CHALMERS

A Study of Software Implemented Fault Tolerance in AUTOSAR Based Systems

Master of Science Thesis
Networks and Distributed Systems Programme

SHOVAN KUMAR PAUL
DEWAN MAHABUB SARWAR

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, March, 2013

A Study of Software Implemented Fault Tolerance in AUTOSAR Based Systems

Shovan Kumar Paul
Dewan Mahabub Sarwar

Examiner: Johan Karlsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden mars 2013

ABSTRACT

The AUTOSAR standard simplifies the complexity of automotive system design with its layered and modular software architecture. Currently, this standard has no support for fault-tolerance. Fault-tolerance will be required in the design of the future automotive systems to avoid catastrophic system failures and hazardous events. In this thesis, we present a study of fault-tolerance by means of software in AUTOSAR based systems. The aim of the study is to investigate how fault-tolerance mechanisms can be implemented in AUTOSAR. To this end, we implemented duplication and comparison, and triple modular redundancy in AUTOSAR in order to investigate how errors can be detected and fault-tolerance can be achieved, respectively. Moreover, the study focuses on the implementation of the distributed consensus protocol to investigate the impact of fault-tolerance in distributed scenario. In addition, we measure the timing overhead of the consensus protocol in which we observe that the execution-time declines with the increasing number of joining nodes.

Keywords: AUTOSAR, fault tolerance, duplication and comparison, triple modular redundancy, distributed consensus protocol.

Acknowledgments

We would like to express sincere gratitude to our Examiner Professor Johan Karlsson and Supervisor Associate Professor Roger Johansson, Department of Computer Science and Engineering at Chalmers University of Technology, for their invaluable supervision throughout this thesis project. Their excellent feedback, encouragement, and continuous support inspire us to carry out the thesis project.

We would also like to thank Mr. Michael Svenstam and Arccore AB technical team for their support and invaluable comments throughout the thesis project.

Special thanks to Dr. Risat Mahmud Pathan, Fatemeh Ayatolahi, Negin Fathollah Nejad, and Behrooz Sangchoolie for their valuable discussion and comments during the thesis project.

Table of Contents

1 Introduction

Today, the usage of electronic systems in automotive applications has introduced novel features. Some of these features are safety-critical such as adaptive cruise control, automated braking with obstacle and pedestrian recognition, and lane departure warning. However, in the early stage of automotive systems manufacturing, vehicles were mechanical systems with complex wiring harness. Such mechanical systems have been substituted by the innovation of the automotive electronics to enhance reliability and safety. One of the purposes of the automotive electronics is to integrate driver assistance functions such as lane change assistance and collision avoidance. In order to make reliable in-vehicle subsystems communication, the wiring harness has been substantially reduced by the invention of the communication protocols. Since modern cars are equipped with a large number of electronic control units (ECUs) for example 70-100, the complexity of car electronics architecture is increasing rapidly. A recent study [1] mentions that over the last 5 years, the number of ECUs increased by a factor of 1.45.

To simplify the design of complex automotive systems, manufacturers and suppliers have introduced a new standard, AUTOSAR (AUTomotive Open System Architecture) [2]. AUTOSAR provides flexible integration of functional subsystems, modular and layered software architecture, and component-based development. The future automotive systems will be equipped with vehicle platooning [3] and autonomous driving [4] functions. These functions require fault tolerance to reduce the risk of hazardous events and catastrophic system failures. Fault tolerance avoids service failure even in the presence of faults and thus ensures reliability and safety. Currently, AUTOSAR has no support for fault tolerance. Therefore, we investigate a study of fault-tolerance by means of software in AUTOSAR based systems.

The aim of this thesis is to investigate how fault-tolerance can be achieved in AUTOSAR based systems. The study focuses on the implementation of fault-tolerance mechanisms - duplication and comparison, and triple modular redundancy (TMR). The duplication and comparison is used to detect errors of the replicated results produced by two redundant modules. The TMR is used to mask errors by taking majority voting from three redundant modules for the same input. In addition, TMR performs forward recovery of the faulty module, which is known as TTR-FR presented in [5]. The forward recovery ensures an error-free state in which the erroneous result of the redundant module is replaced by another redundant module. These mechanisms are implemented in AUTOSAR in a single node and multiple node in order to achieve fault-tolerance. The single node experiment is conducted within an ECU among the redundant modules in AUTOSAR application layer. In contrast, multiple node experiment is conducted with multiple ECUs where each redundant module resides in a single ECU.

Moreover, the study focuses on the implementation of the distributed consensus protocol in AUTOSAR based systems. The aim of this protocol is to reach agreement among the nodes on a specific value. It terminates after two rounds of message communication, i.e., initial values are exchanged in the first round and decision making on a value in the second round. In addition, membership property is included in the consensus protocol where the nodes can join and leave at any time. The theoretical analysis proves that the consensus protocol should tolerate node(s) failure that will exhibit fault-tolerance in distributed scenario. Therefore, this protocol can be considered as a fault-tolerance mechanism. The consensus protocol is implemented in AUTOSAR to achieve fault tolerance in which the membership property is examined by using the time-out mechanism. In addition, the timing overhead is measured to observe the decline of the execution-time with the increasing number of joining nodes.

The remainder of the thesis is organized as follows. We present the theoretical background of the fault-tolerance mechanisms, a brief description of AUTOSAR, and related work in chapter 2. Chapter 3 describes the detailed design of fault-tolerance mechanisms in AUTOSAR. We present the experimental set-up and the implementation of the design in chapter 4. Moreover, we present the timing overhead of the consensus protocol in this chapter. Finally, the conclusions of the study are summarized in chapter 5.

2 Technical Background

The theoretical description of the fault-tolerance mechanisms and an overview of AUTOSAR standard are summarized in this section.

2.1 Fault Tolerance Mechanisms

Duplication and Comparison

The duplication and comparison is an error detection technique. It consists of two redundant modules and one comparator module. The redundant modules feed replicated results to the comparator module and then the comparator module compares the results in order to detect errors. Error detection is indicated by an output flag. The comparator module is a single point of failure in this mechanism.

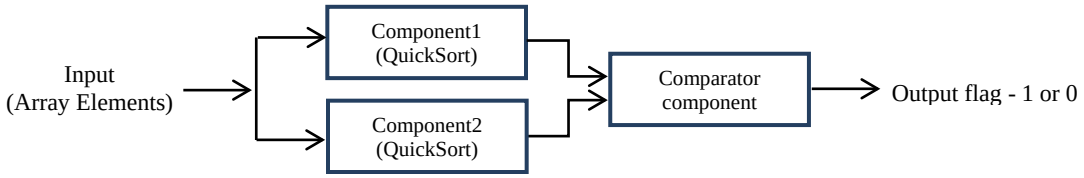

Figure 1: Duplication and comparison in AUTOSAR

As can be seen in figure 1, two redundant components perform the same calculations and feed results to the comparator component. In this case, the redundant components use quick sort application to sort the unsorted array elements. They should produce the identical sorted array elements if there is no software implemented fault injection technique applied. Finally, the comparator component compares the sorted arrays and indicates an output flag 1 if comparison matches or 0 if comparison mismatches.

Triple Modular Redundancy

The triple modular redundancy (TMR) takes majority voting and does forward recovery. The three redundant modules receive the same inputs and perform the same calculations. It performs 2-of-3 majority voting in which the faulty module is replaced by one of the correct redundant modules.

This mechanism can be implemented in replicated runnables across ECUs according to AUTOSAR release 4.0 [6]. As can be seen in figure 2, three components perform quick sort application and produce three results. Then the voting component performs majority voting based on the produced results and generates an output flag. If one of the redundant components produces an incorrect result, it is recovered by another redundant component. The voting element uses majority voting to mask errors.

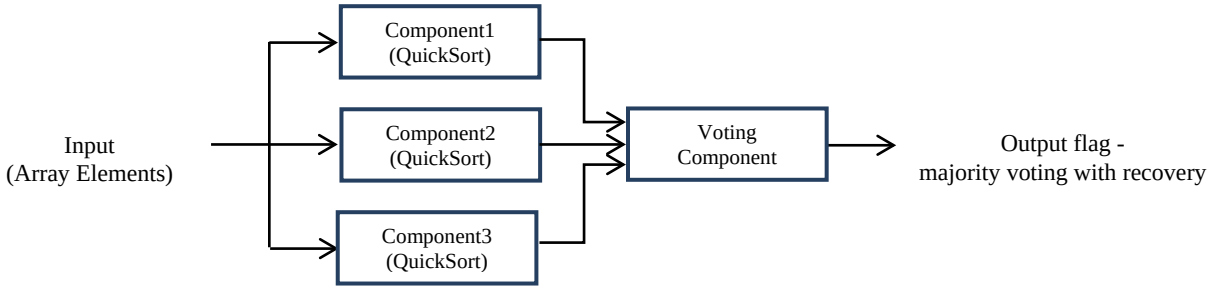

Figure 2: Triple modular redundancy in AUTOSAR

2.2 Distributed Consensus Protocol

The consensus protocol agrees on a specific value among the nodes in distributed scenario. It has three properties - agreement, validity, and finite termination. The protocol should terminate within a finite number of rounds. In our case, two rounds of message exchange have been considered to decide on a maximum value. In the first round, each node broadcasts its local value to the other nodes in the network. Then each node receives the values, and calculates the maximum value. In the second round, each node broadcasts its maximum value. After the second round of message communication, the nodes agree on the same maximum value. Thus, the consensus protocol proves the validity property as each node broadcasts the same value in both rounds of message communication.

The membership property is included in the consensus protocol. This means that the nodes join and leave at any time. The joining node reaches an agreement according to the consensus protocol. The node, which will leave the group, withdraws its proposed maximum value. Then the participating nodes calculate a new maximum value in order to reach agreement in the network.

2.3 Overview of AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a common automotive standard among the automotive manufactures, electronic vendors and suppliers, and embedded software industries. Initially BMW, Bosch, Continental, DaimlerChrysler and Volkswagen in August, 2002 started discussion in order to develop and establish an open standard for automotive electronic/electronics (E/E) architecture [7]. Afterwards, several automotive industries have joined to meet the future vehicle needs in the same framework.

2.3.1 Background

This standard reduces the development complexity of automotive applications, and ensures automotive product quality in a cost effective and significant way. The software components and electronic components can be used from different manufacturers and suppliers. Therefore, they have agreed to work together to address current market needs and the complexity of automotive electronic control units (ECUs). The main goals are to make a standard platform for software upgrades/update, a combination of different software and hardware components from various industries, and to meet the functional domains such as safety requirements, navigational enhancement in high traffic situation, different emergency states in case of critical situation of the future vehicle, and the customer infotainment facilities [7]. Moreover, AUTOSAR has increased scalability and maintainability in its standard specification.

The key features of AUTOSAR include modularity and configurability, standardized interfaces, and RTE [7]. Modularity and configurability gives the provision of HW and SW component independent feature from different suppliers. Standardized interfaces clearly divide the software components and basic software modules by providing API routines, and different standard interfaces among different layers of AUTOSAR. RTE encapsulates interaction between software components and BSW modules, and provides intra- and inter-ECU communication stub routines. The AUTOSAR specification provides standard architectural description, and routines. It has real time system behavior, diagnostic functionalities for error detection and user defined error recovery initiatives, and extended OSEK OS functionalities.

2.3.2 Software Framework

The AUTOSAR software component is independent, and interacts with basic software via RTE and different AUTOSAR interfaces for both intra- and inter-ECU communication. AUTOSAR software component is called Atomic software component and cannot be distributed over several ECUs. In contrast, several software components can be modeled in a group which is called composition. In this case, components in a composition can be distributed over several ECUs. RTE interacts between software components and basic software modules. AUTOSAR ECU software architecture shows in Figure 3, the detail software architecture of an ECU.

AUTOSAR basic software is located below RTE layer that provides services to the software components. It provides System services, Communication service, Operating system services, Microcontroller abstraction, and ECU specific components – ECU abstraction and Complex device drivers [8]. System services include memory related information and diagnostic notifications. Communication services provide message transmission and reception facilities using communication protocol such as CAN, FlexRay, LIN, etc. Operating system specifies scheduling mechanism of tasks based on alarm, priority, event, resource management, and so on. Microcontroller abstraction layer avoids direct access and modifications to the ECU-Hardware registers from the upper layer software. It is a hardware specific layer and informs notification to different processes. ECU abstraction depends on specific ECU and communicates with microcontroller hardware abstraction.

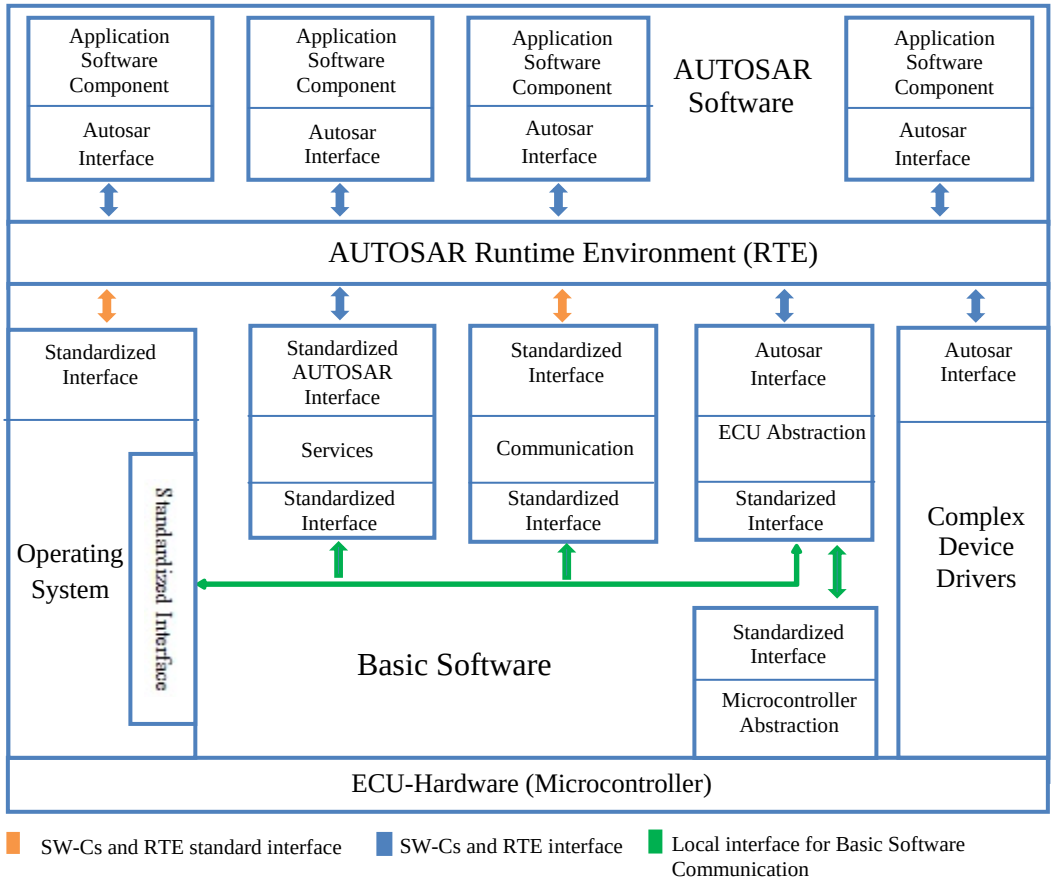

Figure 3: AUTOSAR ECU software architecture [8]

The complex device drivers provide the facility to access ECU-Hardware directly. It also supports nonstandard AUTOSAR drivers to include as complex device driver. There are three different interfaces used in AUTOSAR ECU architecture in Figure 3. They are AUTOSAR interface to exchange message(s) between software components, Standardized AUTOSAR interface which must follow a standard to exchange message(s) within AUTOSAR system services, and Standardized interface follows a standard (a specific programming language) to communicate with software modules in the same ECU [8].

Layered software architecture

AUTOSAR layered software architecture in Figure 4 shows the layer structure. It is also possible to add complex device drivers.

A. Application Layer

Application software components and Sensor/Actuator components are placed in Application layer. Software components communicate via RTE, and it is completely independent from Microcontroller, ECU, and HW perspective.

B. AUTOSAR Run-Time Environment

RTE is ECU specific and generates stub routines for the software components to communicate with BSW modules and other software components. The detail of this module has been explained after the description of layered software architecture.

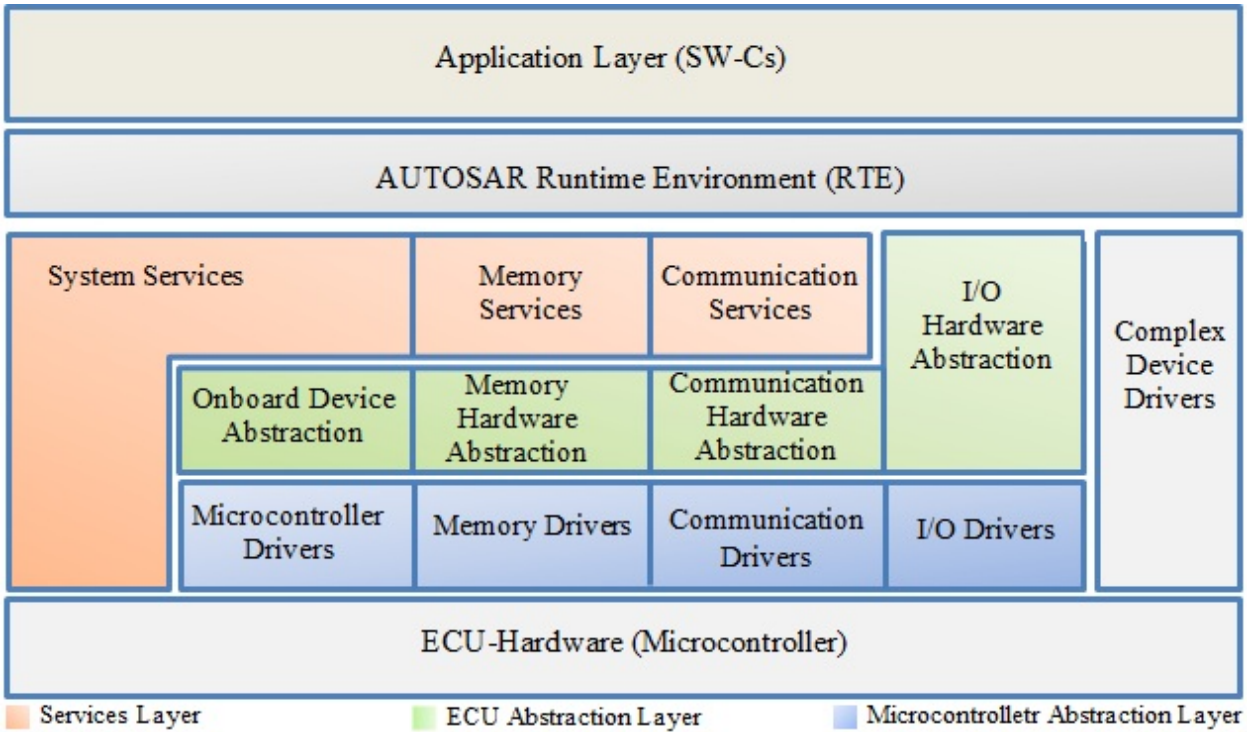

Figure 4: Refined layered software architecture [8]

C. Services Layer

This layer provides memory management services, network communication through communication protocol, diagnostic services, and operating system services. It is the top layer of basic software modules. It provides services to software components, and keeps ECU and Microcontroller layer abstract. The communication services perform vehicle network communication (CAN, FlexRay, etc) and interact with

communication drivers through communication hardware abstraction. Memory services abstract memory location from the application and manage nonvolatile data to the application in a secure way by interfacing with memory drivers. System services provide error detection facilities using some library functions and diagnostic modules.

D. ECU Abstraction Layer

ECU Abstraction Layer abstracts the ECU functionality from the services layer, and communicates with Microcontroller Abstraction layer to avoid direct access to hardware registers. It does the peripheral and handlers functionality to upper layer, and carries the signal to Microcontroller layer for further processing [8]. ECU abstraction has four parts. I/O Hardware abstraction abstracts I/O signals from upper layer and affects I/O drivers. Communication hardware abstraction abstracts communication controller location. It will give equal access to bus channel. Memory hardware abstraction provides equal access right to internal and external memory devices. Onboard device abstraction has drivers for ECU onboard devices (e.g. external watchdog) and provides access to onboard ECU devices via Microcontroller Abstraction Layer.

E. Microcontroller Abstraction Layer (MCAL)

This layer is placed just above Microcontroller Hardware and it is the lowest software layer in basic software modules. MCAL contains drivers software to access memory mapped external devices and internal peripherals [8]. It makes the upper layer Microcontroller independent. It is subdivided into four sections. I/O drivers for analog and digital input output peripherals (e.g. DIO), Communication drivers for vehicle network communication (e.g. CAN, FlexRay) that is same as Data Link Layer of OSI Layer model and communication within the ECU (e.g. SPI, I2C), Memory drivers for external and internal memory devices (e.g. external and internal Flash), and Microcontroller drivers for direct access to microcontroller hardware and internal peripherals (e.g. Watchdog).

Complex Device Drivers

Complex device driver can directly access to ECU-Hardware using complex microcontroller peripherals and specific interrupts. It interacts with complex sensors and actuators component using AUTOSAR interfaces. It is highly application, ECU, and Microcontroller dependent.

Virtual Functional Bus and Run-Time Environment

The AUTOSAR software components (SW-Cs) are independent from the rest of the system architecture. SW-Cs need a communication mechanism to interact with other software components. Virtual Functional Bus (VFB) makes this interaction. The communication can be executed in Intra-ECU i.e. among the components or Inter-ECU using communication protocol e.g. CAN, FlexRay frame through VFB. It works here as a middleware, and provides necessary routines for further communication with BSW modules.

The SW-Cs can be mapped to different ECUs or within a single ECU. Even a single component can communicate between two different ECUs. There are sender-receiver and client-server communication modes in AUTOSAR VFB view, and they can communicate through associated ports and assembly connectors. Sender-Receiver interface supports two type of communication – 1:N (one sender multiple receiver) and N:1 (multiple sender one receiver). Provider port (PPort) and require port (RPort) is used by sender and receiver respectively [9]. During communication, sender writes data element in PPort and receiver reads the data element from RPort. Data elements can be a single entity e.g. Integer, Boolean type, etc or an array of elements.

On the contrary, client server interface only applies to N:1 (one or more clients one server). Clients and server communicate through the RPort(s) and the PPort, respectively.

To communicate among the software components of sender-receiver and client-server interfaces, stub routines of each connector are generated in VFB. Therefore, SW-Cs can easily communicate with BSW modules. VFB gives all sort of facilities described above to SW-Cs including the possibility to have dynamism i.e. combination of SW-Cs from different manufacturers and suppliers. That also gives relocation facility to AUTOSAR software components [9].

The Run Time Environment (RTE) implements VFB interface for a specific ECU, and generates stub routines to communicate with application software components and BSW modules including OS and Communication services module. It hides details of communication mechanism from AUTOSAR software components. RTE generator generates stub routines to make communication between SW-Cs and BSW modules. In RTE contract phase [10], it generates contract file for each software component. On the other hand, RTE generation phase [10] generates all relevant information for each specific ECU. Each ECU must need one RTE to fulfill AUTOSAR system architecture.

The communication must be done through RTE for both the Intra- and Inter-ECU communication. In case of Intra-ECU communication, software components communicate through RTE generated stub routines. User defines specific activities in runnable entities which are called after that specific RTE stub. In contrast, Inter-ECU communication takes place in RTE generated stub code and then RTE sends signal(s) to COM module for further communication. In case of reception, RTE decomposes signal(s) from COM module. RTE maps application level signals which are from SW-Cs to COM signals for transmission and the other way around for reception. This mapping is extremely important for safe communication.

RTE plays an important role in AUTOSAR software architecture by hiding overall infrastructure above the RTE where application software components reside, and below the RTE where BSW modules, OS, communication services, and complex device drivers exist. It uses OS functionalities such as events, tasks, scheduling, and alarms from the user defined configurations of OS module, and schedules the tasks. Each task is triggered by the OS event which is also important to inform the RTE. RTE needs close interaction with OS module for scheduling and COM module for system level signal mapping issues of software components. RTE generator makes these interactions with OS and COM to realize the AUTOSAR VFB impression [10].

2.3.3 Communication Stack

Communication module

The AUTOSAR COM module is placed in the middle of RTE and PDU router which takes signal from RTE, packs them to corresponding I-PDU, and eventually transmits towards PDU-router for further transmission. On the other hand, it unpacks the I-PDU from PDU router, decouples the signals, and then transmits signals towards RTE in case of COM reception. Signal gateway is used to map the signals into a single I-PDU. The COM I-PDUs size is dependent on L-PDU size as the PDU router deploys the L-PDU directly to the communication interfaces. The maximum L-PDU length is bound to 8 Bytes for CAN communication. The same size also applies for LIN but FlexRay uses L-PDU length 254 Bytes [11]. COM manager controls the sending and receiving of I-PDUs through COM module.

Transfer Property	***Description***
TRIGGERED	Immediate transmission of the I-PDUs, if PERIODIC or NONE transmission mode is not defined.
PENDING	No transmission of I-PDUs.
Transmission Modes	
DIRECT	Causes when TRIGGERED transfer property is assigned. It triggers n times transmission immediately where n = 1…m, m<=255.
PERIODIC	Transmission request occurs for an I-PDU to the lower layer from the assigned periodic interval.
MIXED	Mixture of Periodic and Direct.
NONE	No transmission requests from COM.
Signal Processing(RX)	
Immediate	Signal indication takes place in COM_RxIndication which notifies successful message reception.
Deferred	First indicates COM_RxIndication, and then notifies COM_MainFunctionRx for the next call which is done asynchronously. It is used in case of cyclic task.

Table 1: Signal Transfer Properties, Transmission and Signal Processing Modes [11]

In context of AUTOSAR COM, signal is treated as message [11]. COM can also send group signal. In this case, signals are similar to C-structs which will be considered as a single entity in between transmission and reception. The COM communication modes follow a certain Transfer Properties for each signal, Transmission Modes of I-PDUs to transmit them towards PDU router, and Signal Processing Modes for receiving events. Each mechanism is shown in Table-1.

COM module gives the facility to invalidate a signal for both transmission and reception of message(s) [11]. The sender component can invalidate a signal and informs COM to replace the signal by a default value or any chosen value. On the contrary, signal value can be replaced by any default value when receiver receives invalid signal that will be detected from system configuration. COM module also provides deadline monitoring facility of the receiving signals, and can initiate user defined actions through call-back routines. In addition to that, it notifies upper layer i.e. RTE in case of successful message transmission and reception.

Protocol Data Unit Router

The PDU Router is a part of AUTOSAR Basic software, and provides routing facilities of the I-PDUs between different modules such as AUTOSAR COM and communication interface, shown in Figure 5.

It can identify a PDU by the ID, and uses this id along with the configuration table to find destination I-PDU [12]. It is responsible for forwarding I-PDUs without modifying the content. Moreover, it provides APIs to communicate with COM and communication interface modules. PDU router engine does routing operations based on routing tables. During transmission, PDU-Transmit-Request triggers PDU Router to transfer I-PDU from upper layer (COM) to lower layer (CANIF) using PDU ID and information from the routing table. On the other hand, it transfers I-PDU in reverse direction (CANIF to COM) during reception.

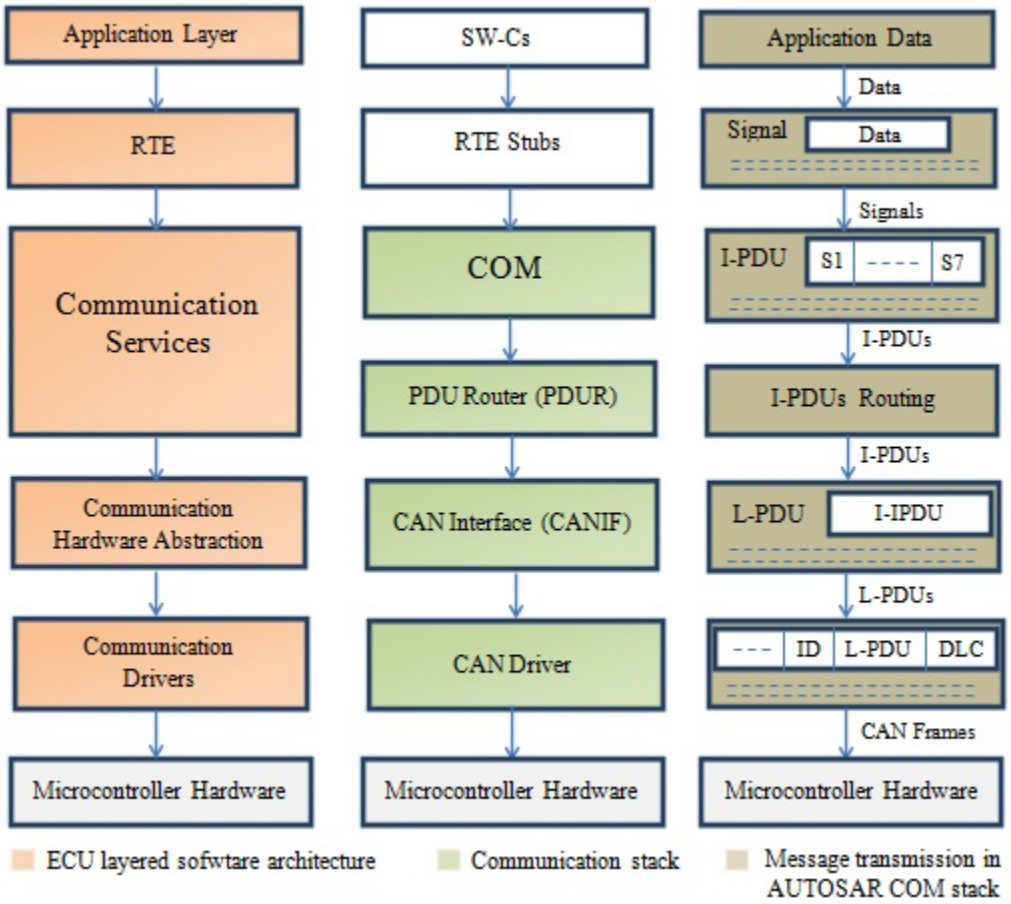

Figure 5: Data transmission in AUTOSAR communication stack

CAN Interface

The CAN Interface module belongs to Communication Hardware Abstraction layer in AUTOSAR layered software architecture. It is situated in between the lower layer i.e. CAN driver, and communication service layer i.e. COM. This module acts as an interface for the communication service layers to manage various CAN hardware devices such as CAN controllers and CAN transceivers. As an example, multiple CAN controllers can be controlled by CAN state manager with the help of CAN interface module.

CAN Interface is responsible for two type of functions. Firstly, it handles various notifications and data processing operations for the communication service layer such as Transmit request processing, Transmit confirmation, Receive indication, and Error notification. Secondly, this interface takes care of CAN controller related work such as start/stop of CAN controller based on the detected events. This module provides location of application data buffer by using CAN Hardware-Object –Handles (HOH) to CAN driver for transmission and reception of data [13]. HRH (Hardware Receive Handle) characterizes the reception unit which contains Hardware Object for reception of data and also used for software filtering. On the other hand, HTH (Hardware Transmit Handle) represents Hardware Object for data transmission. During transmission, upper layer (COM) initiates transmission request by invoking CAN interface service. Then CAN interface calls the interface service of CAN driver with HTH as a parameter. This service completes hardware dependent functions and transmission request.

CAN Driver

The CAN driver performs hardware dependent tasks and provides hardware independent interfaces to upper layer i.e. CAN Interface module. This driver module controls behavior of CAN controller [14]. It provides services which are periodically executed by CAN Interface. During data transmission, CAN driver writes data in the buffer of CAN controller hardware, and converts L-PDU attributes ID and DLC into appropriate hardware dependent format. On the other hand, this driver invokes RX indication callback routine of CAN Interface module by providing appropriate arguments such as ID and DLC during reception of L-PDU.

2.3.4 Systems Services

Operating System

The AUTOSAR OS is based on OSEK/VDX operating system which is commonly used in automotive industry. Some features of OSEK OS are restricted or extended in AUTOSAR OS. It provides services for task management, synchronization, interrupt processing, alarms, and error handling. The control software is divided into several parts which are executed according to their requirements. There are four different states of tasks - running state in which CPU is executing the task, Ready state has all the prerequisites to go in running state, waiting state in which task waits for an event to occur, and suspended state represents termination of a task. Tasks have two different types. They are Basic and Extended task [15]. Basic task has no waiting state and activated only once. It has three states – running, ready, and suspended. On the other hand, extended task has running, ready, waiting, and suspended states. The configurable parameters of tasks are priority and preemptability. Task travels through different states by calling different operating system services.

AUTOSAR Operating System provides Alarm Mechanisms to periodically generate interrupt to application software components which is driven by counters [15]. Tasks can be activated by using Alarm expiration time or by an event which is set by the expiration of Alarm. Alarm depends on the advancement of counter value. The counter mechanism can be implemented as hardware or software timer. In terms of Event in AUTOSAR OS, it is used to synchronize the extended tasks. Extended tasks in the waiting state wait for an event to go to the ready state. Operating system has specific services to set and clear the events. Generally, event is identified by name, and task is the owner of that event.

Resource sharing in AUTOSAR OS can lead to several problems when several concurrent tasks and ISRs (interrupt service routines) compete for the same resource. Resource management ensures coordination of concurrent accesses of tasks and ISRs to shared resources such as scheduler, memory and hardware. It prevents two tasks or ISRs to occupy same resource at the same time, and thus prevents priority inversion and deadlocks. It manages resources between preemptable tasks and ISRs. Moreover, AUTOSAR OS supports error handling using hook functions which are defined by the user, and called by the operating system. For instance, OS uses StartupHook routine for system startup and ShutdownHook for system shutdown [15].

2.3.5 Diagnostic Services

Diagnostic Communication Manager

Diagnostic Communication Manager provides APIs for diagnostic services in AUTOSAR software architecture. This module facilitates communication between on-board AUTOSAR application and

external diagnostic tools e.g. tester or OBD scan tool for data collection. It is responsible for diagnostic data flow and diagnostic status of the system. It receives diagnostic service request message from PDUR module, interprets the message, interacts with other BSW modules and SW-Cs to collect data, and eventually send the message back to PDUR. The sub modules of DCM are DSL (Diagnostic session layer), DSD (Diagnostic service dispatcher) and DSP (Diagnostic service processing) [16]. They are accountable for data flow and timing, states of diagnostic messages (session and security), and data processing. DSL module performs session and time handling. DSD checks the legitimacy of a request message, verifies the session and security access level, and then transmits response message or NRC (Negative response code) code. DSP is responsible for processing of diagnostic service request messages. After reception of request message, DSP analyzes messages, checks message length and format, executes the required function, and finally assembles the response.

Diagnostic Event Manager

Diagnostic Event Manager is located in system services layer in AUTOSAR layered software architecture. This module stores diagnostic events to memory, and processes event related data e.g. sensor values, mileage, etc. It delivers fault information to DCM, and provides interfaces to software components and other diagnostic modules such as DCM and FIM. DEM allows software components to provide and retrieve data to/from DEM through monitor function [17].

Development Error Tracer

DET is a part of AUTOSAR diagnostic modules. Development error in BSW modules and software components are reported to DET. It provides API routines to trace error source (infected modules, functions) and type of errors. For instance, Det_ReportError() [18] API is used to report error.

2.3.6 Peripherals

PORT Driver

Port Driver is situated above Microcontroller Hardware and inside Microcontroller Abstraction Layer in AUTOSAR layered software architecture. Each microcontroller pin is used for certain functions such as communication, general purpose input/output, etc. This driver initializes and configures port and port pins [19]. It also configures direction, mode, and physical level of a port and port pin. After successful initialization of port pins by port driver, these are used by DIO driver. Unless the port and port pins are initialized prior to use DIO functions, the system will exhibit anonymous behavior. Moreover, The Port driver initializes unused ports and port pins in a defined state.

DIO Driver

DIO driver belongs to Microcontroller Abstraction Layer. It works with port driver for reading and writing to/from DIO channels [20]. Thus it allows data transfer between ports and channel groups. After initialization and configuration of port and port pins by port driver, DIO driver uses DIO functions to read and write data in the port and port pins. This driver also allows grouping of several DIO channels into DIO groups and ensures data consistency.

MCU Driver

MCU driver is located in Microcontroller Abstraction Layer in AUTOSAR layered software architecture, and has direct access to controller hardware. It provides services to initialize microcontroller, and control power down and reset functions. This driver contains startup code, and offers services to configure clock

and PLL settings as well as the base address and size of RAM [21]. This driver manages hardware reset through a software triggering event. Moreover, MCU driver provides services to select different power modes of microcontroller which are implemented in hardware.

General Purpose Timer

The General Purpose Timer (GPT) is placed in Microcontroller Abstraction Layer where MCU driver is located as well. It works as a hardware timer, and helps operating system timers when OS timers have extreme overhead. GPT timer depends on MCU time settings. Therefore, changes to MCU system clock also affect GPT timer [22]. It provides diagnostic notifications to AUTOSAR diagnostic framework.

This driver informs error occurrence to DET when development error is detected. It can report production error to DEM module in case of production code error occurrence. This driver also provides starting and stopping of hardware timer instance within a user defined notification period. Therefore, it can estimate elapse and remaining time till the end of notification instance. It can wake up the specific ECU after expiring the timeout period. Notification time out can be absolute or relative which is similar to AUTOSAR OS alarm types – absolute alarm and relative alarm [15]. GPT driver is time based, not event based module.

IO Hardware Abstraction

IO Hardware Abstraction is located above Microcontroller Abstraction Layer and is a part of ECU Abstraction Layer. This module offers signal based interface to software components. IO Hardware Abstraction converts electrical signals to ECU signals and hides these electrical signals from software components. ECU signals are mapped into IO Hardware Abstraction port [23]. IO Hardware Abstraction invokes the API of various drivers such as ADC, PWM and receives notification from them. It provides high level interface to PWM and ADC modules to set and get ECU pin values. IO Hardware Abstraction provides interfaces to control and measure different signals attribute.

2.4 Related work

To implement fault tolerance in AUTOSAR systems, there are several attempts to be found in the literature. The authors' techniques are described briefly as follows.

In [24], Thorsten Piper et al. propose "*instrumentation approach"* in AUTOSAR environment for dependability assessment. Their approach provides the key features of usability, customizability, and efficiency in AUTOSAR. The authors use the term "Instrumentation" as a means of modification of SW-Cs in AUTOSAR by experimenting fault injection in anti-lock brake-by-wire (ABS) system. They do experiment on top of AUTOSAR implementation of two vendors - ETAS Group and OptxWare Research and Development Ltd. In their fault injection experiment, the authors use SWIFI tool to inject fault in ABS system. They transmit 16 bit data, and run the system 17 times of each fault injection campaign. They consider one golden run as a reference to monitor the fault injection. In each fault injection campaign, they flip a single bit in 16 bit data. They observe the deviations from the golden run due to the injected fault in 16 bit data. Moreover, they observe that the lower 8 bit data has a minor impact on their system which can be tolerated within one or two period of execution time. The authors observe the instrumentation overhead in case of runtime execution and memory consumption. They measure the runtime execution of current CPU ticks and estimate the relative comparison of different approaches. They measure the overhead of approximately 50% for OptxWare EA and 38% for ETAS INTECRIO. Moreover, they observe that the fault injector doesn't add measurable overhead in their experimental

result. In case of memory consumption experiment, they observe that the overhead in text segment size of memory using objdump tool varies in between 1.5% and 15% per wrapper. This is due to the fact of different implementation strategy. Moreover, they observe that each wrapper consumes around 33% bytes for ETAS INTECRIO and 30 bytes for OptxWare EA. This difference is due to the usage of different compilers – ETAS relies on MinGW GCC 3.4.2 and EA relies on Cygwin GCC 3.4.2. The authors show that the instrumentation methods in qualitative aspects in case of intrusiveness, implementation effort, automation complexity, required system access, and scalability in AUTOSAR SW-Cs and RTE of .c-file (white-box), .h-file (grey-box), and .o-file (black-box). In their observation, they show that the black-box instrumentation has clear advantages over gray-box and white-box in all qualitative categories except automation complexity. Lastly, they conclude that the instrumentation location (SW-C or RTE) is not clear as SW-C has advantages in intrusiveness and required system access categories of qualitative aspects, and on the other hand, RTE has advantages in other categories. The authors think that this qualitative approach depends on the system evaluator.

In [25], the authors propose an R-FLOW algorithm for Hot Standby and Cold Standby processors to support fault tolerance with bounded recovery times. Then they integrate this algorithm in AUTOSAR to provide dependability. They introduce "*application flow*" concept to exchange information periodically among the sensors and actuators runnables of SW-Cs. An ASIL value is assigned to each SW-C which helps R-FLOW to do replication. They add a new structure in SW-C template to keep up-to-date state in order to synchronize the standby module in Cold standbys. A health status module is added to activate the replicas. In their experiment, R-FLOW replicates software components and maps them to specific ECUs. New tasks are created based on predefined rules of SysWeaver automated tool. Their results show that R-FLOW saves up to 45% of processors when only primary components are active. Moreover, savings can be more than 60% of processors if replicas are used as compare to their previous scheme, R-BATCH.

In [26], Lu et al. suggest "*reflective principle*" approach to achieve fault tolerance in AUTOSAR. This approach introduces defense software which works on logging of information, checking, and recovery. The authors implement this defense software as a monitor which interfaces with functional software using AUTOSAR OS hooks. Hooks are triggered on certain events such as task start/stop, OS error, etc. This defense software requires access to the OS in order to monitor the control and data flow at OS level. The efficiency of defense software is evaluated by fault injection technique.

In [27], Fabre et al. propose "*Multi-level reflection"* approach to achieve fault tolerance and robustness using AUTOSAR as a middleware. In their approach, they implement a defense software which is based on computation reflection [26]. This multi-level layering of functional and non-functional software works as self-checking component. The defense software has two different parts - Error Detection Mechanism (EDMs) and Error Recovery Mechanisms (ERMs). The EDMs consist of runtime assertions that are triggered by sensors. On the other hand, actuators do recovery based on sensors' queries. The implementation is carried out using AUTOSAR OS services and hooks to log suspicious behavior relevant to assertion, and then trigger the verification. Finally, ERMs take the recovery actions.

2.5 Development Environment and Tools

The evaluation board **Vk-EVB-M3** [28] has an ARM 32-bit STM32F107 processor which is based on ARM Cortex M3 architecture. This board has two isolated CAN channels, four user defined LEDs, 20 pins of JTAG connector for downloading and debugging purpose, uSDCard, and Ethernet connector using RJ45. In this project CAN1 channel, JTAG, and user defined LEDs are used.

The following tools [29] developed by ARCCORE AB have been used to design and implement the fault-tolerance mechanisms.

Arctic Studio is a complete AUTOSAR development environment developed by ARCCORE. It is based on Eclipse IDE to develop and compile AUTOSAR projects. Several plugins are integrated in Arctic Studio for the development of AUTOSAR applications.

SWC Builder is a tool for defining and editing different software components. User can easily design software components, interfaces and ports, and can set event to runnable by this tool. Finally it is possible to check the validity of configurations if there is any missing object in SWC components.

Extract Builder helps to add the instantiated **s**oftware components and IO Hardware Abstraction module in the ECU. It provides facilities to connect software components using ports and connectors using "port mappings" section.

RTE Builder is a tool that is used to generate the RTE code. This tool has a XML based editor through which user can specify how runnable will be scheduled by mapping the runnables with OS tasks and specific event for each task which will trigger the runnables. RTE builder uses the extract file as an input.

BSW Builder is a tool that supports configuration and generation of different AUTOSAR basic software modules such as MCU, CAN, COM, OS, etc. User can integrate various AUTOSAR basic software modules in this tool. Moreover, it provides option to import BSW modules from other projects. Multiple modules can be configured at the same time using BSW builder. In the final part, user can check the validity of the configuration and the stub codes will be generated in configuration directory the project.

Can Monitor Lite [30] is a software utility tool developed by WGsoft which is used to monitor CAN messages in the CAN bus. It supports various Baud rates. User can filter and monitor the CAN messages with specific ID in the GUI interface. It works in integration with Lawicel CANUSB adaptor to connect with the CAN bus. User can send messages through the graphical interface and log the messages as well.

Debugger Setup

Arctic Studio is configured to support debugging with the help of MSYS, MinGW, GDB, OpenOCD, and Zylin Embedded CDT. The steps are mentioned in the debugging section of ARCCORE. Zylin Embedded CDT plugin is installed in Arctic Studio to create the debug configuration. GDB commands are introduced to the JTAG interface. User can stop the program at any point and can fix bugs. The R-Link JTAG adapter is used in VK-Board-M3 to facilitate debugging. The debugger setup in arctic studio with raisonance support using gdb and openocd is described in [31].

3 Design

In this section, we present the detailed design of fault-tolerance mechanisms in AUTOSAR framework developed by ARCCORE AB. Each AUTOSAR based application in such framework is designed with Software Component (SWC) Builder, ECU Extract (Extract Builder), and ECU-configuration (BSW Builder). The software components are created and configured in the SWC Builder. These software components are then added in the Extract Builder to instantiate them in an ECU. Lastly, the basic software modules are created and configured in the BSW Builder in which the configured stuff of the SWC and the Extract Builder is used to design the BSW Builder.

3.1 Duplication and Comparison

3.1.1 Single ECU Experiment

Duplication and comparison is designed with two redundant software components (component-1 and component-2), a comparator software component, and an additional software component named as LED component to indicate the outcome of the comparator. As a startup of the design, we added the configuration tools of the SWC, ECU Extract, and ECU-configuration in the project.

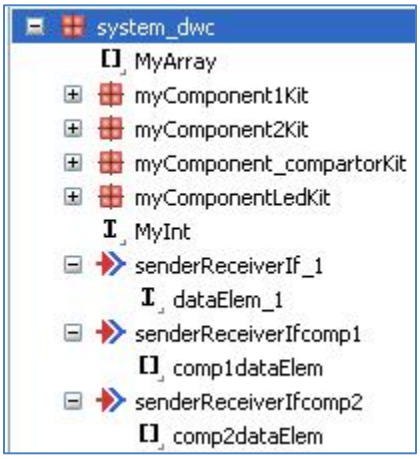

Figure 6: SW-Cs and sender-receiver interfaces in SWC Tool

The SWCs and the sender-receiver interfaces in SWC Builder tool are shown in figure 6. In this configuration, four software components are added. Array type and integer type is included in the configuration in order to use them directly in the implementation phase. The detail design of the LED component is shown in figure 11. The LED component uses the IO Hardware Abstraction module, which is a basic software module. Therefore, we first added the IO Hardware Abstraction basic software module in the ECU-configuration tool, and then we pressed the "generated system model for this module" in order to generate the Digital-Output interface, as can be seen in figure 11 and described of the configuration process.

Figure 6 shows the configuration of the three sender-receiver interfaces. LED component uses the senderReceiverIf_1 interface with the comparator component. In a sender-receiver interface, sender writes data in the provider port and the receiver receives the data from the require port. The data is written in the data-element variable of the interface. In the senderReceiverIf_1 interface, comparator component uses a provider port and LED component uses a require port.

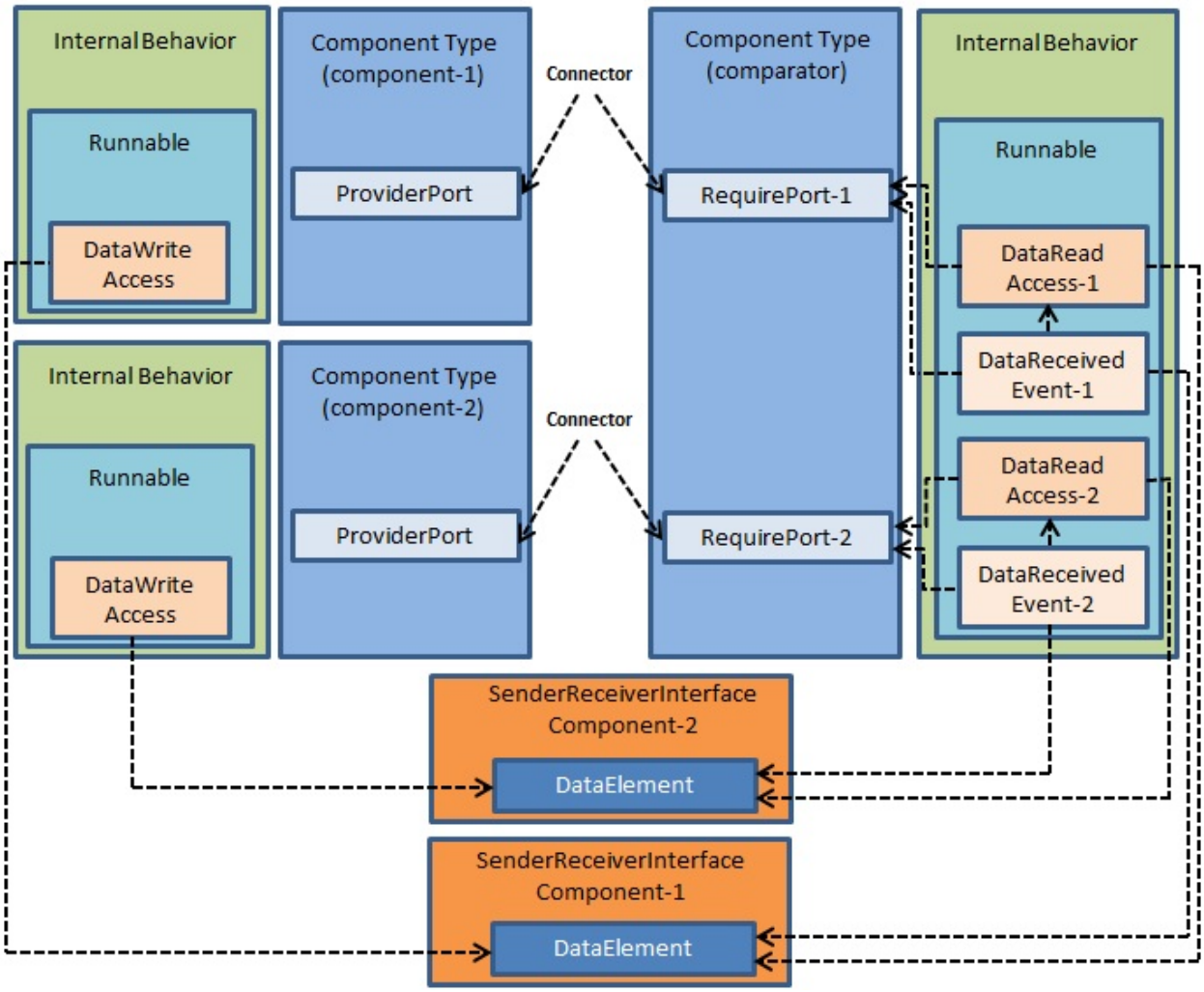

Figure 7: Logical view of duplication and comparison using sender-receiver interfaces

Therefore, the LED component blinks a specific LED number to indicate the outcome of the comparator component. The remaining two sender-receiver interfaces are used between component-1 and comparator as well as between component-2 and comparator component, respectively, in which component-1 and component-2 write data in the provider ports and comparator component uses the unique require port to receive the data.

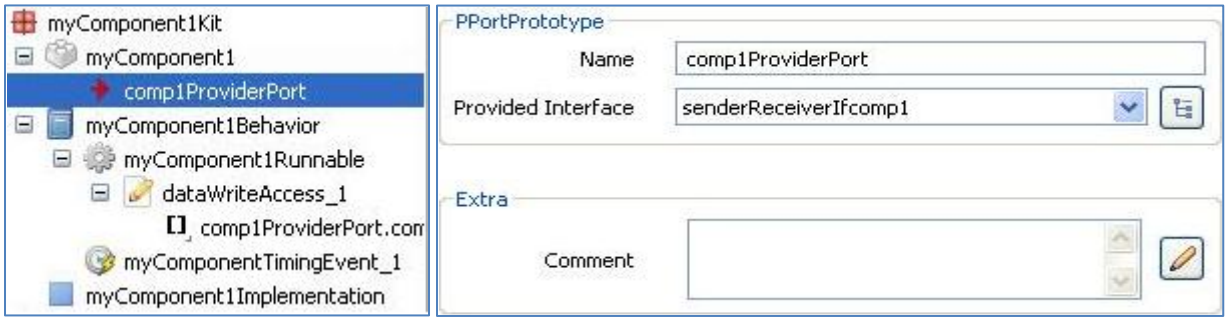

Figure 8: Configuration of software component-1

The logical view of data communication in duplication and comparison mechanism is shown in Figure 7 where the software components write the data in the "data element" variable of the sender-receiver interface and afterwards, the comparator component receives that data. In fact, the sender component writes the data in the data-element of the provider port and the receiver component receives the data from the provider port.

The configuration of component-1 is shown in figure 8 where this component uses a unique sender-receiver interface to connect with the comparator component, and writes the sorted array elements in the provider port of the interface using the "dataWriteAccess_1". In a similar fashion, the second redundant software component is configured shown in figure 9. In this case, component-2 uses a different sender-receiver interface.

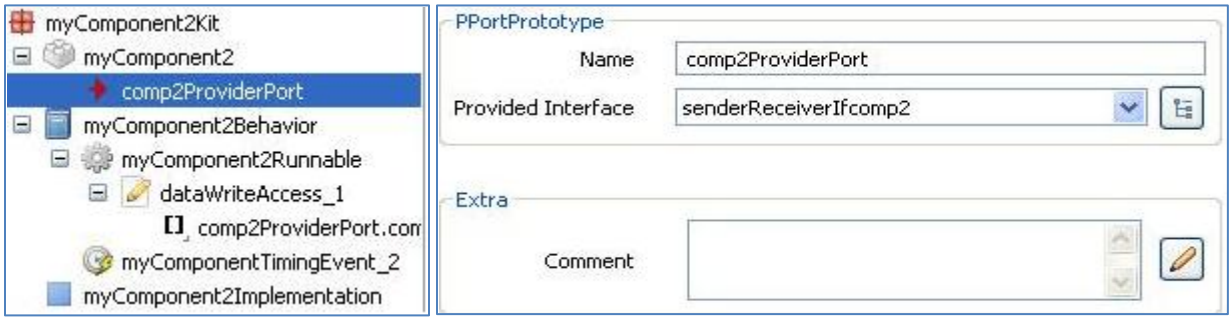

Figure 9: Configuration of software component-2

The comparator software component receives two sorted array elements from the software component-1 and the software component-2 where the comparators receive the sorted array elements from the requireport-1 and the requireport-2, separately. Figure 10 shows two require ports of the comparator component. The sender-receiver interfaces are unique for these require ports. For instance, requireport-1 uses the same interface that the component-1 is using. The "data read access" is used to read the array elements of the require ports.

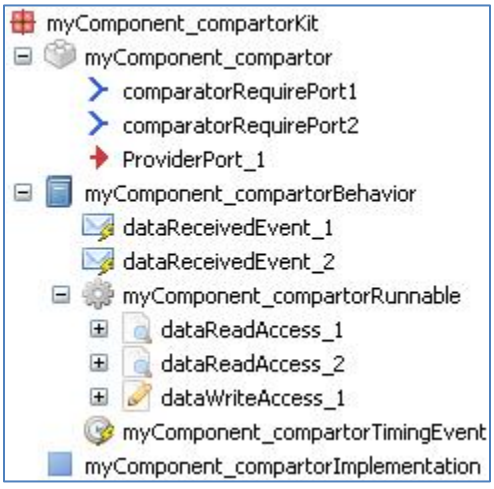

Figure 10: Comparator software component in SWC Tool

The comparator component communicates with the LED component using a sender-receiver interface to blink the specific LED number. In this interface, comparator component uses provider port and writes a specific number in the "data write access" field. Figure 10 shows two data receive events which is needed

if we would have sequence of events. The components work fine for the design of the duplication and comparison mechanism without using the “data receive event” attribute.

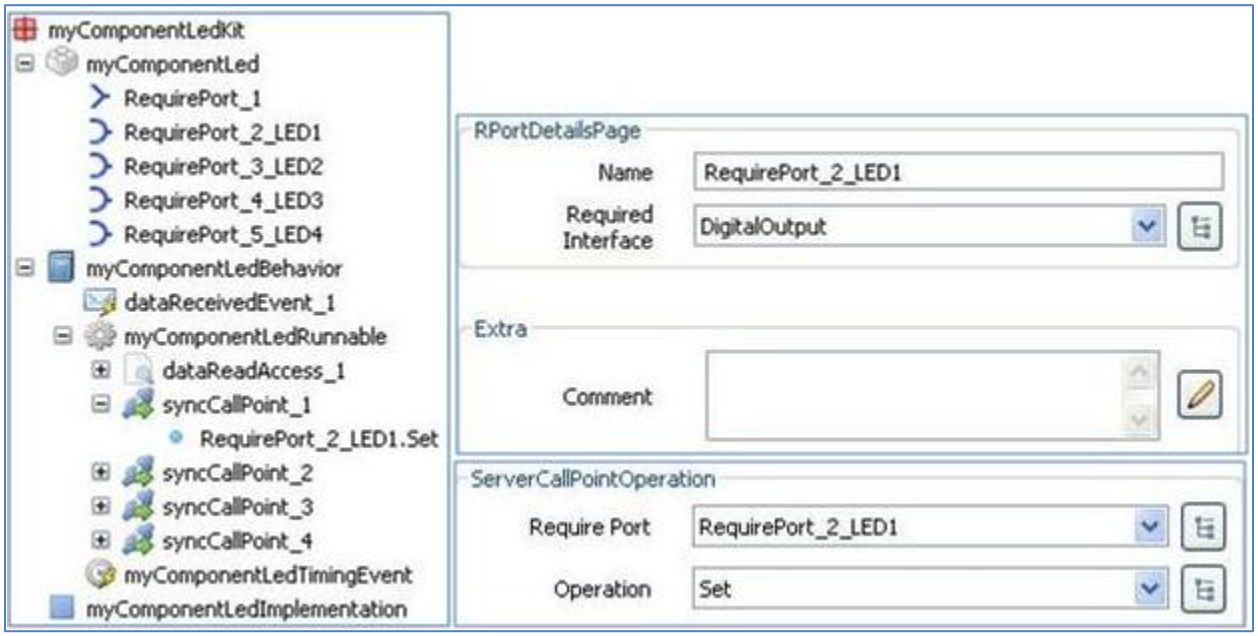

Figure 11: Configuration of LED component in SWC Tool

As can be seen from Figure 11, the LED software component uses a require port, namely requireport-1 which is used to receive the indication of the comparator component. LED component needs to be configured using the synchronous call point so that it can be invoked from the runnable method of the LED software component. This synchronous call point (Set operation in figure 11) invokes the digital-output interface function and afterwards, invokes the IO Hardware Abstraction to blink the desired LED number.

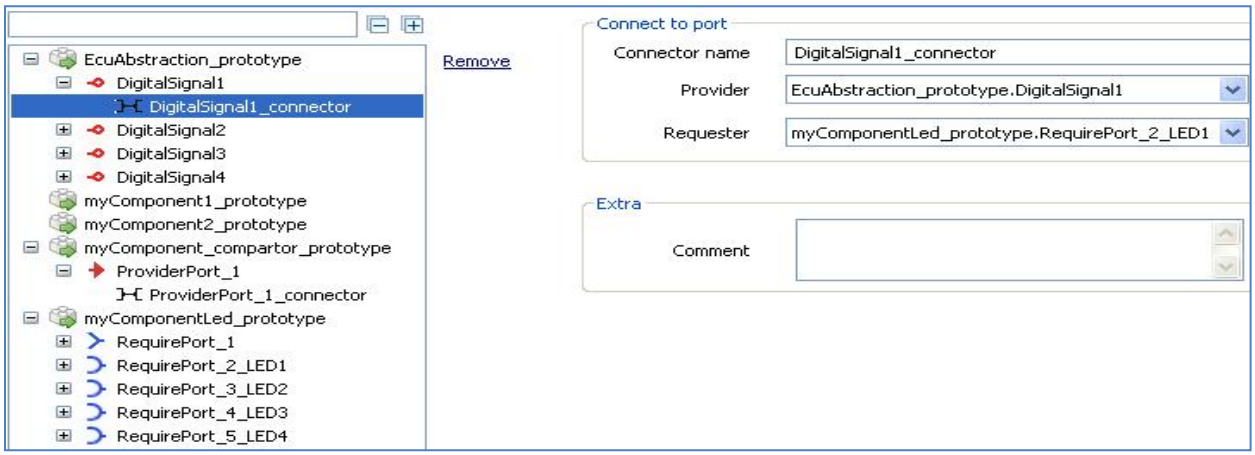

Figure 12: Port mappings in Extract Builder Tool

In the second phase, the prototype of the four software components and the IO Hardware Abstraction is visible in the Extract Builder tool. We added these prototypes in order to instantiate them in the ECU. The Extract Builder makes connection, namely port mappings of all the instantiated components. In this case, port mapping is configured of the sender-receiver interfaces, which are used in the SWC tool. Figure 12

shows the port mapping indication of the provider and the requester port of LED1. The same procedure is followed for the rest of the mappings of the sender-receiver interfaces.

The final phase deals with the basic software configuration. The basic software modules such as Os, EcuM (ECU state manager), and Mcu have been added and then configured. These modules are the core modules, and are needed in all projects. The configuration of the EcuM and Mcu varies in different ECUs. We used the reference manual of the VK-Eval-M3 ECU and the related sample projects in the arctic studio for this ECU to configure the EcuM and Mcu basic software modules.

Alarm Name	Component-1	Component-2	Comparator
Alarm Time	10	20	30
Alarm Cycle Time	50	50	50
Autostart Option	Yes	Yes	Yes
Autosart Type	Absolute	Absolute	Absolute
Alarm Action Type	Set Event	Set Event	Set Event
Set_Event	Component1Event	Component2Event	ComparatorEvent
Set_EventTask	Component1Task	Component2Task	ComparatorTask

Table 2: Os alarms scheduling in intra-ECU duplication and comparison

Table 2 shows the Os alarm triggering time and action in the OS module. Alarm time means initial starting offset of the specific component, and alarm cycle time denotes the periodic activation time of the component. For instance, Component-2 has alarm time 20 and cycle time 50. The alarm fires Component2Event when the 20 Os ticks are elapsed for the first time and afterwards, Set_EventTask activates Component2Task. This task will be triggered again at 70 Os ticks i.e. (20+50) Os ticks.

The execution of the basic task starts at first among all other tasks in AUTOSAR OS tasks configuration. As described in the operation system section of the overview of AUTOSAR chapter, basic task has no waiting state and thus it activates only once. It performs initialization of the ECU after the startup of OS. Therefore, the startup task is given the highest priority task, as can be seen in Table 3.

Task Name	Startup Task	Component1Task	Component2Task	ComparatorTask
Task type	Basic	Extended	Extended	Extended
Schedule Preemptability	Full	Full	Full	Full
Priority	10	2	2	1
Activation Limit	1	-	-	-
Events	-	Component1Event	Component2Event	ComparatorEvent and RTEevent
Autostart	Yes	Yes	Yes	Yes

Table 3: Os tasks scheduling in intra-ECU duplication and comparison

The basic task invokes the Rte_Start routine and then the StartupTask does context switching into the extended task(s) according to the priority assignment. In this design step, Component1Task and Component2Task have been assigned as priority 2, and ComparatorTask has been given the lowest priority because the comparison operation is performed after the execution of the Component1Task and the Component2Task. The OS starts Component1Task and then Component2Task due to the shorter alarm time of the Component1Task as can be seen in in Table 2. The alarm time of the component1Task

set component1Event and afterwards, the Component1Task is activated. After that, the Component2Task and the ComparatorTask are activated accordingly.

To configure port pins, and to enable data transfer, PORT and DIO basic software modules are added in the ECU-configuration (BSW Builder) to control the user defined LEDs.

Figure 13: Port configuration in BSW Builder

The BSW Builder configuration of PORT and DIO is shown in Figure 13 and 14. The reference manual of the VK-Eval-M3 ECU is used to configure the pin numbers and LED numbers. Moreover, the pin number of CAN1 channel is configured in order to exchange messages in inter-ECU communication.

Figure 14: Dio configuration in BSW Builder

The configuration of the IO Hardware Abstraction module includes the Dio channel and the relevant information in order to activate the digital signals of the LEDs. Figure 15 shows the configuration of LED1. Four digital signals are added and configured for the four user defined LEDs.

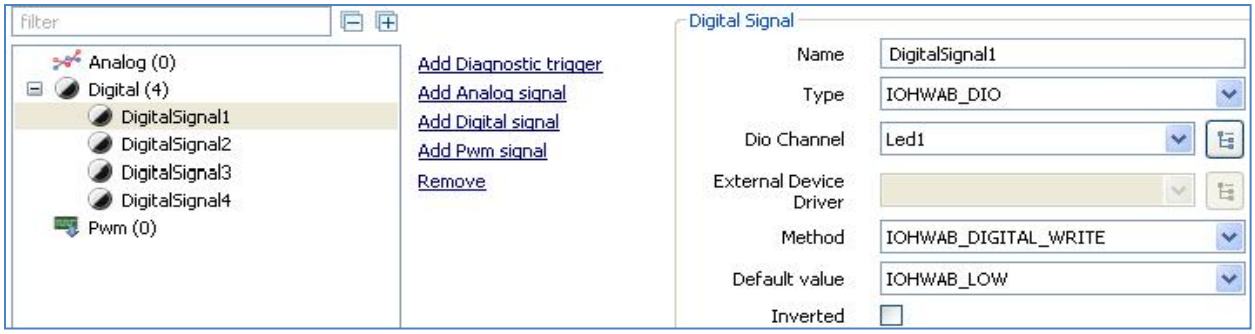

Figure 15: IO Hardware Abstraction configuration of LED1

RTE configuration in BSW tool is shown in figure 16. The first step is to add the instantiated components in the Extract Builder tool. The runnables are mapped with the specific extended tasks and then they are set to trigger by the specific event which are configured in the OS tasks basic software configuration. Basic task has no wait event. Therefore, startupTask has no runnable routine. It is only used in system initialization. The DigitalOutput_Set() and DigitalOutput_ReadBack() runnables are kept unmapped as

there is no relevant tasks for this module. The LED component runnable is mapped under the comparator component as there is no specific task for the LED component and the comparator feeds the outcome to the LED component. After generating the configuration of all basic software modules, RTE creates an event for the LED component, which is mapped with this component shown in figure 16.

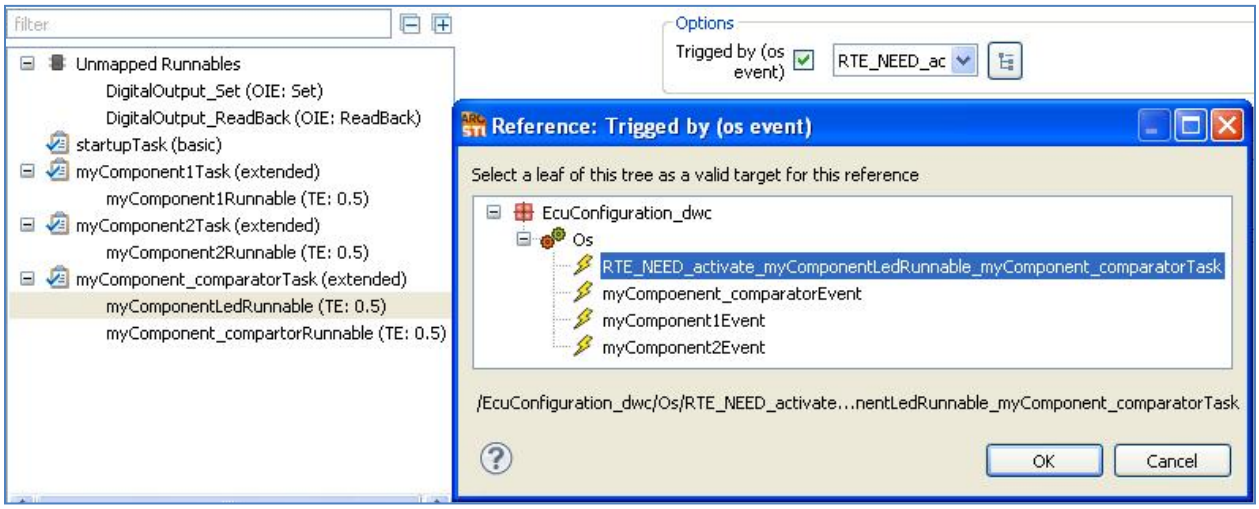

Figure 16: RTE configuration in BSW Builder Tool

3.1.2 Multiple ECU Experiment

To design duplication and comparison in multiple ECU experiment, four software components are added in the SWC Builder tool; sender component, receiver component, comparator component, and LED component. The description of the LED component is similar to section 3.1.1. The sender and receiver components are created to send and receive signals inside their runnable methods. The comparator component checks the computed results of the sender and receiver components and feeds a numerical value to the LED component as an indication of the outcome of the comparator component. In this case, comparator and LED component uses a sender-receiver interface, which is similar to the description of the duplication and comparison in the single ECU experiment. The configuration of the IO Hardware Abstraction and the Extract Builder tool is similar to section 3.1.1.

Alarm Name	senderComponent	receiverComponent	comparatorComponent
Alarm Time	100	100	100
Alarm Cycle Time	100	100	100
Autostart Option	Yes	Yes	Yes
Autosart Type	Relative	Relative	Relative
Alarm Action Type	Set Event	Set Event	Set Event
set_Event	senderEvent	receiverEvent	comparatorEvent
Set_EventTask	senderTask	receiverTask	comparatorTask

Table 4: Os alarms scheduling in inter-ECU duplication and comparison

The configuration of the EcuM, Mcu, Port, Dio, and RTE follow the same procedure in section 3.1.1. The OS module is configured with senderComponent, receiverComponent, and comparatorComponent. The OS alarms triggering and tasks scheduling is shown in table 4 and 5, respectively. In this case, senderComponent has been given alarm time as 100 and alarm cycle time as 100. That means that the

alarm fires the senderEvent when 100 Os ticks are elapsed and afterwards, this event activates the senderTask. It is triggered again at 200 Os ticks i.e. (100+100) Os ticks.
The basic task is assigned as priority 5, which is the highest priority among all tasks. In this design, senderTask, receiverTask, and comparatorTask have been assigned priority 4, 3, and 2 accordingly. Therefore, OS at first starts the senderTask and then the receiverTask based on the priority assignment. The alarm time of the senderTask triggers the senderEvent, and then the senderEvent activates the senderTask. After that, receiverTask is activated. Eventually comparatorTask compares the results of the sender and receiver component components. Table 5 depicts the OS tasks priority and event scheduling in the inter-ECU duplication and comparison.

Task Name	StartupTask	senderTask	receiverTask	comparatorTask
Task type	Basic	Extended	Extended	Extended
Schedule Preemptability	Full	Full	Full	Full
Priority	5	4	3	2
Activation Limit	1	-	-	-
Events	-	senderEvent	receiverEvent	comparatorEvent and RTEevent
Autostart	Yes	Yes	Yes	Yes

Table 5: Os tasks scheduling in inter-ECU duplication and comparison

In order to design multiple ECU experiment, additional basic software modules such as Can, CanIf, Com, EcuC, and PduR are needed in the BSW Builder tool. EcuC module includes Protocol Data Units (PDUs) to traverse them through the AUTOSAR communication stack. These PDUs are used in CanIf module for message transmission and reception. Each PDU size is 64 bits which is the maximum PDU size of the CAN communication protocol in AUTOSAR. The configuration of PDUs in EcuC is shown in figure 17.

Figure 17: PDUs in EcuC BSW Builder Tool

To activate CAN communication, CAN controller and hardware objects are added in CAN driver module. Baud rate is set as 125 kbps as the CAN controller of the VK-Eval-M3 ECU supports this bit rate. In order to transmit and receive messages, two CAN hardware objects are added shown in figure 18 and then configured the necessary field to activate the CAN driver.

Figure 18: Configuration of CAN driver

It is also possible to configure hardware filtering mask in the CAN driver module. This kind of hardware filtering is not used in this project. The loop-back attribute is kept disabled to avoid signal reception by the sender ECU.

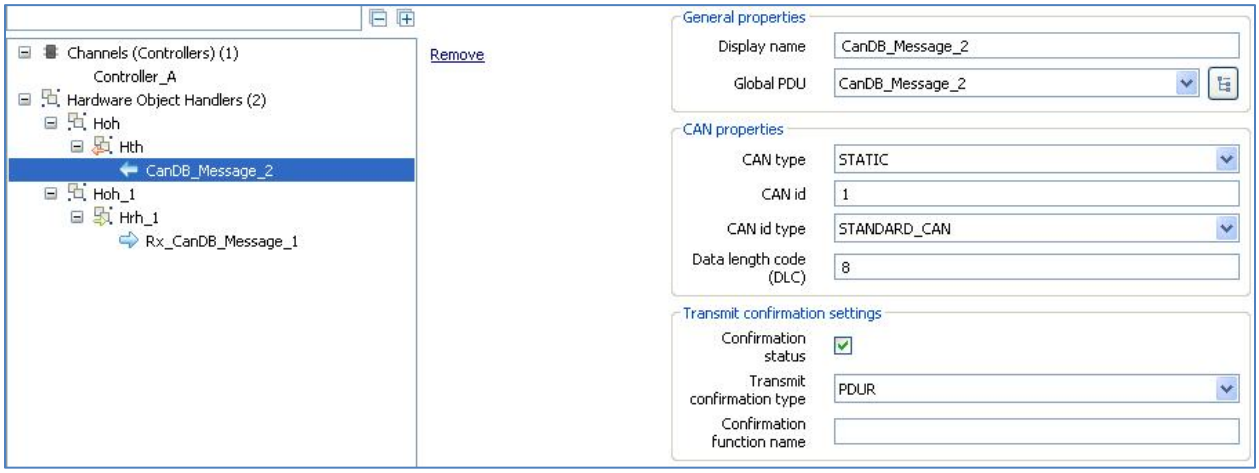

Figure 19: CanIf configuration of transmit-PDU

CanIf basic software module provides message transmission and reception channel with the help of the CAN controller, which is configured in the CAN driver module. Hardware Transmit Handler and Hardware Receive Handler are added in the Hardware Object Handlers (HOH) with a view to transmitting and receiving PDUs. Figure 19 shows the transmission of a PDU in the CanIf basic software module. In HTH section, we added a transmit-PDU. The necessary parameters such as CAN id, global PDUs for transmission, hardware object from CAN driver, and DLC length have been set up. The similar type of configuration is prepared in case of Hardware Receive Handler (HRH) section.

To communicate with the RTE and PduR, I-PDUs are added in the COM basic software module. These I-PDUs are grouped into a group and initiated from the basic task routine. Therefore, CanDImportedPdus is created which is linked in the Protocol Data Unit section shown in Figure 20. Four signals have been added in each I-PDU to transmit and receive four data. The required parameters of each signal has been configured such as signal bit position, size, initial value, signal transfer property, and signal endianess. Each signal must have a unique data configuration under the same I-PDU to avoid signal overlapping. Signal transfer property can be either triggered or pending. Triggered transfer property is used to transmit signal and Pending is used for signal reception, as described in Table 1. Each signal is mapped with respect to each system signal mapping. This is done by using the generating system model of this module. Then the desired signal will be visible in the dropdown box in the “override automatic system signal mapping” section shown in Figure 20.

Figure 20: COM configuration of transmit-PDU

In the final part of the basic software configuration, PduR is configured to indicate the direction of the I-PDUs, which are configured in the COM basic software configuration. It is used to route the I-PDUs towards the configured interfaces such as CAN and FlexRay interface. In this case, we used the CAN interface. Figure 21 shows the routing table configuration in which CanDB_message_1 is configured to receive I-PDUs (CanIf to Com) and CanDB_Message_2 is for transmission (Com to CanIf). The SDU length is same as the length of an I-PDU.

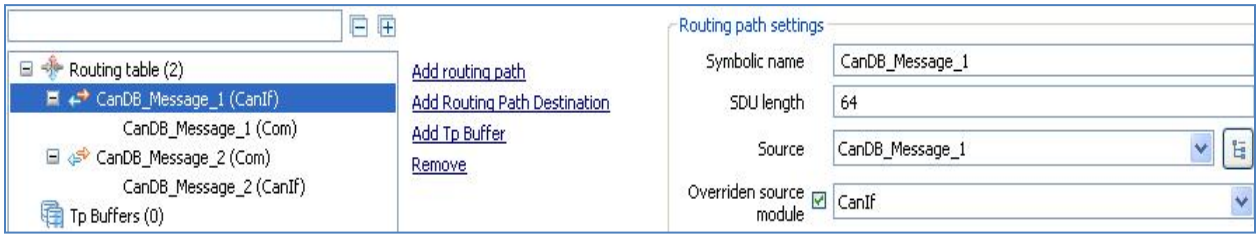

Figure 21: PduR routing table configuration

To make the design steps convenient, we created another project and then configured this project according to the above configuration except in Figure 19. In this figure, we assigned CAN id 2 for transmit-PDU, and CAN id 1 for receive-PDU. In this way, two different executable (.elf) files were created (after implementation work). Finally, we downloaded these executable files in two ECUs. This approach was considered for the remaining implementation of the thesis.

3.2 Triple Modular Redundancy

3.2.1 Single ECU Experiment

The design configuration of SWC and ECU Extraction is almost similar to section 3.1.1 except one additional software component is added in SWC Builder, and array type and integer type is used. The added software component, component3, works as a third redundant software component and the comparator component in section 3.1.1 represents as a voter component. The sorted elements are written one by one to the data elements of each interface. That means, four data elements are written using “data

write access" in each sender-receiver interface of each redundant software component as we have four sorted elements in each software component. On the other hand, voter component reads twelve times using "data read access".

The Os alarm scheduling in Table 6 shows alarm scheduling configuration. Initially component1 is activated due to the shortest alarm time in the design. Afterwards, component2, component3, and voterComponent are triggered accordingly.

Alarm Name	Component-1	Component-2	Component-3	voter Component
Alarm Time	10	20	30	40
Alarm Cycle Time	50	50	50	50
Autostart Option	Yes	Yes	Yes	Yes
Autostart Type	Absolute	Absolute	Absolute	Absolute
Alarm Action Type	Set Event	Set Event	Set Event	Set Event
set_Event	Component1Event	Component2Event	Component3Event	voterEvent
Set_EventTask	Component1Task	Component2Task	Component3Task	voterTask

Table 6: Os alarms scheduling in intra-ECU TMR

The OS task scheduling is similar to section 3.1.1 except one additional component is included in this design. StartupTask has the highest priority in Table 7 and activates once during the ECU initialization. VoterTask has been assigned the lowest priority as it does majority voting based on the produced results from Component1, Component2, and Component3.

Task Name	Startup Task	Component1 Task	Component2 Task	Component3 Task	Voter Task
Task Type	Basic	Extended	Extended	Extended	Extended
Schedule Preemptability	Full	Full	Full	Full	Full
Priority	10	2	2	2	1
Activation Limit	1	-	-	-	-
Events	-	Event1	Event2	Event3	vEvent and RTEevent
Autostart	Yes	Yes	Yes	Yes	Yes

Table 7: Os tasks scheduling in intra-ECU TMR

The design step of PORT, DIO, IO Hardware Abstraction, and RTE is similar to section 3.1.1 except in addition of one more component in RTE module. The new runnable is mapped with new extended task, and triggered with its event.

3.2.2 Multiple ECU Experiment

The design of triple modular redundancy in SWC Builder, Extract Builder, and BSW Builder is almost similar to section 3.1.2 description. In this design approach, sender component transmits PDUs and receiver component receives two PDUs from two ECUs. Then the voter component does majority voting. The additional task of this design with respect to section 3.1.2 is described in the following way.

The variation is made in EcuC, Com, CanIf, and PduR basic software modules. In EcuC module, three PDUs are added, and configured. Then One PDU is added and configured in HRH section of figure 19 as

we have two receiver ECUs in the network. After that, the second receive-PDU is added in Com and configured module similar to existing receive-PDU in figure 20. Finally, routing path is configured in PduR module for the added receive-PDU which similar to figure 21. In case of OS configuration, comparator component is replaced by voter.

To make the design of TMR steps easier, we create three different projects. In the first project, we configure according to the above description. In the second project, we assign CAN id 2 as a transmit-PDU, and CAN id 1 and 3 as receive-PDUs in CanIf BSW module. That means, ECU-2 transmits a PDU and receives PDUs from ECU-1 and ECU-3. Similarly, CAN id 3 is assigned as a transmit-PDU, and CAN id 1 and 2 is assigned as receive-PDUs in CanIf configuration i.e. ECU-3 transmits a PDU, and receives PDUs from ECU-1 and ECU-2. In this way, three different executable (.elf) files are created (after implementation work), and finally downloaded these executable files in three ECUs.

3.3 Distributed Consensus Protocol

To design consensus with membership protocol, four software components are considered and one additional software component to provide LED blinking in IO Hardware Abstraction. Four components are used for two rounds of message transmission and reception in this protocol. In the first round of consensus protocol, sender1 and receiver1 component transmits and receives signals. Then, sender2 and receiver2 component exchange messages in the second round. Later on, LED component is added and configured according to section 3.1.1. The receiver2 component and LED component is connected with sender-receiver interface in which receiver2 component provides a specific value in the provider port and LED component receives that value to blink the LED number.

In the second phase, the prototype of software components and IO Hardware Abstraction is instantiated in Extract Builder. Ports are connected in “port mappings” section of Extract Builder by selecting appropriate providers and requesters.

Alarm Name	Sender1Comp	Rceiver1Comp	Sender2Comp	Receiver2Comp
Alarm Time	100	100	100	100
Alarm Cycle Time	100	100	100	100
Autostart Option	Yes	Yes	Yes	Yes
Autosart Type	Relative	Relative	Relative	Relative
Alarm Action Type	Set Event	Set Event	Set Event	Set Event
Set Event	Sender1Event	Receiver1Event	Sender2Event	Receiver2Event
Set Event Task	Sender1Task	Receiver1Task	Sender2Task	Receiver2Task

Table 8: Os alarms scheduling of consensus protocol

In the third phase, basic software modules – Mcu, EcuM, Port, Dio, and CAN are added according to section 3.1.1 and 3.1.2. Later on, OS module is added in ECU Configuration. The Os alarm triggering and tasks scheduling is shown in table 8 and 9 respectively. In this design, alarm time and alarm cycle time are assigned equally to all components. Tasks are executed according to their assigned priority in Table 9. For instance, Sender1Task is executed at first and then receiver1Task, sender2Task, and receiver2Task consecutively. When Sender1Task is in running state, the remaining tasks will be in the waiting state. After execution of sender1Task, the second highest priority task i.e. receiver1Task is executed. Afterwards, the remaining tasks follow the same cycle.

Task Name	Startup Task	Sender1 Task	Receiver1 Task	Sender2 Task	Receiver2 Task
Task type	Basic	Extended	Extended	Extended	Extended
Schedule Preemptability	Full	Full	Full	Full	Full
Priority	5	4	3	2	1
Activation Limit	1	-	-	-	-
Events	-	Sender1 Event	Receiver1 Event	Sender2 Event	Receiver2 and RTE Event
Autostart	Yes	Yes	Yes	Yes	Yes

Table 9: Os tasks scheduling of consensus protocol

To design this protocol, CanIf, Com, PduR, and EcuC are added. Eight PDUs are added in EcuC module to communicate among the ECUs. Two of them are used for transmission in two round (one in each round) and six receive-PDUs are used for reception (three in each round). Each PDU is 64 bits in size. The design of RTE and IO Hardware Abstraction module is more or less similar to section 3.1.1.

To design this protocol, HOH is configured with Hth and Hrh for message transmission and reception in first round, and Hth_2 and Hth_2 for second round of transmission and reception, shown in Figure 22. Therefore, four CAN ids are used (1 to 4) in the first round and CAN id 5 to 8 are used for second round of communication. For instance, CAN id 1 is assigned in Hth section. It will receive message from the rest of the ECUs in the network e.g. CAN id 2, 3, and 4. The receiving CAN id is included in Hrh section. The same design is considered in Hth_2 and Hrh_2 section with different CAN ids. Figure 22 shows the configuration of one of the receive-PDUs for CAN id 2. The necessary parameters e.g. CAN id, global PDUs for reception, hardware objects from CAN driver, DLC length, etc are configured. In this case, CAN id 1 is assigned in Hth section which is transmitting one PDU (CanDB_Message_1_TX) in the first round and CAN id 5 in Hth_2 section is transmitting another PDU (CanDB_Message_2_TX) in the second round. On the other hand, Can id 1 is receiving PDUs of CAN id 2, 3, and 4 in the first round in Hrh section and CAN id 6, 7, and 8 in the second round in Hrh_2 section in figure 22. The first digit of CanDB message for instance in CanDB_Message_11_RX, the initial 1 represents the first round of message exchange and in CanDB_Message_21_RX, the initial 2 represents the second round of message exchange in consensus with membership protocol.

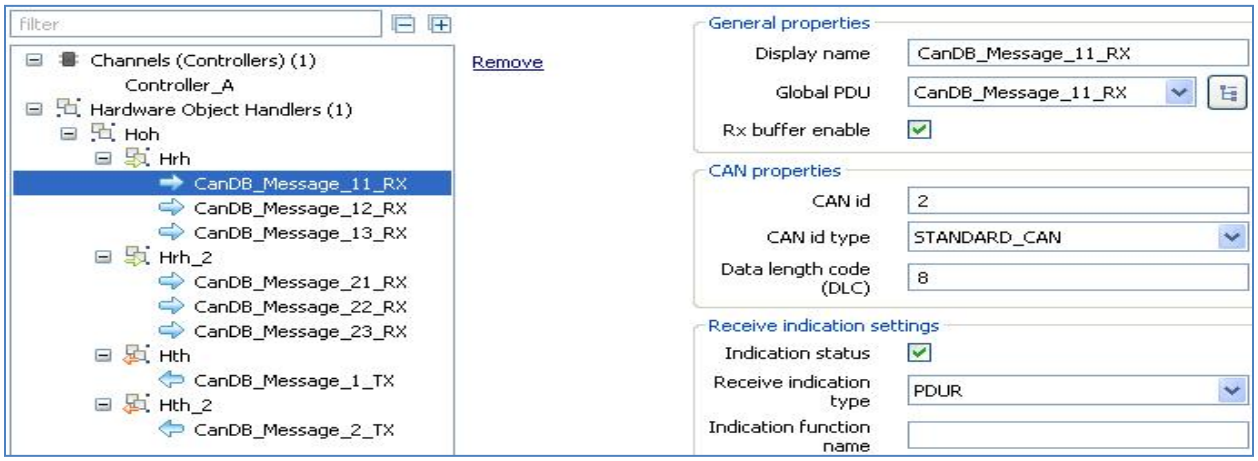

Figure 22: CanIf configuration of receive-PDU

To configure COM, PDUs and signals are added in COM BSW module. The configuration step of this protocol is similar to section 3.1.2. In this setting, eight PDUs are added. Four PDUs are used, one for message transmission and three for message reception from the remaining three ECUs, in the first round of consensus protocol. The remaining four PDUs are used in the same way for second round of message exchange in consensus protocol. Figure 23 shows one of the configurations of receive-PDU which is similar to section 3.1.2. Moreover, timeout and notification are taken into consideration to design membership protocol which is explained later part of this section.

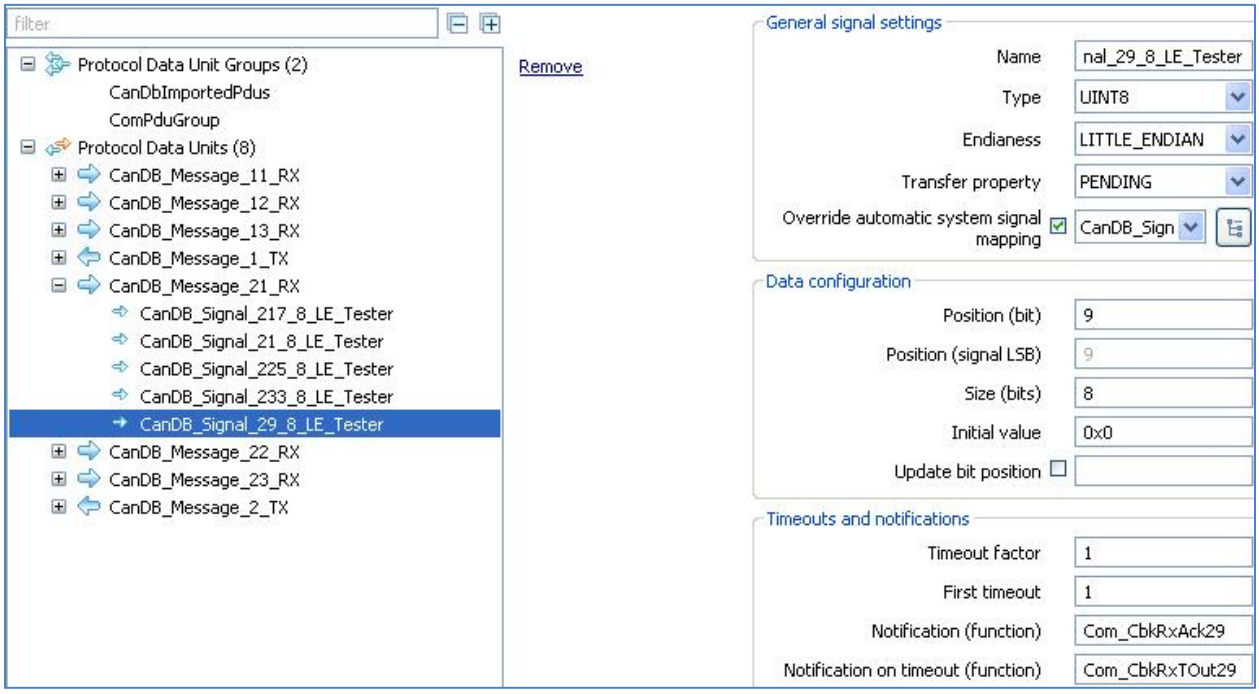

Figure 23: COM configuration of receive-PDU signal

The ECU which is receiving this signal waits for more than one second (one second plus execution time to reach the checking of incoming signals value from the PDU buffer in receiver component) based on the periodic cyclic activation of receiver components. If this signal doesn't arrive within this time, the receiving ECU will replace this signal's value by the initial value. In this case, initial value is assigned as zero.

Figure 24: PduR routing table configuration

The PDU Router (PduR) routes the I-PDUs in AUTOSAR communication stack. The description is similar to section 3.1.2. In this configuration, routing table consists of eight PDUs. Figure 24 shows the configuration of routing table where CanDB_Message_1_TX is configured to transmit I-PDUs (Com to CanIf) and CanDB_Message_21_RX is for reception (CanIf to Com) of I-PDUs. The necessary fields are filled up to run this system properly. “Data provision” indicates the direct transmission of data from CanIf interface for further processing.

To design membership protocol, timeout and notification section are configured in figure 23. Moreover, manual configuration is needed to activate the timeout action. As can be seen in figure 25, the ComRxDataTimeoutAction field is changed from COM_TIMEOUT_DATA_ACTION_NONE to COM_TIMEOUT_DATA_ACTION_REPLACE. The same replacement is manually done to all the incoming signals of the PDUs from three ECUs i.e. six incoming signals of six PDUs, three for each round of message exchange in consensus protocol with membership protocol.

```
.ComSignalInitValue = &Com_SignalInitValue_CanDB_Signal_29_8_LE_Tester
.ComBitPosition = 9,
.ComBitSize = 8,
.ComSignalEndianess = COM_LITTLE_ENDIAN,
.ComSignalType = UINT8,
.Com_Arc_IsSignalGroup = 0,
.ComGroupSignal = NULL,

.ComRxDataTimeoutAction = COM_TIMEOUT_DATA_ACTION_REPLACE,
```

Figure 25: COM timeout in COM post build configuration file (Com_PbCfg.c)

In the final step of the design of consensus with membership protocol, we create four different projects. In the first project, we configure according to the above description. In the second project, we assign CAN id 2 as a transmit-PDU, and CAN id 1, 3, and 4 as receive-PDUs in CanIf BSW module (similar to figure 22). That means, ECU-2 transmits a PDU and receives PDUs from ECU-1, ECU-3, and ECU-4. Similarly, CAN id 3 is assigned as a transmit-PDU, and CAN id 1, 2, and 4 is assigned as receive-PDUs in CanIf configuration i.e. ECU-3 transmits a PDU, and receives PDUs from ECU-1, ECU-2, and ECU-4. Finally, CAN id 4 is assigned as a transmit-PDU, and CAN id 1, 2, and 3 is assigned as receive-PDUs in CanIf configuration. Moreover, the configuration of timeout and notification is configured (similar to figure 22 and 25) in order to activate membership protocol. The incoming signals’ timeout and notification functions are mentioned in the code of each receiver component in Appendix C section of this report. In this way, four different executable (.elf) files are created (after implementation work), and finally downloaded these executable files in four ECUs.

4 Implementation

4.1 Development Methodology in Arctic Core

The methodology steps of Arctic implementation of AUTOSAR are shown in figure 26. The description of each step is explained in the following way.

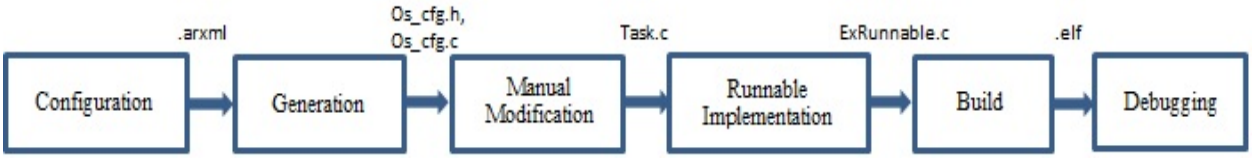

Figure 26: Development methodology in Arctic studio

SWC configuration and BSW configuration: In the first step software components and the required BSW modules are defined. This configuration includes software component definition, communication interfaces and ports. In this phase, the required BSW modules for the specific ECU are configured as well where it includes OS tasks scheduling, signal and PDU configuration, system model generation for IO hardware Abstraction and COM, mapping of communication signals, etc. The configuration phase is accomplished with SWC builder and BSW builder in Arctic Studio.

Generation: After configuring all the BSW modules and software components, it is needed to configure and generate the RTE. In this step previous configuration files are used as input. After this step, all the contract files and stubs are generated. RTE Builder does this procedure in Arctic Studio.

Runnable Implementation: RTE generator generates contract files in which it contains the prototype of the software components runnable method. User needs to implement the original algorithm using these function prototypes.

Manual Modification is needed to adjust and modify some of the generated files because all implementations are not following the AUTOSAR specification. For instance, the extended tasks in Tasks.c file should be removed as the definitions are generated in Rte.c file. Moreover, the necessary routines inside the basic task method in Tasks.c need to invoke to run the configurations smoothly. For instance, EcuM_StartupTwo routine to start up the ECU, Com_IpduGroupStart routine to start up the Com IPDU group, CanIf_SetControllerMode routine to set the CanIf mode settings in the configuration, and so on. These methods, depending on the projects, are needed inside the basic task method as this task initiates the required initialization of the ECU.

Build step compiles all the generated files and runnables, and builds the executable (.elf) file which is loaded in ECU for debugging and testing.

Debugging is done after loading the executable file in the ECU. It helps to check the application whether it is working correctly as anticipated. If modification or correction is required then development process can be resumed again from the previous step.

4.2 Intra-ECU Implementation

The implementation of triple modular redundancy, and duplication and comparison in single ECU are described in the following way.

Triple Modular Redundancy

The implementation of triple modular redundancy uses three redundant software components, one voter component, and one additional component to facilitate LED blinking. The redundant software components write the data elements in the provider port and then the voter component receives the data from the require port of the sender-receiver interface. Each redundant software component uses separate sender-receiver interface where each interface contains four data elements to contain the sorted elements.

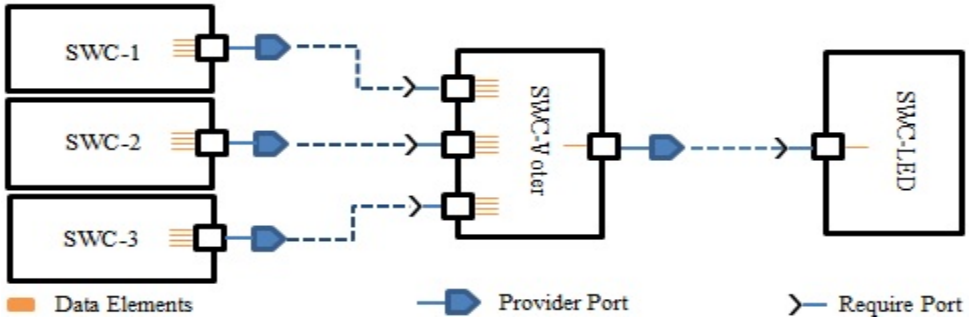

Figure 27: TMR in AUTOSAR software components

The LED component uses RPort to read data element from one additional PPort of voter component. Thus voter component has three RPort and one PPort to receive data from three redundant software components, and it provides data to blink the desired LED number. Moreover, the LED component uses four additional RPort to interface with Digital-Output interface which in turn communicate with IO Hardware Abstraction via RTE. The IO Hardware Abstraction sets and gets the desired LED number using Dio write and read channel. To read or write channel, Dio uses the configured port and pin number to finish the operation.

The runnable methods of redundant software components and voter software component are triggered based on the OS design of triple modular redundancy in single ECU. Thus, the first redundant software component is triggered in the beginning. In the runnable method of first redundant software component, quick sort algorithm is implemented with four elements. The exact name of the runnable method is taken from the RTE generated contract file of this redundant component. The sorted elements are passed as parameters to the four provider method declarations which are also generated in the contract file of first redundant component. The same procedure is followed for the remaining two redundant software components. Figure 27 shows the graphical view of TMR implementation in AUTOSAR software components.

The voter component executes after getting the produced result from the redundant software components. It reads four data-elements of each redundant software component using the generated method declarations in the contract file of the comparator component. Then the TMR algorithm is executed. In this algorithm, faulty redundant part is recovered by one of the correct redundant software components. The desired LED number is passed as parameter to the provider method of sender-receiver interface in which LED component is the receiver part of this interface. In the runnable method of LED component, the data element is received and then the LED number is set to the corresponding require port method. The LED number indicates the recovery part of the specific component e.g. LED 3 blinks when third component is faulty and is replaced by the correct result of the first or second software component.

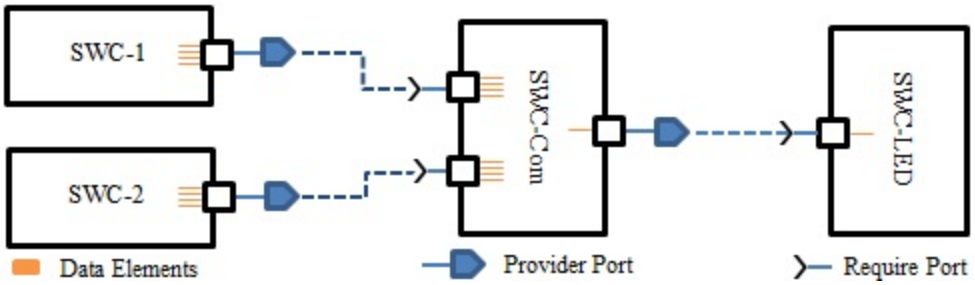

Figure 28: Duplication and comparison in AUTOSAR software components

Duplication and Comparison

To implement the duplication and comparison, the same approach (TMR) is considered. In this case, comparator software component compares the produced result from the two redundant software components. In the final step, LED 2 is blinked to check the comparison operation. Figure 28 shows the graphical view of duplication and comparison in AUTOSAR SW-Cs.

4.3 Inter-ECU Implementation

The implementation steps of triple modular redundancy, duplication and comparison, and consensus with membership protocol in multiple ECUs are organized in the following way.

Triple Modular Redundancy

To implement triple modular redundancy in three ECUs, four software components have been considered – sender component, receiver component, voter component, and LED component. The sender component is triggered in the beginning and then the receiver component. This is due to the OS priority assignment which is mentioned in design chapter. In the runnable method of the sender component, it transmits sorted array elements using Com_SendSignal and Com_MainFunctionRx method.

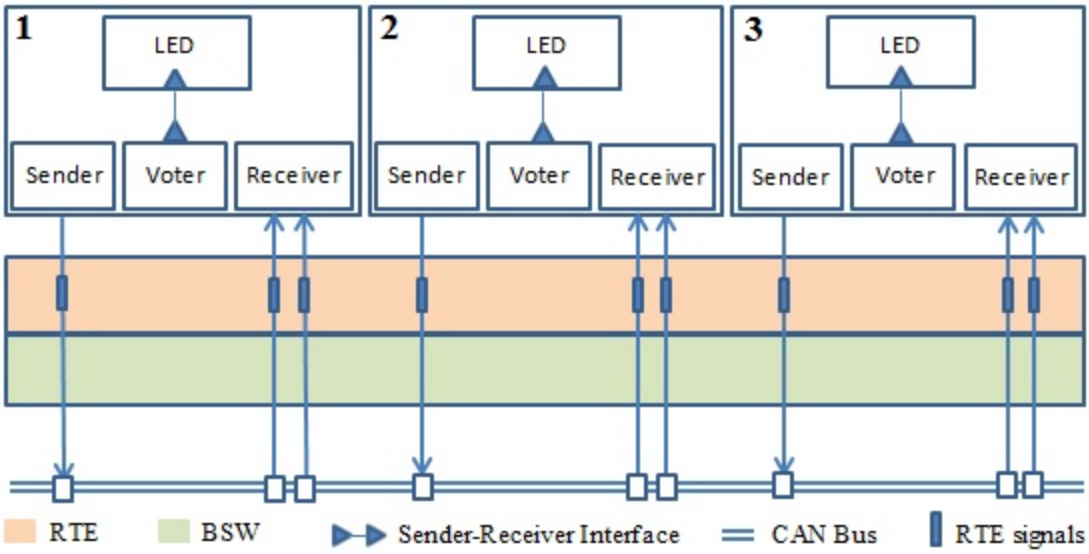

Figure 29: Inter-ECU TMR implementation in AUTOSAR

In this case, four signals are used to transmit four sorted array elements. On the other hand, receiver component receives signals using Com_MainFunctionRx and then Com_ReceiveSignal method in the

runnable method. Therefore, each ECU transmits four signals and receives eight signals from the rest of the two ECUs in the network. Moreover, a state (flag) variable has been assigned at the end of the receiver component to ensure that the compiler has executed this runnable method.

In the runnable method of voter component, TMR algorithm takes majority voting and recovers (if there is any faulty component) based on the stored result in three different arrays. In the beginning of TMR operation, the state variable is used to make sure that the receiver component has been executed. The algorithm follows in the same way of TMR in intra-ECU communication.
The LED component works according to the description of TMR in single ECU. Figure 29 shows the inter-ECU TMR implementation.

Duplication and Comparison

This implementation steps are as same as the description of TMR for inter-ECU communication except some modifications. In this case, two ECUs are used and voter software component is replaced by comparator software component.

The algorithm of comparator software component is similar to intra-ECU duplication and comparison. The implementation work of redundant software components is similar to the implementation of inter-ECU TMR. Figure 30 shows the implementation of inter-ECU duplication and comparison. LED 2 is blinked eventually to indicate the successful duplication and comparison operation, or LED 1 is blinked to detect error between the two ECUs.

Distributed Consensus Protocol

To implement the consensus protocol, five software components have been considered in each ECU. The components are first-sender component (sender1), and first-receiver component (receiver1), second-sender component (sender2) and second–receiver component (receiver2), and LED component. The components are triggered based on the design of OS tasks priority and scheduling, see design chapter.

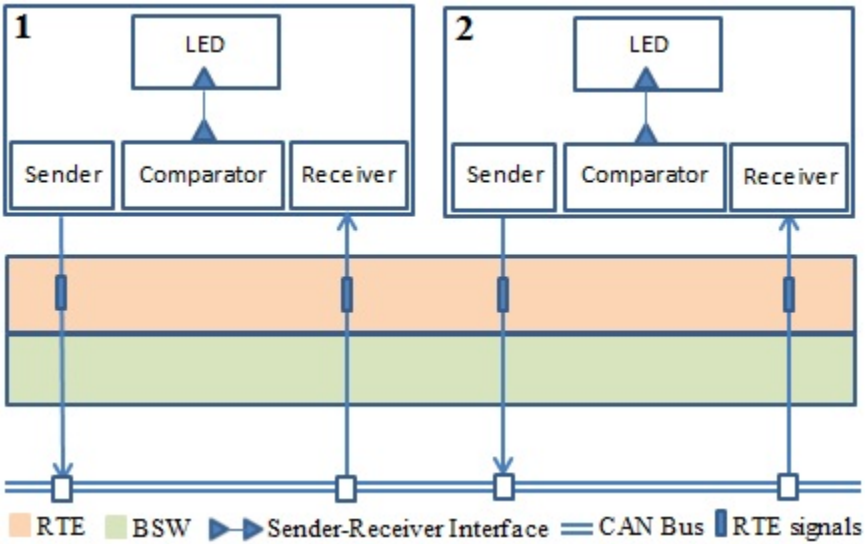

Figure 30: Inter-ECU duplication and comparison implementation in AUTOSAR

In the first round of communication, sender1 component of each ECU transmits signal using Com_SendSignal and Com_MainFunctionTx method. Then the receiver1 component receives signals

using Com_ReceiveSignal routines from the rest of the ECUs in the network. Receiver1 checks the round number and then stores in the first array. Moreover, receiver1 stores its own local value (sent by sender1). Then receiver1 calculates maximum from the first array. In the second round of this protocol, sender2 component of each ECU transmits the maximum value from its local array. Afterwards, the receiver2 component receives the maximum values from all the ECUs in the network and stores in the second array after checking the round number. Then receiver2 calculates maximum value from the second local array and compares this maximum value in each index of that array. If the values are same, the ECUs will reach consensus. In the final step, the agreed number of the second local array invokes the LED component to blink the desired LED e.g. LED 4 blinks when the maximum value is 4 in the network.

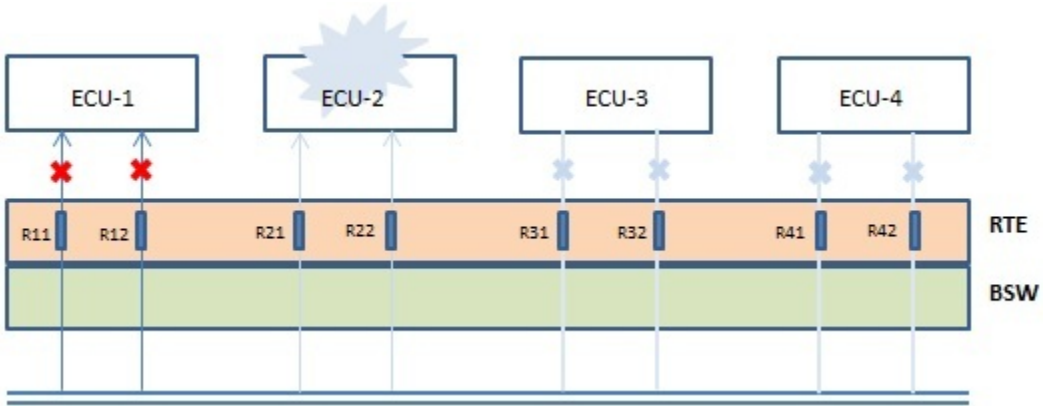

Figure 31: Timeout of signals in consensus protocol for ECU-2

The membership property works according to timeout and notification configuration shown in design chapter. The deadline monitoring for timeout counter is 1. Initial value 0 is replaced when the time expires for the specific signal. In our design (section 3.3), receiver1 and receiver2 are triggered periodically after one second interval and the signals are configured individually for each ECU. In each signal, we have configured 1 as first time out and periodic timeout (time out factor). Therefore, the time out will be – one second plus the execution time in receiver component to check the received signal's value from the PDU buffer. If the received value is zero, then time out occurs. Then the corresponding proposed value of the left ECU disappears. The new maximum value is chosen from the second local array of each ECU, and afterwards blinks the desired LED number. For instance, assuming four ECUs (id - 1 to 4) are in the network and all of them are blinking LED 4 (the maximum number at the end of second round). Then the fourth ECU has been disconnected. After that, all the ECUs in the network will blink LED 3 (the maximum value in the second local array) after the expiration of time out. The fourth ECU can join at any time. If it joins, the ECUs will blink LED 4 again.

The time out and notification callback functions are invoked after time out expiration. Each PDU has four signals. In the runnable method of receiver1 component, the callback functions of the receiving PDUs from the rest of the ECUs are defined with null statements. The same process is followed in the runnable of receiver2. As an example, ECU-1 receives signals from ECU-2, ECU-3, and ECU-4 via RTE. The corresponding variable of ComRxDataTimeoutAction needs to be replaced which is shown in figure 25. The time out of signals in ECU-2 is shown in figure 31. Therefore, receiver1 and receiver2 software components in ECU-1 do not receive signals (red color indicated in figure 31) due to failure of ECU-2.

ECU-3 and ECU-4 do not receive signals from ECU-2 as well. In the implementation of the consensus, all signals are reinitialized to their initial value at the end of receiver component to avoid any garbage value from the network in the next cycle of invocation.

4.4 Timing Overhead in Consensus Protocol

We measure the execution-time (system ticks) of the SW-Cs in the consensus protocol. The SW-Cs are executed by the static priority scheduling of AUTOSAR OS tasks. The system tick is the clock frequency of the VK-Eval-M3 ECU, which is 72MHz. We calculate the execution time in micro second (µs) unit using the clock frequency. We run the system 10 times and observe that the same value appear in each case. Therefore, we mention only one value in each task in Table 10. In our approach, we set a break point in ECU1 after observing the 10 times data flow in CAN monitor pro software. Then, we add ECU2 and take the system tick count from ECU1. We follow the same approach after adding ECU3 and ECU4 accordingly.

Table 10 shows the system ticks of the consensus protocol with respect to the increasing number of ECUs. Sender1Task sends its local value and round number in the first round, and sender2Task sends the calculated maximum value by Receiver1Task and second round number. In Sender2Task, there is one extra statement to fetch the maximum value from the Receiver1Task. Therefore, Sender2Task takes 16 system ticks more than Sender1Task except S2 (ECU1 ECU2 ECU3). Probably, the signals are taking one more system ticks due to bus contention in asynchronous CAN communication.

In the design of consensus with member protocol, Receiver1Task receives first round of messages from the other ECUs. It receives three signals from three ECUs and three more signals to check the round number. The loopback is disabled in CAN configuration. Therefore, the receiver1Task stores its own local value in the array. Then the receiving values are stored after checking the first round number of the ECUs in the network. In our design, Receiver1Task is triggered after one second interval. We have checked the null value of the signal inside Receiver1Task. Therefore, the timeout will be - one second + execution time of the signals' null value in Rceiver1Task. This timeout is represented using the reception deadline monitoring counter which is decremented after checking the initial value in the PDU buffer. The initial counter value is set to 1, and it is decremented to 0 when the signals value is zero in the buffer. In each call of decrement time-out counter, an empty notification function is invoked in Receiver1Task. The notification function names are configured for the three receiving signals in the BSW configuration settings. After storing the signals value, the values and the signals of round number is reset in the buffer. The same approach is considered for Receiver2Task. Therefore, Receiver1Task and Receiver2Task in Table 10 show the clear decrement of the systems ticks with respect to the addition of the remaining ECUs one by one in the network.

For instance, ECU1 takes 14383 system ticks in Receiver1Task. In this case, Receiver1 is waiting for three signal values from other ECUs in the network. But they are dead. Therefore, decrement counter invokes the time-out notification function for each signal. Then, we add ECU2, and measure the system ticks 12493 in ECU1. This is lower than the first approach as the Receiver1Task in ECU1 invokes the time-out notification function for the remaining two signals from the remaining ECUs in the network. After that, we add ECU3 and ECU4 respectively. We can see the decrement as well when ECU3 is in the network with ECU1 and ECU2. In this stage, the measured system tick is lower than the previous one which is 10600. Finally, there will be no time-out when ECU4 is in the network. Therefore, the execution of the number of system ticks is the lowest among the system ticks in Receiver1Task.

No of ECUs	Sys Ticks of Autosar OS Tasks in ECU1			
	Sender1Task (S1)	Receiver1Task (R1)	Sender2Task (S2)	Receiver2Task (R2)
ECU1	8506	14383	8522	14571
ECU1 ECU2	8506	12493	8522	12682
ECU1 ECU2 ECU3	8506	10600	8523	10790
ECU1 ECU2 ECU3 ECU4	8506	8705	8522	8896

Table 10: System Ticks value of AUTOSAR OS Tasks

The difference of system ticks in R1 and R2 is shown in Table 11. In this case, we can see that the difference is 188, 189, 190, and 191 system ticks. This is due to the addition of LED module in Receiver2Task. In addition to that, Receiver2Task calculates agreement to reach consensus. Moreover, the difference is incremented one by one because LED 1 blinks when difference is 188, LED 2 blinks when difference is 189, and so on.

The difference of system ticks in R1 is shown in Table 11. In this case, we can see that the difference is 1890, 1893, and 1895 system ticks. This is due to the maximum calculation, and round number checking and then storing the values in the array. In case of maximum calculation, the more value we have in the array the more system ticks will be required to check maximum and assignment (or reassignment) of the new maximum value. On the other hand, we check the round number and then store its corresponding value in the array.

Difference of R1 and R2 in ECUs	
Subtraction	***Sys Ticks***
R2(ECU1) – R1(ECU1)	188
R2(ECU2) – R1(ECU2)	189
R2(ECU3) – R1(ECU3)	190
R2(ECU4) – R1(ECU4)	191
Difference of R1 OS Tasks in ECUs	
R1(ECU1) – R1(ECU1 ECU2)	1890
R1(ECU1 ECU2) – R1(ECU1 ECU2 ECU3)	1893
R1(ECU1 ECU2 ECU3) – R1(ECU1 ECU2 ECU3 ECU4)	1895
Difference of R2 OS Tasks in ECUs	
R2(ECU1) – R2(ECU1 ECU2)	1889
R2(ECU1 ECU2) – R2(ECU1 ECU2 ECU3)	1892
R2(ECU1 ECU2 ECU3) – R2(ECU1 ECU2 ECU3 ECU4)	1894

Table 11: Difference of System Ticks value in the receiver tasks in AUTOSAR

For instance, difference of system ticks 1890 and 1893 has timeout of signals. Hence, the signals related to round number will be zero and the corresponding values of the signals' (message) statement will not be executed and stored in the array. But, when the difference is 1895, there are two different states – R1(ECU1 ECU2 ECU3) which has one timeout state and R1(ECU1 ECU2 ECU3 ECU4) which has no timeout state. In this case, the measured execution of system ticks is high due to the maximum calculation, and the round number checking and storing in the array.

No of ECUs	Execution Time of Sys Ticks in µs of Autosar OS Tasks in ECU1			
	S1	R1	S2	R2
ECU1	118.139	199.764	118.361	202.375
ECU1 ECU2	118.139	173.514	118.361	176.139
ECU1 ECU2 ECU3	118.139	147.222	118.375	149.861
ECU1 ECU2 ECU3 ECU4	118.139	120.903	118.361	123.556

Table 12: Execution time in µs of the receiving tasks in AUTOSAR

The design and implementation of Receiver2Task is as same as Receiver1Task except the LED module and the calculation of the agreed values in the second array. Therefore, the system ticks vary which is shown in Table 11. In this case, the difference of R2 is 1889, 1892, and 1894. This is due to the LED blinks time increase in the conditional statements, agreement calculation, and round number checking and storing in the second array. The explanation is as same as given above for the difference of R1.

We calculate the execution time of system ticks in micro seconds. The measurement is shown in Table 12 using the clock frequency of the ECU. For instance, S1(ECU1) is 8506 ticks and ECU clock frequency is 72 MHz. Therefore, S1(ECU1) will be 8506/72000000 seconds which is 118.139 micro seconds.

4.5 Alternative ways of Implementation

Duplication and Comparison

There are two ways to implement duplication and comparison mechanism – intra- and inter-ECU experiment. In intra-ECU experiment, redundant software components feed results to the comparator component using sender-receiver interface APIs. In this case, the redundant component writes data in the provider-port and then the comparator component receives the data from the require-port

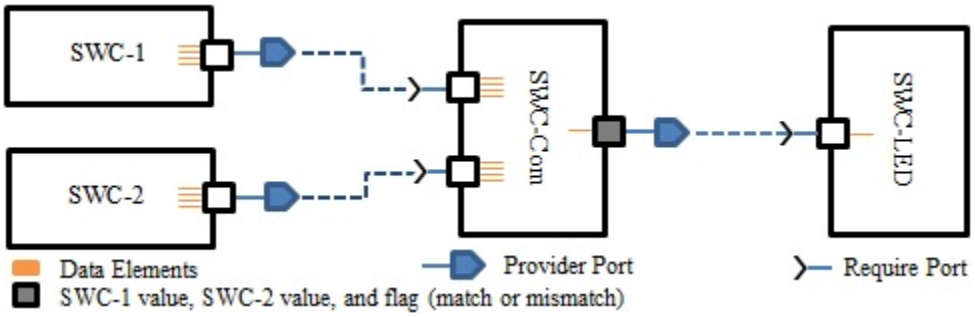

Figure 32: An alternative implementation of duplication and comparison

The data can be lost in both experiments. One of the reasons of the data losses in intra-ECU experiment is due to the failure of the redundant component. In such case, invalid value (uninitialized) is written to the provider port and therefore the comparator component will produce an in-correct result. In our implementation, the comparator component indicates an output flag (match or mismatch). However, in an alternative implementation, the comparator can have three flags, i.e., computed results of the redundant components and the output flag, which can be seen in figure 32. Thus, we can trust the outcome of the comparator as the alternative representation proves the accuracy of the comparator. This helps us to identify omission failure and value failure. The omission failure occurs when the computed result of the redundant component is missing whereas the value failure is detected by the output flag of the

comparator. The comparator component can also be programmed in a way that can detect faulty redundant component by storing previous sessions. Moreover, the comparator can be duplicated as well in a fault-tolerant system.

In contrast, the data can be lost in inter-ECU experiment due to communication failure. In inter-ECU sender-receiver communication, the data is queued in the receiver side. In case of data reception, AUTOSAR provides two categories of data-reception semantics - Last-is-Best semantics and Event-Queuing semantics. The former semantics uses the last received data whereas Event-Queuing semantics uses the history of data. We have used the Last-is-Best semantics because of its simplicity. However, an alternative implementation could be to use Event-Queuing semantics.

The time-out mechanism can be used to restrict the communication delay within a range. In our implementation, we did not consider the time-out mechanism. However, duplication and comparison can be implemented with time-out mechanism.

Triple Modular Redundancy (TMR)

The implementation is conducted similarly as TTR-FR algorithm [5] in intra- and inter-ECU experiment. In the implementation, the voter component takes majority voting, and does forward recovery. The forward recovery ensures an error-free state in which the erroneous result of the redundant component is replaced by another redundant component. Alternatively, the voter component can indicate four flags, which can be seen in figure 33. Thus, we can trust the outcome of the voter component. In order to avoid single point of failure and to mask redundancy, TMR can also be designed with multiple voters and multiple inputs such as multi-stage TMR.

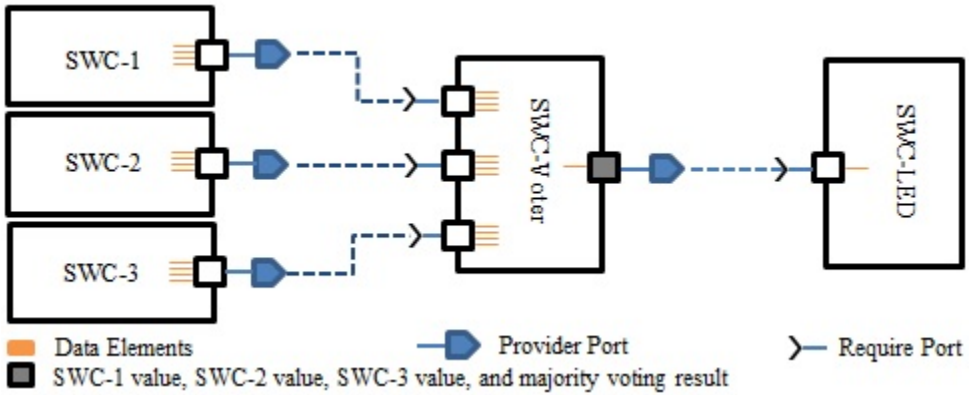

Figure 33: An alternative implementation of TMR

Another implementation of TMR can have the time-out mechanism. In such case, if one of the redundant modules fails, TMR should work similarly as duplication and comparison.

The TMR can be used in various distributed applications in order to achieve fault tolerance. In such applications, the designer must need to prevent a fault-containment region, i.e., a fault in one module causes faults in the other module or the voting element. Therefore, this region is handled with care in distributed TMR system.

Consensus protocol

The consensus protocol can be synchronous or asynchronous depending on the system requirements. The time-out mechanism can be used in the synchronous consensus protocol. We have used the time-out

mechanism in the consensus protocol to implement the membership property. However, the implemented consensus protocol is not synchronous because the joining node does not know exactly which component is currently executing by the existing nodes. Therefore, the nodes are not in the same cycle of their components' execution.

The design of message transmission and reception of the implemented consensus protocol is shown in table-13 where the nodes transmit messages using CAN identifier 1 to 4 in the first round and CAN identifier 5 to 8 in the second round. As can be seen in table-14, the nodes should initiate second round of message transmission after completion of the first round in the synchronous consensus protocol whereas the implemented consensus protocol transmits messages asynchronously.

Nodes	Transmit (CAN ID) Round-1	Receive (CAN IDs) Round-1	Transmit (CAN ID) Round-2	Receive (CAN IDs) Round-2
Node-1	1	2, 3, 4	5	6, 7, 8
Node-2	2	1, 3, 4	6	5, 7, 8
Node-3	3	1, 2, 4	7	5, 6, 8
Node-4	4	1, 2, 3	8	5, 6, 7

Table 13: Message transmission and reception in the implemented consensus protocol

We have measured the execution-time of the implemented consensus protocol, which is described in the timing overhead section of the implementation chapter. An alternative measurement could be based on the message transmission and reception within the time-out limit.

CAN Monitor (CAN IDs)	***CAN Monitor (CAN IDs) in synchronous consensus***
1 ------ Round-1 Node-1	1 ------ Round-1 Node-1
5 ------ Round-2 Node-1	2 ------ Round-1 Node-2
2 ------ Round-1 Node-2	3 ------ Round-1 Node-3
6 ------ Round-2 Node-2	4 ------ Round-1 Node-4
3 ------ Round-1 Node-3	5 ------ Round-2 Node-1
7 ------ Round-2 Node-3	6 ------ Round-2 Node-2
4 ------ Round-1 Node-4	7 ------ Round-2 Node-3
8 ------ Round-2 Node-4	8 ------ Round-2 Node-4

Table 14: Comparison of the implemented consensus and the synchronous consensus message transmission in CAN Monitor

5 Conclusions

The study shows that fault-tolerance mechanisms can be implemented in AUTOSAR application layer in a single node and multiple node experiments. In single node experiment, application software components communicate via RTE. However, this experiment can also be investigated via the RTE and the basic software [24]. In contrast, multiple node experiment is conducted through AUTOSAR communication stack. In this case, CAN communication protocol is used because it is cost-effective and is widely used in automotive systems. In addition, the timing overhead of the consensus protocol shows that the execution-time decreases with the increasing number of joining nodes.

The implemented fault-tolerance mechanisms can be investigated in different ways, which is discussed in the implementation chapter. In such ways, we can add more status information to the outcome of the duplication and comparison and triple modular redundancy.

The implementation of the fault-tolerance mechanisms in AUTOSAR comprises two steps. The first step is to configure the software components and the basic software modules in AUTOSAR. In the second step, these mechanisms are implemented by using the generated skeleton in the configuration step. This implies that the implementation of AUTOSAR based application is different with respect to traditional implementation.

An extension of the thesis would be to investigate the alternative ways of implementing fault-tolerance mechanisms. Additional work includes the implementation of fault-tolerance over Ethernet and then incorporating this system in AUTOSAR. It would be also interesting to investigate fault-tolerance in AUTOSAR basic software layer.

References

[1] Abelein, U.; Lochner, H.; Hahn, D.; Straube, S., “Complexity, quality and robustness - the challenge of tomorrow's automotive electronics”, Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 870-871, March 2012.

[2] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst, “AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures,” in Convergence International Congress & Exposition on Transportation Electronics, pp. 325-332, Oct. 2004.

[3] Erik Coelingh and Stefan Solyomg, “All Aboard the Robotic Road Train”, IEEE Spectrum, November 2012.

[4] Evan Ackerman, “Google's Autonomous Car Takes To The Streets”, IEEE Spectrum, Oct 12, 2010.

[5] Alexandersson, Ruben; Karlsson, Johan: Fault injection-based assessment of aspect-oriented implementation of fault tolerance. 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, 27-30 June 2011, pp. 303-314 . ISBN/ISSN: 978-142449233-6

[6] Explanation of Error Handling on Application Level [PDF], V1.0.0, R4.0, Rev 1: AUTOSAR 2009.

[7] AUTomotive Open Standard ARchitecture, http://www.autosar.org.

[8] Technical Overview [PDF], V2.2.2, R3.1, Rev 0001: AUTOSAR 2008.

[9] Specification of the Virtual Functional Bus [PDF], V1.1.0, R3.1, Rev 5: AUTOSAR 2010.

[10] Specification of RTE [PDF], V2.3.0, R3.1, Rev 5: AUTOSAR 2010.

[11] Specification of Communication [PDF], V3.2.0, R3.1, Rev 5: AUTOSAR 2010.

[12] Specification of PDU Router [PDF], V2.3.0, R3.1, Rev 5: AUTOSAR 2010.

[13] Specification of CAN Interface [PDF], V3.2.0, R3.1, Rev 5: AUTOSAR 2010.

[14] Specification of CAN Driver [PDF], V2.4.0, R3.1, Rev 5: AUTOSAR 2010.

[15] Specification of Operating System [PDF], V3.1.1, R3.1, Rev 0002: AUTOSAR 2009.

[16] Specification of Diagnostic Communication Manager [PDF], V3.3.0, R3.1, Rev 5: AUTOSAR 2010

[17] Specification of Diagnostics Event Manager [PDF], V3.1.0, R3.1, Rev 5: AUTOSAR 2010.

[18] Specification of Development Error Tracer [PDF], V2.2.2, R3.1, Rev 0001: AUTOSAR 2008.

[19] Specification of PORT Driver [PDF], V3.1.0, R3.1, Rev 0004: AUTOSAR 2010.

[20] Specification of DIO Driver [PDF], V2.2.2, R3.1, Rev 0001: AUTOSAR 2008.

[21] Specification of MCU Driver [PDF], V2.3.0, R3.1, Rev 0004: AUTOSAR 2010.

[22] Specification of GPT Driver [PDF], V2.2.2, R3.1, Rev 0001: AUTOSAR 2008.

[23] Specification of I/O Hardware Abstraction [PDF], V2.0.2, R3.1, Rev 0001: AUTOSAR 2008.

[24] T. Piper, S. Winter, P. Manns, N. Suri, “Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework”, IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boson, MA, USA, June 25-28, IEEE Computer Society 2012, ISBN 978-1-4673-1624-8.

[25] J. Kim, G. Bhatia, R. (Raj) Rajkumar, M. Jochim, “An AUTOSAR-Compliant Automotive for Meeting Reliability and Timing Constraints”, SAE 2011 World Congress & Exhibition, Technical paper, April 2011, Detroit, Michigan, United States.

[26] C. Lu, J.-C. Fabre, and M.-O. Killijian, “An approach for improving Fault-Tolerance in Automotive Modular Embedded Software,” in Proc. of the 17th International Conference on Real-Time and Network Systems (RTNS), 2009.

[27] J.-C. Fabre, M.-O. Killijian, and M. Taiani “Robustness of Automotive Applications Using Reflective Computing: Lessons learnt,” in Proc of the ACM Symposium on Applied Computing, pp 230-235, March 2011.

[28] Evaluation board: ARCCORE AB [Online]. http://www.arccore.com/products/vk-evb/

[29] Development Tools: ARCCORE AB [Online]. http://www.arccore.com/products/

[30] CAN Monitor Lite: WGSoft [Online]. http://www.can232.com/?page_id=75

[31] Debugging within Arctic studio with Raisonance support using GDB and openOCD: ARCCORE AB [Online]. [Accessed - December 24, 2012]. http://arccore.com/wiki/Debugging