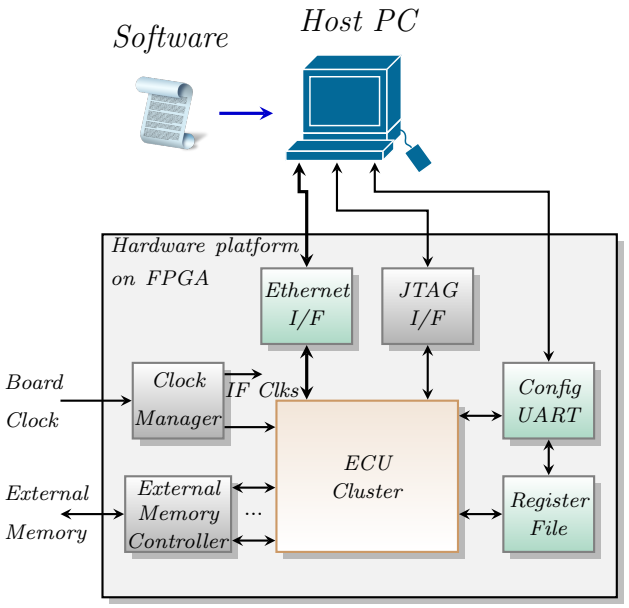


Figure 6.3: System Architecture of the Validation Platform.

6.4 Platform Architecture

The complete validation platform comprises a standard host PC connected to a commercial FPGA board, as shown in Figure 6.3. Software on the host PC controls and configures the FPGA board over standard Ethernet, JTAG, and UART interfaces. The FPGA board integrates a cluster of isolated compute units or ECUs connected through a dual-channel FlexRay communication network. Each ECU may implement a specific automotive function, like park assist or adaptive cruise control, either as a hardware implementation or as software running on a soft processor. Interfaces within the FPGA are built using hard macros or optimised IP to minimise overhead, and allow for complex ECUs.

6.4.1 Hardware Architecture

Figure 6.4 shows the architecture of the validation platform, with the external interfaces and clock domains. The figure describes an example design comprising 6 independent computing units marked ECU_1 to ECU_6. Each ECU is fed with independent isolated clocks, marked as HC_x for ECU_x generated by the hardware clock manager. Each IC_x clock is the interface clock for the FlexRay interface

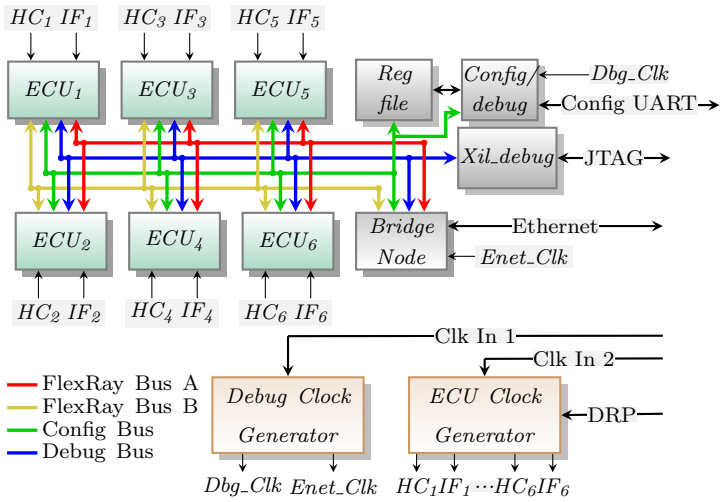


Figure 6.4: Hardware Architecture of the Validation Platform.

within ECU_x, enabling the interfaces to be clocked at a different frequency from the ECU core. The interfaces enable communication between ECUs (over the FlexRay bus) and can be controlled by the host PC.

Host Interfaces and Global Registers: The host PC can communicate with the ECUs over the shared UART *Config/debug* interface, the *Xil_debug* JTAG interface and the *Ethernet interface*, as shown in Figures 6.3 and 6.4. The host PC controls the FPGA platform by accessing the global registers (register file) over the *Config/debug* interface. The *Xil_debug* JTAG interface enables initialisation and debugging of MicroBlaze-based ECUs in the cluster, using the MicroBlaze debugger module (MDM). The host PC uses the Ethernet interface as a real-time debugger for monitoring the state of the FlexRay bus and selected control signals from the ECUs in the cluster.

The Register File implements a set of global registers that are used to configure the interfaces, set platform parameters, and control/configure the special test features. The functionality of each register is described in Table 6.1. The control/configuration registers are used for enabling/disabling the platform, enabling test cases and to configure the operation modes. The *Config UART* is also mapped in the memory space of each ECU, and thus doubles as a debug interface. This enables host software to access debug data and registers using the ECU address registers (*ECU addr reg*) as an indirect addressing register.

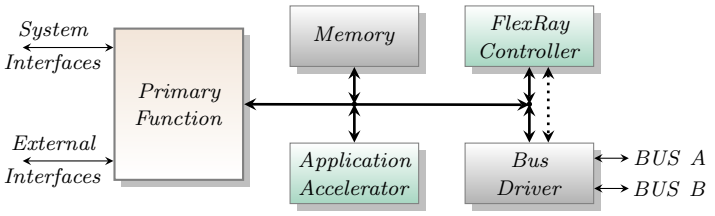


Figure 6.5: Example ECU architecture.

ECU architecture: Compute units implement automotive functions either as hardware functions, or as software on a soft processor. Figure 6.5 illustrates one such model for an ECU. As would be the case in a vehicle, each ECU is completely independent and implemented in isolation. The primary function represents the

Table 6.1: Register file description of the Validation Platform.

Address	Function	Description
0x00	Version register	H/W version number
0x01	Platform control	Controls the interfaces and operative modes of the H/W platform
0x02	Platform status	Interface and mode status register for the H/W platform
0x03	Debug control	Operative modes & parameters for the debug module
0x04	Error injection	Enable/disable error modes - bit error, frame error, frame delay, frame drops
0x05	Bit error config	Specifies the bit/byte(s) to insert error
0x06	Error slot config	Specifies the Slot-ID to insert error
0x07	Delay rate config	Specifies the delay value in ticks
0x08	Frame drop config	Specifies the FlexRay cycle to drop Frame
0x09	ECU addr reg [31:16]	ECU debug : Indirect address register (upper DWORD)
0x0A	ECU addr reg [15:0]	ECU debug : Indirect address register (lower DWORD)
0x0B	ECU access control	Selects ECU and read/write for indirect access
0x0C	ECU data reg [31:16]	ECU debug data (write/read)
0x0D	ECU data reg [15:0]	ECU debug data (write/read)
0x0E	Clock jitter	Select the clock offset for IC_x - frequency/phase and offset value

functional implementation of an automotive algorithm in either software or hardware. The application accelerators are dedicated hardware units like FFT modules or FIR filters that leverage FPGA specific hardware like DSP blocks, or other custom hardware designed for a specific application. The ECU memory is built using BlockRAMs. ECUs also incorporate a dual channel FlexRay Communication Controller (CC) which implements the FlexRay communication protocol. A key aspect of this platform is that each ECU uses a fully featured communication controller for verification, just as would be the case in the final deployment.

The ECUs may interface with subsystems like sensor modules over standard interfaces like SPI, I2C, or other system interfaces. ECUs may also interface with external storage elements like non-volatile memories or high-speed DRAMs. The external memory controller provides multi-channel access to such storage elements, and ECUs can connect to it over a streaming interface. It is configured to ensure that the memory spaces are isolated between the different channels.

FlexRay Communication Controller (CC) and Debugging Extensions: The heart of the validation platform is the FlexRay bus and our custom FlexRay CC. The optimised implementation enables a complete network interface to be integrated with every functional unit, replicating a full ECU. The configurable extensions in the FlexRay CC like timestamping, header insertion, and filtering are utilised during functional evaluation. The synchronised timestamps present a common time-base for evaluating functional performance as well as to determine the predictability of the system when validating critical features like fault-tolerance.

The FlexRay CC at each ECU also integrates a buffer layer (called the *bus buffer*) to model the physical medium interface to support error injection in the messages transmitted by the ECU. The *bus buffer* is composed of a delay line, bit inverter, and a buffer with multiplexers connecting different paths, as shown in Figure 6.6. In the normal operating mode, the multiplexers connect the output of the FlexRay CC directly to the bus by enabling the buffer and bypassing the other elements. The bit inverter is used to inject bit-level errors in the transmitted message, while the memory block is used to model a configurable line delay. The different paths

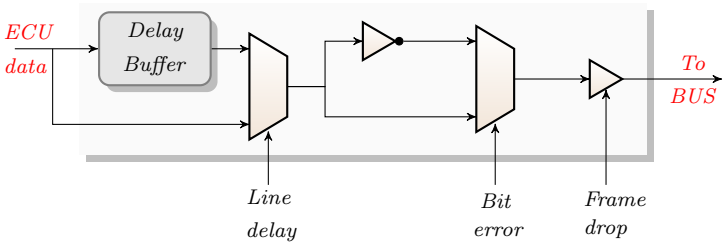


Figure 6.6: Architecture of the Bus buffer module.

are enabled by the configuration in the *register file* and are triggered by a specialised ECU called the *Bridge Node*.

The *Bridge Node* implements two distinct functions; a bus replay module (BRM) function and a bus debug module (BDM) function. The BRM allows logged data from a real experimental setup to be stored in its CC and played back within the FPGA network in a cycle-accurate manner. The BRM handles special test capabilities like bit-error injection, delay and jitter adjustment, as well as preconfigured frame drops during transmission, using the extended capabilities of the custom FlexRay CC and *bus buffers*. The BRM supports specific (e.g. delay slot x in cycle y by t units) and pseudo random specifications for the parameters.

The BRM can also inject random messages, i.e., transmission of data in slots which are not assigned to this node but may or may not be assigned to some other node. Since FlexRay CC implementations only allow transmissions in configured slots, the CC of *Bridge Node* is extended to allow transmission in un-assigned slots. The BRM utilises this extension to inject faults like un-synchronised transmissions, transmissions across slot boundaries, and "babbling idiot" faults. The BRM can also be used to inject specific commands that can trigger actions from the ECUs under test, like triggering a computational mode switch, fault tolerance modes, and task migration actions.

The BDM enables real-time debugging of the FlexRay bus and ECU control signals selected at design-time. The BDM captures the selected signals with a timestamp and encapsulates them into Ethernet frames which are transmitted over the Gigabit Ethernet interface. The BDM also features a trigger mode which can be configured to perform a trigger-based capture of selected signals. Ethernet frames

are decoded into value change dump (VCD) format by the debug tools and can be viewed on the host PC in real-time.

6.4.2 Management of the Platform

The initialisation and management of the verification platform is handled on the host PC using the Python function calls listed in Table 6.2. The individual *init_* functions can be used to initialise the platform by configuring all the ECUs, one specific ECU or the BRM module. The *mode_platform* function controls the error injection capabilities of the platform, while the *mode_debug* function configures and triggers the live-debugger module (BDM).

The code for an example design comprising multiple ECUs is shown in Figure 6.7. The initialisation segment creates a merged bitstream file from the tool-generated bit file by invoking the *init_platform* API call. The function merges the software (elf) for ECU_1 (mb_0) into the tp_top bit file, which is used in subsequent calls that initialise the remaining ECUs. For the final ECU, the *done_flag* is set to 1, triggering the initialisation of the FPGA with the integrated bitstream.

Table 6.2: Software APIs for controlling/configuring the Platform.

Function	Arguements	Description
init_platform()	elf_file, processor_tag, bit_file, done_flag	Initialises the processor *processor_tag* with code *elf_file*, creates bitstream *bit_file*, downloads it to FPGA
mode_platform()	reg_file_address, reg_file_value	Modifies the register file content at address *reg_file_address* with value *reg_file_value*; alters the operating mode of platform
init_processor()	elf_file, processor_tag	Downloads the modified software *elf_file* to the processor *processor_tag* and resets it
init_BRM()	brm_file	Initialises the BRM memory with the captured FlexRay bus data and enables the BRM
mode_debug()	reg_file_value, capture_flag	Alters the behaviour of the Bus Debug Module. Can choose debug signals using *reg_file_value* and alter host data capture using *capture_flag*

Subsequently, tests are performed on the cluster by using the *mode_debug* and *mode_platform* function calls. A change in software or communication schedule is triggered by altering the software routine in the specific ECU, which is triggered by the *init_processor* API call in the example.

```
from init_platform import init_platform
from init_BRAM import init_BRAM
from mode_debug import mode_debug
from mode_platform import mode_platform
from init_processor import init_processor

# define global values
debug_config = #value
test_config0 = #value
...
test_configN = #value
...

# Initialisation
# merge ECU software with bitstream
# generate a merged file for first ECU
init_platform("ecu0.elf","mb_0","tp_top",0)
# use merged file for further ECUs
init_platform("ecu1.elf","mb_1","tp_top",0)
...
# set done_flag to 1 for the final ECU
init_platform(...,1)

# Run Tests
# write debug_configuration and start debug
mode_debug(debug_config,1,1)
mode_platform(reg_addr, test_config0)
...
# stop current test
mode_platform(reg_addr, 0,0)

# Modify ECU S/W
init_processor("elf_new1.elf","mb_1")
# Re-run Tests
# write debug_configuration and start debug
mode_debug(debug_config,1,1)
mode_platform(reg_addr, test_config1)
...


```

Figure 6.7: Python Software Flow.

6.4.3 Accelerated Mode

A normal FlexRay bus provides a serial datapath over an unshielded twisted pair cable. To ensure data is protected in the harsh automotive environment, the FlexRay protocol imposes bit-level redundancy for all transmissions. The decoder must sample these redundant bits and majority vote over a predefined 8-bit window to decide bit polarity. This results in an actual transmission rate of 80 Mbps for

the maximum FlexRay data rate of 10 Mbps. Hence, an 80 MHz sampling and transmission clock are required for the FlexRay CC. This would limit the potential overclocking possible on an FPGA to around 3 × (a frequency of 240 MHz is considered high for complex designs).

Since the entire FlexRay bus is contained within the FPGA, and we can be sure of transmission robustness, we take advantage of the 8 times serial redundancy to make the bus byte-wide. In order not to affect the protocol constraints, only the coder-decoder module within the FlexRay PHY is altered to support byte-wide transmission and reception. This relaxes the clock frequency required for the interface to 10 MHz. Alternatively, the modification allows data transmission to be overclocked to 8 × or more, enabling faster progression of the emulation. With *fast_mode* selected, the achievable cluster acceleration is limited primarily by the acceleration possible for the compute nodes, rather than the communication infrastructure.

For functional validation, the FlexRay CC switches its *operating mode* from normal-serial mode to fast-parallel mode when *fast_mode* is enabled in the *Platform Control* register. The ECUs are issued with a reset signal and the FlexRay interface switches to the parallel mode. The ECUs' software reads the state change and enables faster local clock configuration for the interface, thus accelerating the validation process. However, for HIL tests which should progress at line-speeds, *normal_mode* should be enabled. For *normal_mode*, the FlexRay CC switches to the normal-serial mode which offers complete compliance with the FlexRay interface specification at all levels and is configured with the actual local clock configuration. The verification/HIL tests then progresses at normal hardware speeds.

6.5 Evaluating Automotive ECUs on the Platform

To evaluate the capabilities of the platform we have implemented it on a Xilinx ML605 board that incorporates a Virtex-6 LX240T device and on a Xilinx VC707 board that incorporates a Virtex-7 VX485T. This implementation incorporates 6 compute units including the bridge node. Four of them (ECU_1 to ECU_4) use a MicroBlaze running various algorithms as the primary function. ECU_5 is a hardware-based ECU mimicking a dedicated ASIC, while the *Bridge Node* also uses a MicroBlaze-based ECU with debugging capabilities. The resources consumed by this setup are shown in Table 6.3. For our evaluation, we have configured the ECUs to utilise on chip memory (BRAMs) rather than external memory, resulting in 88% BRAM utilisation. With this exception, the overall FPGA utilisation is below 50% of the Virtex-6 device and hence it is possible to integrate more ECU functions. This can be relaxed by utilising the external DRAM memory over our external memory controller interface; however this could limit the acceleration achievable because of the limits of the underlying DRAM interface controller. On the larger Virtex-7 device (on the VC707 board), the same setup consumes under 40% of resources, and thus it is possible to integrate 10 or more functions.

To quantify the capabilities of the platform (error injection, acceleration and others), we use a number of case studies that involve actual automotive applications mapped as software tasks onto the cluster of ECUs. The park assist, cruise control, and brake-by-wire systems discussed earlier are mapped as the tasks to be validated. Of the three applications, two non-concurrent non-safety-critical functions in the form of park assist and cruise control are evaluated for the first case study, and a highly critical brake-by-wire system is evaluated for the second case study.

The first case study evaluates the performance of a non-critical ECU cluster. Here, ECU_1 and ECU_2 form part of a park assist system with ECU_1 forming the sensor interface and ECU_2 forming the compute and actuator interface. ECU_3 and ECU_4

represent prime and standby logic for an adaptive cruise control system with dedicated hardware accelerators. ECU_5 consists of a hardware implementation of a radar interface for the adaptive cruise control system which passes sampled radar data over the FlexRay bus for processing by the primary and standby units. The *Bridge Node* mimics a centralised fault detection unit which can trigger a switch between the primary and redundant functions. The commands to/from the brake and throttle actuators/sensors are collected/generated by the *Bridge Node*. This can be replaced by actual models in an HIL test setup, or connections to real actuators/sensors.

For the second case study, we use the platform to determine operative bounds of a safety-critical system. Here, the brake sensor tasks are mapped on ECU_1 and brake actuator tasks on ECU_2. Each of them incorporates a fault-tolerant subset (redundant tasks), which guarantees a minimum level of performance. ECU_3 monitors the FlexRay bus and detects errors in ECU_1 tasks and/or communication. The functionality, fault-tolerant behaviour, and fault-recovery actions are evaluated in the presence of faults like bit-errors, frame-drops, and frequency drifts injected through our validation platform. The FlexRay communication parameters used for the two case studies are detailed in Table 6.4.

Table 6.3: Resource utilisation on XC6VLX240T and XC7VX485T.

Function	Virtex - 6				Virtex - 7			
	LUTs	FFs	BRAMs	DSPs	LUTs	FFs	BRAMs	DSPs
ECU_1	11339	7614	77	5	11553	7548	80	7
ECU_2	11334	7614	29	5	11581	7548	30	7
ECU_3	13837	11766	87	47	13337	11644	90	49
ECU_4	13844	11771	87	47	13322	11651	90	49
ECU_5	9006	5604	13	2	9224	5496	13	2
Bridge ECU	12121	8479	79	5	13328	8421	80	7
Debug Logic	791	1010	10	0	821	970	10	0
Total	74186	56184	729	111	73166	53278	741	121
(%)	49%	18%	87.6%	14%	24.1%	8.8%	38.2%	4.3%

6.5.1 Test Cases

To inject errors into the system, we make use of the different capabilities of the platform. For network error injection, we inject bit-errors and frame drops modelling disturbances and loss of communication on the cluster network. Another common error condition in a distributed system is the loss of synchronisation between different nodes due to drifts in their individual clocks. Clock drift within the system is tested by dynamically altering the ECU clock frequency and phase using the clock generator module. For testing selectable functionality like a user configurable drive mode select, the BRM is used to inject specific commands to trigger a corresponding response from an ECU. The effect of a complete outage

Table 6.4: Communication schedule for the Cluster.

Parameters	Assigned Values
Number of Cycles	64, 1 ms per cycle
Number of Static Slots	15 at 32 macroticks each
Payload Length (Static)	2 words
Number of Dynamic Slots	71 (max)
Non-critical ECU Cluster	**Case Study – I**
ECU_1 Data Txn	Cycles 0 to 31 on multiple slots
ECU_2 Data Txn	Cycles 32 to 63 on multiple slots
ECU_3 Data Txn	Cycles 0 to 15, 48 to 63 on multiple slots
ECU_4 Data Txn	Cycles 16 to 47 on multiple slots
ECU_5 Data Txn	All Cycles on Slot 16
Bridge_Node Data Txn	All Cycles on Slot 5
Safety-critical ECU Cluster	**Case Study – II**
ECU_1 Data Txn	All Cycles Slot 3
ECU_1 Fault-tolerant	Odd Cycles Slot 4
ECU_2 Data Txn	All Cycles Slot 13
ECU_2 Fault-tolerant	Odd Cycles Slot 14
ECU_3 Data Txn	All Cycles on Slots 7
Bridge_Node Brake-Input	All Cycles on Slot 15

of the network caused by a faulty node which prevents any meaningful communication on the network is also evaluated. The effects of the different errors are validated against expected behaviours.

6.5.2 Evaluating Case Study 1: Cluster of Non-Critical ECUs

We now evaluate the **park assist** and the **radar front-end** system that are integrated as a cluster on the platform. Errors are injected in the network and at ECU level during this evaluation using the capabilities of the platform. The consolidated results for the different test cases described below are tabulated in Table 6.5.

6.5.2.1 Network Error Injection

For the **park assist** system, ECU_2 samples data from the sensor ECU (ECU_1) over 64 ms and computes the adjustments required to the throttle, steering, and brake controls to complete the parking operation, once the park-mode is chosen by the user. The module should cease its operation if it receives a consistent stream of erroneous sensor data (64 samples) or if it fails to receive valid data over 4 consecutive communication cycles. Bit-errors are injected into the sensor data transmitted by ECU_1 by toggling a pre-configured data-bit (8 bits on the network) in the message for 64 consecutive samples and it was observed that on receiving the 64th consecutive sample with error, ECU_2 transmits an error code in its response message indicating a sensor malfunction, and ceases to provide control information to the throttle and steering ECUs. ECU_1 stays halted even if it receives a stream of error free packets, and must be restarted by re-enabling the park-mode signal, as in the specification.

When data frames from ECU_1 are dropped modelling a communication breakdown between the two modules, it was observed that ECU_2 issues the error code

in its data segment as in the previous case and ceases to provide control information. However, once the communication is re-established, ECU_2 resumes normal operation after receiving a complete set of error-free sensor data, as required by the design.

The **radar front-end** system is based on a 64 ms frequency modulated continuous wave (FMCW) radar scanner. When enabled, ECU_5 models the radar system and transmits the received samples from the radar interface for processing by the target detection ECUs (ECU_3 and ECU_4). These ECUs detect and estimate the distance and velocity of targets from the sample data and issue control signals to the throttle and brake ECUs. ECU_3 and ECU_4 will accept a complete dataset (1024 samples) only if all the bytes are marked error-free. In case of an erroneous cycle, the ECUs must not issue any control commands corresponding to the erroneous data. It was observed that with a single error in a dataset, ECU_3 and ECU_4 flagged erroneous communication in their respective messages without providing any control input to the throttle/brake ECUs. With persistent errors, both ECUs went into fall-back mode which provides minimal functionality based on error-free data.

6.5.2.2 Babbling Idiot Test

This test models a common fault in time-triggered system caused by an out-of-sync node which transmits messages at arbitrary points in time, corrupting the transmission schedule and the messages exchanged. This is modelled by forcing the Bridge_Node to transmit frames in slots not assigned to it, disrupting the communication sequence. During this test, it was observed that the **park assist** and **radar front-end** ECUs that were synchronised to the network prior to the error, fail to stay in synchronisation due to the inconsistency of transmission introduced by the faulty node. Further, they switch to the halt mode and flag a clock synchronisation error, as required by the FlexRay standard. Once restarted under normal conditions, the ECUs reset the FlexRay interface and re-integrate into the cluster, resuming normal communication.

6.5.2.3 Clock Drifts/Jitter

In our test setup, the effect of drift in ECU clocks is modelled by altering the phase and/or frequency of the interface clocks dynamically using the dynamic reconfiguration port (DRP) found in Xilinx Clock Managers. When enabled, a predefined set of test cases that perform phase drifts and frequency drifts can be performed on the nodes. These include shifting any selected clock by 45/90/180 degrees or altering the frequency by fixed steps. Any chosen set will cause all ECUs to be reset and restarted with the selected clock combination.

During our experiments, we observed that phase variations and small frequency

Table 6.5: Evaluation of non-safety-critical park assist and radar system with error injection enabled on the validation platform

Test Case	Expected	Observed
Bit-error	Park assist: ECU_2 fault message	ECU_2 transmits error code indicating sensor fault
	Radar system: reject dataset	Rejects data & flags error; fall-back mode with persistent errors
Frame drop	Park assist: ECU_2 fault message	Transmits error code, recovers when communication re-established.
	Radar system: fall-back then halt	Switches to fall-back mode & halts with persistent errors
Special Frames	Radar ECU: fall-back, then recovery mode	Radar ECU decodes error, triggers fall-back mode & initiates recovery
Random Txn	Synchronisation error	All ECU's flag sync error & halts; recovers on reset
Clock Drift: Phase	Phase drifts absorbed by FlexRay clock synchronisation	
Clock Drift: Frequency	For drifts > 10% interface loses synchronisation & halts, unable to recover	

variations on the clock (less than 10%) were absorbed by the FlexRay clock correction mechanisms, with the reset triggering a re-synchronisation consuming 6 cycles (6 ms). Frequency variations of more than 10% caused nodes to go out of sync, without being able to recover from the error.

6.5.3 Evaluating Case Study 2: Cluster of Safety-Critical ECUs

We now evaluate the critical brake-by-wire ECUs in the presence of different error combinations. The system uses a linear deceleration model with the vehicle normally running at the top speed of 100 km/h. Under error-free conditions, the constant velocity vehicle can be brought to a complete halt in 4.8 s, when the brake is engaged. This is achieved by applying the braking pulse to the actuator every 400 ms periodically. When persistent faults are detected in the ECUs, the fall-back mode takes over, which offers minimum functionality to bring the vehicle to complete halt in double this time (minimum operating mode). The consolidated results of the different tests can be observed in Figure 6.8.

6.5.3.1 Network Error Injection

In every cycle, the sensor ECU (ECU_1) samples the brake input (message received in previous cycle from Bridge_Node and physical I/O connected to buttons) and generates a sensor message for the actuator ECU (ECU_2). The actuator ECU receives the sensor messages, consolidates them over a 400 ms period and applies the braking pulse for 1 ms, if the brakes are to be enabled. It also receives the fault-status message from ECU_3 which indicates if persistent faults have been detected in the system or in the communication from ECU_1. In the absence of faults or with transient faults, the ECUs operate in normal mode, however, if persistent faults are detected they switch to the minimum operative mode.

When bit-errors are injected into the system in a non-consecutive fashion (like periodic bit-errors in messages, in every 3 block periods of 400 ms), the system

continues to operate in normal mode. However, ECU_2 observes the large fraction of errors during consolidation and does not provide actuation outputs corresponding to the missed frame, resulting in increased response time. Similar performance was observed when frames were dropped in a non-consecutive fashion (periodic drop of frames, in every 5 block periods of 400 ms); the system continued in normal mode, but with reduced performance (increased response time).

When errors were made continuous in slot 3 of every cycle, the ECUs were forced into the fall-back mode which uses a different schedule for communication. This sets the minimum operative condition that brings the vehicle to halt from the constant speed in 9.6 s. However, if the error is removed after, say 2 seconds, the systems recovers to the normal operative mode to achieve a better braking performance.

6.5.3.2 Babbling Idiot Test

With the babbling idiot enabled, we observed that the ECUs switched to the fault-tolerant mode as the synchronisation could not be established on the FlexRay network. However, as the fault-tolerant scheme also uses the same network, the system was unable to provide any functionality. Once normal conditions were restored after 4 seconds, the ECUs re-integrated and re-established synchronisation and resumed operation in normal mode within 6 cycles (6 ms).

6.5.3.3 Clock Drifts/Jitter

Here, we first halved the ECU clock frequency of sensor ECU_1 by altering the configuration of the clock generating circuit. This change in system performance was observed by ECU_3 which generates a system fault message over the network, forcing ECUs into fault-tolerant mode, and later to halt. Further, we also observed that frequency variations to the interface clock frequency of under 10% were absorbed by the FlexRay clock correction mechanism, with the reset causing a temporary loss in functionality for 6 cycles (6 ms) due to re-synchronisation. Varying

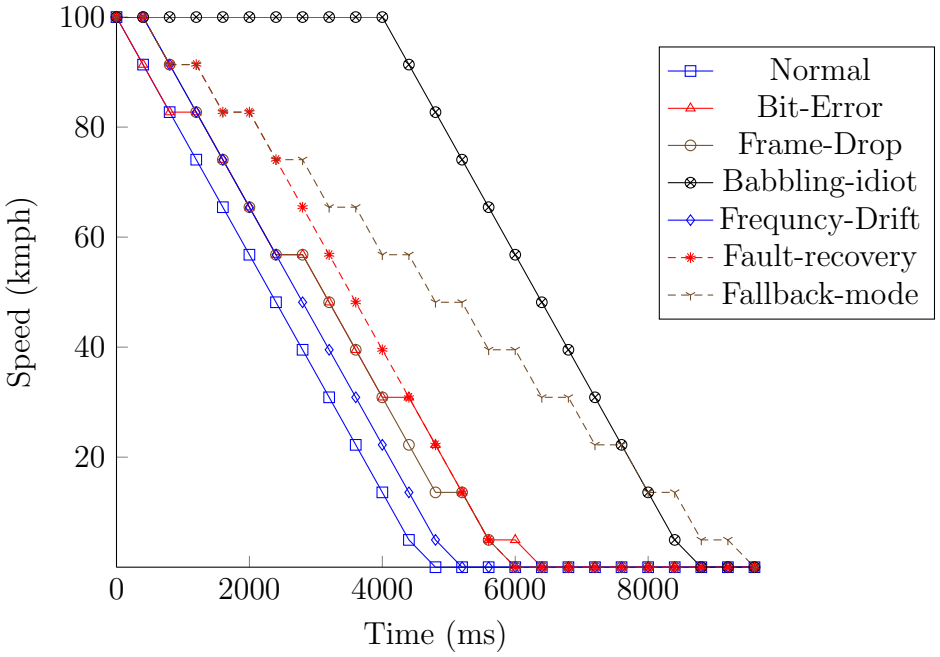


Figure 6.8: Safety-critical brake-by-wire system's performance when evaluated using the platform in presence of different errors.

interface frequency by more than 10% resulted in a complete loss of synchronisation, without the ability to recover from the error (though the fault-tolerance features were enabled).

Table 6.6: Observations during acceleration tests.

Mode Requested	Observed Acceleration
Fast_mode (1×)	Parallel data mode enabled, normal communication
Fast_mode (2×)	Two times acceleration in response & communication
Fast_mode (4×)	Four times acceleration over normal mode
Fast_mode (8×)	ECU_1 to ECU_4 fail, ECU_5 operates at 8× speed

6.5.4 Acceleration Tests

Accelerating the emulation process is one of the key advantages of our platform. To determine the achievable acceleration, we use a 5 ECU setup similar to the non-safety-critical ECU scheme (4 MicroBlaze and 1 hardware logic) described above with the same communication schedule (i.e., all nodes communicate in every cycle). In normal mode, all ECUs are run at 40 MHz (MicroBlaze clock), while the FlexRay network runs at 10 Mbps data rate (full capacity) with an 80 MHz interface clock. When *fast_mode* is selected with default acceleration (1×) in the *platform control* register, the debug waveform shows the ECUs being reset, and clocked with the new frequencies, with the parallel communication interface enabled. Normal communication was established over the parallel bus at 10 Mbps data rate, but with the interface now being clocked at 10 MHz. By choosing an acceleration of 2× in the *platform control* register, the ECUs were fed with 2× clocks by the clock manager, while the interface is also sped up to 20 MHz. Thus the function and interfaces now run twice as fast as in the normal case. With 4× acceleration, MicroBlaze-based ECUs, now clocked at 160 MHz, were at the limits of what is supported, while the interface was still only at 40 MHz, much below the achievable limits. At this speed, reliable and error-free communication was established between all ECUs. MicroBlaze-based ECUs were limited from further acceleration, and only the logic-based ECU_5 was able to run at 8 ×. However, no communication occurred at this stage, since only the logic-based ECU was able to use the FlexRay bus. The results of acceleration tests are tabulated in Table 6.6

6.6 Host Interface over PCIe

One of the bottlenecks for integrating complex clusters with a large number of ECUs on our validation platform was the limited bandwidth of the bus debugger module (BDM) that uses Gigabit Ethernet connectivity to the host PC. Our platform utilised nearly 92% of the Ethernet bandwidth with the 6 ECU setup in the accelerated mode of operation. This limits the number of ECU signals that can

be continuously monitored in the host-PC over a single Ethernet link. It could be possible to use multiple Ethernet links using off-the-shelf FMC Ethernet interfaces, but this would be cumbersome.

An alternative solution is to use PCIe that offers tighter integration and higher throughput between the host PC and board, enabling unified data and control interfaces and a more compact test setup. Freeing up the Ethernet interface also enables validation of future automotive functions that may use Automotive Ethernet interfaces.

The PCIe interface allows use of more abstract programmed-I/O (PIO) transactions with an address-data interface. For ECU initialisation and run-time software debugging, the existing *Xil_debug* interface is also supported. PCIe also supports advanced techniques like multiple streams and re-ordering to achieve high throughput to the host PC.

We have developed a high-performance accelerator framework on FPGA, which can interface with the host PC over PCIe or Ethernet, with management bindings in C that mimic the CUDA API. It utilises a custom designed PCIe interface manager and a multi-port DRAM interface manager that ensures high throughput connectivity from the user accelerator/design on the FPGA to the respective physical interfaces. On the Virtex-6 ML605 board we use for our validation platform, our evaluations show that we are able to achieve 1.4 GBps sustained throughput between the host PC and the platform when buffering data via the DRAM memory and nearly 1.2 GBps throughput directly from the user logic. This throughput

Table 6.7: Resources consumed by interfaces on ML605 board.

Component	Area		
	FFs	LUTs	BRAMs
PCIe Manager	3902	4222	47
Multi-port DRAM	7196	7305	3
Total	11098	11327	50
(%)	3.68%	7.7%	12%

is achieved over a PCIe Gen2×4 interface, significantly higher than the 114 MBps maximum throughput achievable over Gigabit Ethernet. The framework utilises only 12% of the overall resources to achieve such high throughputs, as can be seen from Table 6.7. The low resource overhead allows us to map the same number of ECUs on the ML-605 board, with better debug capabilities.

6.7 Summary

In this chapter, we presented a platform that enables accelerated functional validation of automotive clusters using a hardware-in-the-loop setup on a commercial FPGA. The platform provides a modular mechanism for implementing a cluster of ECUs, and provides special test capabilities for emulating real world conditions. The compliance with automotive specifications is ensured by replicating the actual communication network and interfaces at each ECU. The platform also provides the ability to introduce common network errors like clock jitter, frame drops, and bit-errors, among other possibilities. Furthermore, the platform capabilities as well as the individual ECUs can all be controlled, monitored and debugged from the host PC using a software API. A proof-of-concept implementation on a Xilinx ML605 board was presented, allowing a cluster of 6 ECUs on a FlexRay network to be validated. We used automotive specific functions in our case study to demonstrate the capabilities of the platform and show that acceleration levels of up to 8× are achievable during the validation process. The bandwidth used by the real-time debugging capability can be further enhanced using a PCIe interface, providing unified communication and over 10× higher communication bandwidth to the host machine than over Gigabit Ethernet, allowing more extensive evaluation of the system.

7

Conclusions and Future Research

The complexity of vehicular computation and communication has increased dramatically and will continue to do so with the introduction of future technologies like cooperative communication and driverless vehicles. This thesis has proposed architectures and techniques for enhancing the computational capabilities and communication infrastructure for next generation E/E architectures, by utilising the capabilities of FPGAs. We have shown that enhanced communication interfaces on customisable FPGAs offer tight integration and unique mechanisms for extending communication and management, while being more energy efficient than fixed ASIC solutions. Such extended communication enables novel features in the compute architecture like fault-tolerance and function consolidation for standard ECUs, while hybrid FPGA architectures offer support for modular gateway architectures and scalable ECUs in next generation vehicular applications. We also

showed that the tight integration and flexibility of FPGAs can be leveraged for transparent network-level and system-level security schemes that offer adaptability and run-time configurability, with effectively no latency without affecting the protocol and/or application guarantees/deadlines. A functional validation platform has also been presented that enables automated functional validation of clusters of ECUs at real-time and bit-level precision in an integrated environment, which can also be accelerated to reduce validation time. This chapter draws conclusions from the different contributions described in this thesis and outlines areas for future research.

7.1 Summary of Contributions

We have proposed enhancements to in-vehicle communication at the network level as well as to ECUs at the architectural and system levels. An intelligent communication interface enables additional information to be integrated into the standard-format messages to enhance communication capabilities, while managing run-time extensions in a way that is abstracted from the application. This additional information can be used to manage data packing/un-packing or to detect and prevent replay attacks without software intervention. A similar approach is applied to enable network-level security approaches that enable enhanced cross-layer security and network access control. On the architecture front, FPGA features like partial reconfiguration and parallelism enable scalable compute schemes that integrate hardware-level fault-tolerance, functional consolidation, high performance interconnect, and system-level security. The management of such features has been abstracted from the application by integrating them tightly with the (intelligent) communication interface, presenting these as an architecture feature to the application designers. The validation platform enables such enhanced architectures and/or applications to be validated in an integrated environment.

7.1.1 Extensible Network Interfaces

Existing automotive protocols do not provide any mechanisms for integrating extra features like timestamping as a transparent extension. Instead, application designers mast add these at the application layer, resulting in significant overheads since synchronisation must then be applied at this layer too, and the computations typically entail some latency. In Chapter 3, we demonstrated that many such features can be integrated into the network controllers as a configurable extension at the network layer. Extensions includes timestamps, data-segment headers, as well as data packing/re-ordering logic. These extensions are transparent to the application layer and take advantage of the data segment of the transmission protocol, making them compatible with other standard controllers on the network. We showed how these extensions offer a unique mechanism for adding features by extending existing messages, which are then handled at the network interface and abstracted from the application. The applicability of these extensions in an automotive environment was demonstrated with a number of case studies. While we have used FlexRay to demonstrate extended communication, the same method can be applied to evolving networks standards that may replace FlexRay for safety-critical communication.

7.1.2 Enhanced ECU architectures

We demonstrated in Chapter 4 that the extended communication offered by our intelligent interface can be utilised for exchanging information about system state or mode configuration without software intervention. We also showed that advanced features available on FPGAs like partial reconfiguration can enable novel mechanisms for consolidating non-concurrent functions and hardware-level fault-tolerance. The management of low-level operations associated with partial reconfiguration are abstracted from the application and are managed by the extensions on the network interface, making them appear as a feature of the platform to the application designer. We showed how these could result in significantly lower

turnaround and recovery times. We also showed that hybrid FPGA architectures like the Xilinx Zynq provide an efficient platform for compute intensive ECUs and configurable gateway architectures for future vehicular applications and interconnect.

7.1.3 Network and System-level security

Security of vehicular networks has gained focused attention recently as a result of many high profile demonstrations of current vehicles' weaknesses. Approaches based on software and even offloading to hardware incur considerable latency on a per-message basis. Furthermore, these approaches are unable to prevent integration and tampering of messages on the bus by unauthorised (attacker) devices since they function at higher layers.

In Chapter 5, we demonstrated how the extended network interface enables a high-entropy message encryption system with lightweight symmetric ciphers without incurring additional latency for the encrypt/decrypt operation, forming a completely transparent layer of security. We enhanced this approach further by adding tamper protection, which monitors the boot contents for tampering using a one-way hashing function that does not allow network connectivity in the case of modified contents. Finally, by taking advantage of the extra information available at the network layer we showed how integration of unauthorised devices could be prevented with a header-obfuscation scheme, without affecting the reliability and timing guarantees of the protocol. The system-level security approach was demonstrated on a Xilinx Zynq hybrid FPGA architecture in an integrated test setup.

7.1.4 Functional Validation Platform

Finally, in Chapter 6, we introduced a hardware validation platform that enables functional validation of network clusters with support for these extended features. The platform replicates the cluster of ECUs along with their network interfaces

and connectivity on a large FPGA, emulating the cluster with bit-level precision and real-time performance. The platform also allows injection of network-level and system-level errors, with software bindings to control their rate and type, and observation in real-time on a standard PC. The validation process can also be accelerated up to 8× or more without losing precision, for super-real-time evaluation.

7.2 Future Research

The research presented in this thesis aims at advanced architectures and networks for next generation vehicular embedded systems. We have identified numerous possible extensions to the different aspects of the work presented which can be explored in future research.

7.2.1 Enhancing Evolving Time-Triggered Standards

Safety-critical and advanced adaptive vehicular functions currently use ECUs connected over FlexRay due to its higher bandwidth and determinism. More recently, Automotive Ethernet has been proposed as the replacement to FlexRay in future vehicles. While the principles of the two standards are similar (they are both time-triggered), there are some variations, and some features that extend FlexRay are standard in Automotive Ethernet. The methodologies presented in this thesis can be explored for higher bandwidth Ethernet networks that simultaneously handle critical and non-critical communication, presenting alternative opportunities for optimisation.

7.2.2 Distributed Fault-Tolerance

In Chapter 4, we touched upon the possibility of providing a distributed fault-tolerance mechanism using FPGA-based ECUs where a single fault-tolerant unit

can take over the functionality of different faulty ECUs in a pre-defined cluster. This requires complex exchange of information and task/data migration operations which must be handled at the application layer. The distributed architecture also complicates the reconfiguration management at the fault-tolerant unit, which may also have to consider criticality of ECUs in the case of multiple faulty ECUs. Facilitating such information exchange and managing the recovery process in a manner that does not incur significant latency and software overhead is a non-trivial challenge. An interesting possibility would be to utilise techniques similar to dynamic spectrum access in cognitive radios to exploit free bandwidth for such information exchange and management. Extending communication using techniques similar to those in Flexible Data Rate CAN (CAN-FD) is also of interest [44].

7.2.3 Higher layer security management

In Chapter 5, we presented an approach for system-level security that ensures tamper protection and network access mitigation and a mechanism to adapt keys at run-time. Generation and secure exchange of keys between ECUs was not explored in detail, instead left as the function of the special central agent. Also, since the application is tied to the hardware, any in-field updates would have to securely update hardware functionality. Security infrastructure on a secured central agent that handles key generation, key management, as well as a secure pathway to exchange the keys would enable the secure ECUs to adapt to newer threats. The possibility of utilising our central gateway architecture (AEG) as the central agent can be evaluated, using the capabilities of TrustZone to secure the gateway in addition to our proposed architecture. Layering security policies on top of this infrastructure for enabling over-the-air updates can also be explored to creating a truly dynamic security infrastructure for in-vehicle communication.

7.2.4 Hardware in the Loop Optimisation Flow

In Chapter 6, we presented our validation platform that can evaluate a cluster of ECUs at super-real-time rates with bit-level accuracy. We also presented a PCIe based host interface that can enable better observability due to the higher communication bandwidth between the platform and the host PC. Since the platform models the actual communication and computation with bit-level accuracy, it would be possible to perform design space exploration to determine the optimal combination of architecture, features, and network schedule for a given cluster. The iterative operation would explore different architecture combinations, network interconnection schemes, and message/task schedules while evaluating the functionality of the ECUs to determine their control quality and performance. This requires complex management of reconfiguration as well as extension of our software bindings to enable integration with the optimisation loop. This could also be extended to a distributed scheme where the entire vehicular electronics may be evaluated and combinatorially optimised, within a hardware-in-the loop setup over multiple FPGAs. Such optimisation would be extremely useful for determining optimal architectures and ECU combinations for constrained systems like electric vehicles.

7.3 Summary

This thesis has contributed novel approaches for enhanced computation and communication in vehicles using FPGAs, to enable next generation vehicular applications. The focus was on providing architecture-level and communication capabilities in a manner that is abstracted from the software layer or application, with minimal impact on latency and energy consumption, enabling existing automotive applications to be directly ported to enhanced infrastructure. We believe the presented architectures, techniques, and enhancements will be beneficial to automotive embedded systems designers, as well as offering ideas in the general area of

cyber-physical systems. We are also hopeful that this work will encourage adoption of FPGAs for in-vehicle infrastructure, having demonstrated their significant benefits.

Bibliography

[1] Jonas Bereisa. Applications of Microcomputers in Automotive Electronics. *Transactions on Industrial Electronics*, IE-30(2):87–96, May 1983.

[2] Robert N. Charette. This Car Runs on Code. *IEEE Spectrum*, February 2009.

[3] AMPG Body Electronics Systems Engineering Team. White paper: Future Advances in Body Electronics. Technical report, Freescale Semiconductor Inc., 2013.

[4] Hermann Kopetz. Automotive electronics. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 132–140, 1999.

[5] Open Systems and their Interfaces for the Electronics in Motor Vehicles (OSEK). OS223: OSEK/VDX— Operating System Specification.

[6] AUTomotive Open System ARchitecture GbR. AUTOSAR Technical Overview.

[7] Gabriel Leen, Donal Heffernan, and Alan Dunne. Digital networks in the automotive vehicle. *Computing Control Engineering Journal*, 10(6):257–266, 1999.

[8] Nicolas Navet, Yeqiong Song, Francoise Simonot-Lion, and Cedric Wilwert. Trends in Automotive Communication Systems. *Proceedings of the IEEE*, 93 Issue 6:1204–1223, 2005.

[9] Christopher A. Lupini. In-Vehicle Networking Technology for 2010 and Beyond. *SAE2010– World Conference & Exhibition*, 2010.

[10] Christopher Claus, Johannes Zeppenfeld, Florian Müller, and Walter Stechele. Using Partial-Run-Time Reconfigurable Hardware to accelerate Video Processing in Driver Assistance System. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, 2007.

[11] Javier Díaz, Eduardo Ros, Francisco Pelayo, Eva M. Ortigosa, and Sonia Mota. FPGA-based real-time optical-flow system. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(2):274–279, 2006.

[12] Shanker Shreejith, Kizheppatt Vipin, Suhaib A. Fahmy, and Martin Lukasiewycz. An Approach for Redundancy in FlexRay Networks Using FPGA Partial Reconfiguration. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, pages 721–724, 2013.

[13] Shanker Shreejith, Suhaib A. Fahmy, and Martin Lukasiewycz. Reconfigurable Computing in Next-Generation Automotive Networks. *IEEE Embedded Systems Letters*, 5(1):12–15, 2013.

[14] Shanker Shreejith, Suhaib A. Fahmy, and Martin Lukasiewycz. Accelerating Validation of Time-Triggered Automotive Systems on FPGAs. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pages 4–11, 2013.

[15] Kizheppatt Vipin, Shanker Shreejith, Dulitha Gunasekera, Suhaib A. Fahmy, and Nachiket Kapre. System-Level FPGA Device Driver with High-Level Synthesis Support. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pages 128–135, 2013.

[16] Shanker Shreejith and Suhaib A. Fahmy. Enhancing Communication On Automotive Networks Using Data Layer Extensions. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pages 470–473, 2013.

[17] Shanker Shreejith and Suhaib A. Fahmy. Zero Latency Encryption with FPGAs for Secure Time-Triggered Automotive Networks. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pages 256–259, 2014.

[18] Shanker Shreejith and Suhaib A. Fahmy. Extensible FlexRay Communication Controller for FPGA-Based Automotive Systems. *IEEE Transactions on Vehicular Technology*, 64(2):453–465, 2015.

[19] Shanker Shreejith and Suhaib A. Fahmy. Security Aware Network Controllers for Next Generation Automotive Embedded System. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 39.1–39.6, 2015.

[20] Ulrich Abelein, Helmut Lochner, Daniel Hahn, and Stefan Straube. Complexity, quality and robustness– the challenges of tomorrow's automotive electronics. In *Proceedings of the International Conference on Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 870–871, March 2012.

[21] Renesas Electronics Corporation. *R01CP0003EJ0401: Renesas Microcomputer RL78 Family Product Guide*. Renesas Electronics Corporation.

[22] Freescale Semiconductor. *SG187: Automotive Selector Guide*. Freescale Semiconductor, Inc.

[23] Freescale Semiconductor. *MC9S12XF512 Reference Manual*. Freescale Semiconductor, Inc., rev.1.20 edition, November 2010.

[24] Renesas Electronics Corporation. *R8A77860: Renesas SH7786 Series User Manual*. Renesas Electronics Corporation.

[25] International Organisation for Standardisation (ISO). Road Vehicles— Functional Safety— Part 4:Product development at the system level.

[26] International Organisation for Standardisation (ISO). Road Vehicles— Functional Safety— Part 9:Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses.

[27] Gabriel Leen and Donal Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, 2002.

[28] Leandro D'Orazio, Filippo Visintainer, and Marco Darin. Sensor Networks on the Car: State of the Art and Future Challenges. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, 2011.

[29] International Organisation for Standardisation (ISO). Road Vehicles— Low Speed Serial Data Communication— Part 2: Low Speed Controller Area Network, 1994.

[30] International Organisation for Standardisation (ISO). Road Vehicles— Interchange of Digital Information— Controller Area Network for High-Speed Communication, 1994.

[31] Robert Bosch GmbH. CAN Specification, Version 2.0, 1991.

[32] Manuel Barranco, Guillermo Rodriguez-Navas, Julian Proenza, and Luís Almeida. CANcentrate: an active star topology for CAN networks. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 219–228, 2004.

[33] Giuseppe Buja, Alberto Zuccollo, and Juan Pimentel. Overcoming babbling-idiot failures in the FlexCAN architecture: a simple bus-guardian. In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 461–468, 2005.

[34] Martin Lukasiewycz, Michael Glaß, Christian Haubelt, Jürgen Teich, Richard Regler, and Bardo Lang. Concurrent topology and routing optimization in automotive network integration. In *Proceeding of the ACM/IEEE Design Automation Conference (DAC)*, pages 626–629, 2008.

[35] Khawar M. Zuberi and Kang G. Shin. Non-preemptive scheduling of messages on Controller Area Network for real-time control applications. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 240–249, 1995.

[36] Khawar M. Zuberi and Kang G. Shin. Design and implementation of efficient message scheduling for Controller Area Network. *IEEE Transactions on Computers*, 49(2):182–188, 2000.

[37] K. Anwar and Z. A. Khan. Dynamic priority based message scheduling on Controller Area Network. In *Proceedings of the International Conference on Electrical Engineering (ICEE)*, pages 1–6, 2007.

[38] Hüseyin Aysan, Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Efficient fault tolerant scheduling on Controller Area Network (CAN). In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2010.

[39] Xilinx Inc. *DS265: Xilinx CAN v3.2 Product Specification*, April 2010.

[40] Altera. *C_CAN User Manual Revision 1.2*, June 2000.

[41] Tobias Ziermann, Stefan Wildermann, and Jürgen Teich. CAN+: A new backward-compatible Controller Area Network (CAN) protocol with up to 16x higher data rates. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, 2009.

[42] Gianluca Cena and Adriano Valenzano. Overclocking of Controller Area Networks. *IEE Electronics Letters*, 35(22):1923–1925, 1999.

[43] Imran Sheikh and Michael Short. Improving information throughput in CAN networks: Implementing the dual-speed approach. In *Proceedings of the International Workshop on Real-Time Networks (RTN)*, 2009.

[44] Robert Bosch GmbH. CAN with Flexible Data-Rate 1.0, 2012.

[45] Andreas Kern, Thilo Streichert, and Jürgen Teich. An automated data structure migration concept– From CAN to Ethernet/IP in automotive embedded systems (CANoverIP). In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, pages 1–6, 2011.

[46] International Organisation for Standardisation (ISO). Road Vehicles— Controller Area Network (CAN)— Part 4: Time-Triggered Communication, 2000.

[47] Klaus Schmidt and Ece Guran Schmidt. Systematic Message Schedule Construction for Time-Triggered CAN. *IEEE Transactions on Vehicular Technology*, 56(6):3431–3441, 2007.

[48] FlexRay Consortium. FlexRay Requirements Specification, Version 2.1, 2005.

[49] Feng Luo, Zhiqi Chen, Juexiao Chen, and Zechang Sun. Research on FlexRay communication system. In *Proceedings of the IEEE Vehicle Power and Propulsion Conference (VPPC)*, pages 1–5, 2008.

[50] Josef Berwanger, Anton Schedl, and Martin Peller. BMW– First Series Cars with FlexRay in 2006. *Hanser Automotive Magazine, FlexRay Special Edition*, pages 6–8, 2005.

[51] Robert Shaw and Brendan Jackman. An Introduction to FlexRay as an Industrial Network. In *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE)*, 2008.

[52] Jens Kötz and Stefan Poledna. Making FlexRay a Reality in a Premium Car. In *Proceedings of the SAE International*, 2008.

[53] FlexRay Consortium. FlexRay Communications System, Protocol Specification Version 2.1 Revision A, December 2005.

[54] Christoph Heller, Josef Schalk, Stefan Schneele, and Reinhard Reichel. Approaching the Limits of FlexRay. In *Proceedings of the IEEE International*

Symposium on Network Computing and Applications (NCA), pages 205–210, 2008.

[55] ASAM e.V. FIBEX– Field Bus Exchange Format Version 3.1, January 2009.

[56] Martin Lukasiewycz, Michael Glaß, Jürgen Teich, and Paul Milbredt. FlexRay Schedule Optimization of the Static Segment. In *Proceedings of the IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES + ISSS)*, 2009.

[57] Klaus Schmidt and Ece Guran Schmidt. Message Scheduling for the FlexRay Protocol: The Static Segment. *IEEE Transactions on Vehicular Technology*, 58, No. 5, 2009.

[58] Ece Guran Schmidt and Klaus Schmidt. Message Scheduling for the FlexRay Protocol: The Dynamic Segment. *IEEE Transactions on Vehicular Technology*, 58, No. 5, 2009.

[59] Jimmy Jessen Nielsen, Amen Hamdan, and Hans-Peter Schwefel. Markov Chain-based Performance Evaluation of FlexRay Dynamic Segment. In *Proceedings of the International Workshop on Real Time Networks*, 2007.

[60] Bonjun Kim and Kiejin Park. Probabilistic Delay Model of Dynamic Message Frame in FlexRay Protocol. *IEEE Transaction on Consumer Electronics*, 55, Issue 1:77–82, Feb. 2009.

[61] Xuewen He, Qiang Wang, and Zhenli Zhang. A Survey of Study of FlexRay Systems for Automotive Net. In *Proceedings of the International Conference on Electronic & Mechanical Engineering and Information Technology (EMEIT)*, 2011.

[62] Robert Bosch GmbH. *Product Information : E-Ray IP Module*, July 2009.

[63] IPextreme Inc. *FRCC2100 : Product Brochure, Freescale FlexRay Communications Controller Core*. IPextreme Inc.

[64] T. Forest, Alberto Ferrari, G. Audisio, Marco Sabatini, A. Sangiovanni-Vincentelli, and Marco Di Natale. Physical Architectures of Automotive Systems. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, pages 391–395, 2008.

[65] Jayant Y. Hande, Milind Khanapurkar, and Preeti Bajaj. Approach for VHDL and FPGA Implementation of Communication Controller of FlexRay Controller. In *Proceedings of the International Conference on Emerging Trends in Engineering and Technology (ICETET)*, 2009.

[66] Yi-Nan Xu, Yong-Eun Kim, Kyung-Ju Cho, Jin-Gyun Chung, and Myoung-Seob Lim. Implementation of FlexRay Communication Controller Protocol with Application to a Robot System. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2008.

[67] Yi-Nan Xu, I. G. Jang, Y. E. Kim, J. G. Chung, and Sung-Chul Lee. Implementation of FlexRay protocol with an automotive application. In *Proceedings of the International SoC Design Conference (ISOCC)*, 2008.

[68] P. M. Szecowka and M. A. Swiderski. On Hardware Implementation of FlexRay Bus Guardian Module. In *Proceedings of the International Conference on Mixed Design of Integrated Circuits and Systems (MIXDES)*, 2007.

[69] Gang-Neng Sung, Chun-Ying Juan, and Chua-Chin Wang. Bus Guardian Design for Automobile Networking ECU Nodes Compliant with FlexRay Standards. In *Proceedings of the International Symposium on Consumer Electronics*, 2008.

[70] Christoph Schmutzler, Abdallah Lakhtel, Martin Simons, and Jürgen Becker. Increasing energy efficiency of automotive E/E-architectures with Intelligent Communication Controllers for FlexRay. In *Proceedings of the International Symposium on System on Chip (SoC)*, 2011.

[71] AUTOSAR. Specification of FlexRay Interface, v 3.2.0, October 2010.

[72] Paul Milbredt, Bart Vermeulen, Gökhan Tabanoglu, and Martin Lukasiewycz. Switched FlexRay: Increasing the Effective Bandwidth and Safety of FlexRay Networks. In *Proceedings of the Emerging Technologies and Factory Automation (ETFA)*, 2010.

[73] Thijs Schenkelaars, Bart Vermeulen, and Kees Goossens. Optimal scheduling of switched FlexRay networks. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, 2011.

[74] Martin Lukasiewycz, Samarjit Chakraborty, and Paul Milbredt. FlexRay Switch Scheduling – A Networking Concept for Electric Vehicles. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, 2011.

[75] Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. Timing analysis of the FlexRay communication protocol. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 203–216, 2006.

[76] Jin Ben, Bian Yongming, and Li Anhu. A method for response time computation in FlexRay communication system. In *Proceedings of the IEEE International Conference on Intelligent Computing and Intelligent Systems,(ICIS)*, volume 3, pages 47–51, 2009.

[77] Andrei Hagiescu, Unmesh D. Bordoloi, Samarjit Chakraborty, Prahladavaradan Sampath, P. Vignesh V. Ganesan, and Sethu Ramesh. Performance Analysis of FlexRay-based ECU Networks. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2007.

[78] Matthias Heinz, Verena Hoss, and Klaus D. Müller-Glaser. Physical Layer Extraction of FlexRay Configuration Parameters. In *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*, pages 173–180, 2009.

[79] Arkadeb Ghosal, Haibo Zeng, Marco Di Natale, and Yakov Ben-Haim. Computing robustness of FlexRay schedules to uncertainties in design parameters. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, pages 550–555, 2010.

[80] Inseok Park and Myoungho Sunwoo. FlexRay Network Parameter Optimization Method for Automotive Applications. *IEEE Transactions on Industrial Electronics*, 58(4):1449–1459, 2011.

[81] Reinhard Schneider, Dip Goswami, Sohaib Zafar, Samarjit Chakraborty, and Martin Lukasiewycz. Constraint-driven synthesis and tool-support for FlexRay-based automotive control systems. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS*, pages 139–148, 2011.

[82] Stephan Reichelt, Oliver Scheickl, and Gökhan Tabanoglu. The influence of real-time constraints on the design of FlexRay-based systems. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, pages 858–863, 2009.

[83] Reinhard Schneider, Unmesh Bordoloi, Dip Goswami, and Samarjit Chakraborty. Optimized Schedule Synthesis under Real-Time Constraints for the Dynamic Segment of FlexRay. In *Proceedings of the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 31–38, 2010.

[84] Reinhard Schneider, Dip Goswami, Samarjit Chakraborty, Unmesh Bordoloi, Petru Eles, and Zebo Peng. On the quantification of sustainability and extensibility of FlexRay schedules. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 375–380, 2011.

[85] Wenchao Li, Marco Di Natale, Wei Zheng, Paolo Giusto, Alberto Sangiovanni-Vincentelli, and Sanjit A. Seshia. Optimizations of an

application-level protocol for enhanced dependability in FlexRay. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, pages 1076–1081, 2009.

[86] Paul Milbredt, Andreas Steininger, and Martin Horauer. Automated Testing of FlexRay Clusters for System Inconsistencies in Automotive Networks. In *Proceedings of the IEEE International Symposium on Electronic Design, Test and Applications (DELTA)*, pages 533–538, 2008.

[87] Martin Horauer, Oliver Praprotnik, Martin Zauner, Roland Holler, and Paul Milbredt. A Test Tool for FlexRay-based Embedded Systems. In *Proceedings of the International Symposium on Industrial Embedded Systems (SIES)*, pages 349–352, 2007.

[88] Kuen-Long Leu, Yung-Yuan Chen, Chin-Long Wey, and Jwu-E Chen. A verification flow for FlexRay communication robustness compliant with IEC 61508. In *Proceedings of the International Conference on Industrial Mechatronics and Automation (ICIMA)*, volume 1, pages 228–231, 2010.

[89] Paul Milbredt, Martin Horauer, and Andreas Steininger. An investigation of the clique problem in FlexRay. In *Proceedings of the International Symposium on Industrial Embedded Systems (SIES)*, pages 200–207, 2008.

[90] Kay Klobedanz, Gilles B. Defo, Henning Zabel, Wolfgang Mueller, and Yuan Zhi. Task migration for fault-tolerant FlexRay networks. In *Distributed, Parallel and Biologically Inspired Systems*, pages 55–65. Springer, 2010.

[91] Kay Klobedanz, Gilles B. Defo, Wolfgang Mueller, and Timo Kerstan. Distributed coordination of task migration for fault-tolerant FlexRay networks. In *Proceedings of the International Symposium on Industrial Embedded Systems (SIES)*, pages 79–87, 2010.

[92] Kay Klobedanz, Andreas Koenig, Wolfgang Mueller, and Achim Rettberg. Self-Reconfiguration for Fault-Tolerant FlexRay Networks. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 207–216, 2011.

[93] Hung-Manh Pham, Sebastien Pillement, and Didier Demigny. Reconfigurable ECU communications in AUTOSAR Environment. In *Proceedings of the International Conference on Intelligent Transport Systems Telecommunications (ITST)*, pages 581–585, 2009.

[94] Jianmin Duan, Liang Zhu, and Yongchuan Yu. Research on FlexRay communication of Steering-by-Wire system. In *Proceedings of the IEEE Intelligent Vehicles Symposium*, pages 824–828, 2009.

[95] S. P. Mane, S. S. Sonavane, and P. P. Shingare. A Review on Steer-By-Wire System Using FlexRay. In *Proceedings of the International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE)*, 2011.

[96] IEEE. Std 802.3– 2005 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. *IEEE Std 802.3-2005 (Revision of IEEE Std 802.3-2002 including all approved amendments)*, Section 1:1–594, 2005.

[97] Wilfried Steiner. *TTEthernet Specification.* TTTech Computertechnik AG, Vienna, Austria, November 2008.

[98] SAE International. AS6802– Time-Triggered Ethernet, November 2011.

[99] TTTech Computertechnik AG. TTEthernet A Powerful Network Solution for All Purposes. Technical report, 2010.

[100] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The Time-Triggered Ethernet (TTE) design. In *Proceedings of the International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 22–33, 2005.

[101] IEEE Standards Association. Ethernet– EtherType Field Public Assignments.

[102] Hermann Kopetz, Astrit Ademaj, and Alexander Hanzlik. Integration of internal and external clock synchronization by the combination of clock-state

and clock-rate correction in fault-tolerant distributed systems. In *Proceedings of Real-Time Systems Symposium*, pages 415–425, 2004.

[103] Wilfried Steiner. *Startup and Recovery of Fault-Tolerant Time-Triggered Communication*. PhD thesis, Vienna University of Technology, Real-Time Systems Group, Vienna, Austria, 2004.

[104] Klaus Steinhammer, Petr Grillinger, Astrit Ademaj, and Hermann Kopetz. A Time-Triggered Ethernet (TTE) Switch. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, pages 1–6, 2006.

[105] Astrit Ademaj and Hermann Kopetz. Time-Triggered Ethernet and IEEE 1588 Clock Synchronization. In *Proceedings of the IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS)*, pages 41–43, 2007.

[106] Till Steinbach, Franz Korf, and Thomas C. Schmidt. Comparing Time-Triggered Ethernet with FlexRay: An evaluation of competing approaches to real-time for in-vehicle networks. In *Proceedings of the International Workshop on Factory Communication Systems (WFCS)*, 2010.

[107] BroadCom Inc. *Product Brief : BCM89810 Single-port BroadR-Reach automotive Ethernet transceiver*, October 2011.

[108] MOST Cooperation. MOST Specification Rev. 3.0 E2, July 2010.

[109] Wang Jian-guo and Yang Bing. Realization of audio transmission node for vehicular MOST network. In *Proceedings of the International Conference on Computer, Mechatronics, Control and Electronic Engineering (CMCE)*, pages 193–195, 2010.

[110] Mu-Youl Lee, Sung-Moon Chung, and Hyun-Wook Jin. Automotive network gateway to control electronic units through MOST network. In *Proceedings of the International Conference on Consumer Electronics (ICCE) Digest of Technical Papers*, pages 309–310, 2010.

[111] Jürgen Becker, Adam Donlin, and Michael Hübner. New Tool Support and Architectures in Adaptive Reconfigurable Computing. In *Proceedings of IFIP International Conference on Very Large Scale Integration (VLSI–SoC)*, 2007.

[112] Actel. *ProASIC3 Flash FPGAs*, 2010.

[113] Mike Peattie. Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode. Technical report, Xilinx Inc., 2009.

[114] Xilinx Inc. *DS586: LogiCORE IP XPS HWICAP*, 2010.

[115] Xilinx Inc. *XAPP151: Virtex Series Configuration Architecture User Guide*, 2004.

[116] Xilinx Inc. *DS031:Virtex-II Platform FPGAs*, 2003.

[117] Xilinx Inc. *DS083: Virtex-II Pro and Virtex-II Pro-X Platform FPGAs*, 2011.

[118] Kizheppatt Vipin and Suhaib A. Fahmy. Efficient Region Allocation for Adaptive Partial Reconfiguration. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 1–6, 2011.

[119] Xilinx Inc. *UG632: PlanAhead User Guide*. Xilinx Inc., 2010.

[120] Xilinx Inc. *UG909: Vivado Design Suite User Guide : Partial Reconfiguration*. Xilinx Inc., 2013.

[121] Naoya Chujo. Fail-safe ECU System Using Dynamic Reconfiguration of FPGA. *R & D Review of Toyota CRDL*, 37:54–60, 2002.

[122] Sébastien Le Beux, Philippe Marquet, Ouassila Labbani, and Jean-Luc Dekeyser. FPGA Implementation of Embedded Cruise Control and Anti-Collision Radar. In *Proceedings of the EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 280–287, 2006.

[123] B. Magaz and M. L. Bencheikh. An efficient FPGA implementation of the OS-CFAR processor. In *Proceedings of the International Radar Symposium*, pages 1–4, 2008.

[124] Felix Eberli. Automotive embedded driver assistance: A real-time low-power FPGA stereo engine using semi-global matching. In *Proceedings of the International Conference on Computer Design (ICCD)*, 2010.

[125] Naim Harb, Smail Niar, Mazen A. R. Saghir, Yassin El Hillali, and Rabie Ben Atitallah. Dynamically reconfigurable architecture for a driver assistant system. In *Proceedings of the IEEE Symposium on Application Specific Processors (SASP)*, pages 62–65, 2011.

[126] Oliver Sander, Benjamin Glas, Christoph Roth, Jürgen Becker, and Klaus D. Müller-Glaser. Design of a Vehicle-to-Vehicle communication system on reconfigurable hardware. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 14–21, 2009.

[127] Woong Cho, Kyeong-Soo Han, Hyun Kyun Choi, and Hyun Seo Oh. Realization of anti-collision warning application using V2V communication. In *Proceedings of the IEEE Vehicular Networking Conference (VNC)*, pages 1–5, 2009.

[128] Angelos Amditis, George C. Kiokes, and Nikolaos K. Uzunoglu. An FPGA-based physical layer implementation for vehicle-to-vehicle (V2V) communication. In *Proceedings of the European Signal Processing Conference (EUSIPCO)*, 2010.

[129] Sang-woo Chang, Jin-soo Jung, Jin Cha, and Sang-sun Lee. Implementation of DSRC Mobile MAC for VANET. In *Proceedings of the International Conference on Advanced Communication Technology (ICACT)*, pages 1502–1505, 2011.

[130] Francisco Fons and Mariano Fons. FPGA-based Automotive ECU Design Addresses AUTOSAR and ISO 26262 Standards. *Xcell journal*, Issue 78:20–31, 2012.

[131] Luca Sterpone and Massimo Violante. A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs. *IEEE Transactions on Nuclear Science,*, 52(6):2217–2223, 2005.

[132] Luca Sterpone and Massimo Violante. Analysis of the robustness of the TMR architecture in SRAM-based FPGAs. *IEEE Transactions on Nuclear Science*, 52(5):1545–1549, 2005.

[133] Brian Pratt, Michael Caffrey, James F. Carroll, Paul Graham, Keith Morgan, and Michael Wirthlin. Fine-grain SEU mitigation for FPGAs using Partial TMR. In *Proceedings of the European Conference on Radiation and Its Effects on Components and Systems,(RADECS)*, pages 1–8, 2007.

[134] Brian Pratt, Michael Caffrey, James F. Carroll, Paul Graham, Keith Morgan, and Michael Wirthlin. Fine-Grain SEU Mitigation for FPGAs Using Partial TMR. *IEEE Transactions on Nuclear Science*, 55(4):2274–2280, 2008.

[135] Cristiana Bolchini, Antonio Miele, and Marco D. Santambrogio. TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs. In *Proceedings of the International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 87–95, 2007.

[136] José Rodrigo Azambuja, Conrado Pilotto, and Fernanda Lima Kastensmidt. Mitigating soft errors in SRAM-based FPGAs by using large grain TMR with selective partial reconfiguration. In *Proceedings of the European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pages 288–293, 2008.

[137] Keith S. Morgan, Daniel L. McMurtrey, Brian H. Pratt, and Michael J. Wirthlin. A Comparison of TMR with alternative Fault-Tolerant Design Techniques for FPGAs. *IEEE Transactions on Nuclear Science*, 54(6):2065–2072, 2007.

[138] Sébastien Varrier, Ruben Morales-Menendez, Jorge De-Jesus Lozoya-Santos, Diana Hernandez, John Martinez Molina, and Damien Koenig. Fault detection in automotive semi-active suspension: Experimental results. Technical report, SAE Technical Paper, 2013.

[139] Matthias Müller, Axel Braun, Joachim Gerlach, Wolfgang Rosenstiel, Dennis Nienhüser, J. Marius Zöllner, and Oliver Bringmann. Design of an automotive traffic sign recognition system targeting a multi-core SoC implementation. In *Proceedings of the Design, Automation Test in Europe Conference (DATE)*, pages 532–537, March 2010.

[140] Willie Tsui, Mohamed Slim Masmoudi, F. Karray, Insop Song, and M. Masmoudi. Soft-computing-based embedded design of an intelligent wall/lane-following vehicle. *IEEE Transactions on Mechatronics*, 13(1):125–135, Feb 2008.

[141] Sergio Saponara, Nicola EL Insalata, Tony Bacchillone, Esa Petri, Iacopo Del Corona, and Luca Fanucci. Hardware/Software FPGA-based Network Emulator for High-speed On-board Communications. In *Proceedings of the Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 353–359, 2008.

[142] Tzuu-Hseng S. Li, Shih-Jie Chang, and Yi-Xiang Chen. Implementation of Human-like Driving Skills by Autonomous Fuzzy Behavior Control on an FPGA-Based Car-Like Mobile Robot. *IEEE Transactions on Industrial Electronics*, 50:867–880, 2003.

[143] Shehryar Shaheen, Donal Heffernan, and Gabriel Leen. A gateway for time-triggered control networks. *Microprocessors and Microsystems*, 31(1):38–50, 2007.

[144] Oliver Sander, Benjamin Glas, Christoph Roth, Jürgen Becker, and Klaus D. Müller-Glaser. Priority-based packet communication on a bus-shaped structure for FPGA-systems. In *Proceedings of the Internation Conference on*

Design, Automation & Test in Europe Conference (DATE), pages 178–183, 2009.

[145] Georg Maier, Alexander Paier, and Christoph F. Mecklenbrauker. Performance evaluation of IEEE 802.11p infrastructure-to-vehicle real-world measurements with receive diversity. In *Proceedings of the International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1113–1118, 2012.

[146] Oliver Sander, Benjamin Glas, Christoph Roth, Jürgen Becker, and Klaus D. Müller-Glaser. Testing of an FPGA based C2X-Communication prototype with a model based traffic generation. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, pages 68–71, 2009.

[147] Tiago M. Fernández-Caramés, Miguel González-López, and Luis Castedo. FPGA-based vehicular channel emulator for real-time performance evaluation of IEEE 802.11p transceivers. *EURASIP Journal on Wireless Communications and Networking*, 2010:4, 2010.

[148] Jan Sobotka and Jiri Novak. FlexRay controller with special testing capabilities. In *Proceedings of the Conference on Applied Electronics (AE)*, pages 269–272, 2012.

[149] Oliver Sander, Jürgen Becker, Michael Hübner, Michael Dreschmann, Jürgen Luka, Matthias Traub, and Thomas Weber. Modular system concept for a FPGA-based Automotive Gateway. *VDI BERICHTE*, 2000:221, 2007.

[150] Oliver Sander, Michael Hübner, Jürgen Becker, and Matthias Traub. Reducing latency times by accelerated routing mechanisms for an FPGA gateway in the automotive domain. In *Proceedings of the International Conference on ICECE Technology (FPT)*, pages 97–104, 2008.

[151] Andreas Puhm, Peter Rössler, Marcus Wimmer, Rafael Swierczek, and Peter Balog. Development of a flexible gateway platform for automotive networks. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 456–459, 2008.

[152] Rob Miller Ishtiaq Rouf, Hossen Mustafa, Sangho Oh Travis Taylor, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskar. Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study. In *Proceedings of the USENIX Security Symposium*, 2010.

[153] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental Security Analysis of a Modern Automobile. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 447–462, May 2010.

[154] Michael Müter and Felix C. Freiling. Model-based security evaluation of vehicular networking architectures. In *Proceedings of the International Conference on Networks (ICN)*, pages 185–193, 2010.

[155] Olaf Henniger, Ludovic Apvrille, Andreas Fuchs, Yves Roudier, Alastair Ruddle, and Benjamin Weyl. Security requirements for automotive on-board networks. In *Proceedings of the International Conference on Intelligent Transport System Telecommunications (ITST)*, 2009.

[156] Andre Groll and Christoph Ruland. Secure and Authentic Communication on Existing In-Vehicle Networks. In *Proceedings of the IEEE Intelligent Vehicles Symposium (IVS)*, 2009.

[157] Hendrik Schweppe and Yves Roudier. Security and privacy for in-vehicle networks. In *Proceedings of the International Workshop on Vehicular Communications, Sensing, and Computing (VCSC)*, pages 12–17, 2012.

[158] Jun Luo and Jean-Pierre Hubaux. A Survey of Inter-Vehicle Communication. Technical Report IC/2004/24, School of Computer and Communication Sciences, EPFL, CH-1015 Lausanne, Switzerland, 2004.

[159] P. Caballero-Gil. *Mobile Ad-Hoc Networks: Applications*, chapter 4 : Security Issues in Vehicular Ad Hoc Networks. InTech, 2011.

[160] Vineetha Paruchuri. Inter-vehicular communications: Security and reliability issues. In *Proceedings of the International Conference on ICT Convergence (ICTC)*, pages 737–741, 2011.

[161] Pierre Kleberger, Tomas Olovsson, and Erland Jonsson. Security aspects of the in-vehicle network in the connected car. In *Proceedings of the IEEE Intelligent Vehicles Symposium (IV)*, pages 528–533, 2011.

[162] Olaf Henniger, Alastair Ruddle, Hervé Seudié, Benjamin Weyl, Marko Wolf, and Thomas Wollinger. Securing Vehicular On-Board IT Systems: The EVITA Project. In *Proceedings of the VDI/VW Automotive Security Conference*, 2009.

[163] Chanbok Jeong. Cryptography Engine Design for IEEE 1609.2 WAVE Secure Vehicle Communication using FPGA, 2015.

[164] Imran Sheikh, Musharraf Hanif, and Michael Short. Improving information throughput and transmission predictability in Controller Area Networks. In *Proceedings of the International Symposium on Industrial Electronics (ISIE)*, pages 1736–1741, 2010.

[165] Infineon Technologies AG. *SAK-CIC310-OSMX2HT, FlexRay Communication Controller Data Sheet*, June 2007.

[166] Jean Saad, Amer Baghdadi, and Frantz Bodereau. FPGA-based Radar Signal Processing for Automotive Driver Assistance System. In *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 196–199, 2009.

[167] Greg F. EE170 : Estimating Power for the ADSP-TS201S TigerSHARC Processors. Technical report, Analog Devices, 2006.

[168] Xilinx Inc. *DS280: OPB HWICAP*, July 2006.

[169] Christopher Claus, Florian H. Müller, Johannes Zeppenfeld, and Walter Stechele. A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration. In *Proceedings of IEEE International Symposium on*

Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2007.

[170] Ming Liu, Wolfgang Kuehn, Zhonghai Lu, and Axel Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploitation. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, 2009.

[171] Kizheppatt Vipin and Suhaib A. Fahmy. A High Speed Open Source Controller for FPGA Partial Reconfiguration. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, 2012.

[172] Willibald Prestl, Thomas Sauer, Joachim Steinle, and Oliver Tschernoster. The BMW active cruise control ACC. In *SAE Paper 2000–01–0344*, 2000.

[173] Ardalan Vahidi and Azim Eskandarian. Research advances in intelligent collision avoidance and adaptive cruise control. *Transactions on Intelligent Transportation Systems (ITS)*, 4(3):143–153, Sept 2003.

[174] Julien Henaut, Daniela Dragomirescu, and Robert Plana. FPGA based high date rate radio interfaces for aerospace wireless sensor systems. In *Proceedings of the International Conference on Systems (ICONS)*, pages 173–178, March 2009.

[175] Brock J. LaMeres and Clint Gauer. Dynamic reconfigurable computing architecture for aerospace applications. In *Proceedings of the IEEE Aerospace Conference*, pages 1–6, March 2009.

[176] Jörg Lotze, Suhaib A. Fahmy, Juanjo Noguera, Barıs Özgül, Linda Doyle, and Robert Esser. Development Framework for Implementing FPGA-Based Cognitive Network Nodes. In *Proceedings of the IEEE Global Communications Conference*, 2009.

[177] Muhammad Atif Tahir, Ahmed Bouridane, and Fatih Kurugollu. An FPGA based coprocessor for GLCM and haralick texture features and their application in prostate cancer classification. *Transactions on Analog Integrated Circuits and Signal Processing*, 43(2):205–215, 2005.

[178] Jason H. Anderson and Farid N. Najm. A novel low-power FPGA routing switch. In *Proceedings of the Custom Integrated Circuits Conference*, pages 719–722, 2004.

[179] Daniel Kohler. A practical implementation of an IEEE1588 supporting Ethernet switch. In *Proceedings of the International Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS)*, pages 134–137, 2007.

[180] Gonzalo Carvajal, Miguel Figueroa, Robert Trausmuth, and Sebastian Fischmeister. Atacama: An Open FPGA-based Platform for Mixed-Criticality Communication in Multi-segmented Ethernet Networks. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 121–128, 2013.

[181] David V. Schuehler and John W. Lockwood. A modular system for FPGA-based TCP flow processing in high-speed networks. In *Field Programmable Logic and Application*, pages 301–310. Springer, 2004.

[182] Brad L. Hutchings, Rob Franklin, and Daniel Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–120, 2002.

[183] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion detection system. In *Proceedings of the International Conference on Field Programmable Logic and Application (FPL)*, pages 880–889, 2003.

[184] Insop Song, Keith Gowan, Jason Nery, Henrick Han, Tony Sheng, Howard Li, and Fakhreddine Karray. Intelligent Parking System Design Using FPGA.

In *Proceedings of the International Conference on Field Programmable Logic and Applications, (FPL)*, pages 1–6, 2006.

[185] Shanker Shreejith, Bezborah Anshuman, and Suhaib A. Fahmy. Accelerated Artificial Neural Networks on FPGA for Fault Detection in Automotive Systems. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, pages 37–42, 2016.

[186] Xilinx Inc. *UG585: Zynq-7000 All Programmable SoC Technical Reference Manual*, Mar. 2013.

[187] Kizheppatt Vipin, Shanker Shreejith, Suhaib A. Fahmy, and Arvind Easwaran. Mapping Time-Critical Safety-Critical Cyber Physical Systems to Hybrid FPGAs. In *Proceedings of the International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 31–36, 2014.

[188] Philipp Mundhenk, Florian Sagstetter, Sebastian Steinhorst, Martin Lukasiewycz, and Samarjit Chakraborty. Policy-based Message Scheduling Using FlexRay. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 19:1–19:10, 2014.

[189] Hyung-Taek Lim, Benjamin Krebs, Lars Völker, and Peter Zahrer. Performance evaluation of the inter-domain communication in a switched Ethernet based in-car network. In *Proceedings of the Conference on Local Computer Networks (LCN)*, pages 101–108, 2011.

[190] Jin Ho Kim, Suk-Hyun Seo, Nguyen Tien Hai, Bo Mu Cheon, Young Seo Lee, and Jae Wook Jeon. Gateway framework for in-vehicle networks based on CAN, FlexRay and Ethernet. *IEEE Transactions on Vehicular Technology (TVT)*, 64(10):4472–4486, 2015.

[191] Jae-Sung Yang, Suk Lee, Kyung Chang Lee, and Man Ho Kim. Design of FlexRay-CAN gateway using node mapping method for in-vehicle networking systems. In *Proceedings of the International Conference on Control, Automation and Systems (ICCAS)*, pages 146–148, 2011.

[192] Klaus-Robert Müller, Till Steinbach, Franz Korf, and Thomas C Schmidt. A real-time Ethernet prototype platform for automotive applications. In *Proceedings of the International Conference on Consumer Electronics (ICCE)*, pages 221–225. IEEE, 2011.

[193] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the USENIX Security Symposium*, 2011.

[194] Dennis K. Nilsson, Phu H. Phung, and Ulf E. Larson. Vehicle ECU classification based on safety-security characteristics. In *Proceedings of the Conference on Road Transport Information and Control*, 2008.

[195] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[196] Ivan Studnia, Vincent Nicomette, Eric Alata, Yves Deswarte, Mohamed Kaaniche, and Youssef Laarouchi. Survey on security threats and protection mechanisms in embedded automotive networks. In *Proceedings of the Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2013.

[197] Philipp Mundhenk, Sebastian Steinhorst, Martin Lukasiewycz, Suhaib A. Fahmy, and Samarjit Chakraborty. Security analysis of automotive architectures using probabilistic model checking. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 38:1–38:6, 2015.

[198] Gabriel Pedroza, Muhammad Sabir Idrees, Ludovic Apvrille, and Yves Roudier. A Formal Methodology Applied to Secure Over-The-Air Automotive Applications. In *Proceedings of the Vehicular Technology Conference (VTC Fall)*, pages 1–5, 2011.

[199] Hendrik Schweppe, Benjamin Weyl, Yves Roudier, Muhammad Sabir Idrees, Timo Gendrullis, Marko Wolf, Gabriel Serme, Santana Anderson

De Oliveira, Herve Grall, Mario Sudholt, et al. Securing Car2X Applications with effective Hardware-Software Co-Design for Vehicular On-Board Networks. *VDI Automotive Security*, 27, 2011.

[200] Tim Leinmüller, Levente Buttyan, Jean-Pierre Hubaux, Frank Kargl, Rainer Kroh, Panos Papadimitratos, Maxim Raya, and Elmar Schoch. SEVECOM – Secure Vehicle Communication. In *Proceedings of the IST Mobile and Wireless Communication Summit*, 2006.

[201] Marko Wolf and Timo Gendrullis. Design, Implementation, and Evaluation of a Vehicular Hardware Security Module. In *Proceedings of the International Conference on Information Security and Cryptology (ICISC)*, pages 302–318. Springer, 2012.

[202] John Giffin, Rachel Greenstadt, Peter Litwack, and Richard Tibbetts. Covert messaging through TCP timestamps. In *Proceedings of the International Workshop on Privacy Enhancing Technologies*, 2003.

[203] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[204] Xin Xin, J. Kaps, and Kris Gaj. A configurable ring-oscillator-based PUF for Xilinx FPGAs. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 651–657, 2011.

[205] Philipp Mundhenk, Sebastian Steinhorst, Martin Lukasiewycz, Suhaib A. Fahmy, and Samarjit Chakraborty. Lightweight Authentication for Secure Automotive Networks. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, 2015.

[206] Elliot Barlas and Tevfik Bultan. NetStub: A Framework for Verification of Distributed Java Applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 24–33, 2007.

[207] Sebastian Uchitel. Partial behaviour modelling: Foundations for incremental and iterative model-based software engineering. In *Formal Methods: Foundations and Applications*, pages 17–22. Springer, 2009.

[208] Seyedehmehrnaz Mireslami and Behrouz H. Far. A system-level approach for model-based verification of distributed software systems. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2545–2550, Oct 2013.

[209] K. Mani Chandy, Brian Go, Sayan Mitra, Concetta Pilotto, and Jerome White. Verification of distributed systems with local–global predicates. *Formal aspects of computing*, 23(5):649–679, 2011.

[210] Hristo Nikolov, Mark Thompson, Todor Stefanov, Andy Pimentel, Simon Polstra, Raj Bose, Claudiu Zissulescu, and Ed Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 574–579, 2008.

[211] Rainer Ohlendorf, Thomas Wild, Michael Meitinger, Holm Rauchfuss, and Andreas Herkersdorf. Simulated and measured performance evaluation of RISC-based SoC platforms in network processing applications. *Journal of Systems Architecture*, 53(10):703–718, 2007.

[212] Michael Gschwind, Valentina Salapura, and Dietmar Maurer. FPGA prototyping of a RISC processor core for embedded applications. *IEEE Transactions on VLSI Systems*, 9(2):241–250, 2001.

[213] Juan J. Rodriguez-Andina, María José Moure, and María Dolores Valdes. Features, design tools, and application domains of FPGAs. *IEEE Transactions on Industrial Electronics*, 54(4):1810–1823, 2007.

[214] Ubaid R. Khan, Henry L. Owen, and Joseph LA Hughes. FPGA architectures for ASIC hardware emulators. In *Proceedings of the ASIC Conference and Exhibit*, pages 336–340, 1993.

[215] Liu Jianhua, Zhu Ming, Bian Jinian, and Xue Hongxi. A debug sub-system for embedded-system co-verification. In *Proceedings of the Conference on ASIC*, pages 777–780, 2001.

[216] Jonathan Babb, Russell Tessier, and Anant Agarwal. Virtual wires: overcoming pin limitations in FPGA-based logic emulators. In *Proceedings of the Workshop on FPGAs for Custom Computing Machines (FCCM)*, pages 142–151, 1993.

[217] Tiago M. Fernandez-Carames, M. Gonzalez-Lopez, and L. Castedo. FPGA-based vehicular channel emulator for evaluation of IEEE 802.11p transceivers. In *Proceedings of the Conference on Intelligent Transport Systems Telecommunications,(ITST)*, pages 592–597, 2009.