electronics

MDPI

Article

Design and Implementation Procedure for an Advanced Driver Assistance System Based on an Open Source AUTOSAR

Jaeho Park and Byoung Wook Choi *

Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul 01811, Korea; jaeho@seoultech.ac.kr

* Correspondence: bwchoi@seoultech.ac.kr; Tel.: +82-2-970-6412

Received: 9 August 2019; Accepted: 11 September 2019; Published: 12 September 2019

Abstract: In this paper, we present the detailed design and implementation procedures for an advanced driver assistance system (ADAS) based on an open source automotive open system architecture (AUTOSAR). Due to the increasing software complexity of ADAS, portability, component interoperability, and maintenance are becoming essential development factors. AUTOSAR satisfies these demands by defining system architecture standards. Although commercial distributions of AUTOSAR are well established, accessibility is a huge concern as they come with very expensive licensing fees. Open source AUTOSAR addresses this issue and can also minimize the overall cost of development. However, the development procedure has not been well established and could be difficult for engineers. The contribution of this paper is divided into two main parts: First, we provide the complete details on developing a collision warning system using an open source AUTOSAR. This includes the simplified basic concepts of AUTOSAR, which we have organized for easier understanding. Also, we present an improvement of the existing AUTOSAR development methodology focusing on defining the underlying tools at each development stage with clarity. Second, as the performance of open source software has not been proven and requires benchmarking to ensure the stability of the system, we propose various evaluation methods measuring the real-time performance of tasks and the overall latency of the communication stack. These performance metrics are relevant to confirm whether the entire system has deterministic behavior and responsiveness. The evaluation results can help developers to improve the overall safety of the vehicular system. Experiments are conducted on an AUTOSAR evaluation kit integrated with our self-developed collision warning system. The procedures and evaluation methods are applicable not only on detecting obstacles but other variants of ADAS and are very useful in integrating open source AUTOSAR to various vehicular applications.

Keywords: AUTOSAR; ADAS; open source; design methodology; performance evaluation

1. Introduction

Recently, as traffic volumes have become increasingly complex, the need for an advanced driver assistance system (ADAS) has emerged to reduce life-threatening situations caused by traffic accidents. ADAS uses sensors and electronics to help drivers make better decisions [1]. With the remarkable advances of sensors and electronics in recent years, the expectations for ADAS have significantly increased. In such, various studies have been conducted related to tracking a vehicle using a camera [2], measurement of the driver's heart rate using sensors mounted on the seat of the vehicle [3], notification of the recommended vehicle speed based on weather, road, and vehicle condition [4], and communication between autonomous vehicles [5]. With the advancements in

ADAS, the underlying software has become more complex; where interoperability, portability, and maintainability became crucial requirements. These requirements are addressed by automotive software platforms such as Open Systems and their Interfaces for the Electronics in Motor Vehicles (OSEK) [6] and AUTOSAR [7–9].

Traditional original equipment manufacturers (OEMs) and electronic control unit (ECU) manufacturers use a variety of hardware and software without a uniform standard to develop the ECU of vehicles. Without a uniform standard, ECUs are developed in a hardware-dependent structure that requires new software to be created when the hardware is changed. In this way, the developed ECU reduces the reusability and reliability of software and increases development costs. To address such problem, German automotive companies jointly started OSEK with the aim to develop "an industry standard for an open-ended architecture for distributed electronic control units in automobiles." It provides standard interfaces incorporating hardware and software for easier development and reusability. However, the constant increase in system complexity of automobiles leads to the current development of automotive open system architecture (AUTOSAR). AUTOSAR aims to develop the software independent of the hardware and can be distributed or reused in different components of ECUs [10,11]. AUTOSAR separates hardware-independent application software from the hardware-oriented software through the runtime environment layer. In this study, we focus on the development of a collision warning system based on AUTOSAR.

Solutions for AUTOSAR development are divided into commercial [12] and open source distributions [13]. In a commercial AUTOSAR solution, the vendor provides a well-established development methodology from the development environment to the implementation in accordance with the development tool. Thus, it is relatively easier to use in project development but with higher licensing costs. Conversely, open source solutions can decrease development costs as the licenses are free and can easily be acquired. However, documentation and implementation procedures are not well-defined. For example, Sun et al. [14] built a software framework based on controller area network (CAN) bus following the AUTOSAR acceptance test. Similarly, Jansson et al. [15], implemented FlexRay communication using the AUTOSAR communication stack. Both researchers offered performance evaluation of the communication stack and the required implementation mechanisms. However, these studies did not offer actual automotive application. A vehicle seat heating system based on the open source AUTOSAR is presented by Melin et al. [13] and the application of the message authenticated controller area network (MaCAN) protocol for vehicular applications [16]Kulaty et al. is proposed. However, all of these studies focus only on the implementation of the communication stack or the specific application. Design and implementation procedures were not considered. For this reason, engineers and practitioners mentioned complexity, learning curve, and bad documentation as the recognizable drawbacks of AUTOSAR [10].

To address these shortcomings, this paper aims to provide a comprehensive reference for practitioners, especially beginners, on the design and implementation of open source AUTOSAR for automotive applications. To begin, we organized the basic concepts of AUTOSAR to make it easier to understand the architecture and each fundamental component. Although the AUTOSAR manual [17] describes most of the parts, this study offers a simplified explanation based on the first-hand experience which is very helpful especially in practical implementation. Also, we propose an improvement of the existing development methodology [18–21] based on an open source AUTOSAR. This focuses on defining the underlying tools required at each development stage which is very necessary for designing flexible and reusable software modules for a complex system such as an ADAS.

In order to validate the methodology, we developed a collision warning system with actual ultrasonic sensors and light-emitting diodes (LEDs) connected to an AUTOSAR evaluation kit, NXP MPC574XG-324DS [22], including the complete details and implementation procedures. Visualization of the sensor data was performed in a virtual robot experience platform (V-REP) [23,24]. Because an ADAS such as the collision warning system is composed of complex software and electronic hardware devices, precise control period and minimal computational delay should be ensured for proper device handling. Thus, the entire system should display deterministic behavior

and responsiveness, i.e. real-time performance. Due to the lack of documentation and established methods in determining the performance of open source AUTSOAR, this study also provides various evaluation methods aiming to measure the real-time performance of the tasks running in the runtime environment of AUTOSAR. These include the task periodicity test to ensure that the system meets real-time constraints and latency tests on each layer of the AUTOSAR communication stack. The results of these evaluation methods can help developers in improving the overall safety of their developed vehicular system. Although we have implemented the proposed procedures and evaluation methods in a collision warning system, these are applicable to any other types of ADAS applications and is very useful as a guideline for integration open source AUTOSAR to more complex vehicular projects.

In summary, the contribution of this study is twofold: First, we provide the complete details on developing a collision warning system using an open source AUTOSAR; including the simplified basic concepts of AUTOSAR and improvement of the existing AUTOSAR development methodology. Second, we propose various evaluation methods measuring the real-time performance of the tasks in the runtime environment and the overall latency of the communication stack. This paper is organized as follows: The second section presents the simplified basic concepts of AUTOSAR. The proposed ADAS development process using an open source AUTOSAR is described in Section 3. Experiment procedures and results are shown in Section 4. Evaluation of the implemented system using the proposed evaluation methods are performed in Section 5 and the last section summarizes the concluding remarks.

2. AUTOSAR Basic Concept

AUTOSAR is a standard software consisting of three major layers: the application software component (ASW), runtime environment (RTE), and basic software (BSW). Each layer is modularized into various software components, connected with each other over a virtual network called the virtual functional bus (VFB) [25]. Figure 1 shows the basic software architecture of AUTOSAR [26].

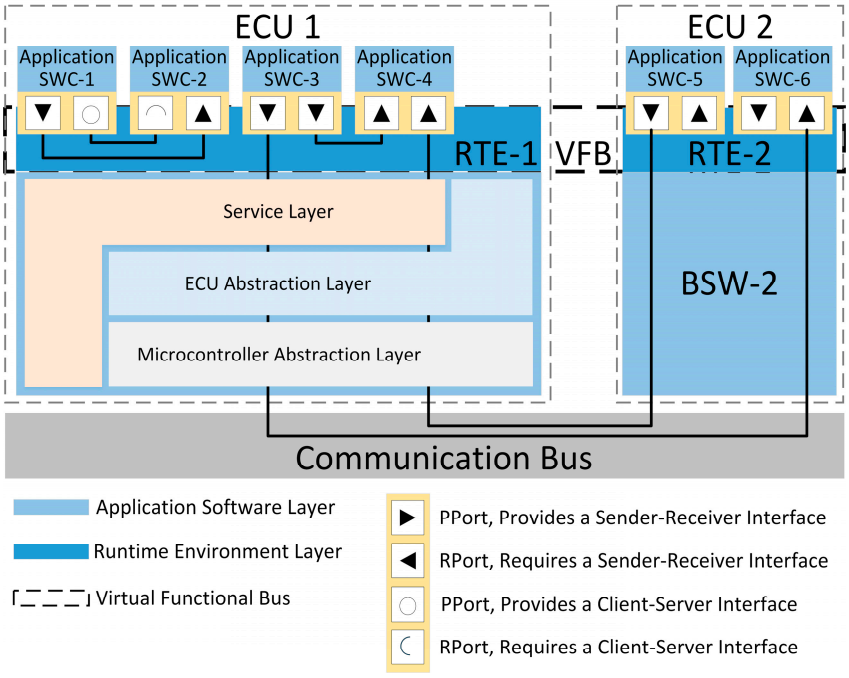

Figure 1. Simplified AUTOSAR software architecture.

The ASW layer consists of the AUTOSAR software component (SWC) mapped into a specific ECU. The SWC is separated into minimum units that can be reused based on the function of the AUTOSAR application, in terms of information hiding and encapsulation and is independent of the

hardware and network bus. Communication between different SWCs or between an SWC and BSW is performed through a VFB. It defines an interface for an access point that can be input and output data to the SWC, and a communication method to communicate with other SWCs or BSWs. The SWC can send and receive necessary data through the port and the interface regardless of the ECU in which the communication object is located. The communication interface includes a sender-receiver interface and a client-server Interface. In the case of the sender-receiver interface, data is transmitted from the sender to the receiver by the signal passing method. The data type of the transmitted data from the sender must match the data type specified by the receiver. In the case of the client-server interface, the function of the server is called by the client in the function call method. The data type of the parameter to be used when calling the server function in the client should match the data type specified in the server.

The RTE layer serves as a middleware for managing communication between the ASW layer and the BSW layer of the same ECU. The RTE provides the same abstracted interface regardless of whether the communication is within the ECU or through the external communication network, so the SWCs of the ASW layer are independent of those of the BSW layer. The VFB of the RTE layer provides the AUTOSAR communication mechanism for the client-server and sender-receiver interfaces and provides communication service to the SWC. The VFB is a technical concept that enables the development of the functional architecture of the entire system, independent of the actual hardware topology of the ECUs and the network [25]. In Figure 1, SWC-1 and SWC-2 communicate within ECU 1, but in the case of SWC-3 to SWC-6, communication is performed between the SWCs of ECU 1 and ECU 2. Since all SWCs are designed to communicate in one VFB, they are designed the same as SWC-1 and SWC-2 when developing SWC-3 to SWC-6, regardless of the position of the mapped ECU. After the SWCs are mapped to the ECU, the VFB is implemented as RTE-1 and RTE-2 in each ECU as shown in Figure 1. As a result, RTE-1 and RTE-2 play an individual role in VFB.

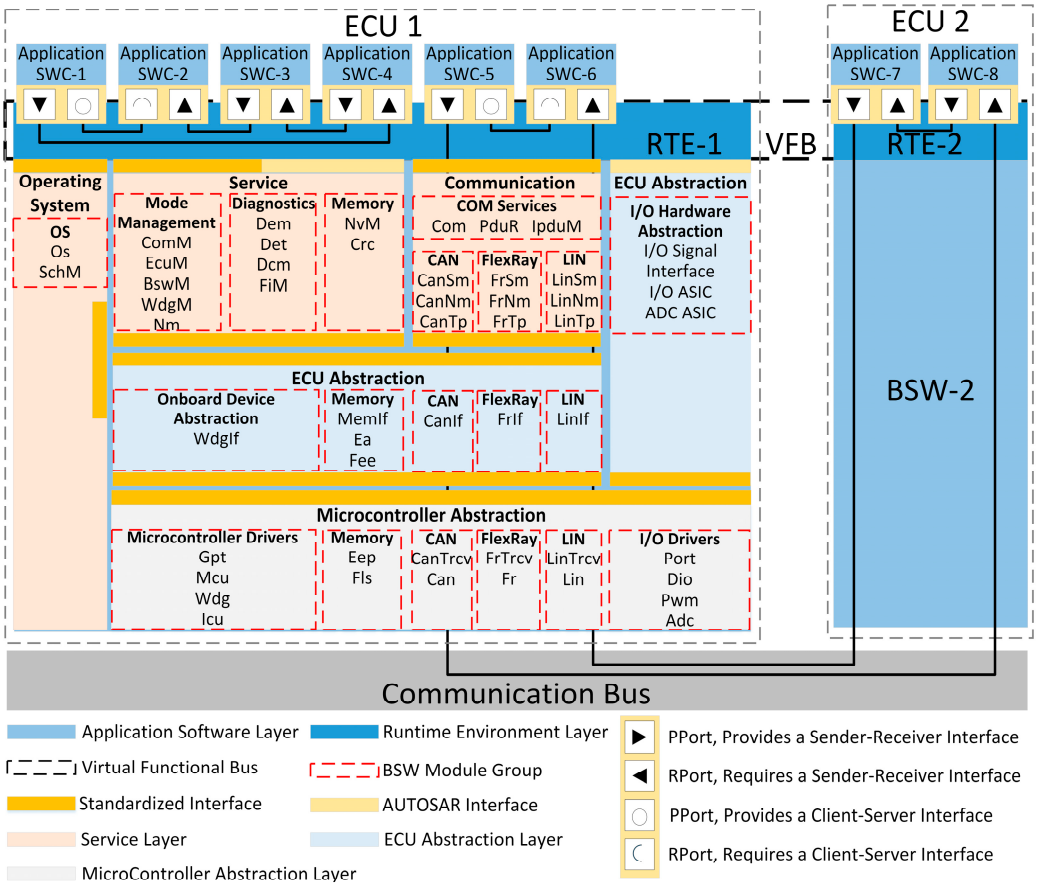

Figure 2. Detailed AUTOSAR software architecture.

Figure 2 shows the detailed software architecture of AUTOSAR, which was unified based on our own first-hand experience and interpretation of the AUTOSAR manual [26]. BSW, which was simply presented in Figure 1, is detailed in Figure 2.

The BSW is also a standard software layer that provides the services required for the ASW layer and the SWCs to perform specified tasks. The BSW consists of a service layer, an ECU abstraction layer, and a microcontroller abstraction layer (MCAL). The service layer is divided into system services, memory services, and communication services depending on the functions to be provided. System services are a group of modules and functions that can be used in all layers of the module. System services provide a real-time operating system, vehicle network communication and management, the ECU status management function, WatchDog, and the diagnostic service function. In Figure 2, the OS module group, mode management module group, and diagnostics module group correspond to system services. Memory services consist of a group of modules responsible for nonvolatile data management. The standard AUTOSAR interface provides nonvolatile data to the ASW layer; memory location and attribute abstraction; a nonvolatile data management mechanism for storage, load, checksum protection, and verification; and stable storage. In Figure 2, the memory module group is provided by the memory service.

Communication services are a group of modules that perform functions such as CAN, local interconnect network (LIN), and FlexRay to provide communication to higher layers through a unified interface. Communication services provide a service for communication network management and a unified interface that eliminates the dependency of the lower layers. Additionally, it accesses the communication driver through the abstracted communication driver and has a structure that is independent of the communication driver of the MCAL layer. Communication services provide a unified interface that eliminates the dependency of the lower layers on diverse applications and vehicle network diagnostic communications, allowing applications to be developed without consideration of protocol and message attributes. Internally, there is a network manager (NM), state manager (SM), and transport protocol (TP) for communication networks, such as CAN, LIN, and FlexRay. Furthermore, modules such as communication (Com) and protocol data unit router (PduR) exist. In Figure 2, the service module groups CAN state manager (CanSm), CAN network manager (CanNm), CAN transport protocol (CanTp), FlexRay state manager (FrSm), FlexRay network manager (FrNm), FlexRay transport protocol (FrTp), LIN state manager (LinSm), and LIN transport protocol (LinTp) are provided in the communication services [27,28].

The ECU abstraction layer serves as an interface and driver for creating an upper layer of software, independent of hardware so that MCAL can be used. The ECU abstraction layer is independent of the microcontroller, but it is dependent on the ECU board used. The ECU abstraction layer is divided into onboard device abstraction, memory hardware abstraction, communication hardware abstraction, and input/output (I/O) hardware abstraction depending on the functions to be provided. Onboard device abstraction contains drivers for the ECU onboard devices not visible as sensors or actuators, such as internal or external watchdogs. The function of this module group is to abstract from ECU specific onboard devices. In Figure 2, the watchdog interface (WdgIf) module is included in the onboard device abstract. Memory hardware abstraction is a group of modules that abstracts internal or external memory devices. It provides a mechanism for accessing internal or external memory devices, allowing access through the same interface regardless of the type of memory, such as electrically erasable programmable read-only memory (EEPROM) or Flash. In Figure 2, the memory abstraction interface (MemIf), electrically erasable programmable read-only memory abstraction (Ea), and flash electrically erasable programmable read-only memory emulation (Fee) modules are included in the onboard device abstraction. Communication hardware abstraction is a group of modules that abstract communication hardware. To use a specific communication protocol, you must implement the communication hardware abstraction module for that protocol. In Figure 2, the CanIf, FrIf, and LinIf modules are included in the communication hardware abstraction. I/O hardware abstraction is a group of modules that abstract the I/O hardware. The main purpose of I/O hardware abstraction is to provide I/O access to the ASW layer and the SWCs. It can be accessed from the upper layer through the I/O signal interface without going through the service layer. In

Figure 2, the Port, Dio for digital input/output (DIO), Pwm for pulse width modulation (PWM), and Adc for analog to digital converter (ADC) modules are included in the I/O hardware abstraction.

The microcontroller abstraction layer (MCAL) is the lowest software layer of a BSW. In order to avoid direct access to the microcontroller register in the high layer software, access to the hardware is done through the MCAL device driver, which includes hardware-dependent drivers such as ADC, PWM, DIO, EEPROM, and Flash. Access to the microcontroller's registers is routed through MCAL, which makes the upper software layer independent of the microcontroller. Depending on the function, the microcontroller driver, memory driver, communication driver, and I/O driver are categorized. This provides an application programming interface (API) for devices and their connection to the microcontroller.

3. Collision Warning ADAS based on an Open Source AUTOSAR

In this section, we describe the design and implementation procedures of a simple collision warning system comprised of ultrasonic sensors and LEDs, with the aim to provide an in-depth reference for an open source AUTOSAR. We present the improved design methodology of developing AUTOSAR applications, which focuses on defining the underlying tools required at each development stage. We also describe the complete design and implementation procedures of a collision warning system including the step-by-step process from configuring both the SWC and BSW. This includes RTE and SWC runnable implementations for developing ADAS applications.

3.1. Design Methodology

The existing AUTOSAR development methodology [18–21] has difficulty in clarifying the development process because determining the underlying tools at each development stage is not clearly defined. For commercial AUTOSAR, it is not a problem because the vendor providing the solution provides detailed manuals, but because the open source AUTOSAR does not have a clear methodology for development, there are many challenges for the developers. In this paper, a specialized procedure is proposed for development using open source AUTOSAR. The existing AUTOSAR development process has been reconfigured in five stages. The proposed procedure using ARCCORE's open source AUTOSAR development solution is shown in Figure 3.

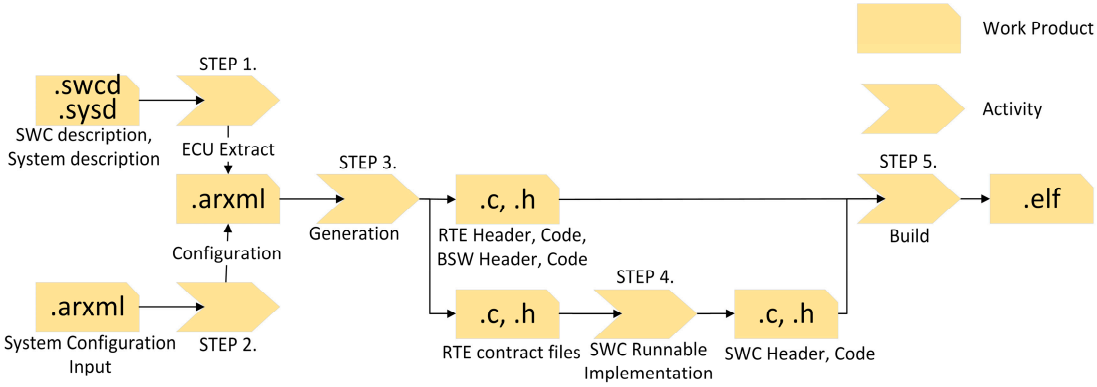

Figure 3. Open source AUTOSAR procedure.

Step 1: This step is performed in the SWCD Editor of Arctic Studio. The SWC description and system description should be defined for the applications in each software component of .arxml files. We classify the SWCs according to the function of the AUTOSAR software to be implemented and define the communication interface, port, and data type in each SWC. After defining the SWCs, the defined SWCs are used to create prototypes that are objects of the SWC. Then, the communication ports of the prototypes are connected for communication and the signals to the external ports for outside the ECU are mapped.

Step 2: This step is performed in the BSW Editor of Arctic Studio. The BSW module for providing the service required by the SWC is defined and configured in an export.arxml. Modules that should be configured by default at this stage are OS modules for task scheduling, EcuM modules for ECU management and BswM modules for BSW management. Other modules are added to perform their functions according to the functions required by the AUTOSAR SWC. If GPIO is to be used, add the I/O modules. To implement communication between other ECU, add the communication modules. You can configure the detailed features of each module for all added modules. For OS modules, task creation, Priority and Period can be configured. The addition and configuration of modules performed at this stage are all made in the GUI environment. Finally, the ECU extract method is conducted to create an extract.arxml file.

Step 3: This step is the RTE configuration step performed in the RTE Editor and BSW Editor of Arctic Studio. RTE configuration is based on the extract.arxml files created in Step 1 and 2. As a result of this step, source code and header file for the RTE and BSW are generated. The RTE configuration process instantiates the SWC prototype created in Step 1 and maps the tasks and events created in the OS module configuration of Step 2 to the runnable of the instantiated SWC. Through this process, the runnable of the SWC is scheduled and executed in the AUTOSAR OS. After the RTE configuration is completed, RTE and RTE contract files are generated to connect the BSW and the ASW using the generate function provided by the RTE editor. In the RTE contract file, an API for calling the service provided by the BSW layer in the application of the ASW layer is defined. Finally, generate the BSW code using the generate function provided by BSW editor.

Step 4: The runnable, which is a function that defines the SWC, is created in C language. The runnable refers to the RTE contract file which defines the API for communication through the RTE which is generated as a result of the execution of Step 3.

Step 5: The source code and header files generated as a result of Steps 3 and 4 are built and used to generate executable files that will run on the ECU. The tool chain and environment variables of the MCU are specified and used in the ECU. These are performed in Arctic Studio.

3.2. Designing a Collision Warning System

A collision warning system is a system that allows the driver to recognize and avoid an obstacle by generating a warning to the driver according to the distance between the obstacle and vehicle when the obstacle exists in a blind spot that the driver cannot recognize. In this paper, the distance from an ultrasonic sensor to the obstacle is measured and transmitted to the ECU. The ECU determines the warning level according to the distance data. As a demonstration system, V-REP is used to show the visualization of sensor data, which is transferred through the CAN bus. The control signal is also displayed with an LED indicator. The following sections provide a detailed design procedure of a collision warning system described in the previous section.

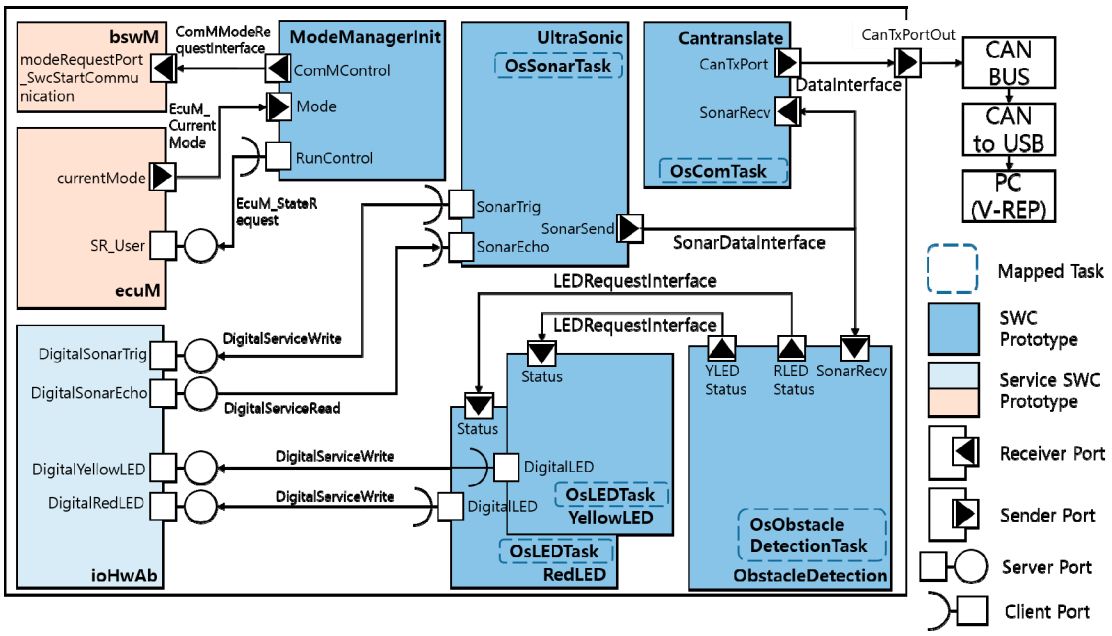

Figure 4. Collision warning application layer overview.

3.2.1. SWC Configuration

The Communication interface at SWC is classified into standardized interface and AUTOSAR interface according to the communication target module as shown in Figure 4. The standardized interface is used with the BSW, and the AUTOSAR Interface is used with the SWC. Standardized interface is provided by the AUTOSAR framework and does not need to be defined. AUTOSAR Interface design is classified according to the size and type of data. Three interfaces are defined in this paper: The sonar data interface (SonarDataInterface) is used to transmit the measured ultrasonic sensor data, the data interface (DataInterface) is used for CAN data transmission, and the LED request interface (LedRequestInterface) is used for LED control request. When communicating using the SonarDataInterface and DataInterface, data represented distances and four bytes of an unsigned int data type are used. When communicating using LedRequestInterface, the data only needed to indicate whether to turn on/off the LED for control, so a one byte Boolean data type is used. In Figure 4, those are used to connect the communication ports of SWC Prototypes.

The SWCs that constitute the communication port and internal behavior are designed by classifying the operations performed in the ECU into five functions: the distance measurements to the obstacle, data transmission with the CAN bus, warning level judgment, LED control for the warning, and ECU initialization. Additionally, the runnable (executable file) is configured for each SWC. Designed SWCs provide a structure for instantiating SWC prototypes. The application layer with the detailed operation and connection of each SWC prototype is shown in Figure 4. In the developed collision warning system, the SWCs are divided as the UltraSonic, CanTranslate, ObstacleDetection, LEDActuator, and. The UltraSonic converts the measured data from the ultrasonic sensor into distance data in meters. The calculated distance data is then transmitted to the CanTranslate and ObstacleDetection for further processing. After receiving the distance data, the CanTranslate relays them to the CAN bus through the CAN communication stack. On the other hand, the ObstacleDetection determines the color of the LED to be controlled according to the same distance data received from the UltraSonic and signals the LEDActuator. The LEDActuator performs the actual control of the connected LEDs, whether YellowLED or RedLED. The ModeManagerInit transmits the ECU control signals to the ecuM and bswM, and lets the BSW perform Ecu, Gpt (general purpose timer), and communication initialization functions. Through these operations .swcd and .sysd files are finally created for SWC description and an .arxml file is also generated through ECU Export.

3.2.2. BSW Configuration

In order to provide the services that the SWC needs to perform specific functions, the modules utilized in the BSW are defined and configured. In such, the UltraSonic SWC requires timer services for measuring the return time of the Digital I/O and ultrasonic waves. CanTranslate needs the CAN communication services. As the ObstacleDetection SWC does not request any hardware resources, a BSW service is not necessary. On the other hand, hardware-related SWCs such as the YellowLED and RedLED require the Digital I/O service. The service required by ModeManagerInit is an ECU management function to perform the initialization of the ECU, timers, and communication. In addition, OS service is required for task creation and scheduling. The Com, PduR, CanIf, and Can modules are used to implement the CAN communication stack [14,28]. The CanSM module is used to implement the control flow of the CAN BUS. The Dio, Port, and IoHwAb modules for Digital I/O control are used; the Gpt module for timer control is used; the EcuM, BswM, Mcu, and EcuC modules are used for system management; and the Os module is used to manage the creation and scheduling of AUTOSAR tasks. Figure 5 shows the layered structure of the collision warning system after performing SWC and BSW configurations.

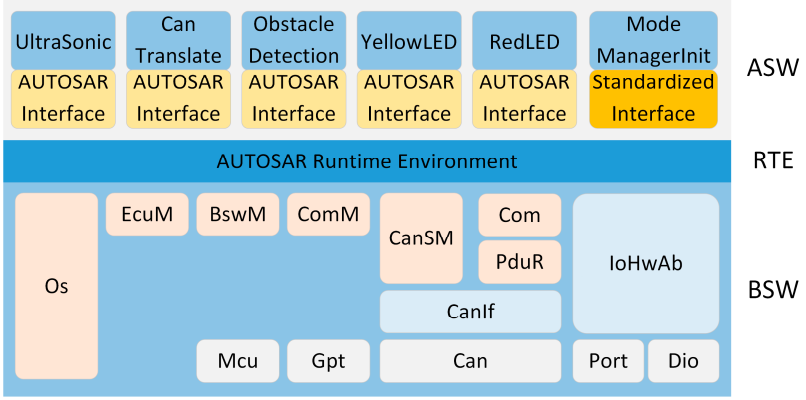

Figure 5. Layered collision warning system architecture.

3.2.3. BSW and RTE Generation

The next procedure is to generate the RTE and BSW codes. RTE is configured by the .arxml file from the SWC and BSW configuration mentioned in the previous subsections. The RTE configuration process instantiates the SWC prototype configured during the SWC configuration process, such that the RTE can recognize the SWC as part of the ASW layer. After instantiating the SWC prototype, the runnable is mapped to the Task and Event configured in the Os module in the BSW configuration. Through this process, the SWC runnable is mapped to a task managed by the OS scheduler and is scheduled in a task unit.

The tasks mapped to SWCs in the collision warning system are displayed as a dotted line in Figure 4. Those are OsObstacleDetectionTask, OsLEDTask, and OsSonarTask, OsComTask. The OsStartupTask initializes the BSW module by calling EcuM_StartupTwo as the first task to be executed. This task is not mapped and runs only once when the OS is executed. OsBswServiceTask, which is also not mapped, calls the MainFunction of the ComM, Com, EcuM, CanSM, Can, and BswM modules, which are BSW modules that should be called periodically to provide services. This task is also not mapped and is scheduled by the BSW Scheduler to run in polling method every 10 ms. The OsObstacleDetectionTask is triggered every 20 ms and the runnable in the ObstacleDetection SWC Prototype is mapped. The OsLEDTask is triggered every 10 ms and the runnables in the YellowLED and RedLED SWC Prototypes are mapped. The OsSonarTask is triggered every 40 ms and the runnable in the Ultrasonic SWC Prototype are mapped. The OsComTask is triggered at data is received at SonarRecv port, and the runnable in the CanTranslate SWC Prototype are mapped.

3.2.4. SWC Runnable Implementation

The algorithm of the SWCs are designed in the SWC configuration process; a .c file is implemented for each SWC and the runnable is implemented as a function inside the .c file. This implemented the communication between an SWC and other SWCs or communication between the SWC and BSW, by referring to the API defined in the RTE contract file generated as a result of RTE generation.

4. Experiment Results

Experiments were conducted to verify the operation of the developed collision warning system based on the proposed methodology in the previous section. The collision warning system operates by acquiring the distance from ultrasonic sensors, and depending on the measured distance of the obstacle, the state of the corresponding LED changes. In this section, we specify the hardware and software environment of the experimental testbed, verified the operation of the developed system on top of a mobile platform, and visualized the sensor data using the visualization software, V-REP.

4.1. Hardware Environment

In our developed system, we used the MPC574XG-324DS board with MPC5748G MCU as the ECU. The USB Multilink Universal debug probe is used for debugging and software upload. The USB to CAN converter is used for communication between a PC and the ECU through CAN protocol. The data transferred to the PC are visualized using the tool, V-REP [23,24]. Ultrasonic sensors are used to measure the distances from obstacles in the collision warning system. Red and yellow LEDs inform the user about the risk of the obstacles. An oscilloscope is used to identify the CAN message frames. The complete hardware environment is shown in Figure 6.

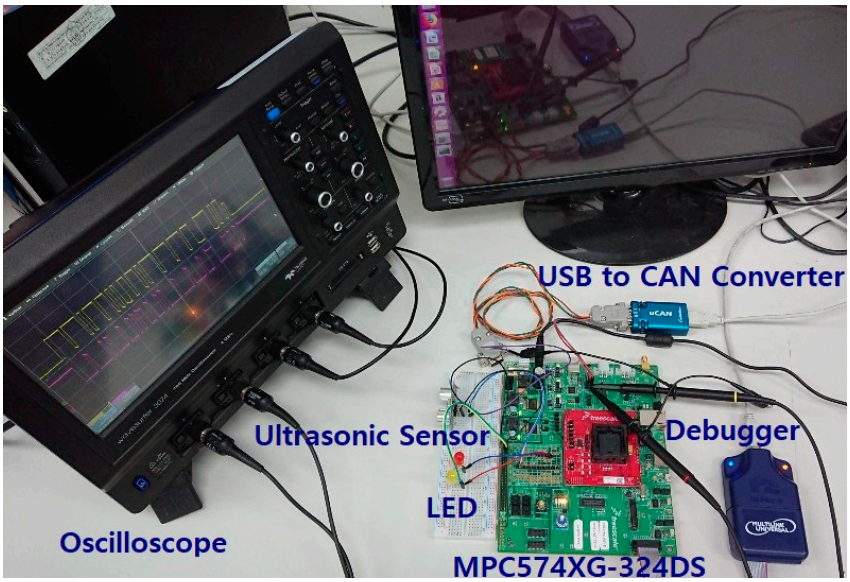

Figure 6. Hardware environment.

4.2. Software Environment

In this study, the open-source AUTOSAR design is based on the Arctic Core 21.0.0 from ARCCORE along with the integrated development environment (IDE), Arctic Studio 21.0.0, running under Eclipse. To develop AUTOSAR software for the MPC5748G MCU, a toolchain of the PowerPC architecture supported by the IDE is required. Currently, supported toolchains are CodeWarrior, Diab, and Greenhills. However, CodeWarrior (the toolchain for NXP boards) does not support the MPC57xx series of MCUs. Therefore, we used the toolchain provided by the "S32 Design Studio for Power Architecture" IDE provided by NXP for the PowerPC architecture.

After installing the toolchain, the environment variables to build the AUTOSAR project is configured in Arctic Studio. These include BOARDDIR, COMPILER, and CROSS_COMPILE. BOARDDIR, for example. COMPILER is set to "gcc" to enable the usage of the gcc cross compiler and CROSS_COMPILE defines the directory where the toolchain is installed. In order to generate the code for the BSW that can be operated on the target board, the MCAL item, found in the ECU information and options of the BSW Editor of Arctic Studio, were set to MPC5748G, the MCU of the target board. Finally, a robot simulator, V-REP PRO EDU, is installed to visualize the behavior of the ECU on the PC.

4.3. Collision Warning System Operation Test

In order to verify the operation of the collision warning system in a real environment, the change of the LED was observed while moving the distance of an obstacle from 0.5 to 2.0 m. The red LED turns on when the distance from the obstacle is less than 1.0 m, the yellow LED lights up when it is between 1.0 m and less than 2.0 m, and both LEDs remain off if the obstacle is more than 2.0 m away. The test results are shown in Figure 7, showing the distance of the obstacle from the collision warning system in 0.5 m increments. The size of one block of the tile is 0.5 m in width and 0.5 m in height. Through these results, we have verified the normal operation of the collision warning system.

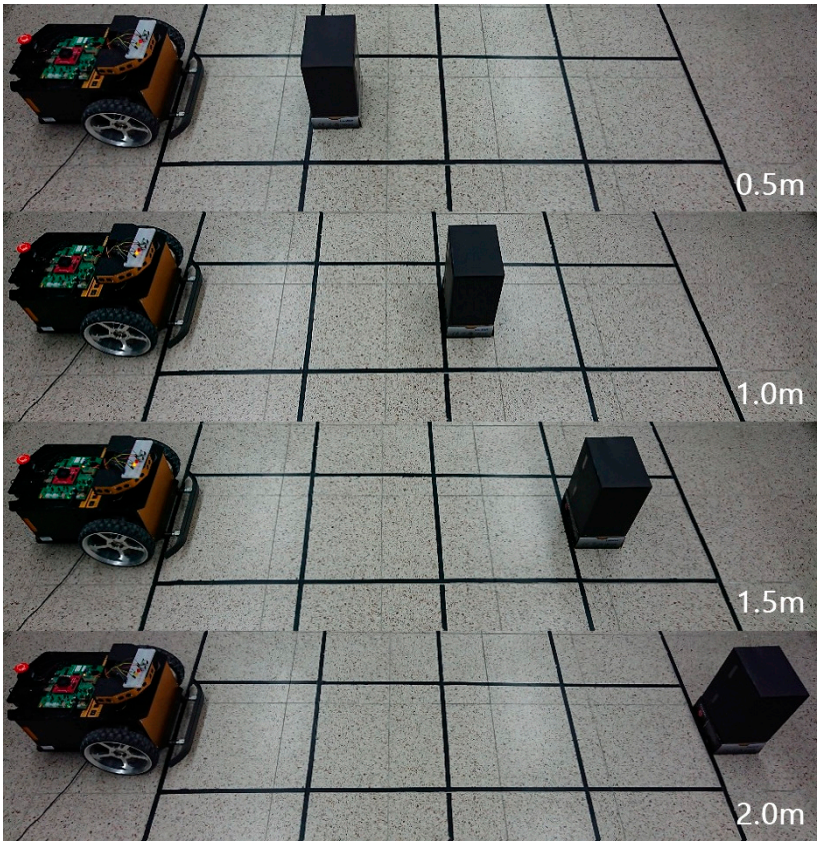

Figure 7. State when detecting the distance.

4.4. Sensor Data Visualization Using V-REP

In this section, V-REP visualization is performed to visually confirm the operation of the collision warning system and the data of the ultrasonic sensor as described in the previous subsection. Figure 8 shows the result of the visualization of ultrasonic sensor data sent from ECU to CAN bus using V-REP of host PC using USB to CAN Converter. To maintain consistency with the actual experiment environment, the size of one block of tile was set to 0.5 m for both width and height. The ultrasonic sensor was attached to the front part of the vehicle, and the position of the obstacle is indicated by a

blue disk. The LEDs to indicate the risk of the obstacles were composed of the same red and yellow as the actual hardware configuration. The UI on the left shows the data of the ultrasonic sensor, and the view from the side, and the view from above. We confirmed that the movement of the blue disc and the state of the LED change according to the distance between the ECU and the obstacle. The results show the same behavior with the actual experiment in the previous section.

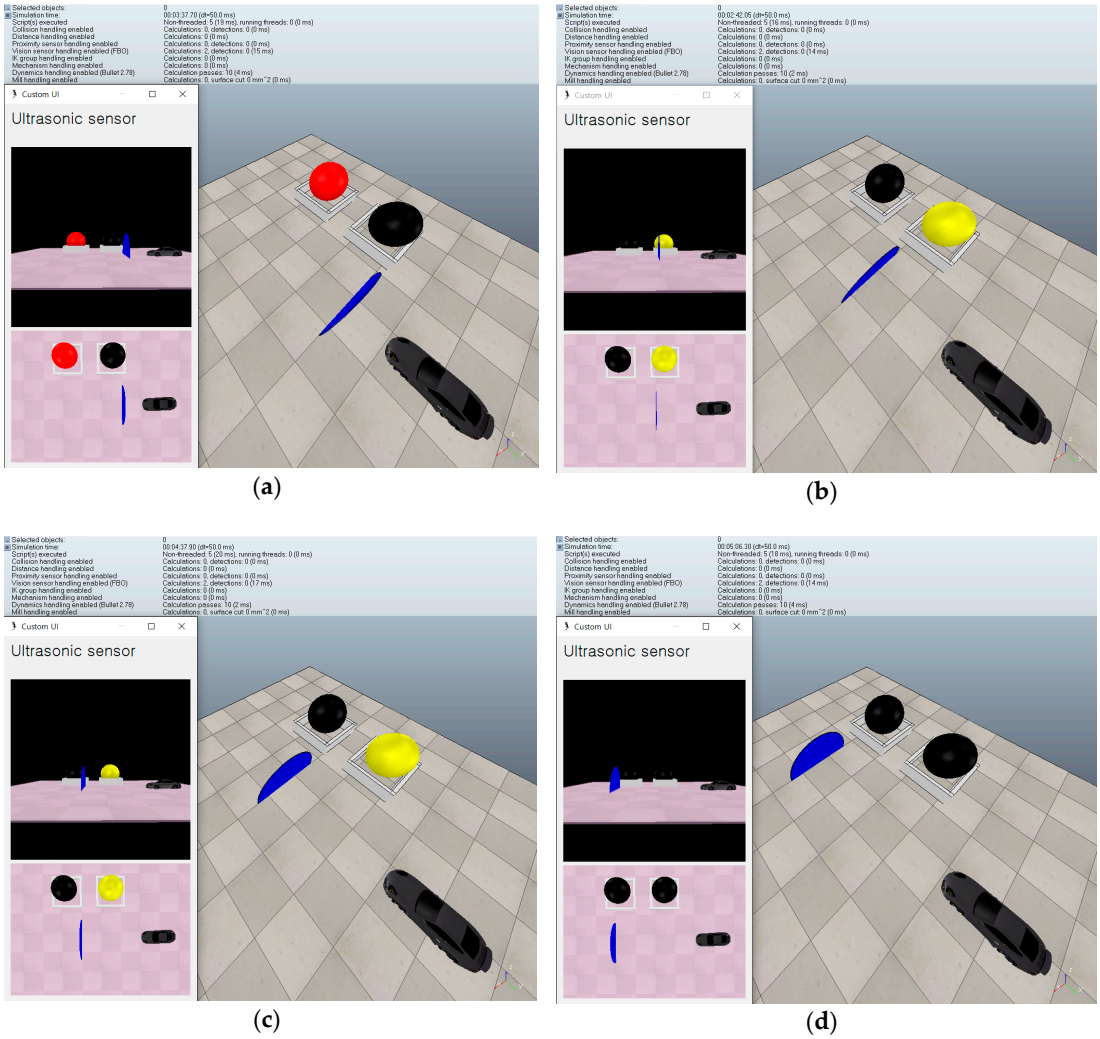

Figure 8. State when the detection distance is: (**a**) 0.5 m; (**b**) 1.0 m; (**c**) 1.5 m; (**d**) 2.0 m.

5. Performance Evaluation

As the performance of open source AUTOSAR is not proven and there is a lack of existing methods benchmarking open source distribution of AUTOSAR, this section presents the metrics for evaluating the real-time performance and latency of the communication stack of AUTOSAR focusing on the developed collision warning system in the previous section. Real-time performance tests of the Os tasks were performed to verify that the real-time constraints were satisfied; specifically, the periodicity of the tasks was measured. As the communication stack is implemented in a hierarchical manner, this can contribute to the overall latency of the task and could affect the real-time performance. Thus, we have also evaluated the latency at each layer and calculated the overall latency from the SWC to the CAN bus.

5.1. Real-Time Performance

This section presents the real-time performance evaluation [29,30] of the collision warning system. AUTOSAR OS is based on the Open Systems and their Interfaces for the Electronics in Motor Vehicles (OSEK) OS, which is an OSEK/vehicle distributed executive (VDX) compliant real-time operating system and supports a priority-based preemption real-time scheduling function [31]. In order to verify that the AUTOSAR OS satisfied the real-time constraints, the periodicity of the tasks constructed in the OS module of the BSW is evaluated. This is used to confirm whether the system displayed deterministic behavior and responsiveness, which are critical to the stability of the system [29]. The tasks and events configured in the OS module were mapped to the runnables of the SWC in the RTE configuration step. Then the RTE source code was generated. Among them, the source code, where the task, event, and the runnables were mapped, were implemented in the Rte.c file. The tasks implemented in the Rte.c file are executed repeatedly, waiting for the event mapped to the task. When a mapped event occurred, the task is executed again. Using this characteristic, the real-time property of the AUTOSAR OS was checked by measuring the time from immediately after the event occurred to when the event occurred again. T_{period} represents an actual time that it takes for one cycle of the task. E_{now} is the time at which the event mapped to the current task occurred, and E_{prev} is the time at which the event occurred before. Its relationship with the period is defined by the following equation:

$$T_{period} = |E_{now} - E_{prev}| \quad (1)$$

Experiments were performed for 30 min and the real-time performance is evaluated for the OsLEDTask task (τ_1), OsObstacleDetectionTask (τ_2), and OsSonarTask (τ_3), which were the events and runnable mapped tasks in the OS module configuration. τ_1 had the highest priority and is executed in a 10 ms T_{cycle}. τ_2 had a medium priority and is executed in a $T_{cycle}$of 20 ms. τ_3 had the lowest priority and is executed at a $T_{cycle}$of 40 ms. T_{cycle} is the expected cycle of the real-time task. The results are summarized in Table 1 with the statistical average (avg), maximum (max), minimum (min), and standard deviation (σ) values for each timing metric. T_{period} represents the actual time that it takes for one cycle of the task. Its relationship with the jitter is defined by the following equation [32,33]:

$$T_{jitter} = |T_{cycle} - T_{period}| \quad (2)$$

The results show that the task with the highest priority had the best performance with the lowest deviation to the statistical average of period and jitter. Therefore, it is confirmed that the AUTOSAR OS performed scheduling according to priority and satisfied periodicity.

Table 1. Statistical analysis of the periodicity of tasks used in the collision warning system.

Task	τ_1 **High Priority**		τ_2 **Middle Priority**		τ_3 **Low Priority**	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	9.984680	0.022939	20.007748	0.023442	40.000080	0.026439
max.	10.077800	0.106275	20.199025	0.205300	40.502225	0.502225
min.	9.893725	0.001025	19.794700	0.002375	39.596200	0.000000
st.d (σ)	0.024978	0.017580	0.030898	0.022102	0.043450	0.034481

5.2. AUTOSAR Communication Stack

In order to apply the ECU that implements the ADAS to a real vehicle, it must be able to communicate with the various ECUs installed in the vehicle. AUTOSAR provides a communication stack for communication between these ECUs. The communication stack is a hierarchical structure of communication-related modules existing in the service layer, the ECU abstraction layer, and MCAL of the BSW lay. The Com and PduR modules are used in the service layer. The Com module controls communication transmission and converts the signal used in RTE to interaction layer protocol data unit (I-PDU), which is used in the communication stack. The PduR module routes the I-PDU received

from the Com module to the interface (If) module of the ECU abstraction layer according to the specified communication method.

In the ECU abstraction layer, an If module is used according to the communication method. The interface module provides the interface between the service layer PduR module and the MCAL communication driver module and initializes the driver module. MCAL's communication driver module controls the ECU's communication controller. The signal transmitted from the SWC to communicate with the SWC of another ECU passes through each layer due to the hierarchical structure, and this creates latency. This can affect the real-time behavior of AUTOSAR. Therefore, the latency consumed in each layer in the AUTOSAR communication stack is measured to take this into account. The layers of the AUTOSAR communication stack use the API that calls the next layer to transmit data to the next layer and request data processing. The data processing time is measured by installing a timer provided by the Gpt module in the API that calls the next layer in each layer. The communication stack source code is automatically generated by the AUTOSAR development tool based on the BSW configuration, so the timer is installed after the BSW configuration is finished. Figure 9a is a pseudo code that calls the COM module to use the communication stack in the RTE. When the Com_MainFunctionTx API is called, the data to be transmitted is sequentially called from the COM module to the driver module, which is the lowermost module of the communication stack. Figure 9b shows the process of calling the PduR_ComTransmit API to send data to the next layer, PduR module, after finishing data processing in COM module. The measured rte_latency in Figure 9a is the sum of the latencies at all layers of the communication stack, and the com_latency measured in Figure 9b is the sum of the latencies at all layers below the COM module. Therefore, the latency in the COM module is the value obtained by subtracting com_latency from rte_latency. The remaining layers are measured using the same method.

```
make gpt_timer0;
read end_time;
while(1)
    wait event;
    get event;
    call Com_MainFunctionTx;
    read start_time;
    rte_latency = start_time - end_time;
    end_time = start_time;
endwhile;
```

(**a**)

```
make gpt_timer1;
read end_time;
call PduR_ComTransmit;
read start_time;
com_latency = start_time - end_time;
latency = rte_latency - com_latency;
end_time = start_time
```

(**b**)

Figure 9. Pseudo code for measuring the latency of communication stack: (**a**) from Com module to Driver module; (**b**) from PduR module to Driver module.

We measured the latency of each layer in the CAN communication stack used in the collision warning system. The latency was measured by installing a timer in a routine displayed in Figure 10 that passes data from the layer constituting the communication stack to the next layer. The figure illustrates the latency of each layer of the CAN communication stack and the routines that pass data to the next layer. Analysis of the latency of each layer in the communication stack shows that the exchange of data between the SWC and the RTE has a low latency because the data are stored in a shared structure. The Com module converted the signal used in the RTE to that of the I-PDU used in the communication module of BSW. This task determined the number of bytes to be converted for endianness. Collision warning systems use 32-bit little endian CAN messages. This data conversion operation consumes the most time in the Com module and the longest latency was measured. The latency from the SWC to the CAN controller is 36 μs during the transmission and 34 μs during the reception.

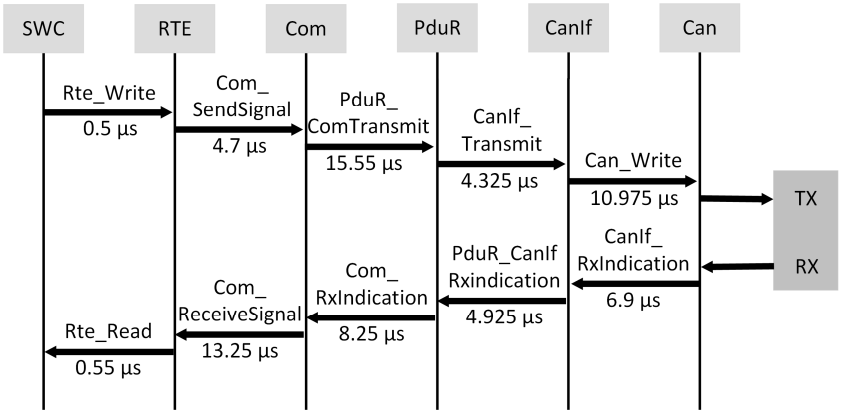

Figure 10. AUTOSAR controller area network (CAN) communication stack latency in each layer.

6. Conclusions

In this paper, a procedure was presented to develop an ADAS using open source AUTOSAR. It was addressed using open source AUTOSAR to ensure the interoperability, portability, and maintenance of complex ADAS software. To develop a system using AUTOSAR, a development solution must first be selected. The development procedure of commercial solutions is well-defined and easy to develop but it is difficult to acquire a license. Licenses can be easily acquired for an open source solution, but there is no established procedure for development, making it difficult for users to develop. Therefore, the detailed design procedure for ADAS based on open source AUTOSAR needs to be defined for most engineers.

Here, we have presented a development procedure that includes design, implementation, and evaluation using open source solutions. To verify the proposed procedure, we implemented a simple collision warning system for ADAS using ARCCORE's open source AUTOSAR solution Arctic tool and an MPC574XG-324DS board equipped with NXP's MPC5748G MCU. Real-time performance evaluation, visualization of the sensor data, and communication stack evaluation were performed to test the implemented systems. We confirmed that the implemented system satisfied the real-time constraints and verified the sensor data through visualization. This paper will be a promising result for engineers who want to develop a more complicated ADAS using open source AUTOSAR, from the development environment to implementation and evaluation.

We have provided the entire AUTOSAR and V-REP software for the collision warning system in this paper available at [34] enumerating the BSW modules and their usage.

Author Contributions: J.P. surveyed the background of this research, presented the procedure including the ADAS design, procedure, and evaluation methods using open source AUTOSAR, and performed the experiments to verify the proposed procedure. B.W.C. supervised and supported this study.

Acknowledgments: This work was supported by the Human Resources Development of the Korea Institute of Energy Technology Evaluation and Planning (KETEP) grant funded by the Ministry of Trade, Industry & Energy of the Korea government (No. 20174030201840).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

Abbreviation	Full form	Software Module
AUTOSAR	AUTomotive Open System ARchitecture	
ADAS	Advanced driver-assistance systems	
OSEK	Open Systems and their Interfaces for the Electronics in Motor Vehicles	
VDX	Vehicle Distributed eXecutive	
OEM	original equipment manufacturer	
ECU	Electronic Control Unit	
ASW	Application Software	
SWC	Software Component	
RTE	Run-Time Environment	
VFB	Virtual Functional Bus	
BSW	Basic Software	
MCAL	Microcontroller Abstraction Layer	
EEPROM	electrically erasable programmable read-only memory	
I-PDU	Interaction Layer Protocol Data Unit	
CAN	Controller Area Network	
LIN	Local Interconnected network	
Adc	Analog-Digital Converter	MCAL-I/O Drivers
Pwm	Pulse Width Modulation	MCAL-I/O Drivers
Dio	Digital Input / Output	MCAL-I/O Drivers
IoHwAb	I/O Hardware Abstraction	ECU Abstraction-I/O Hardware Abstraction
Gpt	General Purpose Timer	MCAL-Microcontroller Drivers
EcuM	ECU State Manager	Service-Mode Management
BswM	Basic Software Mode Manager	Service-Mode Management
Mcu	Microcontroller Unit	MCAL-Microcontroller Drivers
OS	Operating System	Operating System-OS
PduR	Protocol Data Unit Router	Communication-COM Services
Com	Communication	Communication-COM Services
CanIf	CAN Interface	ECU Abstraction-CAN
CanSm	CAN State Manager	Communication-CAN
CanNm	CAN Network Management	Communication-CAN
CanTp	CAN Transport Protocol	Communication-CAN
WdgIf	Watchdog Interface	ECU Abstraction-Onboard Device Abstraction
LinIf	LIN Interface	ECU Abstraction-LIN
LinSm	LIN State Manager	Communication-LIN
LinNm	LIN Network Management	Communication-LIN
LinTp	LIN Transport Protocol	Communication-LIN
FrIf	FlexRay Interface	ECU Abstraction-FlexRay
FrSm	FlexRay State Manager	Communication-FlexRay
FrNm	FlexRay Network Management	Communication-FlexRay
FrTp	FlexRay Transport Protocol	Communication-FlexRay
MemIf	Memory abstraction interface	ECU Abstraction-Memory
Ea	EEPROM Abstraction	ECU Abstraction-Memory
Fee	Flash EEPROM Emulation	ECU Abstraction-Memory

References

1. Lu, M.; Wevers, K.; van der Heijden, R.; Heijer, T. ADAS applications for improving traffic safety. In Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583), The Hague, The Netherlands, 10–13 October 2004; Volume 4, pp. 3995–4002.
2. Zou, Y.; Zhang, W.; Weng, W.; Meng, Z. Multi-Vehicle Tracking via Real-Time Detection Probes and a Markov Decision Process Policy. *Sensors* **2019**, *19*, 1309.
3. Wusk, G.; Gabler, H. Non-Invasive Detection of Respiration and Heart Rate with a Vehicle Seat Sensor. *Sensors* **2018**, *18*, 1463.
4. Galanis, I.; Anagnostopoulos, I.; Gurunathan, P.; Burkard, D. Environmental-Based Speed Recommendation for Future Smart Cars. *Future Internet* **2019**, *11*, 78.
5. Hobert, L.; Festag, A.; Llatser, I.; Altomare, L.; Visintainer, F.; Kovacs, A. Enhancements of V2X communication in support of cooperative autonomous driving. *IEEE Commun. Mag.* **2015**, *53*, 64–70.
6. Sun, Y.; Huang, W.L.; Tang, S.M.; Qiao, X.; Wang, F.Y. Design of an OSEK/VDX and OSGi-based embedded software platform for vehicular applications. In Proceedings of the 2007 IEEE International Conference on Vehicular Electronics and Safety, Beijing, China, 13–15 December 2007; pp. 1–6.
7. Kutila, M.; Pyykonen, P.; van Koningsbruggen, P.; Pallaro, N.; Perez-Rastelli, J. The DESERVE project: Towards future ADAS functions. In Proceedings of the 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), Agios Konstantinos, Samos, Greece, 14–17 July 2014; pp. 308–313.
8. Jo, K.; Kim, J.; Kim, D.; Jang, C.; Sunwoo, M. Development of Autonomous Car—Part I: Distributed System Architecture and Development Process. *IEEE Trans. Ind. Electron.* **2014**, *61*, 7131–7140.
9. Jo, K.; Kim, J.; Kim, D.; Jang, C.; Sunwoo, M. Development of autonomous car—Part II: A case study on the implementation of an autonomous driving system based on distributed architecture. *IEEE Trans. Ind. Electron.* **2015**, *62*, 5119–5132.
10. Martínez-Fernández, S.; Ayala, C.P.; Franch, X.; Nakagawa, E.Y. A Survey on the Benefits and Drawbacks of AUTOSAR. In Proceedings of the First International Workshop on Automotive Software Architecture—WASA '15, Montreal, QC, Canada, 4–8 May 2015; pp. 19–26.
11. Fürst, S.; Mössinger, J.; Bunzel, S.; Weber, T.; Kirschke-Biller, F.; Heitkämper, P.; Kinkelin, G.; Nishikawa, K.; Lange, K. AUTOSAR–A Worldwide Standard is on the Road. In Proceedings of the 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden, Germany, 7–8 October 2009; Volume 62, p. 5.
12. Wozniak, E.; Tucci-Piergiovanni, S.; Mraidha, C.; Gerard, S. An Integrated Approach for Modeling, Analysis and Optimization of Systems whose Design Follows the EAST-ADL2/AUTOSAR Methodology. *SAE Int. J. Passeng. Cars Electron. Electr. Syst.* **2013**, *6*, 276–286.
13. Melin, J.; Boström, D. Applying AUTOSAR in Practice: Available Development Tools and Migration Paths. Master's Thesis, School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden, April 2011.
14. Sun, B.; Huang, S.T. AUTOSAR Acceptance Test of Communication on CAN Bus. Master's Thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, November 2017.
15. Jansson, J.; Elgered, J. AUTOSAR Communication Stack Implementation with FlexRay. Master's Thesis, Chalmers University of Technology, Gothenburg, Sweden, March 2012.
16. Kulatỳ, O. Message Authentication for CAN Bus and AUTOSAR Software Architecture. Master's Thesis, Czech Technical University in Prague, Prague, Czech Republic, 2015.
17. AUTOSAR Methodology. Available online: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_RS_Methodology.pdf (accessed on 3 May 2019).
18. Chaaban, K.; Leserf, P.; Saudrais, S. Steer-By-Wire system development using AUTOSAR methodology. In Proceedings of the 2009 IEEE Conference on Emerging Technologies & Factory Automation, Palma de Mallorca, Spain, 22–25 September 2009; pp. 1–8.
19. Kumar, M.; Yoo, J.; Hong, S. Enhancing AUTOSAR methodology to a cotsbased development process via mapping to V-Model. In Proceedings of the 2009 IEEE International Symposium on Industrial Embedded Systems, Lausanne, Switzerland, 8–10 July 2009; pp. 50–53.
20. Sung, K.; Han, T. Development Process for AUTOSAR-based Embedded System. *Int. J. Control Autom.* **2013**, *6*, 10.

21. Hebig, R. *Methodology and Templates in AUTOSAR*; Technical Report; HassoPlattner-Institut für Softwaresystemtechnik: Potsdam, Germany, 2009.
22. MPC5748G EVB User Guide. Available online: https://www.nxp.com/files-static/microcontrollers/doc/user_guide/MPC5748GEVBUG.pdf (accessed on 15 May 2019).
23. Freese, M.; Singh, S.; Ozaki, F.; Matsuhira, N. Virtual robot experimentation platform v-rep: A versatile 3d robot simulator. In Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots, Darmstadt, Germany, 15–18 November 2010; pp. 51–62.
24. Rohmer, E.; Singh, S.P.N.; Freese, M. V-REP: A versatile and scalable robot simulation framework. In Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, 3–7 November 2013; pp. 1321–1326.
25. Naumann, N. *Autosar Runtime Environment and Virtual Function Bus*; Technical Report; Hasso-Plattner-Institut: Potsdam, Germany, 2009; p. 38.
26. AUTOSAR Layered Software Architecture. Available online: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (accessed on 26 August 2019).
27. Warschofsky, R. *AUTOSAR Software Architecture*; Hasso-Plattner-Institute für Softwaresystemtechnik: Potsdam, Germany, 2009.
28. Gosda, J. *Autosar Communication Stack*; Technical Report; Hasso-Plattner Institute fur Softwaresystemtechnik: Potsdam, Germany, 2009; pp. 13–23.
29. Delgado, R.; Park, J.; Choi, B.W. Open embedded real-time controllers for industrial distributed control systems. *Electronics* **2019**, *8*, 223.
30. Koh, J.H.; Choi, B.W. Real-time performance of real-time mechanisms for rtai and xenomai in various running conditions. *Int. J. Control Autom.* **2013**, *6*, 235–246.
31. Anssi, S.; Tucci-Piergiovanni, S.; Kuntz, S.; Gérard, S.; Terrier, F. Enabling scheduling analysis for AUTOSAR systems. In Proceedings of the 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, Newport Beach, CA, USA, 28–31 March 2011; pp. 152–159.
32. Delgado, R.; You, B.J.; Choi, B.W. Real-time control architecture based on xenomai using ros packages for a service robot. *J. Syst. Softw.* **2019**, *151*, 8–19.
33. Delgado, R.; Choi, B.W. Network-Oriented Real-Time Embedded System Considering Synchronous Joint Space Motion for an Omnidirectional Mobile Robot. *Electronics* **2019**, *8*, 317.
34. Park, J. Collision Warning System. Available online: https://github.com/qkrwoghsla12/ARCCORE-collision-warning-system (accessed on 26 May 2019).