Getting started with the power liftgate system built with a model-based design approach in the AutoDevKit ecosystem

Introduction

With the electrification mega-trend, more and more body and convenience applications are electrifying the actuations that used to be manual. Among these automotive applications, one with renovated interest is the power liftgate.

The aim of this mockup is to manage and automate the car trunk opening/closing while simplifying the trunk access.

We propose an innovative and more reliable approach compared to the solutions currently used in cars. This approach takes advantage of two Time-of-Flight (ToF) sensors placed under the car bumper to open/close the trunk through a predefined foot gesture.

1 Getting started

1.1 Overview

The power liftgate main goal is to ease the car trunk access. Several mid-to-high end car models already implement this capability by allowing the car trunk opening/closing through a kick in the air performed underneath the bumper. We estimate that the power liftgate market size is going to be greater than 5 billion USD by 2026, with a 6% CAGR over the forecast period.

Figure 1. Power liftgate: use case

Some blogs in the internet report that the kick to open the trunk is difficult to operate, and some users admit that it works only a few times. Generally, the activation is successful once out of three times. For the same reason, several car makers have posted videos to explain to the users how to properly kick under the sensor.

Other car manufacturers require you to stand three seconds in front of the car. Neither of these solutions works perfectly, as you can lose the stability by kicking while wearing high heels and standing in front of the trunk for three seconds could feel like an eternity if you are carrying heavy loads.

Figure 2. **Power liftgate opening**

In the power liftgate system, the usage of Time-of-Flight (ToF) sensors represents an innovative approach while the NFC and Bluetooth® Low Energy are redundant car access methods.

The proposed foot detection algorithm with ToF sensors has a high stability in recognizing the gesture performed.

1.2 Power liftgate features

- Control of two linear DC motor actuators to open and close the trunk
- Motion detection to prevent the trunk opening/closing if the vehicle is not still
- Trunk locking and unlocking
- Visual alert through turning lights blinking at the beginning of the trunk opening/closing phases
- Acoustic alert through a beeper at the beginning of the trunk opening/closing phases
- Trunk opening/closing with an NFC tag key
- Trunk opening/closing via mobile phone app employing Bluetooth® Low Energy
- Display of trunk state through a mini-infotainment touch display
- Self-calibration of the system motors
- Trunk opening/closing jam detection
- Trunk opening height adjustment

1.3 Safety mechanisms

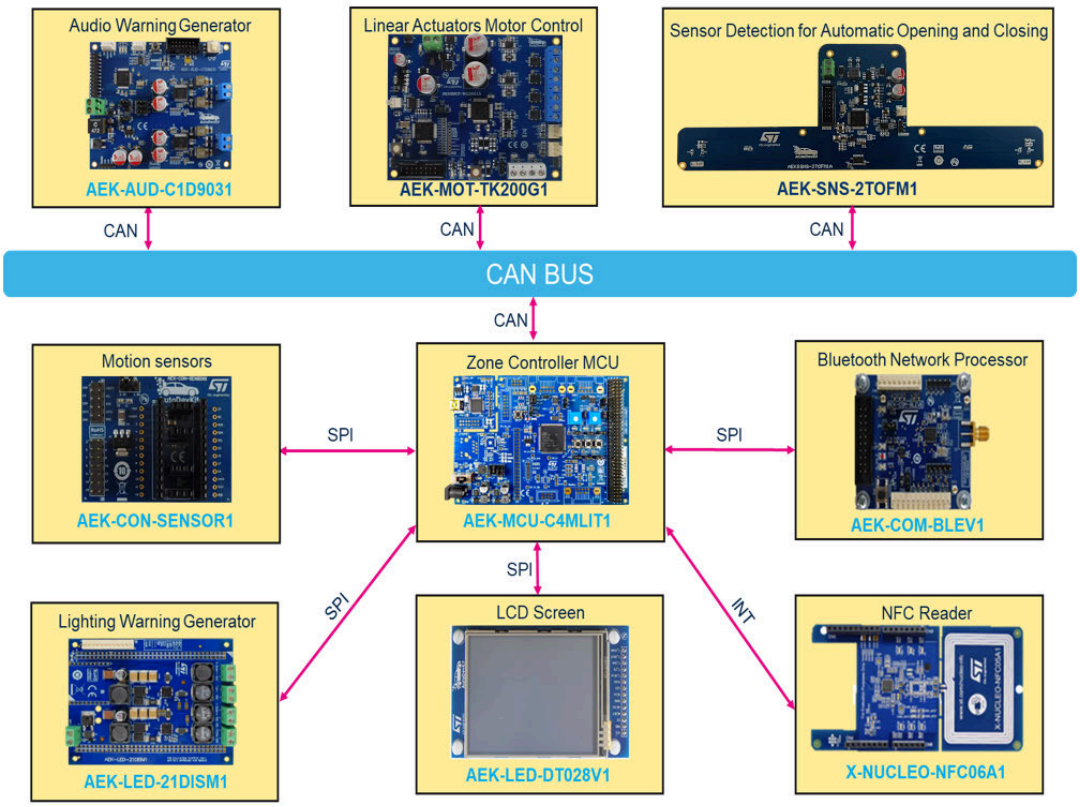
The power liftgate features the following safety mechanisms:

1. Detection of possible obstacles that hinder the trunk opening/closing. When detecting obstacles, the system stops the motors and reverses them with a slow speed sufficient to step away from the contact point.
2. Prevention of the trunk opening/closing when detecting the car motion via a MEMS accelerometer.
3. Continuous monitoring of the position variation of both motors to prevent fault conditions due to the disconnection of the motors and their related encoders. If the motor position remains unchanged for a determined time interval, the system stops the trunk linear motor driving, thus arresting the opening/closing actuation.

2 Hardware architecture

The power liftgate architecture consists of a set of subsystems managed by an AEK-MCU-C4MLIT1, which acts as a domain/zone controller (for further info, refer to AutodevKit).

Figure 3. Power liftgate architecture



To open/close the trunk or stop it in a desired position, you can send commands through:

- a gesture detected by the AEK-SNS-2TOFM1;
- communication from the mobile phone app via the Bluetooth® Low Energy to the AEK-COM-BLEV1;
- an NFC device in proximity of the X-NUCLEO-NFC06A1 reader.

Before opening/closing the trunk, the AEK-LED-21DISM1 and AEK-AUD-C1D9031 emit visual and acoustic signals, respectively, to alert the users about the actuation.

2.1 AEK boards in the power liftgate system

The power liftgate system uses the following AEK boards:

- AEK-MCU-C4MLIT1
- AEK-MOT-TK200G1
- AEK-SNS-2TOFM1
- AEK-COM-BLEV1
- X-NUCLEO-NFC06A1
- AEK-CON-SENSOR1
- AEK-LCD-DT028V1
- AEK-LED-21DISM1
- AEK-AUD-C1D9031

The AEK-MCU-C4MLIT1 board works as a domain/zone controller. The board manages the power liftgate system via in-vehicle communication through several protocols (CAN, SPI, I²C, etc.).

The AEK-MOT-TK200G1 board manages the linear DC and the trunk-lock motors. It also manages the trunk interior illumination and guarantees high levels of safety and reliability. In addition, the board provides the system standby functionality, where the power outputs are turned off.

The AEK-SNS-2TOFM1 board opens and closes the trunk through the foot detection algorithm. It manages the foot recognition through the two embedded Time-of-Flight (ToF) sensors. At power-on, if no error occurs in the sensor initialization, the board turns on the two LEDs placed near the sensors. When the predefined gesture is performed, the two on-board green LEDs blink three times. Upon correct recognition, the microcontroller sends a CAN message to the domain/zone controller via a CAN connector.

The AEK-COM-BLEV1 board is based on the BlueNRG-1 Bluetooth® Low Energy smart system-on-chip. This board is connected to a microcontroller via SPI. It sends actuation commands to the power liftgate application via Bluetooth® Low Energy and a mobile phone app.

The X-NUCLEO-NFC06A1 NFC card reader expansion board manages the frame coding and decoding in reader mode for standard NFC applications. Upon NFC tag recognition, the power liftgate opens/closes the trunk. For further details on how to use the NFC technology with AutoDevKit and X-NUCLEO-NFC06A1, see Section Appendix A NFC technology with AutoDevKit and X-NUCLEO-NFC06A1.

The AEK-LCD-DT028V1 board is used as a mini-infotainment GUI that shows the system status. The screen gives a simple and fast way to display data or menus when prototyping. The displayed system status depends on different events (for example, an NFC key detection).

The AEK-LED-21DISM1 board drives two LED strings to alert about the trunk opening/closing. During the actuation, the two LED strings represent the car turning lights. The same visual alert occurs when the power liftgate system detects a failure.

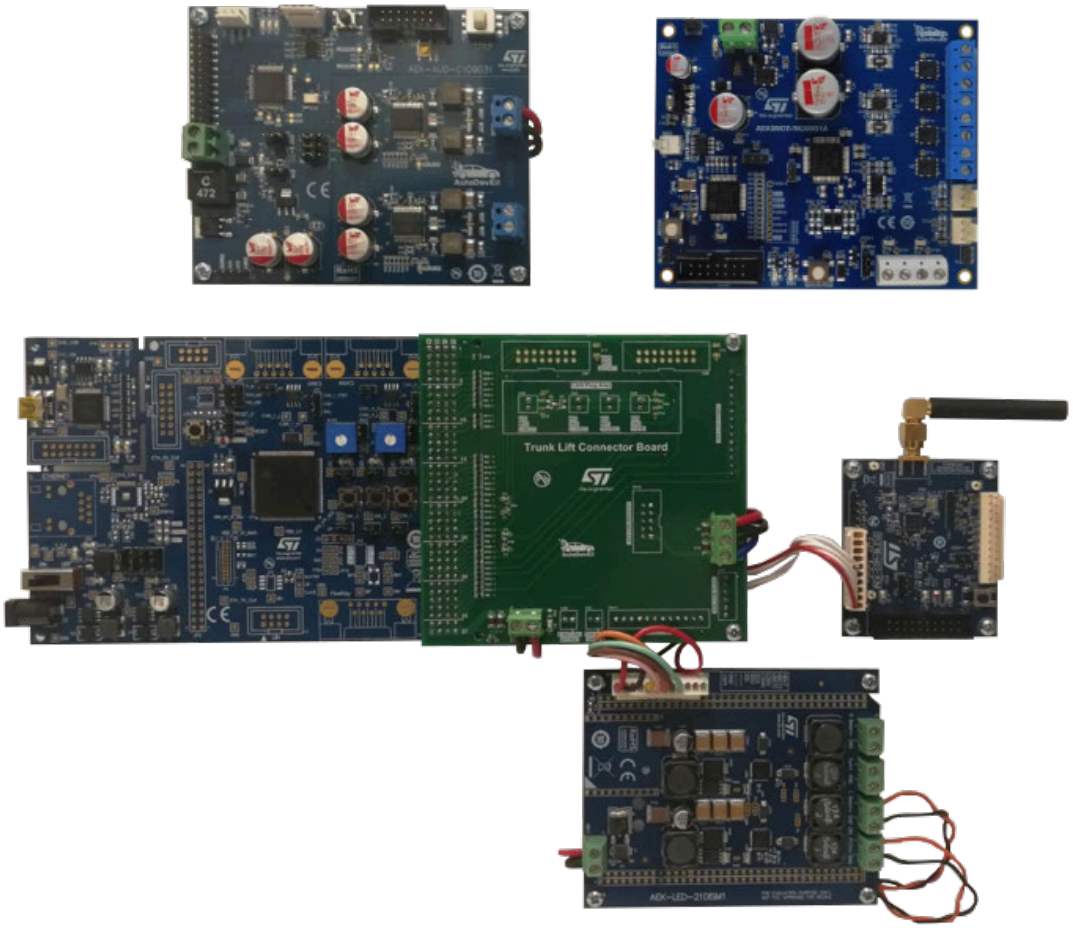
The AEK-CON-SENSOR1 board hosts an automotive 6-axis inertial module. The accelerometer is used to detect the vehicle motion.

The AEK-AUD-C1D9031 is a very compact acoustic vehicle alerting system (AVAS) that emits warning sounds to alert pedestrians of the presence of e-vehicles. This board generates a beep at the beginning of the trunk opening/closing process. Moreover, the acoustic alert is generated also when failures are detected.

Note: For further details on how to use the NFC technology with AutoDevKit and X-NUCLEO-NFC06A1, see Section Appendix A NFC technology with AutoDevKit and X-NUCLEO-NFC06A1.

For further details on how to use the Bluetooth® Low Energy technology with AutoDevKit and AEK-COM-BLEV1, see Section Appendix B How to use Bluetooth® Low Energy with AutoDevKit and AEK-COM-BLEV1.

Figure 4. **Power liftgate system**

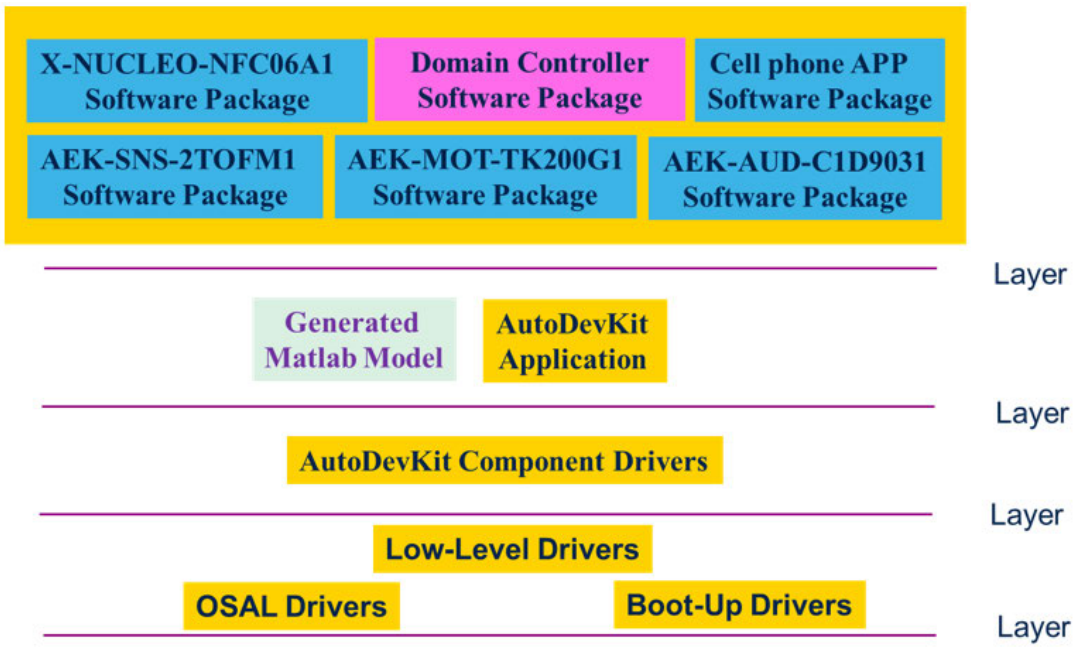


3 Software architecture

The software architecture is based on the AutoDevKit paradigm with additions coming from the MATLAB® tools.

These software architecture layers ensure the encapsulation and abstraction for each layer towards the subsequent one. Thus, the layers ensure reuse and portability. Each layer has defined interfaces for the connection to the upper layer.

Figure 5. Software architecture layers



Just over the hardware layer, there is the hardware abstraction layer (HAL). This layer consists of:

- low-level drivers: used to drive the microcontroller peripherals and enable basic communication protocols (I²C, SPI, CAN, Ethernet);
- boot-up drivers: used to boot the microcontroller and ensure an adequate security level;
- OSAL drivers: used to manage the interrupt handling, core activation, time keeping, and basic memory.

Above the HAL, there is the AutoDevKit component driver layer. This layer contains all the drivers for the external functional boards connected to the microcontroller, that is for example, the component for linear actuators based on the AEK-MOT-TK200G1 board and the component for the LCD touchscreen based on the AEK-LCD-DT028V1 board.

The above layer is the platform configuration layer, where all the platform settings are performed automatically. According to the AutoDevKit component configuration, pins are allocated and the microcontroller peripherals initialized for the proper communication with the functional boards.

One step above, there is the application layer, where the application is written using the high-level APIs available for each of the AutoDevKit component instantiated. In this level, the application source code is combined with the one generated with the *Embedded Coder*, starting from the model in Simulink.

The resulting complete code is compiled and downloaded in the microcontroller for execution.

3.1 Power liftgate application

The complete power liftgate application consists of several software packages.

The main application code for the domain controller is downloaded and run on the SPC58EC Chorus 4M microcontroller of the AEK-MCU-C4MLIT1 board.

A specific driver code for the L99DZ200G device is downloaded in the SPC582B Chorus 1M microcontroller of the AEK-MOT-TK200G1 board. This driver establishes communication with the L99DZ200G via SPI and CAN with the SPC58EC main domain controller of the AEK-MCU-C4MLIT1 board.

Another specific driver code for the audio board is downloaded in the SPC582B Chorus 1M microcontroller of the AEK-AUD-C1D9031 board. This driver establishes I²S and I²C communication with the FDA903x class D audio amplifiers and CAN communication with the domain controller.

For the management of the Time-of-Flight (ToF) sensors, a dedicated driver in the SPC582B Chorus 1M (or 2M) microcontroller of the AEK-SNS-2TOFM1 board is present. This driver reads the data from the sensors via SPI and recognizes the appropriate gesture for the trunk opening and closing. The driver communicates with the domain controller via a CAN bus.

Also for the NFC tag recognition, a separate software package is required, which runs on the SPC582B (or SPC584B) Chorus 1M (or 2M) microcontroller of the SPC582B-DIS (or SPC584B-DIS). Through the Arduino connector, this board is connected to the X-NUCLEO-NFC06A1 board that hosts the ST25R3916 NFC tag reader. The driver manages the tag recognition and informs the domain controller by generating an interrupt when the NFC tag is correctly recognized.

The AEK Controller APP .apk file is loaded in an Android mobile phone and the app is launched to control the trunk via Bluetooth® Low Energy.

As the drivers are embedded directly in the domain controller code, there are no specific packages for the Bluetooth® Low Energy protocol management and command parsing, for the motion sensor detection, for the LCD screen control, and the LED control board. The LCD screen, motion sensor, and LED driver use the SPI protocol to communicate with the domain controller while the Bluetooth® Low Energy uses the serial protocol.

The combination of a model-based design and the AutoDevKit allows designing and validating a model independently from the computing platform. Thus, it is possible to focus on implementing the model for a complex system without dealing with the running platform details.

The AutoDevKit generates and configures a computing platform in line with the system requirements where it is possible to host the software generated from the model. Once the code is combined, you can start the hardware-in-the-loop (HIL) validation of the complete system. Several fine tunings are possible during the validation phase as both AutoDevKit and MATLAB® regenerate the appropriate code as soon as the developer applies the modifications.

Another clear advantage is that the code generated from the model is prevalidated in the simulation with the model-in-the-loop (MIL). This significantly reduces the number of interactions that occur with the HIL validation only model.

The combined approach of AutoDevKit and MATLAB® is effectively unleashing the possibility of proceeding with a fast prototyping and an early validation of the system to implement. This innovative technique is unique and fundamental to be compliant with the more and more stringent requests for a reduced time-to-market.

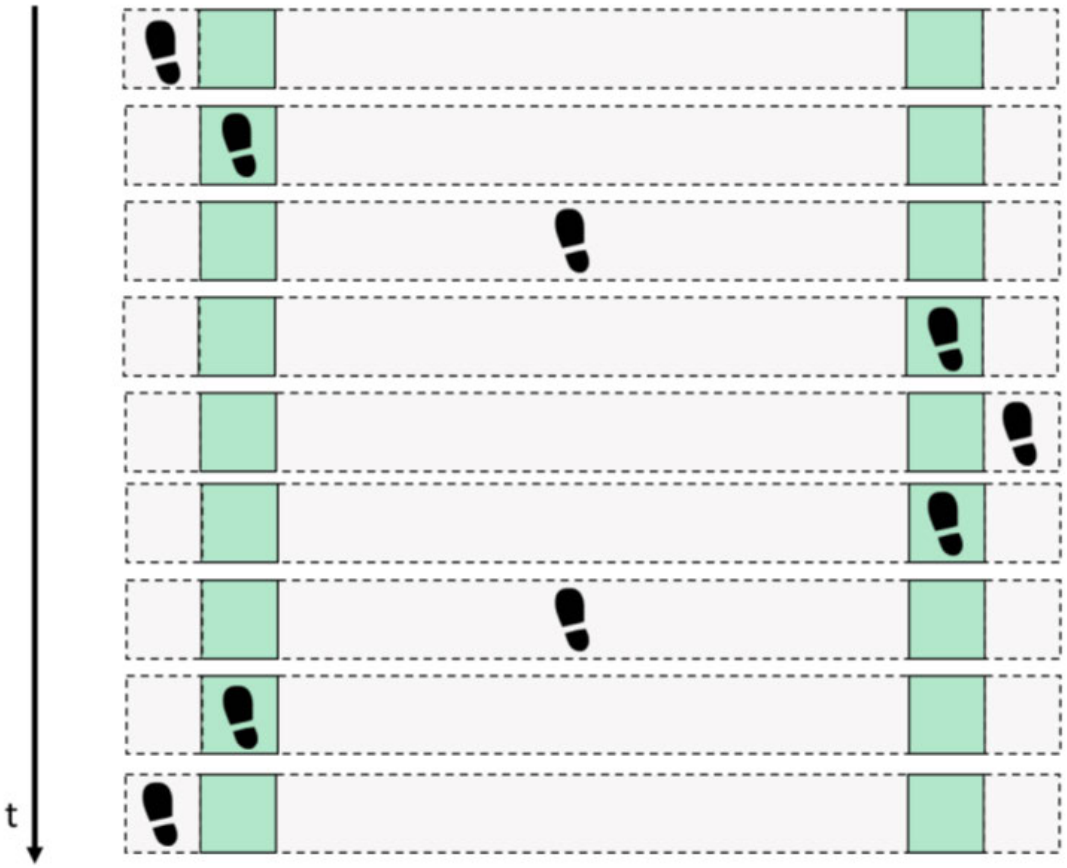
3.1.1 Gesture recognition algorithm with ToF sensors

A gesture recognition algorithm has been implemented for the power liftgate system.

To open the trunk by performing a gesture with the foot under the bumper, a specific gesture has been chosen to increase reliability and avoid undesired activations.

The system installed under the bumper consists of two ToF sensors. The correct gesture involves a detection sequence involving both sensors.

Figure 6. **Accepted foot sequence to open/close the trunk**



The gesture consists in crossing the detection area of each sensor in both directions.

Each sensor has a maximum distance detection threshold; therefore, the foot is detected within this distance. The detection has to happen within a specific time interval. If it is too fast or too slow, the sequence is not recognized. If both foot sequence and timing are valid, the gesture is recognized and the trunk starts opening (or closing).

In the power liftgate system, the maximum admissible distance has been reduced to place the demo on a table. At power-up, the sensor board calibrates the maximum distance at about 132 millimeters.

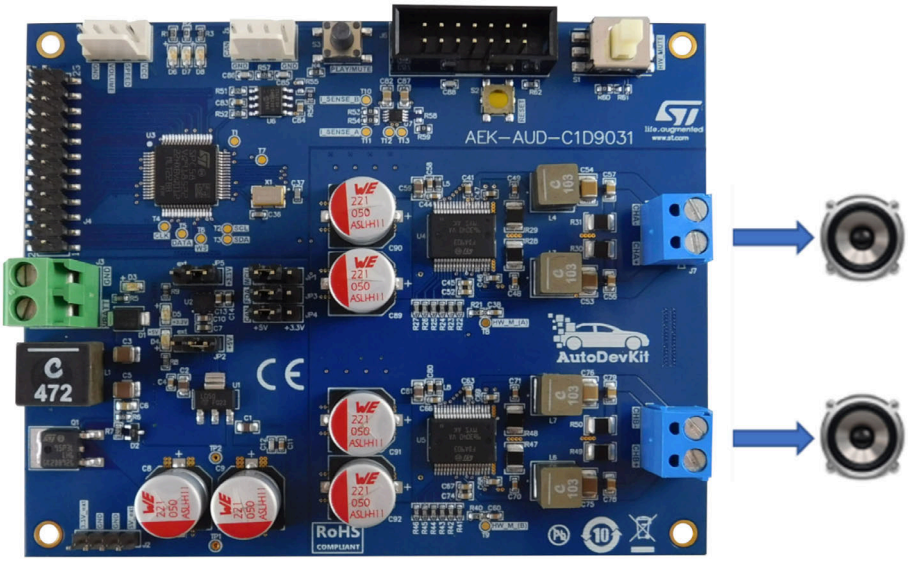
Note: For further details on the implementation of the gesture recognition algorithm and the AEK-SNS-2TOFM1 board, see UM3006.

3.1.2 Acoustic alerting

To create a simple acoustic alert that simulates a beep sound, the AEK-AUD-C1D9031 AVAS board has been modified to play a digital sound sequence stored in the microcontroller flash memory. The sound is stored in digital samples of 32 bits. The two on-board class D audio amplifiers uses only 24 bits out of the available 32.

The dedicated SPC582B chorus 1M on-board microcontroller reads the data from the flash memory and sends the audio sample to the audio amps via the I²S protocol.

Figure 7. **Acoustic alert**



A .wav file contains the beeping sound, sampled at 44.1 kHz to be compliant with the playback system implemented in the board. The .wav file is divided in several blocks. Each of them is loaded into a specific dedicated location of the microcontroller flash memory.

The CAN messages sent by the domain controller drive the audio-alerting subsystem operation. The domain controller puts the system in play or stop status. In the power liftgate, the audio alerting is normally played during the normal opening and closing of the trunk. In addition, the beeping sound is played also when failures are detected.

Note: For further details on the AEK-AUD-C1D9031, see UM2719

3.1.3 Mini-infotainment subsystem

The mini-infotainment subsystem is implemented with a color LCD touchscreen. It shows the power liftgate system status. The domain controller directly controls the LCD screen.

Figure 8. **LCD display**

The system status depends on different events:

- the AEK-MOT-TK200G1 communicates the trunk status to the domain controller via the CAN bus (for example, trunk closing, trunk closed, etc.)
- the NFC key detection is communicated to the domain controller via a dedicated interrupt. The detection message is then displayed on the AEK-LCD-DT028V1 LCD
- when the ToF sensor board detects and recognizes the appropriate gesture sequence, the domain controller is informed, and the status displayed on the LCD screen

- the domain controller displays a specific status when the calibration starts, is complete, or in progress.

Figure 9. Printing “closed” message

```
case CLOSED_TRUNK:
        aek_ili9341_clearScreen(AEK_LCD_DEV0, BLUE);
        aek_ili9341_drawString(AEK_LCD_DEV0, 33, 125, "CLOSED", color, font24pt);
        break;
```

Since the system dynamic is fast and the SPI protocol throughput limits the display refresh, the LCD display management is in the second core available in the domain controller board.

The core dedicated to the LCD message handling has an internal FIFO queue of N elements. These elements are used to store the received display requests. This ensures that status messages are displayed in the correct time order.

Note: For further details on the AEK-LCD-DT028V1, see UM2999.

3.1.4 LED light visual alerting

The AEK-LED-21DISM1 board drives up to four parallel LED strings. In the power liftgate system, it gives a visual alert to the users when the trunk is opening or closing.

During the actuations, the board drives two LED strings that represent the car turning lights.

The same visual alert is given when the power liftgate system detects a failure.

The two main API functions used are the `Activate` and `Deactivate` buck, which, respectively, turn the LED strings on and off.

Figure 10. LED alerting code

```
void lights_blinking_routine(uint8_t n, uint8_t wait_time){
    int i;
    for(i = 0; i<n; i++){
        ActivateBuckDev(AEK_LED_21DISM1_DEV0, DEV1, BUCK1);
        ActivateBuckDev(AEK_LED_21DISM1_DEV0, DEV1, BUCK2);
        osalThreadDelaySeconds(wait_time);
        DeActivateBuckDev(AEK_LED_21DISM1_DEV0, DEV1, BUCK1);
        DeActivateBuckDev(AEK_LED_21DISM1_DEV0, DEV1, BUCK2);
        osalThreadDelaySeconds(wait_time);
    }
}
```

3.1.5 Bluetooth® Low Energy communication subsystem

The AEK-COM-BLEV1 implements a network processor compliant with the Bluetooth® Low Energy.

This board is connected to the domain controller via a serial interface.

The domain controller manages the availability of the Bluetooth® Low Energy services. At power-up, it sends a discovery packet that contains the network information to enable the pairing with a mobile phone device.

Upon successful connection, a dedicated app is able to request services like opening and closing the power liftgate.

A token encrypted with public and private cryptographic keys ensures the connection between the mobile app and the domain controller. To connect the mobile app, a valid token has to be generated and sent to the domain controller. This simulates the capability of having a car key in the mobile app.

3.1.5.1 *Bluetooth® Low Energy architecture*

The power liftgate domain controller (AEK-MCU-C4MLIT1) communicates via the UART protocol with the Bluetooth® Low Energy host-controller device (AEK-COM-BLEV1). The latter is configured as a GATT server with a peripheral role. In this scenario, a typical GATT client (that is, a smartphone with the AEK controller app) is able to connect to the server to perform the trunk opening/closing and the self-learning procedure (that is, the calibration).

To allow only authorized users to perform these procedures, a secure algorithm, based on a token key exchange, has been implemented.

The AEK-COM-BLEV1 is a Bluetooth® Low Energy host/controller device, driven by a domain control unit (AEK-MCU-C4MLIT1), configured with:

- a primary service (UUID-128 bit) that consists of:
  - a status characteristic (UUID-128 bit) used to share the status of the trunk (open, closed, etc.) with a GATT client (AEK controller app)
  - a command characteristic (UUID-128 bit) used to write/read the power liftgate command (opening/closing/self learning).
- a secondary service (UUID-128 bit) that consists of:
  - a token characteristic (UUID-128 bit) used to exchange a key with the GATT client (AEK controller app). This key allows accessing the power liftgate system

The next section details each characteristic to explain how the GATT server interacts with the authorized GATT client.

3.1.5.2 *Token key algorithm*

An Android device with the AEK controller app installed can communicate with the power liftgate system only if the domain controller unit authorizes it. When the access is authorized, you can open/close the trunk and run a self-learning procedure by using a common smartphone as a remote car key.

When an Android device becomes a GATT client, a unique and random 4-byte integer code (a token key) identifies it. If the GATT client token key is already stored in the EEPROM list of active tokens of the power liftgate controller unit, the access is authorized and the communication starts.

There are two kinds of token keys: master and secondary.

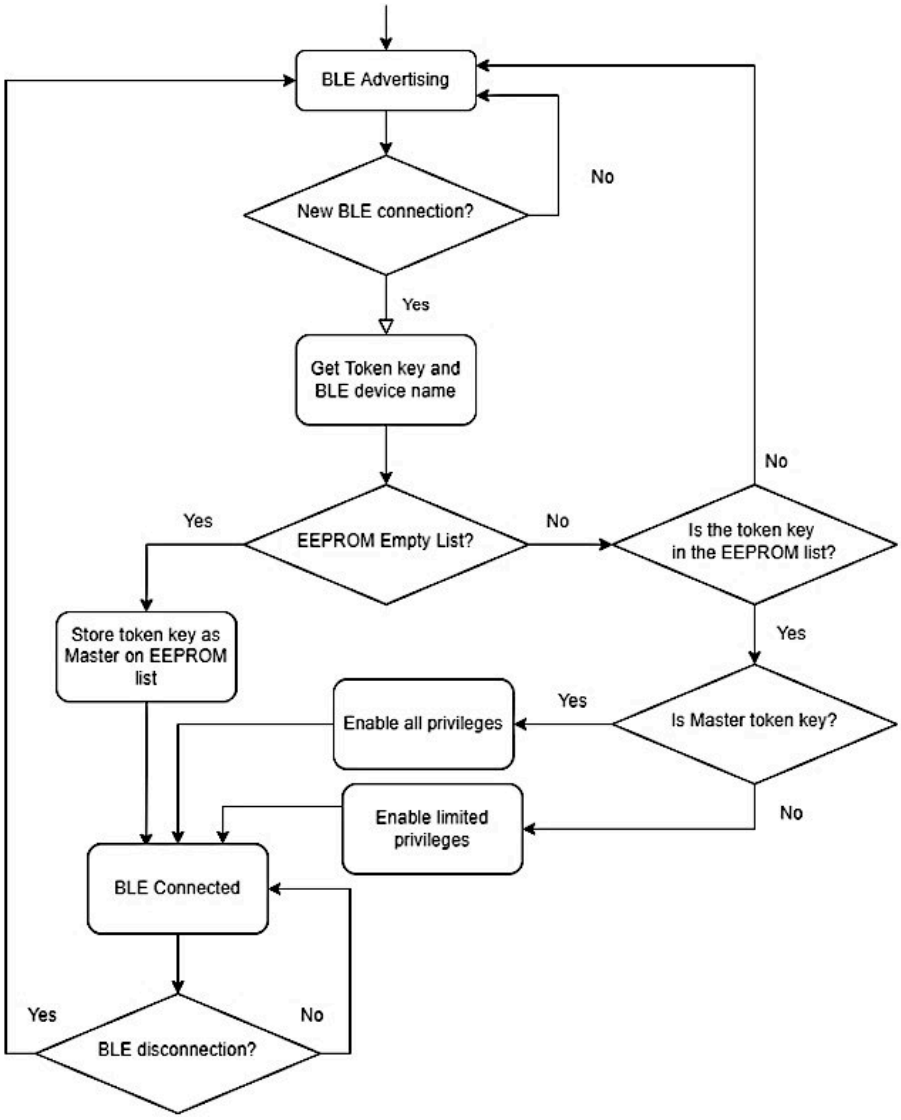
A device with a master token has the following privileges enabled:

- opening/closing/self-learning procedure allowed
- add a new secondary token key
- remove a specific secondary token key
- remove itself as a master token key

A device with a secondary token key has the following limited privileges enabled:

- opening/closing/self-learning procedure allowed

Figure 11. **Block diagram of the token key algorithm**



In the initial state, the token key EEPROM list of the power lift gate domain controller unit is empty. The first Android device (with the AEK controller app installed) which automatically connects via Bluetooth® Low Energy becomes a device with a master token. Otherwise, if the token key EEPROM list of the power lift gate domain controller unit is not empty, the system checks if the received token key is stored in the list. If yes, it checks if it is a master or a secondary token key. Otherwise, the device is automatically disconnected.

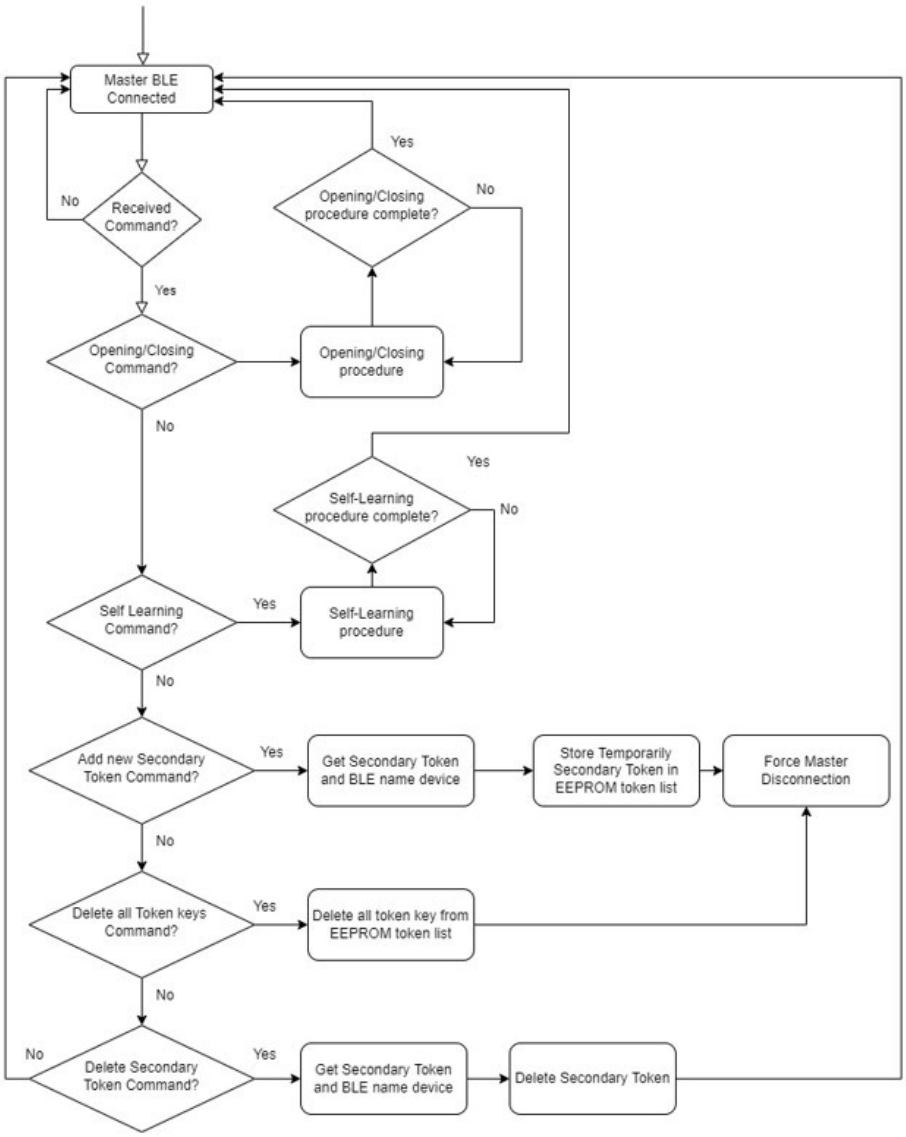
The EEPROM token list can store a maximum of five tokens (one master token and four secondary tokens).

3.1.5.2.1 Master token: key privileges

When a device with a master token key is connected and authorized, it is the only device with all the privileges enabled:

- opening/closing/self-learning procedure allowed
- add a new secondary token key
- remove a specific secondary token key
- remove itself as a master token key

Figure 12. **Block diagram of the master token privileges**



The commands to add a new secondary token or delete all token keys from the EEPROM list generate an automatic disconnection after execution.

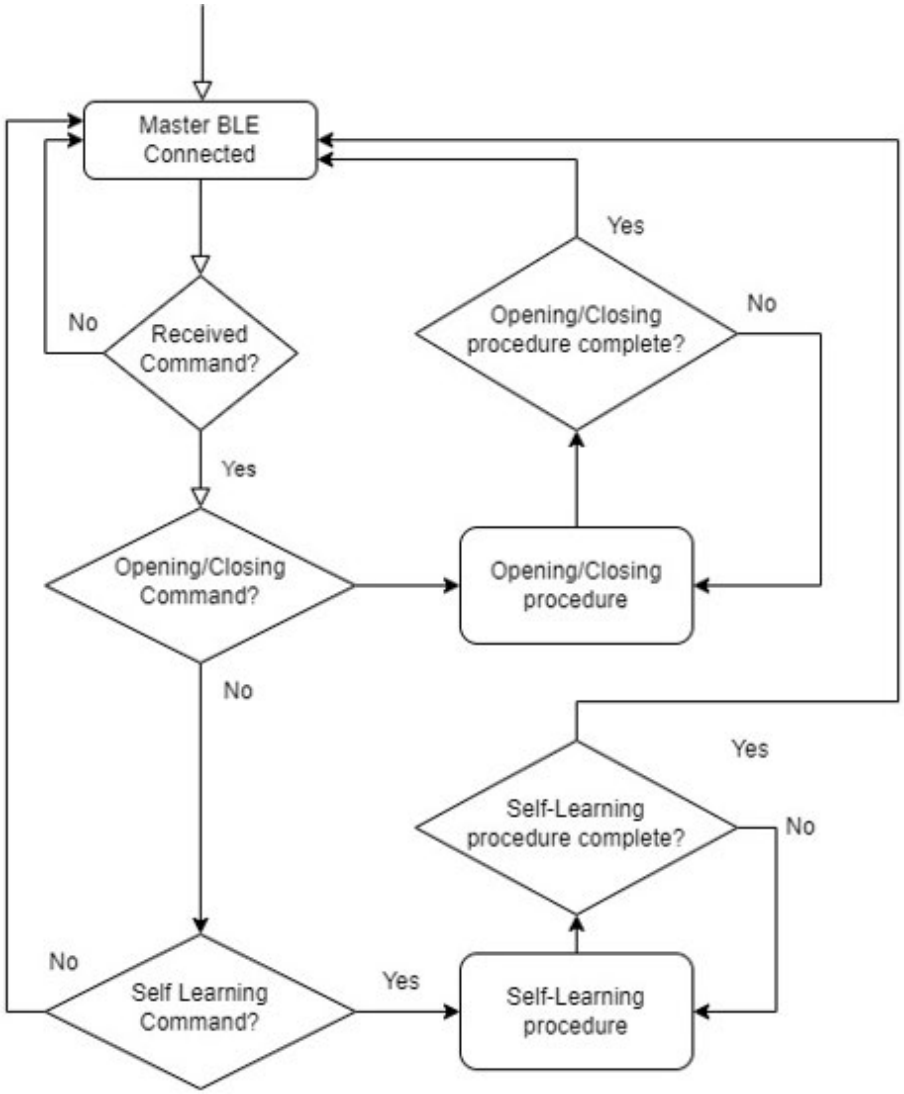
If a secondary token is added, after the master device disconnection, the newly secondary token has only 20 seconds to perform the connection with the domain controller. Otherwise, it is removed from the EEPROM token key list.

3.1.5.2.2 Secondary token: key privileges

When a device with a secondary token key is connected and authorized, it has limited privileges enabled:

- opening/closing/self-learning procedure allowed

Figure 13. **Block diagram of the secondary token privileges**



3.1.5.2.3 Manual removal of all tokens from the list

If a factory reset is needed, you can activate a hidden manual procedure to delete all the tokens from the EEPROM token list.

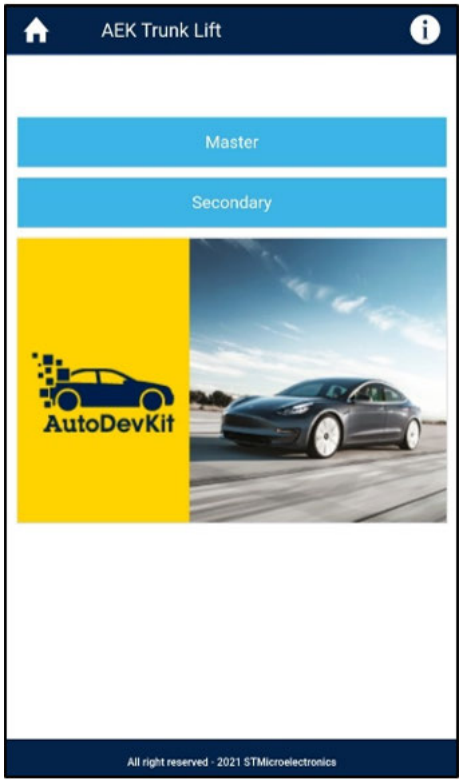
Quickly press the [**Failsafe**] button (red button) on top of the power liftgate demo 10 times. Restart the power liftgate system to empty the EEPROM token list.

3.1.6 AEK controller app

The AEK controller app offers the user the possibility to control remotely the opening/closing of the power liftgate system. The app has the following requirements:

- each power liftgate system can accept only one AEK controller as a master
- only the AEK controller configured as a master can associate a secondary AEK controller installed on another mobile to the power liftgate system
- the communication between the power liftgate system and the app is validated by exchanging encrypted tokens
- while connected, the power liftgate system can interact only with one AEK controller at a time

Figure 14. **App main screen: master and secondary view**



There is only one mobile app that can play the role of “master car key”. There is also the possibility to elect secondary car keys subject to the master car key app approval. The master app can revoke the permissions to the secondary car key.

The app can also start the initial calibration and self-learning procedure of the power liftgate system.

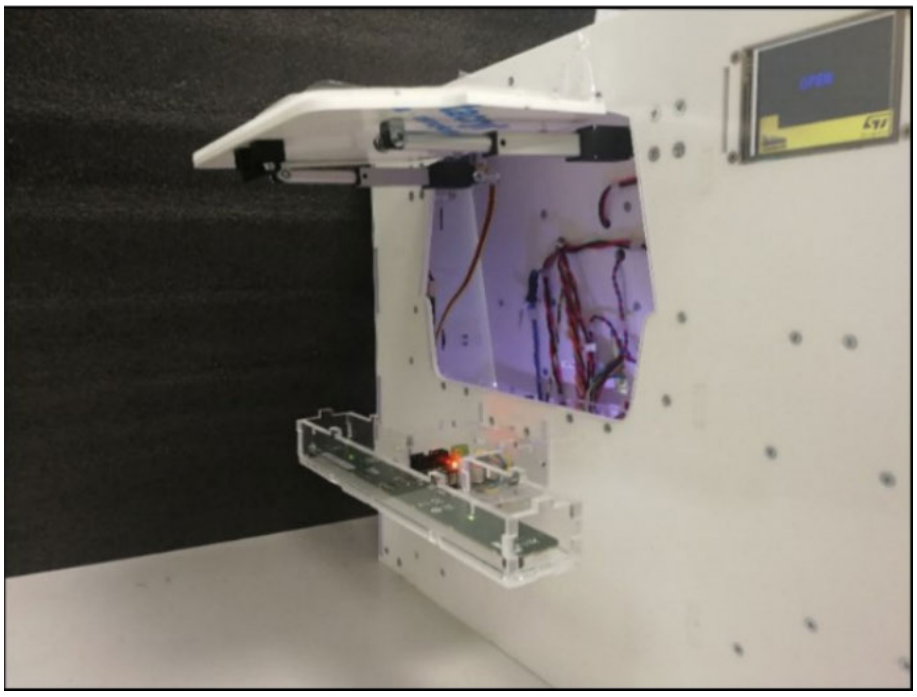
Figure 15. **App input commands**

For further details on the AEK controller, see appendix C.

4 Power liftgate design

The first challenge in the power liftgate system is to drive the two linear motors placed on the trunk tailgate. These two motors have to move synchronously in the same direction and remain aligned while opening and closing the trunk.

Figure 16. Power liftgate system



The control system required for this scenario is quite complex. Therefore, we have decided to take advantage of model-based design.

5 Model-based design overview

The embedded control system development poses several challenges. Expertise in multiple technical fields is required, particularly in electronics and software engineering. Furthermore, product requirements change with changing customer needs and dynamic market conditions. Time-to-market and quality complete the complexity of the product development for revenue and market share increase.

The model-based design main goal is to ensure that the final product meets the customer's needs and requirements through a rigorous design and testing process. You can perform development and test tasks in parallel with the system building throughout the entire process.

A defect discovered after a product release can cost up to 10 times more than fixing the defects discovered during the requirement phase. By applying a rigorous validation process at each phase of the development process, you can drastically reduce the cost of defect fixing.

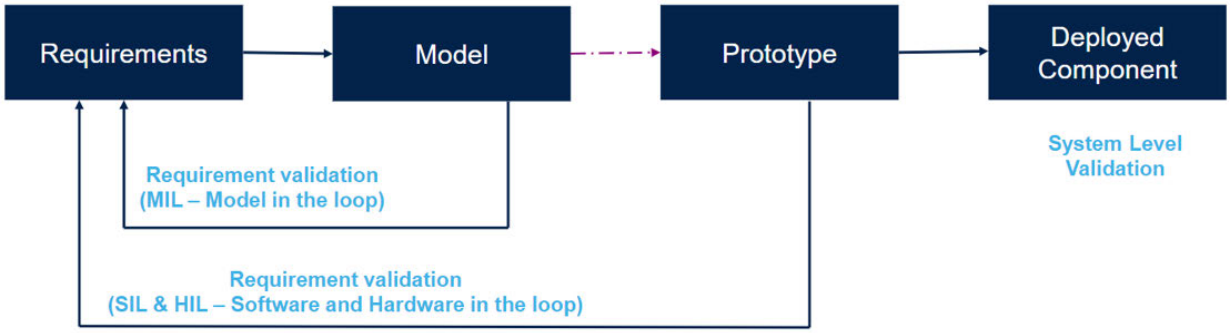
The model design approach links the requirements, design, testing, documentation, and code by using a modeling approach.

Verification and validation at every stage of the design process ensure the product quality.

Various simulation techniques are used for verification: model in the loop (MIL), software in the loop (SIL), hardware in the loop (HIL), and rapid prototyping.

The automatic code generation is another key benefit of these models. It saves costs and development time, improving the overall product quality.

Figure 17. Model-based design workflow



5.1 Modeling fundamentals

A model represents a physical phenomenon and what is needed to control it. It represents the system at a mathematical and behavioral level. To generate a model, it is necessary to observe the system, identify the variables involved in its evolution and measure them. The model is an abstract representation of reality, that is, a process of formalizing the knowledge of an observed physical phenomenon. With this approach, the relationship between the input and output parameters is represented through mathematical relationships.

In the model-based design (MBD), the basic concept is a visual representation of the model. This approach simplifies the design of complex control, communication, and signal processing systems. Models are dynamic. Their response is a mathematical function based on the input values, current state, and current timestamp.

Modeling is possible only if the system involved is observable and controllable. Otherwise, the system would be unpredictable. A system is observable when the outputs can be recorded, and the state of a system can be measured. If a system is not observable, then it cannot be controllable. A system is controllable when it can reach any phase space point computed as finite output variables based on varying input values. In addition to observability and controllability, it must be assessed whether the system has slow or fast dynamics.

The system identification aims at formalizing a physical phenomenon that evolves over time by trying to identify a mathematical relationship between the inputs and outputs of the system. The identification process can be performed with three different approaches:

- white box: in this case, the mathematical relationship between inputs and outputs is known and is represented by a given law (Y=f(X))
- gray box: also in this case, the mathematical relationship between inputs and outputs is known but may slightly vary in time due to some additional parameters (Y=a*f(X)). For example, *a* could depend upon the wearing-out of a mechanical system
- black box: the relationship between inputs and outputs is unknown; therefore, the only possibility is to record the output data in a look-up table when stimuli are applied in the inputs

The black box is a very common procedure in the automotive industry as it considers the disturbances affecting the outputs. It is also possible to apply both the white box and the black box identification on the same process with the goal to obtain a redundant system or a validation system. The obtained outputs are not identical, but the trend should be aligned. The identification phase is applied to describe the behavior of a real system that a system control model controls.

Typically, a controlled system is called a plant. Its control model consists of:

- the control algorithm
- one or more state machines
- the input and output elements mimicking the system physical components

When the control and plant model are linked in a closed loop, we obtain an environment with executable specifications, executable constraints, and direct link to the requirements. In this environment, we can perform an early testing and validation by simulating the entire system, including the external environment. This leads to the model refinement and clear system specifications that screen several errors before the actual design implementation.

5.2 Modeling toolchain

There are several modeling tools like Modelica for domain-specific modeling and MATLAB® Simulink® for safety-critical systems.

MathWorks® toolchain can be used in the automotive and transportation domain. You can develop a typical system model using the Simulink® block diagrams, Stateflow, state machines, and embedded MATLAB® code.

The Simulink® model contains a hierarchy of components to model the complexity of real systems in a better way. This is achieved through virtual components, that is, graphical block containers of the real blocks under simulation (non-virtual blocks). The latter are treated as atomic units, affected by an execution order.

Stateflow is a control logic tool used to model reactive systems via state machines and flow charts within a Simulink® model. The embedded MATLAB® is a subset of the MATLAB® language that supports both simulation and C source code generation.

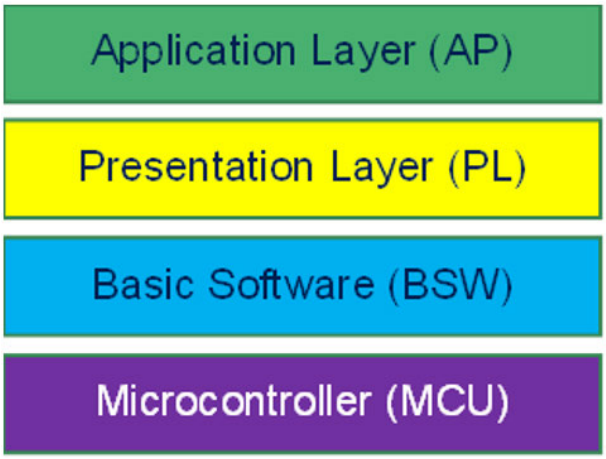
The Embedded Coder is used to for code generation and offers several output code options related to efficiency, traceability, and verifiability.

The requirements can be written in text form in any document editor. You can manually or automatically link them to the model that uses the specific IDs (one ID per requirement).

6 Open software architecture

The model-based design approach follows the open software architecture shown in the following diagram.

Figure 18. Diagram of the open software architecture



The application layer is the uppermost layer of the open software architecture. It supports custom functionality implementations. This layer consists of the specific software components, which are a group of interconnected modules that performs the scheduled tasks. The MathWorks toolchain is used to model the behavior and the interfaces of all the application layer components. C libraries describe these components and automatically generate them.

The presentation layer provides hardware-independent interfaces to the application layer. It implements a communication service between the software components and the basic software layer.

The basic software is the lowest layer that consists of software modules and internal drivers that directly access the microcontroller and its internal peripherals.

The open software architecture allows OEMs, suppliers, and product engineering providers to collaborate on a project without worrying about the underlying hardware platform.

7 Power liftgate model

The power liftgate system consists of two linear actuators that open and close the trunk according to the commands received.

To open and close the trunk, you have to control properly the two linear actuators. The starting point for the control algorithm development is the creation of the model for the ACTUONIX linear actuator used. The extension range of the linear actuators is related to the opening angle of the trunk, subject to the mechanical constraints and supports.

Figure 19. ACTUONIX linear actuator

The power liftgate model implementation is performed using the MathWorks tool chain to ensure that safe-critical properties are included in the generated code.

This portion belongs to the application layer component in the open software architecture stack.

7.1 Plant

In the control theory, a plant is the system to control. Usually, a transfer function represents it to indicate the relationship between the input and output signals.

In the power liftgate system, the linear actuators define the plant to open and close the trunk. The actuators convert the plant input signal (typically a PWM signal) into a physical displacement, following a specific mathematical law.

The ACTUONIX system is a linear actuator consisting of:

- A DC motor driven by a pulse width modulation (PWM) signal. Through a specific frequency and a specific duty-cycle, the PWM signal generates an average voltage value to apply to the armatures of the DC motor.
- A gear system with a 100:1 ratio able to multiply the torque of the DC motor and apply it to the linear actuator.
- A linear actuator defined by a nut screw with an extension of 100 mm.

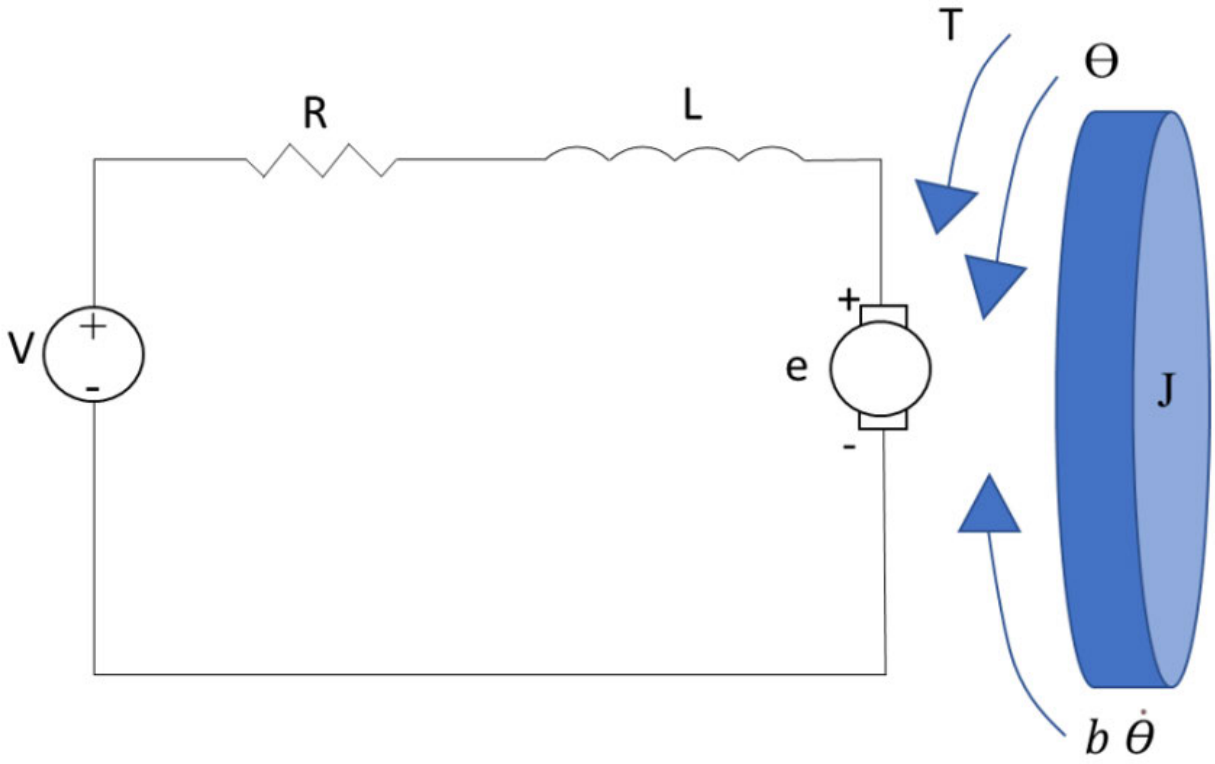
Through the white box identification scheme, we obtain a transfer function. This function binds the extension of the linear actuator (output) with the voltage applied to the DC motor armatures (input). The applied voltage generates an angular velocity of the DC motor converted in linear motion on an internal guide via a nut screw. Thus, the DC motor plays the key role in the system. Therefore, we need to build a model for it.

The DC motors are actuators that convert the electrical energy into mechanical energy.

The parameters to be considered for our DC motor modeling are:

- Moment of inertia (J)
- Motor viscous friction constant (b)
- Electromotive force constant (Ke)
- Motor torque constant (Kt)
- Electric resistance (R)
- Electric inductance (L)
- Input (V)
- Output (Θ)

Figure 20. **DC motor: mathematical model**



The mathematical laws for the DC motor are:

- the torque of the motor is related to the armature current by a constant factor:

$$T = Kt\, i$$

- the back electromotive force relates with a constant to the angular velocity:

$$e = Ke\, \theta$$

The angular velocity represents the variation of the angle with respect to time. We combine the previous formulae with the analysis of the circuit of the figure above and we apply the Newton's law and the Kirchhoff's law.

The result is:

$$J\ddot{\theta} + b\,\dot{\theta} = K\, i$$

$$L\frac{\delta i}{\delta t} + R\, i = V - K\,\dot{\theta}$$

where we assume *Ke* = *Kt* = *K*.

To proceed with the stability analysis, consider the time domain. As this calculation is quite complex, we switch to the Laplace domain.

Using the Laplace transform, we obtain a transfer function:

$$s(Js + b)\theta(s) = K\, I(s)$$

$$(Ls + R)\, I(s) = V - K\, \theta(s)$$

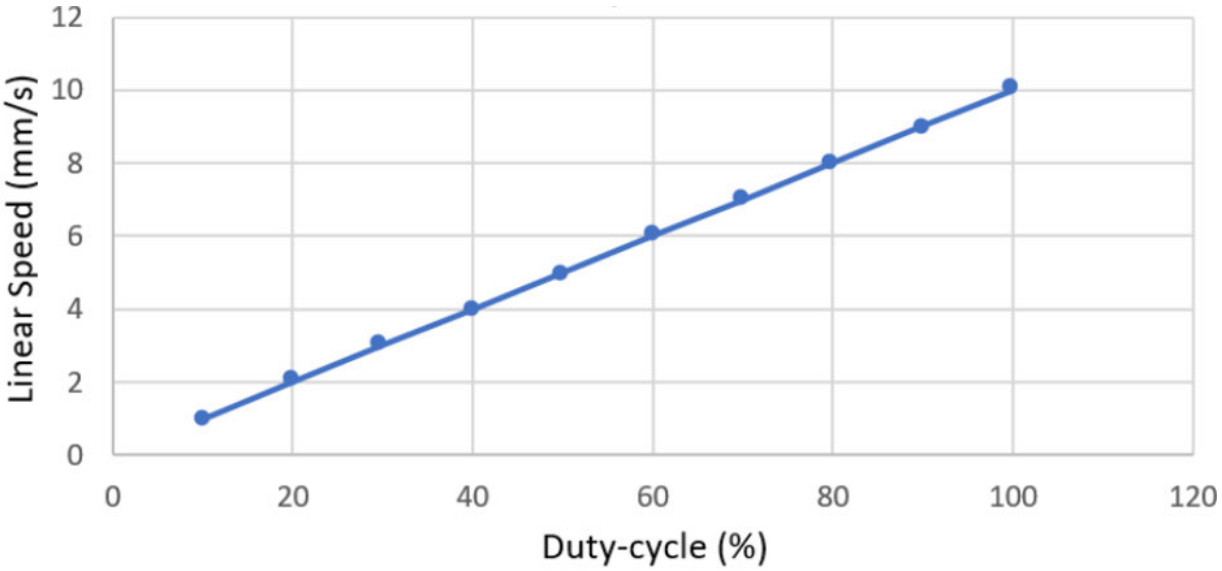
From the two equations, the resulting relationship is:

$$\frac{\dot{\theta}}{V} = \frac{K}{(J\, s + b)(L\, s + R) + K^2}$$

To apply this transfer function in our case, we need to characterize the formula parameters. Despite our characterization effort, these parameters could vary, based on the wear and tear of the system considered. Another disadvantage is that these parameters are not commonly present in the datasheet of the actuator.

Due to the previous mentioned disadvantages, we decided to proceed with a black-box identification scheme. The black box identification involves stimulating the linear actuator through the voltage input (PWM). It consequently records the outputs (actuator extension) in the look-up table. The recording is performed on the entire data range, increasing the duty cycles from 10% to 100%. From the described black box identification, we obtain the following graph.

Figure 21. DC motor look-up table



Using this method, it is possible to put typical linear actuator disturbances into the feedback chain. Indeed, the black-box model is more reliable than the white-box model since the errors are reduced.

This linear actuator model can be used for the power liftgate model. A finite state machine of the system completes the final model is completed. The same model is used for the trunk lock.

7.2 Control algorithm

The control algorithm controls, coordinates, and optimizes all the operations that the power liftgate performs (opening/closing, locking/unlocking, etc.).

The control algorithm compares the system current state to a reference (or set point). The difference between the actual and desired state is an error signal. It is applied as feedback to generate a control action, which takes the system to the desired set point.

Despite several control modeling strategies based on a formal specification, the most used high-level method is the approach based on a finite state machine (FSM). It describes a system state by specifying the value of all the system variables at certain moments or interval of times. It also defines the table of changes of system variables in the subsequent discrete states.

The transactions among the states are possible only when the given state is active. The transaction condition specified in the arc (connecting two states) is met. Furthermore, changing the state causes a change in the value of the discrete output values.

All the system variables related to a specific state of the power liftgate are defined within a certain sample time, which depends on the dynamic behavior of the controlled system.

The state of the power liftgate is described by:

- potentiometer values for both linear actuators
- trunk state (open/closed)
- lock state (unlocked/locked)

The output-discrete variables of the power liftgate are the PWM signals applied to both DC motors.

By analyzing the data collected in Figure 21. DC motor look-up table, a reasonable sampling time is 50 milliseconds. The choice of this sample time was weighted according to the dynamics of the system to be controlled (linear actuator model).

The power liftgate control model includes the following algorithms:

- external command control block
- trunk-locking control block
- high-level trunk lift control block
- fault detection control block
- obstacle detection control block
- self-calibration control block
- motion detection control block

7.2.1 External command control block

The goal of this algorithm is to manage the external commands received in relation to the trunk opening/closing/stopping.

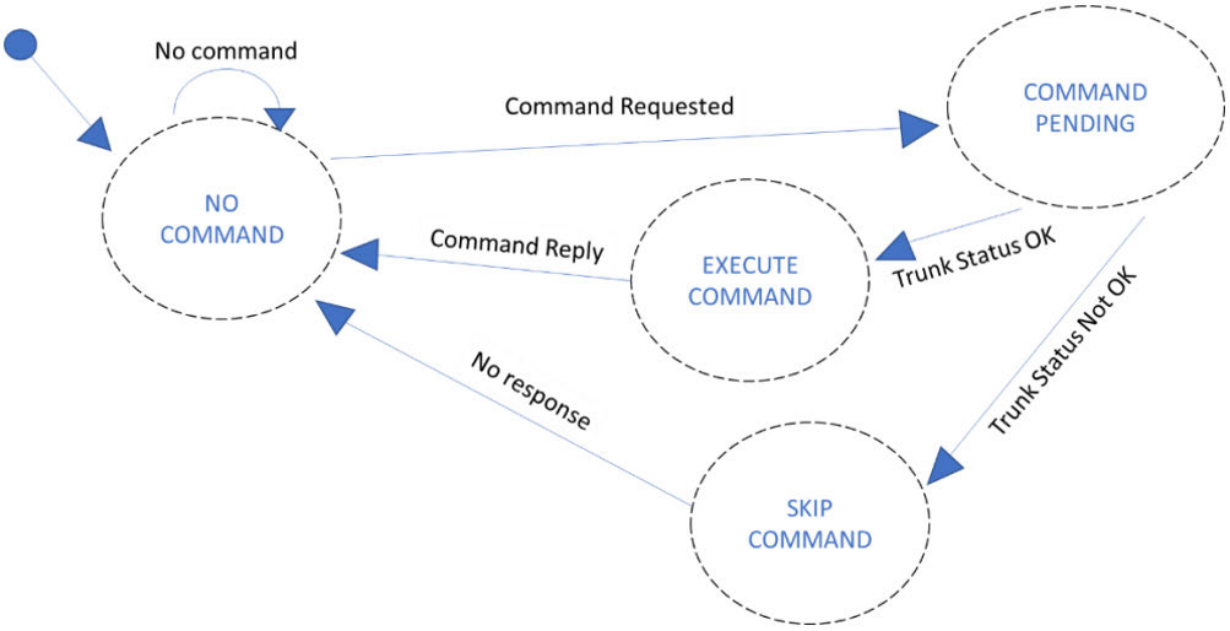
The mobile app sends these external inputs via Bluetooth® Low Energy. An NFC device or a gesture in front of the Time-of-Flight (ToF) sensors can represent external inputs, too.

The model stores each command request. It evaluates the trunk state before proceeding with the command execution. On the basis of the current state, the algorithm decides whether to skip or not the received command.

For example, if the detected state is car motion, the opening/closing commands are skipped. This mechanism prevents possible damages to the system.

Another example of command skipping is during the self-calibration and initialization procedures. Any other command cannot interrupt both procedures until their completion.

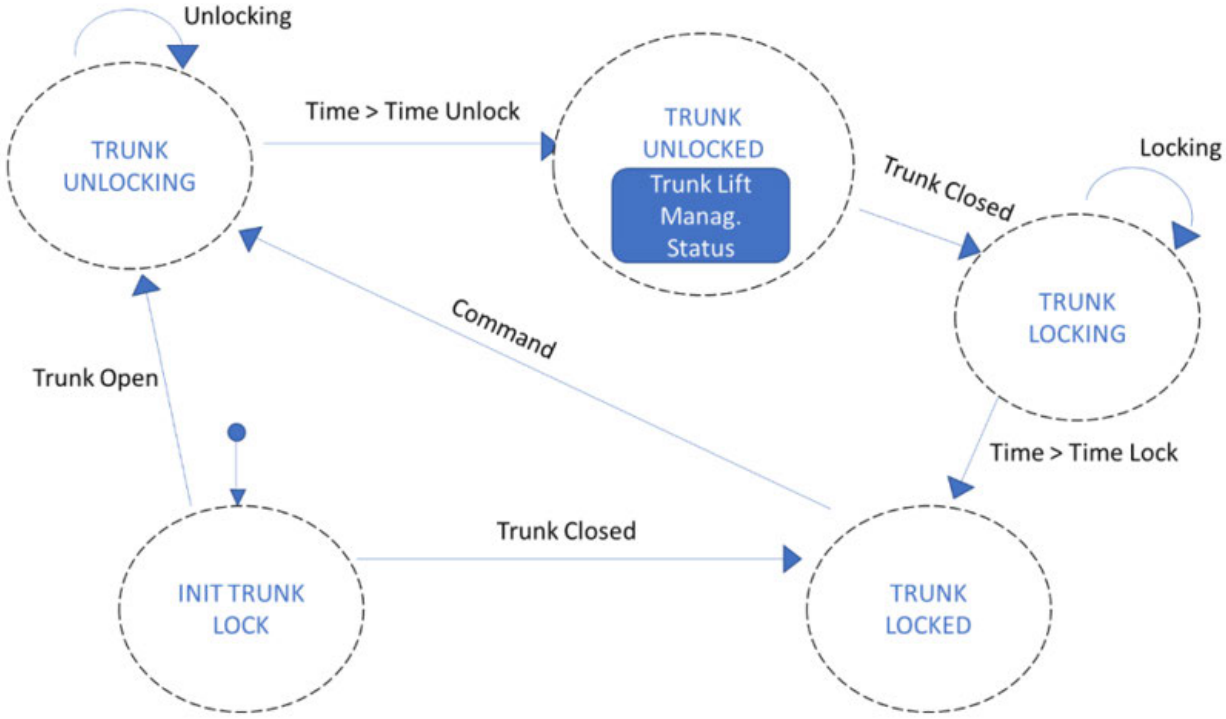
Figure 22. FSM of the external command control block



7.2.2 Trunk locking control block

The goal of this FSM is to manage the trunk locking/unlocking. Since the motor used in this model does not have feedback signals, a time-based logic is used.

Figure 23. **FSM of the trunk locking control block**



The trunk lock management has an `Init` state. It checks the lock status with respect to the trunk status.

This state aim is to take the trunk system to the trunk closed and locked state (default state). Therefore, if the trunk is closed, the FSM directly enters the trunk-locked state. Instead, if the trunk is open, the lock motor is activated to ensure it is completely unlocked.

Once completed, the FSM enters the trunk-unlocked state. At this point, the FSM waits for the trunk closed status from the trunk lift management FSM to lock the trunk. Then, it enters the trunk-locked state. After that, the FSM enters the normal operation mode.

In the trunk-locked state, the FSM is ready to receive commands for the trunk opening procedure. Indeed, when it receives the commands, the unlocking phase starts and the lock motor is activated.

Once a time interval is exceeded, it is assumed that the motor lock is open. As soon as the trunk is unlocked, the liftgate opening can begin. When the trunk is closed, the trunk-locking phase can start. When a time interval is exceeded, we assume that the motor lock is closed. At the end of this last phase, the trunk lock management awaits receiving other commands.

7.2.3 Trunk lift control block

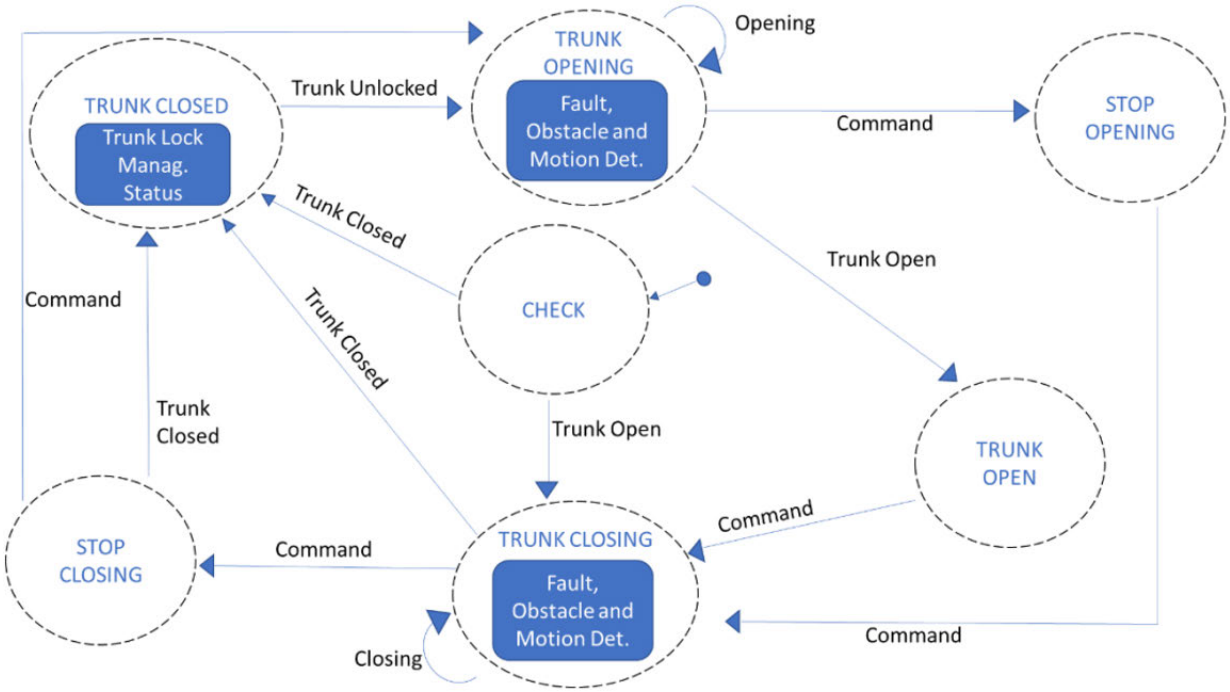
The goal of this FSM is to open, close, and stop the trunk.

In the `Init` state, the position check FSM verifies both motors position. The position feedback indicates whether the trunk was fully closed or not. In the latter case, close the trunk and to take it to the default state (trunk closed and locked).

From this state, the `Init` phase is concluded. Then, the normal operation is resumed to receive commands. Once an input command is received, the trunk is unlocked. After the trunk lock management FSM receives the trunk-locked status, the trunk opening phase starts. In this state, the two PWM signals activate the two linear actuators. The PWM duty cycle values are calculated separately, according to the position that both potentiometers have measured.

During the actuation, there is a continuous fault and object detection monitoring to ensure safety. Moreover, the vehicle motion detection is continuously monitored. If another input command is received, the trunk is stopped. Otherwise, the opening phase continues until the trunk reaches the maximum opening threshold. In both cases, the next command starts the closing phase.

Figure 24. **FSM of the trunk lift control block**



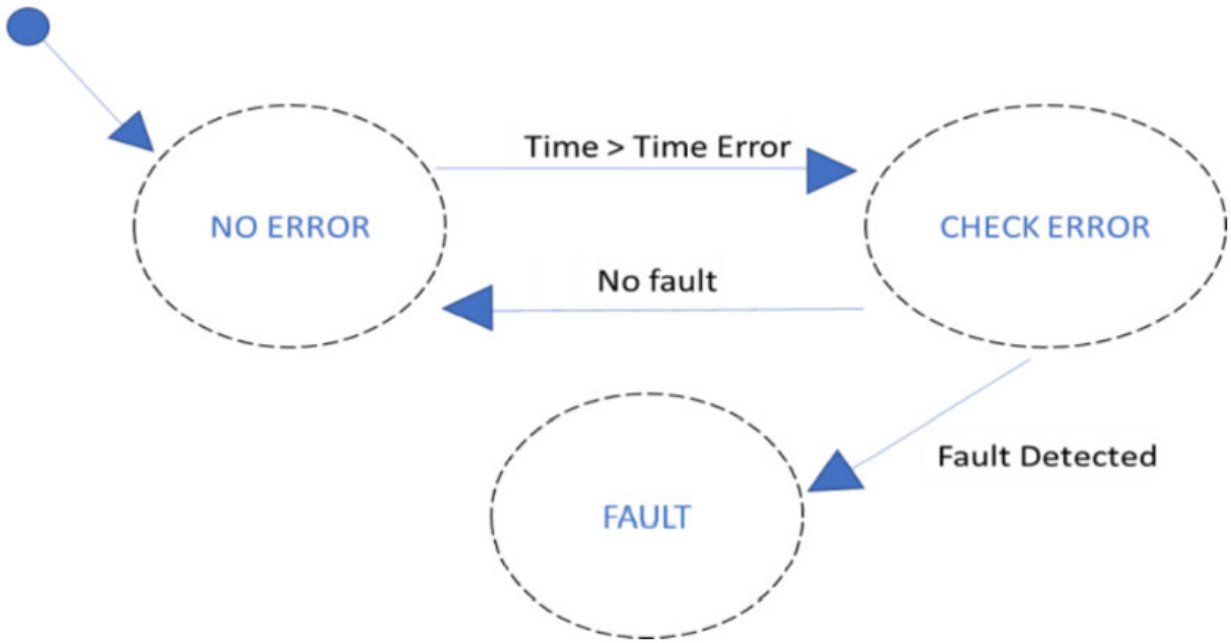
7.2.4 Fault detection control block

The fault detection algorithm detects possible failures of the linear actuators. The possible fault conditions are:

- no displacement variation of the linear actuators
- the gap between the feedback of the two potentiometers is too wide

If these conditions are verified, the algorithm blocks the whole system to prevent any damage.

Figure 25. **FSM of the fault detection control block**



The algorithm starts when the opening/closing procedures are activated.

The first state stores the current positions of the two linear actuators. It starts counting a fixed-time interval. This interval is calculated according to Figure 21. DC motor look-up table.

When exceeding the fixed-time interval, the FSM enters the check error state. In this state, the algorithm compares the current positions of the two linear actuators with the ones previously stored. If the difference is less than a given threshold, a fault is detected. Then the FSM enters the fault state.

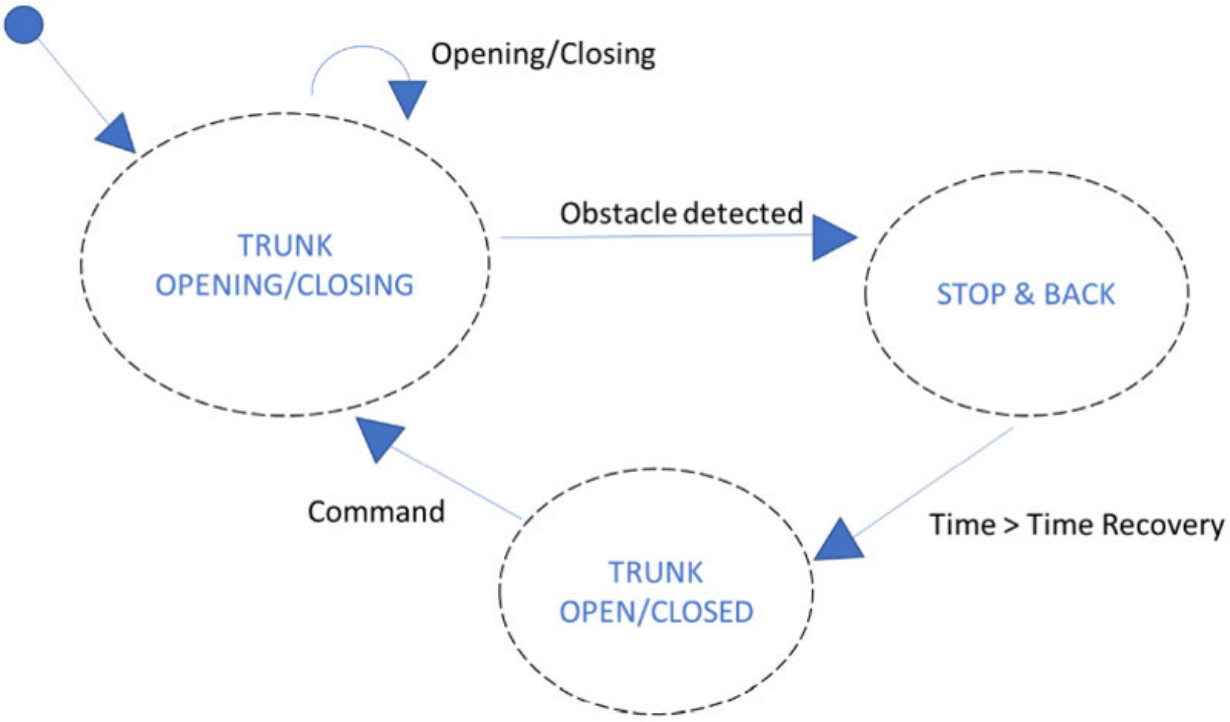
In this state, both linear actuators are stopped while visual and acoustic alerts are emitted. Otherwise, the algorithm assumes that there is no error and returns to no error state.

You can simulate examples of failure conditions by using the manual fault injection mechanism.

7.2.5 Obstacle detection control block

The goal of this algorithm is to block the trunk in case of jam detection. To be safe, the system stops the linear actuators when obstacles are detected while opening/closing the trunk.

Figure 26. FSM of the obstacle detection control block



During the opening/closing phase, a possible obstacle detection is monitored on both linear actuators.

The position check FSM identifies two zones:

- critical zone - the obstacle detection algorithm is deactivated. This facilitates the trunk closing procedure (this zone is few centimeters)
- noncritical zone - the obstacle detection is enabled. When a jam is detected, the two linear actuators are stopped. Their direction is reversed with a slow speed in order to step away from the contact point

For safety reasons, after the opening command, if obstacles are detected, the next command executes the closing procedure.

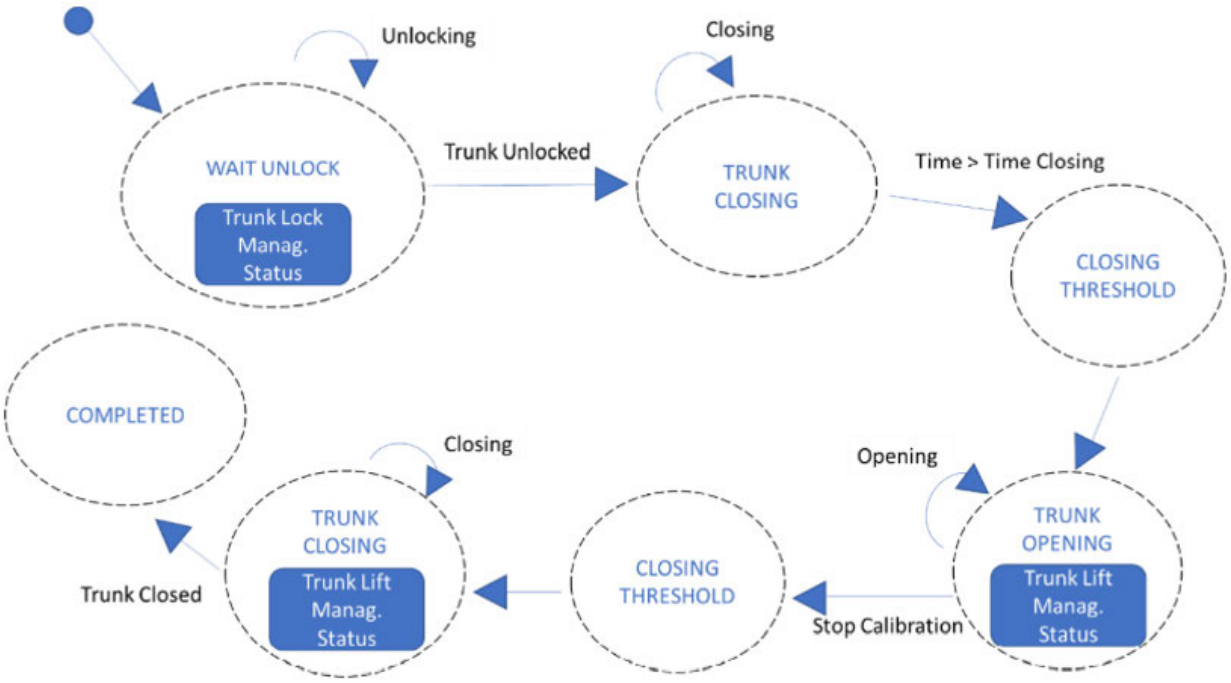
7.2.6 Self-calibration control block

The main goal of this algorithm is to calibrate the two linear actuators. As the two linear actuators consist of mechanical parts, the tear and wear effect might affect them. They could start skipping some steps. This condition could impact the correct trunk positioning and compromise the trunk hinges.

You can define a trunk opening threshold via a specific command through the mobile app.

As an alternative, this algorithm can tune the maximum height of the trunk opening. The self-calibration feature is executed only when the trunk is closed. In other states, the self-calibration command of the mobile app is disabled.

Figure 27. **FSM of the self-calibration control block**



The self-calibration algorithm starts after another FSM sets the trunk-unlocked status.

The first step consists in closing the trunk. This phase is executed for a fixed time interval to ensure that the trunk is correctly closed. The time interval is based on the slowest closing time of the linear actuators. This allows aligning the two linear actuators. At this point, the closing thresholds are stored in the microcontroller flash memory.

In the next step, the trunk opening phase starts until one of the two linear actuators reaches the maximum extension. This is the calibration stop point, which sets the new opening thresholds to store in the microcontroller flash memory.

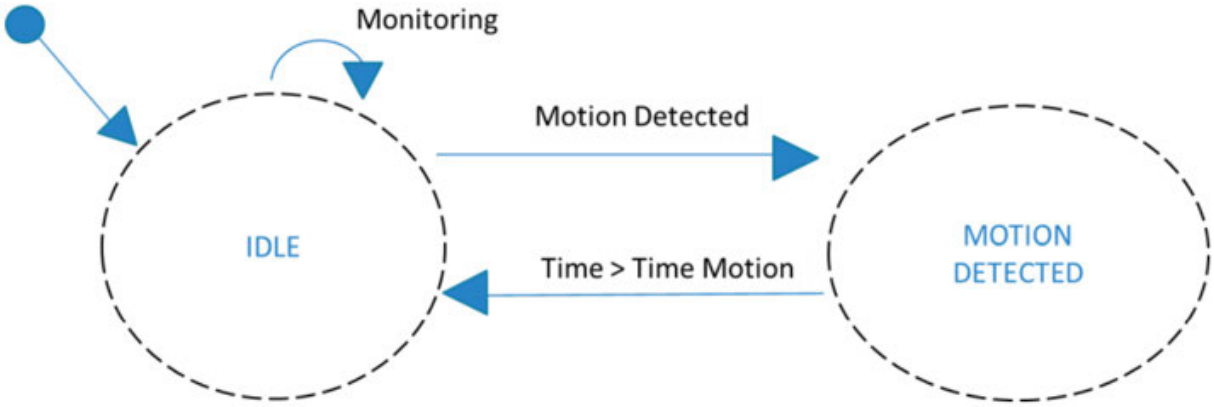
Another way of stopping the calibration phase through user-defined thresholds during the opening phase. These thresholds are used for the future power liftgate operations, until another self-calibration phase starts. Before restarting a command-driven operation, the trunk is closed to return to its default state.

7.2.7 Motion detection control block

The goal of this algorithm is to disable the opening/closing of the trunk during the car motion, either with the engine on or off.

This feature prevents possible damages to the car and the surrounding environment. The motion detection is a signal coming from a motion MEMS sensor.

Figure 28. **FSM of the motion detection control block**



In the default state, the system checks if a movement is detected. For safety reasons, when a movement is detected, the opening/closing phases are not allowed. Thus, the trunk-opening/closing input commands are skipped.

Once the motion detection is asserted, the actuation is prevented for a fixed interval of time. When the interval expires, the motion status is checked again. If the motion continues, the actuation remains disable. Otherwise, the status is cleared and the trunk operation can restart.

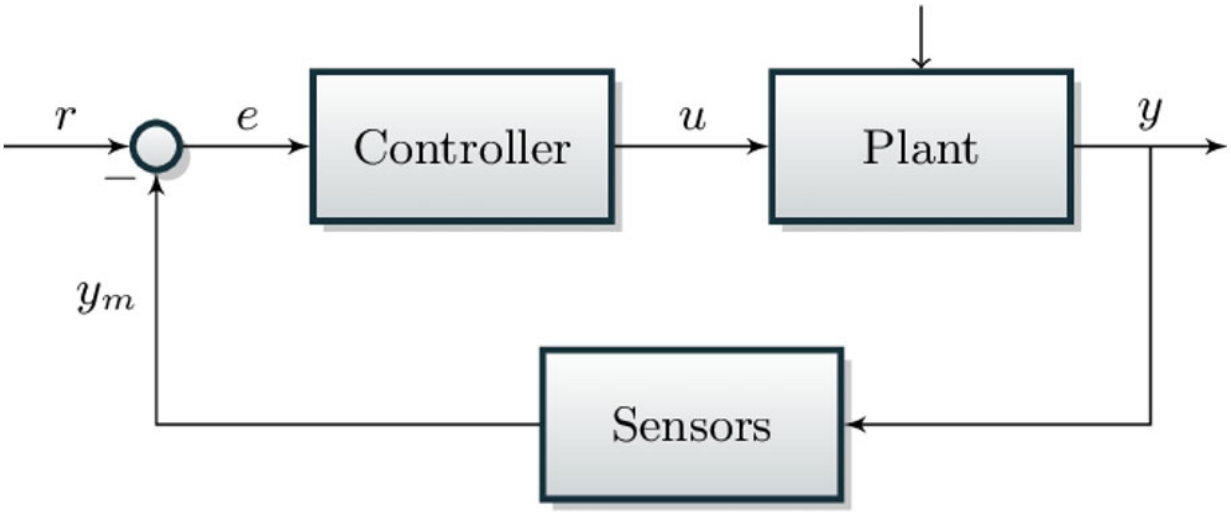
If the motion is detected while the trunk is in operation, a similar behavior occurs.

8 Model-in-the-loop (MIL) validation

In the model-in-the-loop (MIL) validation approach, the control algorithm model is linked to the plant in a closed loop.

This technique simulates a complete environment to test the control laws, correct mechanics, electronics. It also spots the mistakes about the model and requirements before prototyping the development phase.

Figure 29. Closed loop system



The reference signal (`r`) is the desired power liftgate state (locked/unlocked or open/closed). The `Sensors` block represents the linear actuators potentiometers that measure the angular position of the trunk (`Ym`). The `Controller`, implemented with the FSM modeling, generates actuations to drive both DC motors according to the error, which is calculated as the difference between `Ym` and `r`.

The MIL is a method for creating, managing, and testing the power liftgate system by using a non-intrusive test harness to evaluate the system coverage and the software requirements.

9 Code generation

The model-based code generation is one of the main advantages of the model-based development. Through the MathWorks toolchain, C/C++ code is generated.

The automatic code generation can be used to:

- accelerate the model execution compared to the MATLAB® interpreted environment
- create standalone executables for the actual hardware
- incorporate the legacy C/C++ code into the models
- create all the components of the application layers of the open software architecture

10 Software-in-the-loop (SIL) validation

The SIL validation describes a test methodology. The executable code, such as a control algorithm already generated in C, is tested within a modeling environment that can help to prove or test the software.

The generated code is encapsulated in a Simulink® function that represents the control algorithm implemented in a C function.

The SIL testing is performed by using the same test suite created in the MIL validation process.

If behavioral differences are spotted, the C code generation has to be relaunched with a different set of parameters.

11 Basic software integration

The generated code consists of:

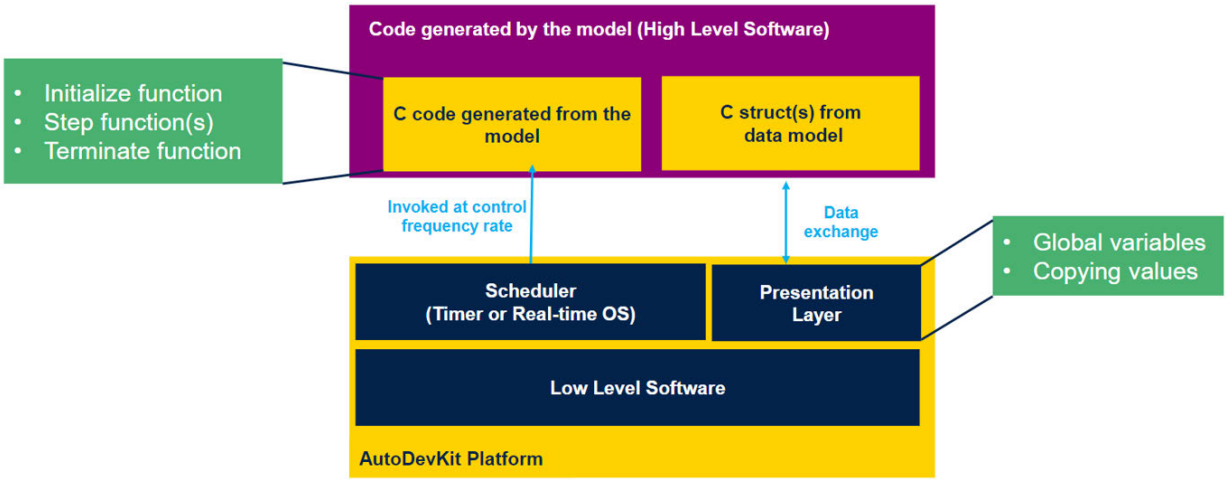
- an `Init` function, which contains the initialization code
- a `Step` function, which represents the entire control algorithm

The `Step` function execution is scheduled according to the sample time chosen in the identification phase of the plant. It is based on the number of ticks (related to the MCU system clock). In our case, the application layer task is invoked every 50 milliseconds.

The model C code generated must be integrated with the basic software described in the AEK-MOT-TK200G1 drivers.

To ensure a proper data exchange between the application layer and the basic software, a specific structure is generated from the model and imported in the code. The data structure that represents the presentation layer is generated as global external variables. Thus, their visibility is extended to the entire code.

Figure 30. Basic software integration



- The variables included in the data structure are:
  - Input variables:
    - obstacle detection right/left
    - command/self-calibration request
    - position left/position right
  - Output variables:
    - PWM left/PWM right/PWM lock
    - trunk direction/lock direction

The obstacle detection signals are obtained through the current sensing on the linear actuators. The analog/digital converter (ADC) reads the current value and compares it with a predefined threshold.

If the value read is higher than the threshold, the value ‘1’ is assigned to the obstacle detection variable and passed to the model. Otherwise, ‘0’ is passed.

Figure 31. **Obstacle detection code**

```
if(currentSensingLeft > THRESHOLD_SENSING){
    objectDetectionLeft = 1;
}
else {
    objectDetectionLeft = 0;
}
```

The variables of the command and self-calibration requests represent the commands that a domain controller receives via the CAN protocol. The commands are sent to manage the trunk opening/closing and to execute the self-calibration mechanism.

In the initialization phase, if there are no opening and closing stored thresholds, the model uses its default values. Otherwise, the model reads the stored thresholds from the EEPROM.

Figure 32. **Command and self-calibration code**

```
void mcanconf_rxreceive(uint32_t msgbuf, CANRxFrame crfp) {
    (void) msgbuf;
    if (crfp.ID == TRUNK_RECEIVE) {
        if (crfp.data32[0] == TRUNK_REQUEST)
        {
            cmdInput = 1;

        } else if (crfp.data32[0] == TRUNK_CALIBRATION)
        {
            selfLearning = 1;
        } else {}
    }
    if (crfp.ID == SID_VEHICLE_SPEED) {
        vehicleSpeed = crfp.data32[0];
    }
}
```

The EEPROM stored values are calculated during the self-calibration.

Figure 33. **EEPROM code**

```
if (liftGateStatus == LIFT_CALIBRATION_COMPLETED) {
    buffer1[0].thresholdPositionOpenLeft = thresholdOpenLeft_out;
    buffer1[0].thresholdPositionOpenRight = thresholdOpenRight_out;
    buffer1[0].thresholdPositionCloseLeft = thresholdCloseLeft_out;
    buffer1[0].thresholdPositionCloseRight = thresholdCloseRight_out;
    returnvalue = eeprom_write(&EEPROMD, 1,
            (uint16_t) (sizeof(buffer1[0])), (uint32_t) buffer1);
}
```

The position feedback of the linear actuators corresponds to the values from the ADC connected to the potentiometer. The ADC voltage value is converted to a linear position. The model reads the actuator linear positions. It then generates the specific PWM duty-cycle as well as the trunk direction to open/close the trunk through the linear actuators and lock the motor.

Figure 34. **Position feedback code**

```
if (liftGateStatus == LIFT_CALIBRATION_COMPLETED) {
    buffer1[0].thresholdPositionOpenLeft = thresholdOpenLeft_out;
    buffer1[0].thresholdPositionOpenRight = thresholdOpenRight_out;
    buffer1[0].thresholdPositionCloseLeft = thresholdCloseLeft_out;
    buffer1[0].thresholdPositionCloseRight = thresholdCloseRight_out;
    returnvalue = eeprom_write(&EEPROMD, 1,
            (uint16_t) (sizeof(buffer1[0])), (uint32_t) buffer1);
}
```

11.1 ADC filtering function

The analog-to-digital converter (ADC) converts the analog signals into digital ones to allow microcontroller processing. Since the system noise affects the input analog signals, the input signal processing is implemented through a digital filter.

In our system, the ADC is connected to a Hall sensor potentiometer, which is included in the linear actuator.

This potentiometer is used to compute the linear actuator positions. The goal of this digital filtering algorithm is to remove possible spikes in the input signal.

Figure 35. **Digital filtering code**

```
float readADCLeft(void) {
    uint8_t i, j;
    float tmp, tmpAverage = 0;
    for (i = 0; i < N_ADC_SAMPLES; i++) {
        saradc_lld_start_conversion(sarAdc);
        adcValuesRawLeft[i] = adcValueRawLeft;
        saradc_lld_stop_conversion(sarAdc);
    }

    //sort
    for (i = 0; i < N_ADC_SAMPLES - 1; i++) {
        for (j = i + 1; j < N_ADC_SAMPLES; j++) {
            if (adcValuesRawLeft[i] > adcValuesRawLeft[j]) {
                tmp = adcValuesRawLeft[j];
                adcValuesRawLeft[j] = adcValuesRawLeft[i];
                adcValuesRawLeft[i] = tmp;
            }
            else {}
        }
    }

    //Average less MIN and MAX value
    for (i = LOSS; i < N_ADC_SAMPLES - LOSS; i++) {
        tmpAverage += adcValuesRawLeft[i];
    }
    adcAverageLeft = tmpAverage / (N_ADC_SAMPLES - 10);
    AEK_MOT_TK200G1_CheckWDExpired();
    return adcAverageLeft;
}
```

The first step of the algorithm consists in activating the ADC. Each time a value is converted in the ADC, a specific interrupt is generated. Our digital filter performs the ADC acquisition *n* times and stores the value in a dedicated array (N_ADC_SAMPLES).

The array is then sorted in an ascending value order. The resulting highest and the lowest values are discarded. The number of discarded values depends on the value set by the loss parameter definition written in the source code.

The resulting reduced array is then used to compute the average value. Computing the average value on the reduced array leads to an increase in the sample accuracy compared with an average value computed on the original array.

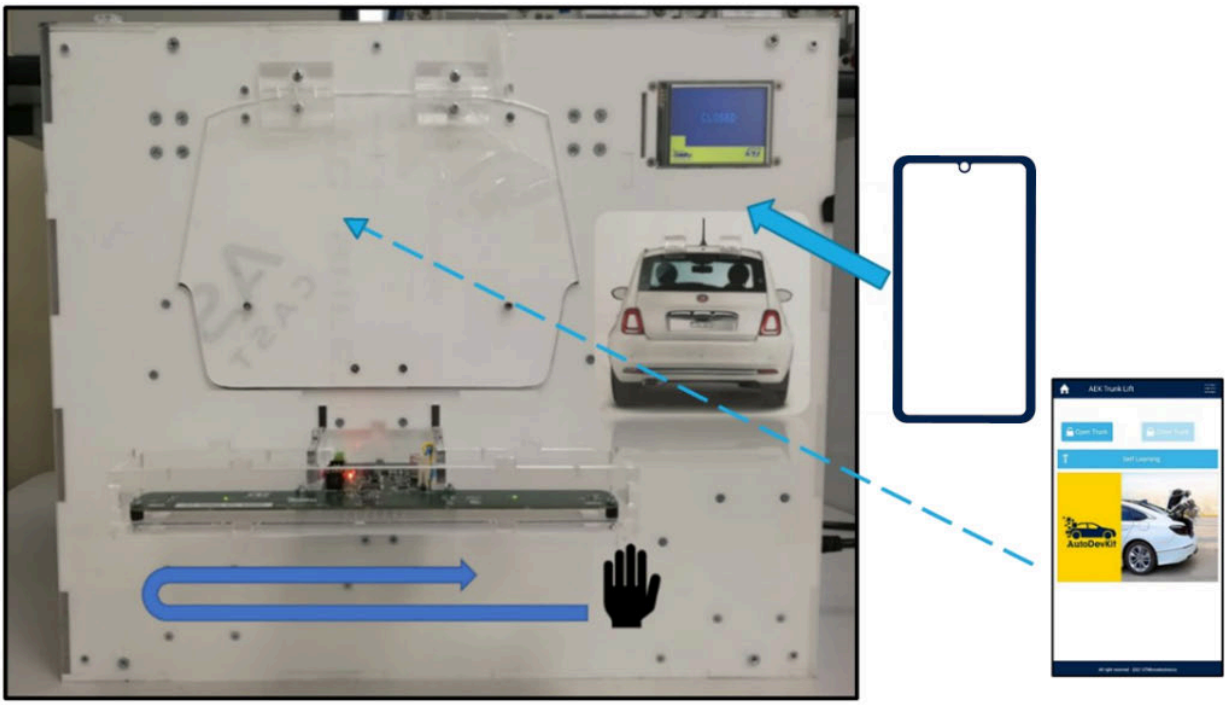
12 System validation

All the features of the power liftgate are now validated with the system mock-up.

At power-up, the power liftgate waits until an input command is received. Three different input commands can be used to open/close the trunk:

- Foot gesture detection - using the AEK-SNS-2TOFM1, you can open/close the trunk by performing a gesture with the correct sequence. The foot detection has to happen within the range of a specific time interval. For example, if it is too fast or too slow, the sequence is not recognized.

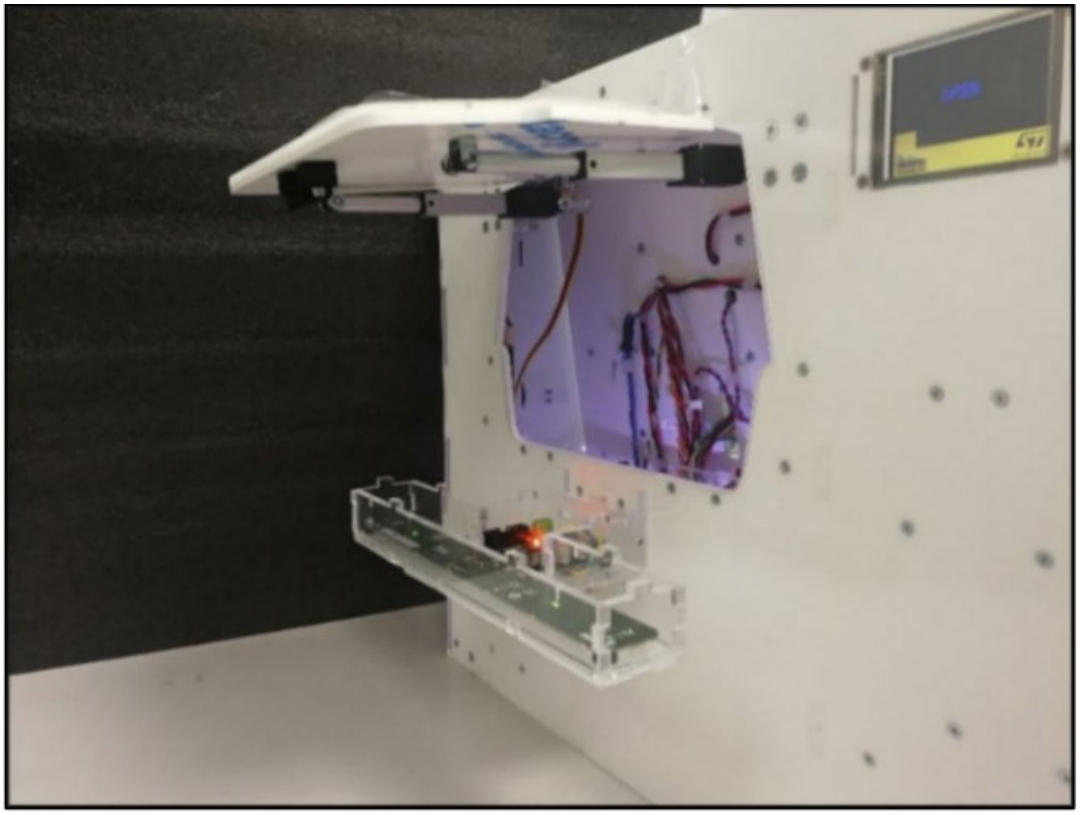
Figure 36. Trunk opening options: gesture, app, NFC



- Smartphone application - to interact with the system via Bluetooth® Low Energy (AEK-COM-BLEV1) to manage the actuations.
- NFC key - to allow a contactless tailgate opening/closing when the key is close to the power lift gate.

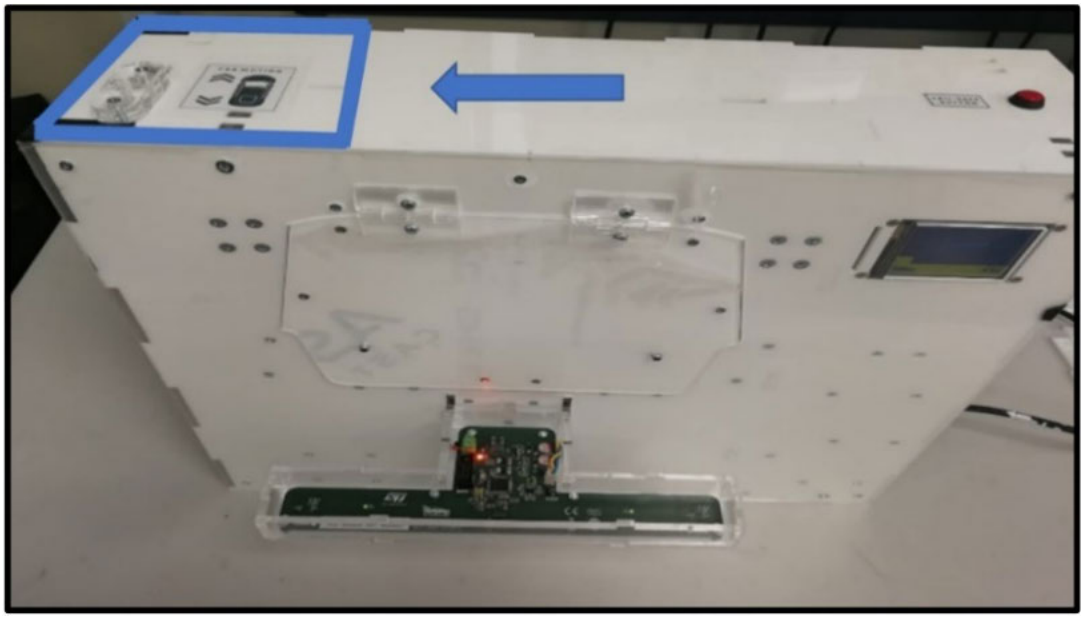
After the trunk has received the input commands, the acoustic and visual signals signal are presented and the actuation starts (opening/closing). For example, in the trunk opening phase, the power liftgate opens the trunk up to the opening thresholds. In the meantime, if another input command is received, the power liftgate stops the trunk opening. Through the same input commands, you can close the power liftgate.

Figure 37. **Power lift gate fully open**



A box that contains the motion MEMS sensor detects accelerations. This box emulates a moving car to test that the trunk actuations are disabled for safety reasons.

Figure 38. **Vehicle motion emulation**



The object detection feature has been implemented to add a safety mechanism. In fact, during the opening/closing phase, if the trunk detects an obstacle, the system stops the movement of the linear actuators and reverses the direction for a few centimeters. The next received command performs a reverse operation.

Figure 39. **Obstacle detection**

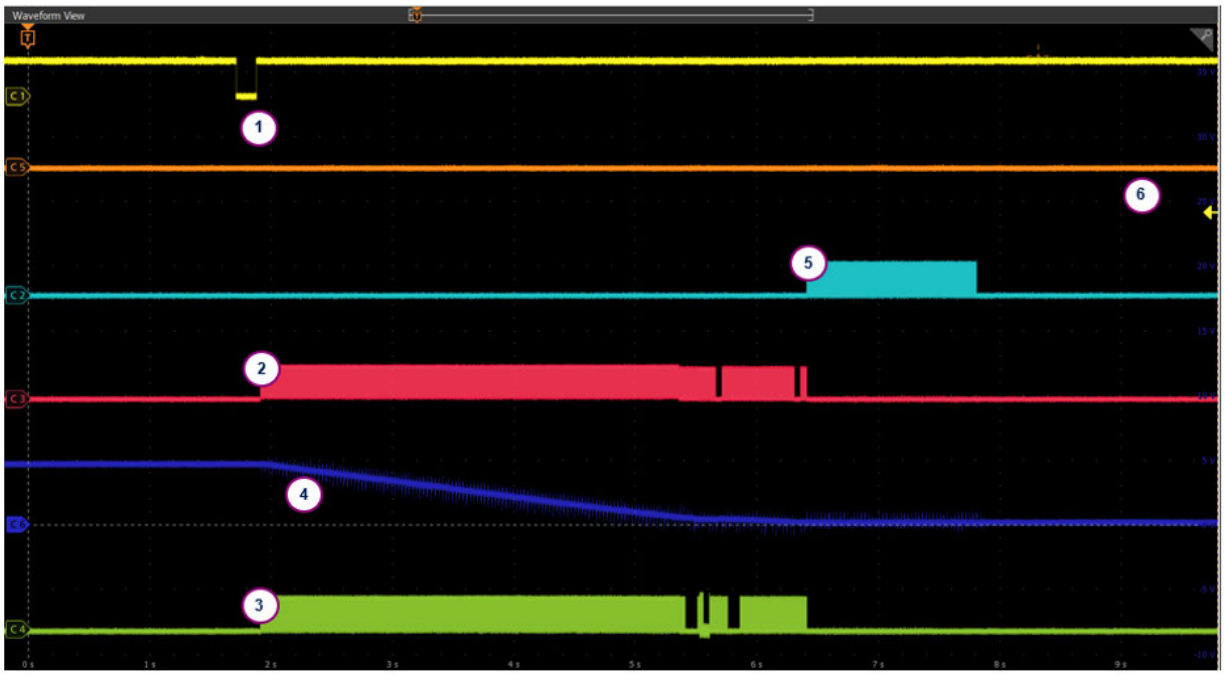
13 Power lifgate waveforms

13.1 Closing waveform

The following figure shows the waveform of the system signals in case of trunk closing via the fail-safe button.

Figure 40. Trunk closing via the fail-safe button

1. Interrupt when the fail-safe button is pushed
2. PWM signal applied to the linear actuator of the right-hand side
3. PWM signal applied to the linear actuator of the left-hand side
4. Right (or left) potentiometer signal related to the trunk positioning during the closure
5. PWM signals applied to the linear actuator during locking
6. PWM signals applied to the linear actuator in case of unlocking

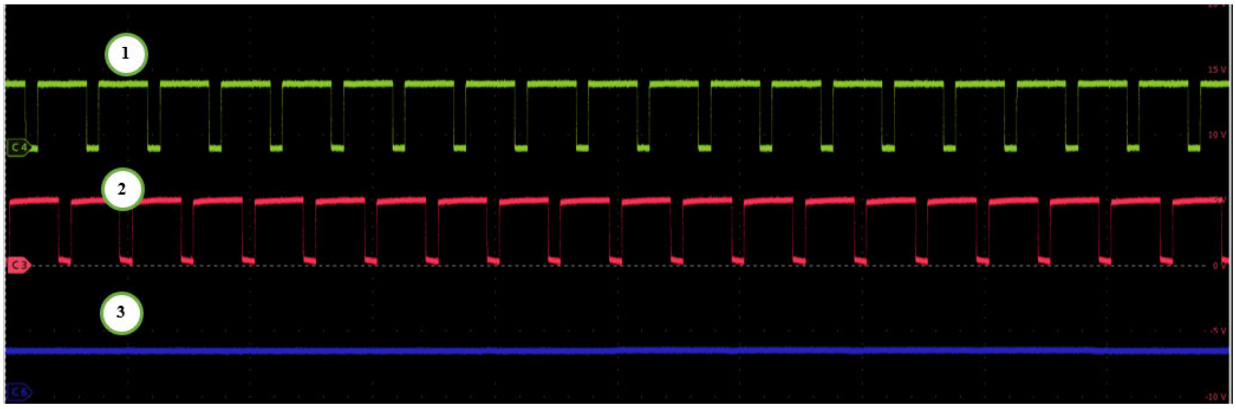


When the trunk is closed unlocking, the PWM duty cycle is equal to zero.

When the trunk is almost completely closed, the control algorithm manages the PWM signals to align perfectly with the box.

The figure below shows a zoomed view of the PWM signals applied to the right (1) and left (2) linear actuators during the closure. It also shows a zoomed view of the right (or left) potentiometer signal (3) related to the trunk positioning.

Figure 41. **Zoom of the PWM signals**

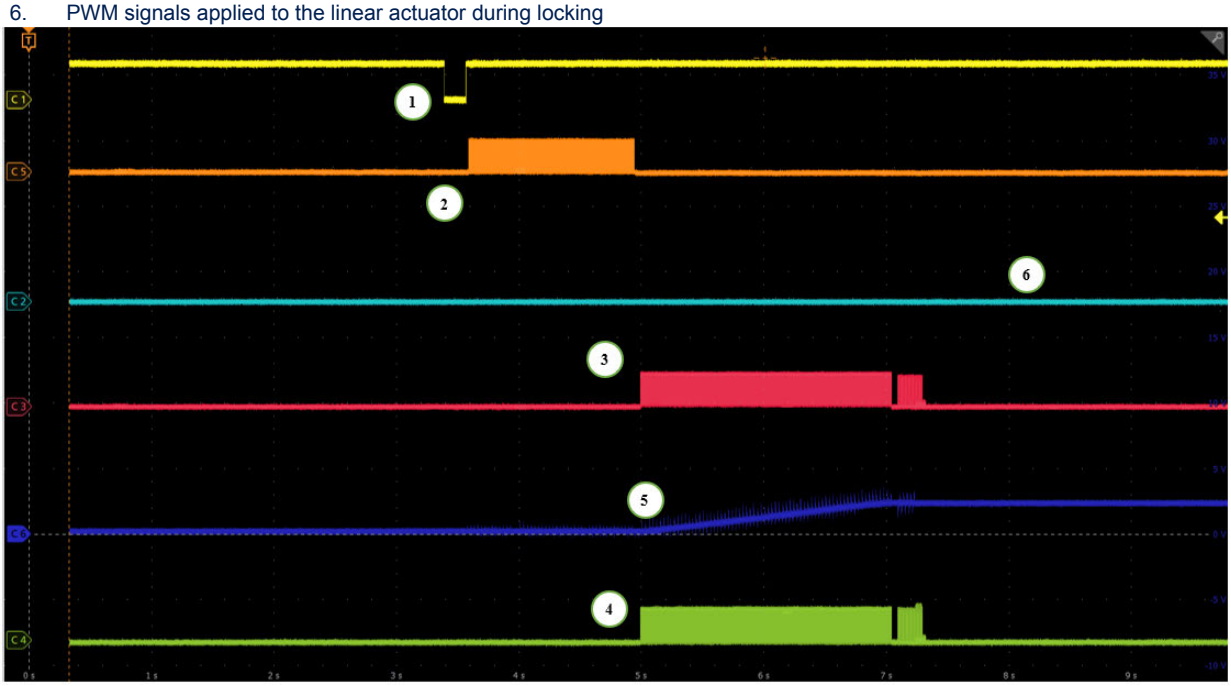


13.2 Opening waveform

The following figure shows the waveform of the system signals in case of trunk opening via the fail-safe button.

Figure 42. **Trunk opening via the fail-safe button**

1. Interrupt when the fail-safe button is pushed
2. PWM signals applied to the linear actuator during unlocking
3. PWM signal applied to the linear actuator of the right-hand side
4. PWM signal applied to the linear actuator of the left-hand side
5. Right (or left) potentiometer signal related to the trunk positioning during the closure
6. PWM signals applied to the linear actuator during locking

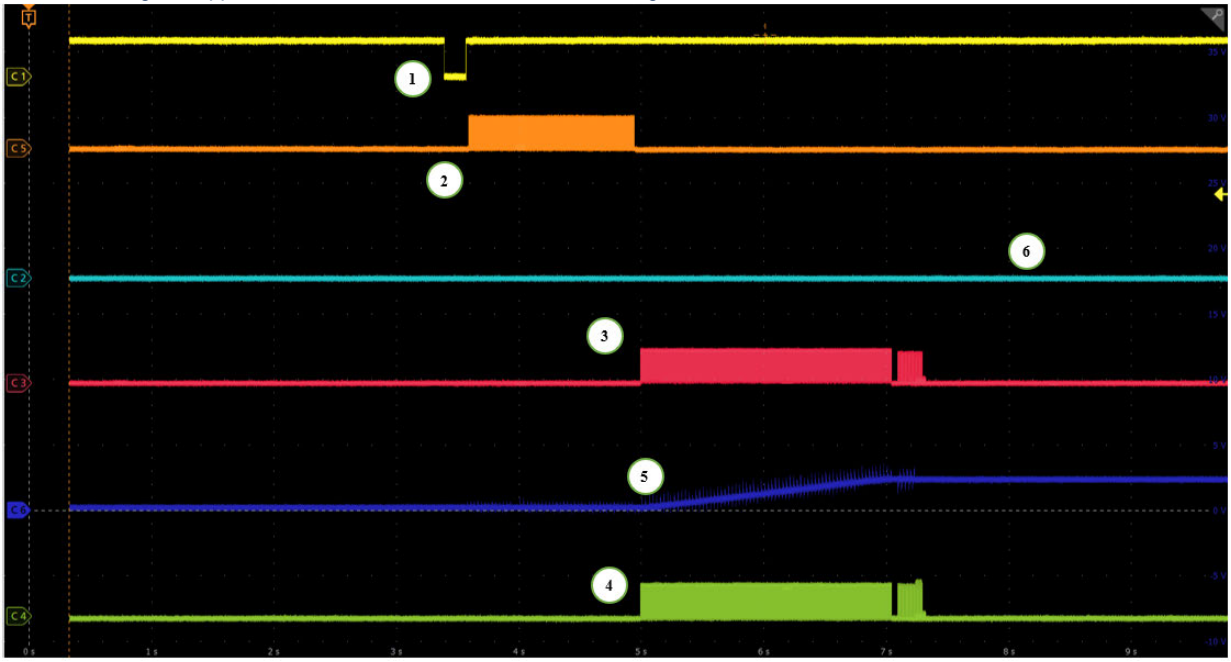


13.3 Object detection during the closing waveform

The following figure shows the waveform of the system signals when an object is detected during the trunk closing.

Figure 43. **Object detection**

1. Interrupt when the fail-safe button is pushed
2. PWM signal applied to the linear actuator of the right-hand side
3. PWM signal applied to the linear actuator of the left-hand side
4. Right (or left) potentiometer signal related to the trunk positioning during the closure
5. PWM signals applied to the linear actuator during locking
6. PWM signals applied to the linear actuator in case of unlocking



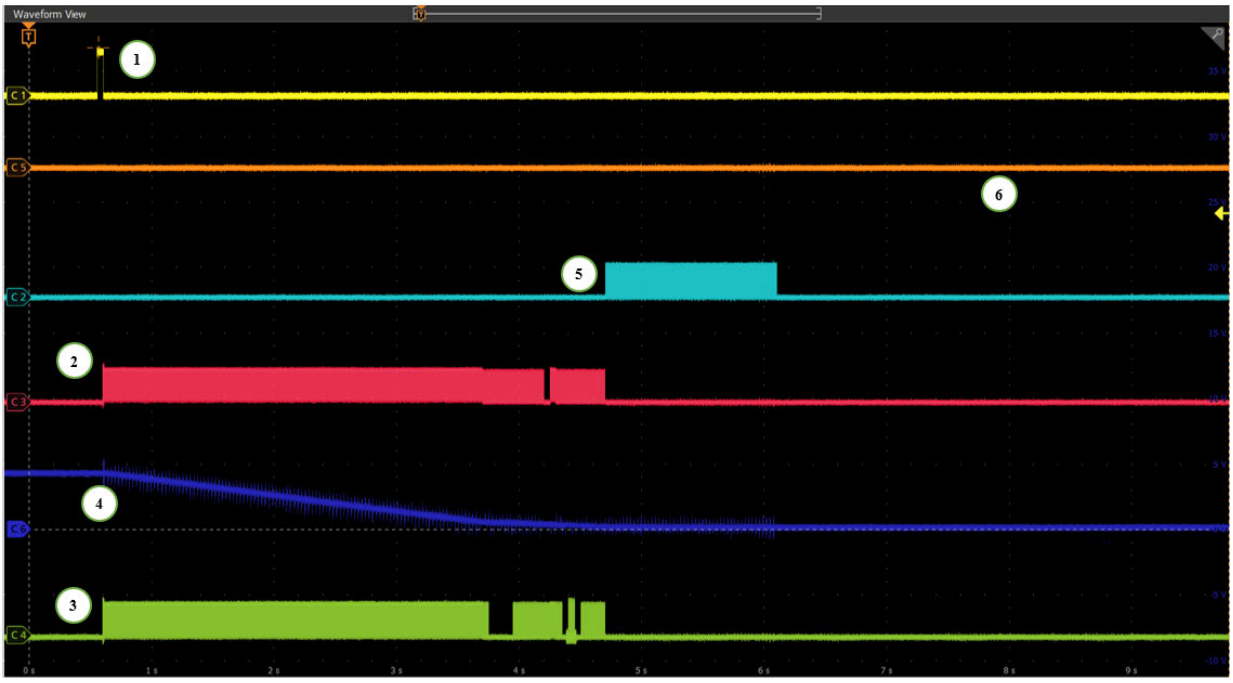
Due to the detected object during the closing, the position of the linear actuators does not reach the set point. Thus, the PWM duty cycle related to the locking is equal to zero.

13.4 NFC closing waveform

The following figure shows the waveform of the system signals in case of trunk closing via NFC.

Figure 44. Trunk closing via NFC

1. NFC interrupt signal
2. PWM signal applied to the linear actuator of the right-hand side
3. PWM signal applied to the linear actuator of the left-hand side
4. Right (or left) potentiometer signal related to the trunk positioning during the closure
5. PWM signals applied to the linear actuator during locking
6. PWM signals applied to the linear actuator in case of unlocking



Appendix A NFC technology with AutoDevKit and X-NUCLEO-NFC06A1

The near-field communication (NFC) technology is spreading in the automotive and transportation domain. Today many car makers are introducing this technology for several vehicle applications. For example:

- **Access and start**:
  - to simplify accessing the car
  - to secure the engine start
  - to ease the cloud-based distribution of virtual car keys to NFC-enabled phones
- **Pairing, setup, and safety**:
  - Bluetooth® Low Energy pairing with the car using the NFC stored data
  - automatic driver identification
  - personal settings configuration (seat, temperature, etc.)
  - NFC credit card secured payments
- **Consumable and diagnostics**: the NFC tags enable vehicle consumable authentication (air filter, oil filter, etc.) and vehicle diagnostic without a physical connection (ODB)

Figure 45. Automotive applications for NFC

The AutoDevKit ecosystem has been extended with the AEK-COM-NFC06A1 NFC component to support the above-mentioned application prototypes.

Note: The AEK-COM-NFC06A1 component has been created using the libraries coming from the X-CUBE-NFC6. The STM32 hardware abstraction layer (HAL) of the library has been modified to make it compatible with the microcontrollers that belongs to the SPC58 Chorus family. The main interface used is the SPI protocol.

A.1 X-NUCLEO-NFC06A1 board overview

The X-NUCLEO-NFC06 is an STM32 Nucleo shield based on the ST25R3916 high-performance universal HF/NFC and EMVCo front-end IC, AEC-Q100 automotive qualified (-40°C to 105°C).

The ST25R3916 directly controls the oscillator, the counters, and the status registers. An external MCU can connect via SPI or I²C to modify the device settings.

The board supports a wide range of NFC and HF RFID standards such as:

- NFC forum-compliant NFC universal device
- EMVCo 3.0 compliant contactless payment terminal
- ISO14443 and ISO15693 compliant general-purpose NFC device
- FeliCa™ reader/writer
- Support of all five NFC forum tag types in the reader mode
- Support of all the common proprietary protocols, such as Kovio, CTS, B'

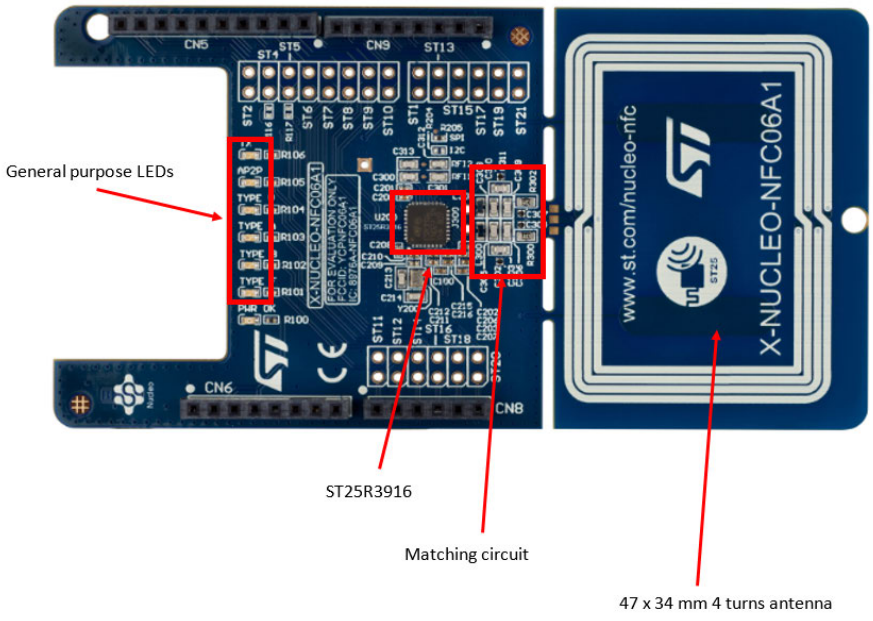
The key board features are:

- Low-power wake-up
- Excellent P2P interoperability
- EMVCo 3.0 certification without any external power amplifier
- Automatic antenna tuning
- Active wave shaping
- 1.6 W output power
- NFC forum universal device (with CE mode)

The low-power capacitive sensor detects the presence of a card without switching on the reader field.

Moreover, measuring the amplitude or phase of the antenna signal can detect the presence of a card. The automatic antenna tuning (AAT) technology enables the operation close to the metallic parts and/or in changing environments.

Figure 46. X-NUCLEO-NFC06A1 component placement



A.2 X-NUCLEO-NFC06 in the AutoDevKit ecosystem

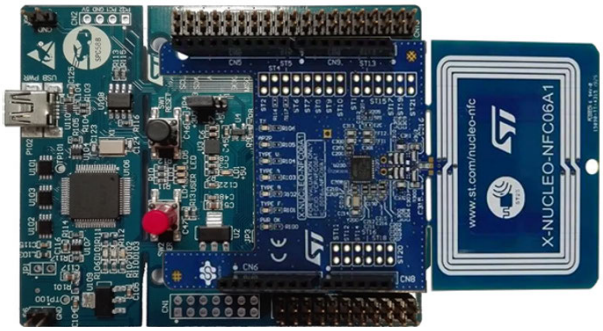
The X-NUCLEO-NFC06 has been added into the AutoDevKit ecosystem as is. It features an Arduino U3 connector that can be plugged easily onto the SPC582B-DIS or SPC584B-DIS automotive MCU evaluation boards.

A.2.1 Hardware connection

To use the X-NUCLEO-NFC06 with the SPC582B-DIS, follow the steps below.

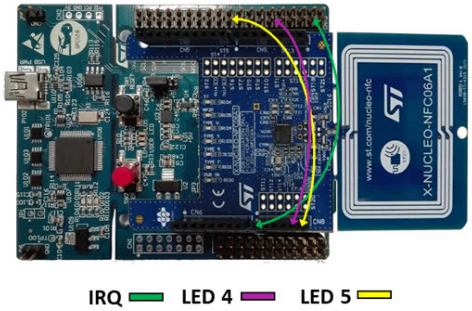
Step 1. Plug the X-NUCLEO-NFC06 on the SPC582B-DIS using the U3 connector.

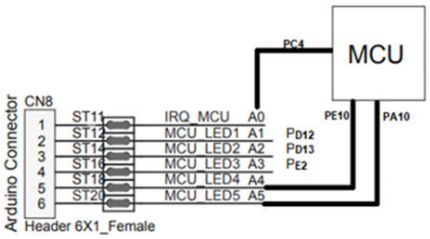
Figure 47. X-NUCLEO-NFC06A1 and SPC582B-DIS connection



Step 2. Wire the IRQ, LED4, and LED5 pins as shown below.

Figure 48. Wire connection





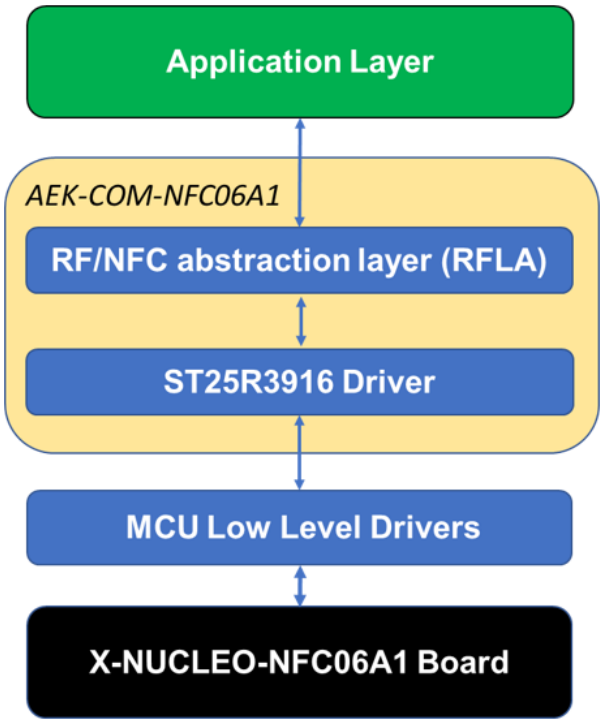
Connection Table

X-NUCLEO-NFC06 connector	Pin	SPC582B-DIS connector	Pin
CN8	1	CN10	35
CN8	5	CN10	27
CN8	6	CN10	19

A.2.2 AutoDevKit software library for the X-NUCLEO-NFC06A1

The AutoDevKit library (STSW-AUTODEVKIT), from version 1.7.0, supports the AEK-COM-NFC06A1 component. This component, written in C, manages the X-NUCLEO-NFC06 when used with the automotive microcontrollers that belong to the SPC58 family. It enhances reuse and simplifies the maintenance.

Figure 49. **Software stack**



A.2.2.1 RF/NFC abstraction layer (RFLA)

The STSW-ST25RFAL002 is the implementation of the RF/NFC abstraction layer (RFAL).

The RFAL simplifies the development of applications by embedding the device and protocol details.

Note: For further information about the RF/NFC abstraction layer, see UM2890.

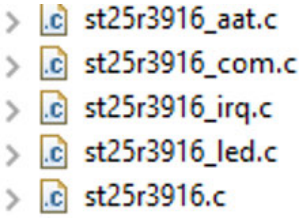
A.2.2.2 ST25R3916 driver

In the AutoDevKit, the driver used to manage all the ST25R3916 features is the same as the one developed for the STM32 ODE.

The X-CUBE-NFC6 driver has been reused by changing only the low-level driver API, that is, only the SPI protocol has been changed.

In the st25r3916_com.c file, we have replaced the `platformSpiTxRx(NULL, rxBuf, rxLen); //` function call (compatible with STM32 ODE) with `spi_lld_exchange(AEK_NFC6_ARRAY_DRIVER[dev], length, txData, rxData);`, which reads and writes the SPI for the SPC58 microcontrollers, based on the SPC5-STUDIO environment.

Figure 50. **ST25R3916 driver files**



A.2.3 How to create an application with AEK-COM-NFC06A1

This example shows how to create an application, which implements the RFLA abstract layer, from scratch.

The RFLA recognizes the followings tags:

- ISO 18092 passive/active initiator, ISO18092 passive/active target
- NFC-A/B (ISO 14443A/B) reader that includes higher bit rates
- NFC-F (Felica™) reader
- NFC-V (ISO 15693) reader up to 53 kbps
- NFC-A and NFC-F card emulation

To create a new application using SPC5-STUDIO and AutoDevKit, follow the procedure below.

Step 1. Create a new SPC5-STUDIO application for the SPC582B platform.

Step 2. Add the following components:

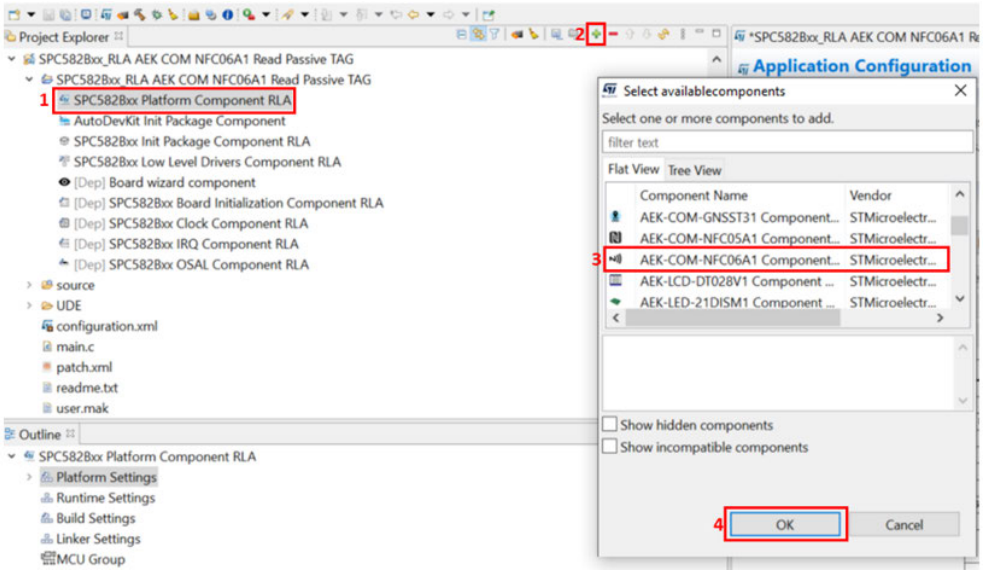
- `SPC58ECxx Init Package Component RLA`
- `SPC58ECxx Low Level Drivers Component RLA`

Note: Add these components at the beginning. Otherwise, the remaining components are not visible.

Step 3. Add the other components:

- `AutoDevKit Init Package Component`
- `SPC58ECxx Platform Component RLA`
- `AEK-COM- NFC06A1 Component RLA`

Figure 51. Adding components

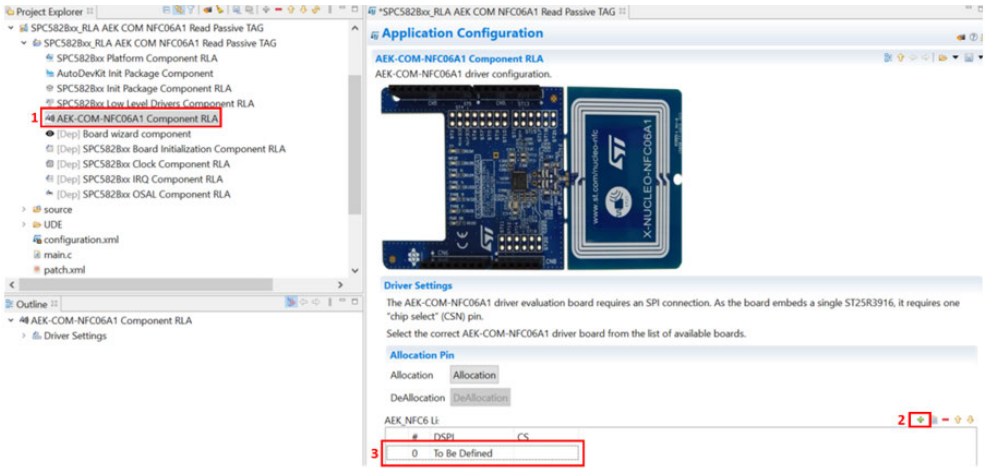


Step 4. Click on the [**AEK-COM-NFC06A1 Component RLA**] to open the [**Application Configuration**] window and configure the SPI interface.

Step 5. In the application configuration window, click on [**+**] to add a new element to the [**AEK-COM-NFC06A1 list**].

Step 6. Double-click on the newly added element to configure the SPI interface.

Figure 52. SPI configuration

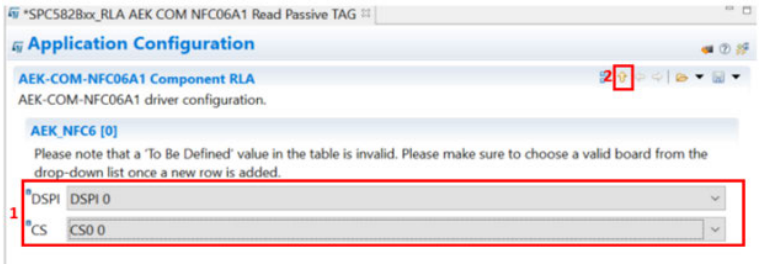


After clicking on the list item, the SPI configuration window opens.

Step 7. Select the SPI and the chip select (CS) to save.

Step 8. Return to the previous window by clicking on the arrow shown below.

Figure 53. Application configuration

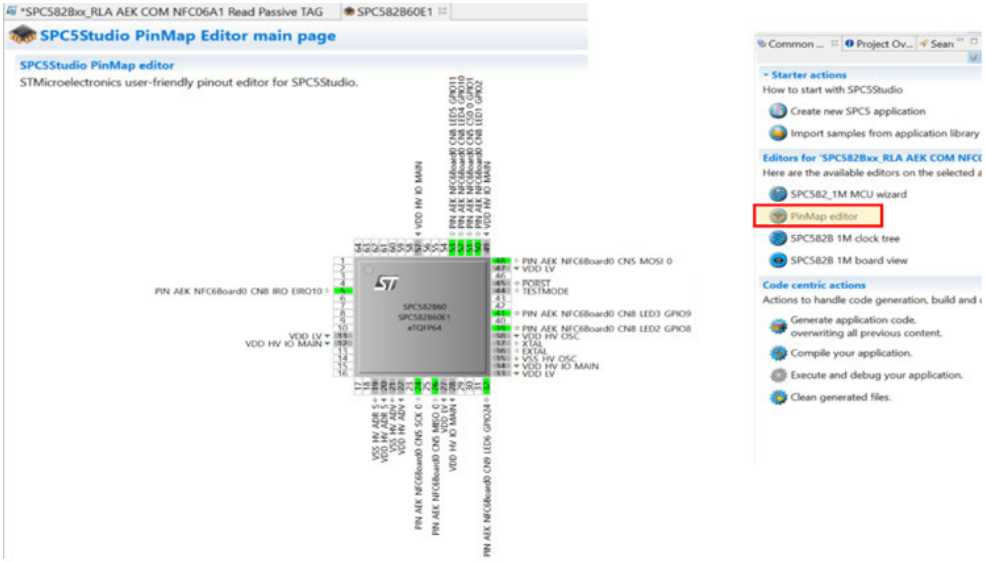


Step 9. Click the on [**Allocation**] button to delegate the automatic pin allocation to the AutoDevKit and click [**OK**] in the confirmation window.

If the allocation process ends successfully, you can open the PinMap editor to check the allocated pins.

Note: The system prompts a message if the selected SPI (DSPI) port is not available. In this case, restart from step 7 and select another SPI port or another chip select (CS).

Figure 54. PinMap editor



Step 10. Close the PinMap editor and save the application.

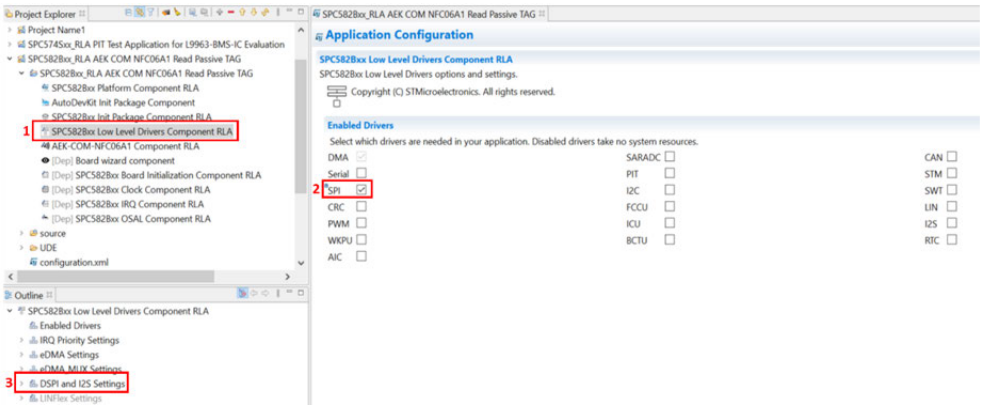
Step 11. Generate and build the application using the appropriate icons in the SPC5-STUDIO.

The project folder is then populated with the new files, including the main.c and the components folder with the `aek_com_nfc06a1_component_rla` driver.

Step 12. Click on the low-level driver component to check whether the required interface drivers have been properly enabled.

Step 13. Double-click on [**DSPI and I2S Settings**] to verify the configuration.

Figure 55. Configuration check



Step 14. In the newly opened window, scroll down to view the [**SPI Configuration**] section.

Step 15. Double-click on the configuration row to open the [**SPI Configuration Settings**]. Then, check the SPI interface configuration.

*Note: The transmission between the master (SPC582B-DIS) and the slave (AEK-COM-NFC06A1) starts when the chip select (CS) enters the low state. A GPIO emulates the CS. Therefore, the CS is set as software in the [**Chip select mode**] section.*

Step 16. Open the generated main.c file. In the header section, include the following lines for the SPI interface and the AEK-COM-NFC06A1.

```
#include "spi_lld_cfg.h"
#include "st25r3916.h"
#include "st25r3916_irq.h"
#include "st25r_platform.h"
```

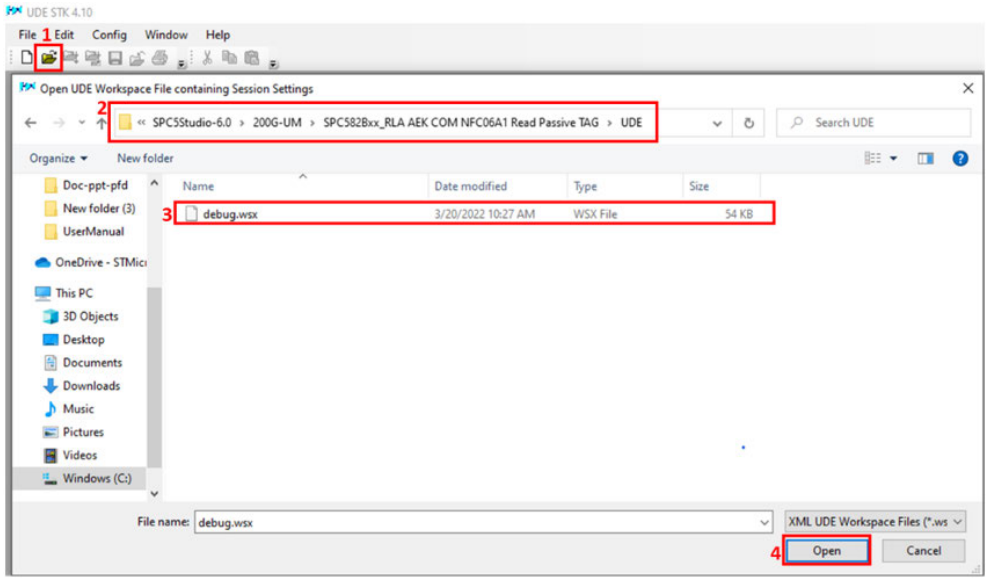
Step 17. In the main section, uncomment the `irqIrsEnable()` function. Then, add your application code and save, generate, and compile it.

```
int main(void)
{
 componentsInit();
/* This function initializes all the imported components. It is present in the
generated file. */
           irqIsrEnable(); /* This function deals with interrupt management */
```

Step 18. Connect the X-NUCLEO-NFC06A1 expansion board to the SPC582B-DIS board as described in Section A.2.1 .

Step 19. Connect the discovery board programmer to a free USB port on your PC using a USB to mini-USB cable. Then, launch SPC5-UDESTK and open the debug.wsx file located in the UDE folder of your application project. Once the loading is completed, run and debug your code.

Figure 56. Code debugging



A.2.4 Available demos for AEK-COM-NFC06A1

The AutoDevKit library includes four demos for the AEK-COM-NFC06A1 component:

- `SPC58ECxx_RLA AEK COM NFC06A1 Read Passive TAG`
- `SPC584Bxx_RLA AEK COM NFC06A1 Read Passive TAG`
- `SPC582Bxx_RLA AEK COM NFC06A1 Read Passive TAG`
- `SPC582Bxx_RLA AEK COM NFC06A1 Key Detection - Trunk System Control`

The purpose of the first three demos is to recognize the passive tags with the X-NUCLEO-NFC06A1 and the SPC58 microcontrollers.

The last one is used to open/close the car trunk (see the next section for further details). More demos might become available with new AutoDevKit releases.

Note: In the SPC58EC demo, the Arduino connector is not present. Therefore, you have to jump-wire the X-NUCLEO-NFC06A1 pins to the AEK-MCU-C4MLIT1.

A.3 Access key to open/close the car trunk system

The `SPC582Bxx_RLA AEK COM NFC06A1 Key Detection - Trunk System Control` demo has been developed to control the opening/closing of a car trunk system.

Conceived as an ECU, the demo provides a redundant keyless access to the power liftgate. It uses the SP582B-DIS (or SPC584B-DIS) and the X-NUCLEO-NFC06A1.

The application changes the state of the GPIO 24 pins, when it reads a passive type A tag.

Upon successful initialization, the X-NUCLEO-NFC06A1 LEDs flash six times. Soon after that, the *Tx LED* starts blinking.

In case of an initialization failure, the X-NUCLEO-NFC06A1 LEDs continuously flash, blocking all the operations. After the initialization phase, the application runs a polling procedure to identify the objects that meet the following requirements:

- ISO 18092 passive/active initiator, ISO18092 passive/active target
- NFC-A/B (ISO 14443A/B) reader that includes higher bit rates
- NFC-F (Felica™) reader
- NFC-V (ISO 15693) reader up to 53 kbps
- NFC-A and NFC-F card emulation

We are interested in detecting only the NFC-A tag, that is the `RFAL_NFC_LISTEN_TYPE_NFCA` case available in the *demo_polling.c*:

```
case RFAL_NFC_LISTEN_TYPE_NFCA:

                    platformLedOn(PLATFORM_LED_A_PORT, PLATFORM_LED_A_PIN);
                                                                  … … …
                                                                  … … …
                                                                  … … …
                    if (callInt==1){
                        pal_lld_setpad(PORT_PIN_GPIO24_KEY,PIN_GPIO24_KEY);
                        platformDelay(50);
                        pal_clearpad(PORT_PIN_GPIO24_KEY,PIN_GPIO24_KEY);
                    }
                    break;
```

Upon collision (`callInt` variable), if the NFC-A tag is found, the above procedure changes the state of the GPIO24 from low to high for 50 milliseconds.

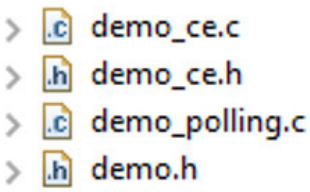
Note: If the tag is left on the antenna, the GPIO24 does not change its state because the `callInt` variable is not reinitialized to 0.

The source code folder is divided into four files: demo_ce.c, demo_ce.h, demo_polling.c, and demo.h.

The demo_ce contains the card emulation demos.

The demo_polling shows how to poll for several types of NFC cards/devices. It also shows how to exchange data with these devices using the RFAL library.

Figure 57. Source folder files



Appendix B How to use Bluetooth® Low Energy with AutoDevKit and AEK-COM-BLEV1

B.1 Power liftgate Bluetooth® Low Energy overview

In case the trunk sensors placed under the bumper are not properly working due to mud/dust/street flood, a secured wireless communication with the car to open/close the trunk represents a valid alternative to foot detection.

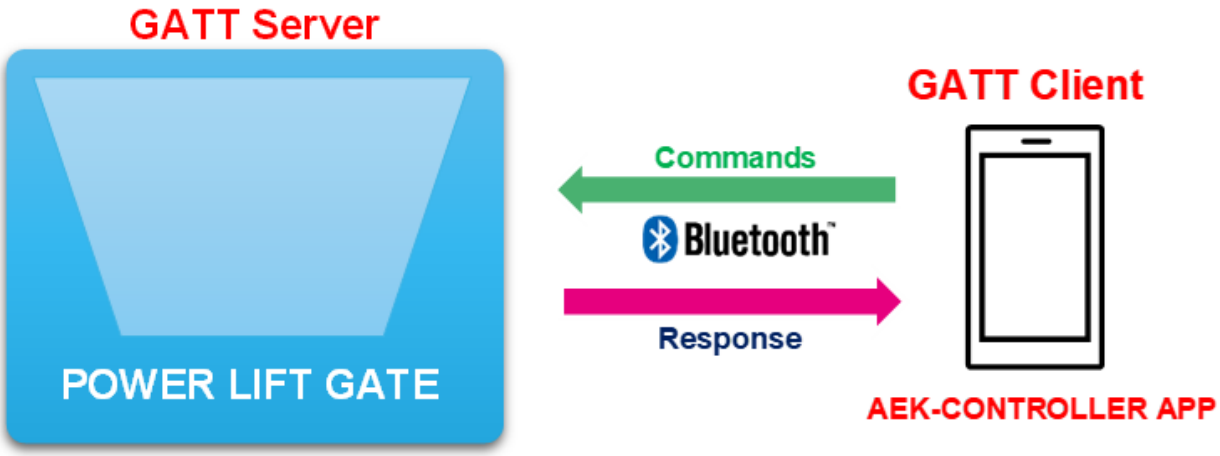
A mobile phone app connected to the trunk system via Bluetooth® Low Energy can implement the wireless communication.

To exchange data between the power liftgate and an app running on an Android mobile phone, an implemented architecture based on the Bluetooth® Low Energy protocol stack (IEE 802.15.1 standard) is required for both devices.

The power lift gate acts as a server. It accepts the incoming commands (opening/closing) sent via app mobile phone. The latter acts as a client. It monitors the status (open/closed) of the power lift gate according to the responses from the server.

An Android mobile runs the AEK-CONTROLLER app available on Google Play. This app is based on the IONIC framework. It can send commands and receive responses from a Bluetooth® Low Energy server.

Figure 58. Bluetooth® Low Energy architecture in the power liftgate



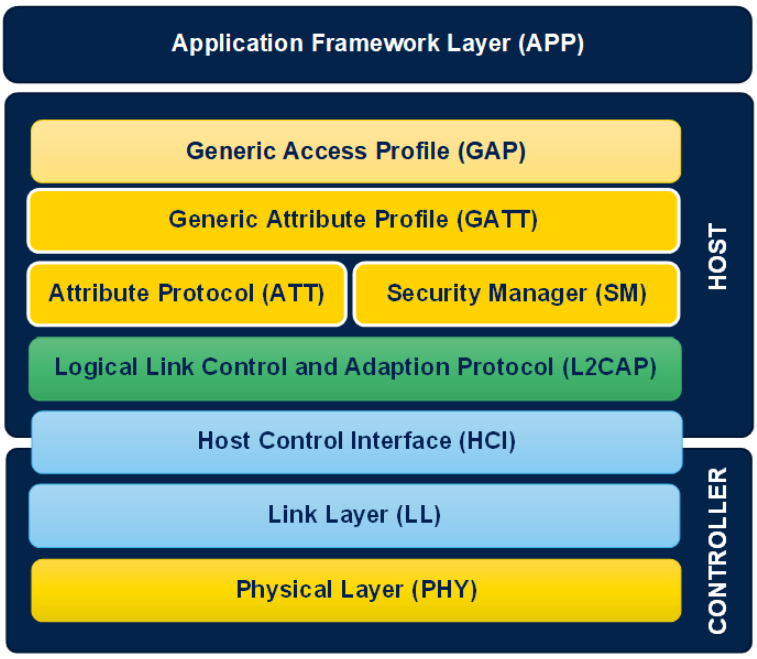
The following section describes the fundamentals of the Bluetooth® Low Energy stack protocol and its implementation for both the server and client devices to exchange data.

For further details on the Bluetooth® Low Energy stack, refer to the IEEE 802.15.1 standard.

B.2 Bluetooth® Low Energy stack fundamentals

The Bluetooth® Low Energy is a low-power wireless technology used to connect devices with each other. It operates in the 2.4 GHz band and targets power consumption-aware applications.

Figure 59. **Bluetooth® Low Energy architecture**



The Bluetooth® Low Energy protocol stack consists of a controller and a host.

The controller includes the physical layer and the link layer.

The host includes:

- logical link control
- adaptation protocol (L2CAP)
- security manager (SM)
- attribute protocol (ATT)
- generic attribute profile (GATT)
- generic access profile (GAP)

The host controller interface (HCI) is the interface between the two components.

B.2.1 Generic access profile (GAP)

The generic access profile (GAP) controls the Bluetooth® Low Energy connections and advertising.

The GAP allows your device to be visible and determines the device interactions.

B.2.1.1 Role of the Bluetooth® Low Energy devices

The GAP defines the following roles when operating over the Bluetooth® Low Energy physical channel:

- Broadcaster: a device that sends out advertisements. It does not receive packets and does not allow connection from other devices
- Observer: a device that listens to the other devices and sends out the advertising packets. It does not initiate a connection with the advertising device
- Central: a device that discovers and listens to other devices that are advertising. A central device can also connect to an advertising device
- Peripheral: a device that advertises and accepts the connection from the central devices

B.2.2 Generic attribute profile (GATT)

The GATT defines how two Bluetooth® Low Energy devices transfer data back and forth, using the services and characteristics.

After the GAP advertising process, and a dedicated connection has been established between the two devices, the GATT starts playing its role.

A Bluetooth® Low Energy peripheral can be connected only to a central device at a time. As soon as a peripheral connects to the central device, it stops advertising itself. Then, the other devices cannot see it or connect to it until the existing connection is closed.

Establishing a connection is the only way to make a central device communicate, sending meaningful data to the peripheral device and vice versa.

B.2.2.1 GATT roles

Bluetooth® Low Energy data communication is implemented in the GATT layer using the attribute protocol (ATT). The ATT is a simple client/server-stateless protocol.

The following GATT roles are defined:

- GATT client: is the device that is waiting for the data. It sends commands and requests to the GATT server
- GATT server: is the device that owns the data and accepts the incoming commands and requests coming from the GATT client. It sends responses, indications, and notifications to the GATT client

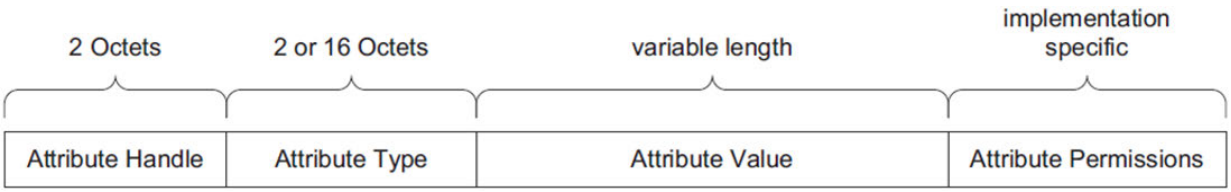
B.2.2.2 GATT attributes

A GATT Server contains data organized in the form of attributes. An attribute is a piece of labeled, addressable data, or metadata about the attribute:

- contained in the server
- accessed by the client

An attribute has the following structure:

Figure 60. GATT attribute format



The attribute handle is a unique 16-bit identifier, which:

- makes the attribute "addressable"
- does not change

Handle values grow in an ordered sequence on a server (gaps are allowed). During the discovery procedure, the client discovers them.

The attribute type determines the kind of data in the attribute value and uses a 2-byte or 16-byte UUID.

The examples include:

- Service UUID
- Characteristic UUID
- Profile UUID
- Vendor-specified UUID

The attribute value holds the actual data content, which is accessible by a client device. It also holds metadata about the attribute (depending on the type).

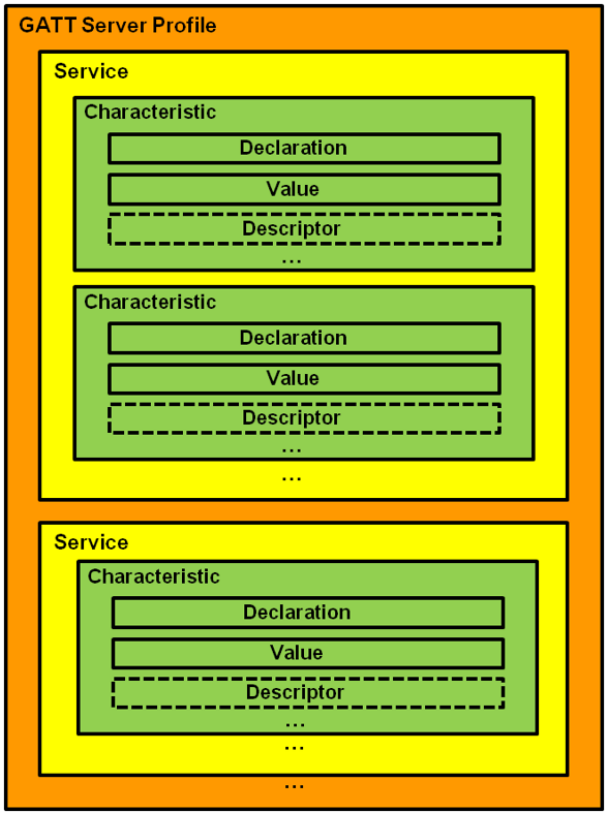
Attribute permissions are attribute metadata that specify:

- ATT access operations allowed on the attribute value
  - Read operations
  - Write operations
  - No operation
- Security requirements
  - Encryption (level required)
  - Authorization required (Yes/No)

B.2.2.3 GATT data hierarchy

The GATT establishes a hierarchy to organize attributes.
The attributes, organized as a GATT server profile, are grouped into Services that contains Characteristics.
The characteristics contain Attributes (declaration, value, descriptor (optional)).

Figure 61. Architecture of the GATT server profile



A service definition is a collection of data and the associated behaviors to accomplish a particular function. A GATT Client discovers the primary services via a GATT primary service discovery procedure.
Services are classified as public or private:

- public is defined as per the Bluetooth® Low Energy SIG (16-bit UUID)
- private is vendor-defined (128-bit UUID)

Characteristics are essentially containers for the user data. They contain a minimum of two attributes:

- characteristic declaration attribute - metadata for the value attribute
- characteristic value attribute

These are special attributes used to expand further the metadata contained in the declaration attribute.
The common descriptor attributes defined by GATT include:

- extended properties
- additional declaration property bits
- characteristic user description
- a user-readable description for the characteristic in which it is placed, that is the client characteristic configuration descriptor (CCCD)
- a switch, which enables/disables the server-initiated updates

B.2.3 Attribute protocol (ATT)

The attribute protocol layer defines the client/server architecture. It enables the GATT server device to expose a set of attributes and their associated values to a peer device (GATT client).

The attributes that the GATT server exposes can be discovered, read, and written by a GATT client, while the GATT server is notified.

B.2.4 Security manager (SM)

The security manager protocol defines the procedures and behaviors to manage pairing, authentication, and encryption between the devices.

These include:

- Encryption and authentication
- Pairing and bonding
- Key generation for a device identity resolution, data signing, and encryption
- Pairing method selection based on the IO capability of the GAP central and GAP peripheral device

B.2.5 L2CAP

L2CAP provides a connectionless data channel.

The Bluetooth® Low Energy L2CAP features:

- Channel multiplexing, which manages three fixed channels. Two channels are dedicated for higher protocol layers like ATT, SMP. One channel is used for the Bluetooth® Low Energy L2CAP protocol-signaling channel for its own use
- Segmentation and reassembly of packets whose size is up to the Bluetooth® Low Energy controller managed maximum packet size
- Connection-oriented channels over a specific registered application using the protocol service multiplexer (PSM) channel. It implements a credit-based flow control between two Bluetooth® Low Energy L2CAP entities. This feature can be used for Bluetooth® Low Energy applications that require transferring large chunks of data

B.2.6 Host controller interface (HCI)

The HCI layer implements command, event, and data interface. This interface allows the upper layers, such as GAP, L2CAP, and SMP, to access the link layer.

B.2.7 Link layer (LL)

The LL protocol manages the physical Bluetooth® Low Energy connections between the devices.

It supports all the LL states such as advertising, scanning, initiating, and connecting (master and slave).

It implements all the control procedures of the key link:

- Bluetooth® Low Energy encryption
- Bluetooth® Low Energy connection update
- Bluetooth® Low Energy channel update
- Bluetooth® Low Energy ping

B.2.8 Physical layer (PHY)

The Bluetooth® Low Energy physical layer contains the analog communications circuitry responsible for the translation of digital symbols to over-the-air waves. It is the lowest layer of the protocol stack and provides its services to the link layer.

B.2.9 Application framework layer (APP)

The application framework level is the application code. This layer can interact with the Bluetooth® Low Energy hardware through an API set. The platform library of the API set depends on the Bluetooth® Low Energy device (native development kit for the Android device).

B.3 AEK-COM-BLEV1 - Bluetooth® Low Energy hardware architecture

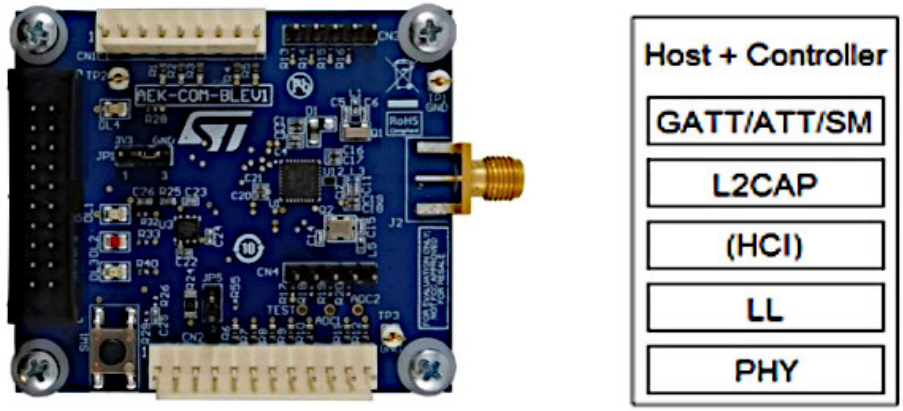
A typical Bluetooth® Low Energy system consists of a Bluetooth® Low Energy controller and host.

In many designs, the Bluetooth® Low Energy controller and the host reside in two separate silicon chips, controlled by two different microcontrollers. The communication between them is allowed by using a common protocol such as UART, USB, SPI, etc. In this scenario, the host can send HCI commands to control the Bluetooth® Low Energy controller.

In an advanced scenario, the Bluetooth® Low Energy controller and the host can be combined into a single chipset. This is a cost-saving solution as it reduces the number of silicon chips used. Moreover, the hardware connection between the host and the controller is no longer required. In this scenario, the host can directly control the LL/PHY. In this way, there is no need to generate the standard HCI commands, transmit the commands to the LL, and then process them. As a result, the execution time is improved and the calculation load on the CPU is reduced.

The combined host/controller solution has been implemented on the AEK-COM-BLEV1. The functional board hosts the BlueNRG-1 Bluetooth® Low Energy system-on-chip. It is part of the AutoDevKit ecosystem.

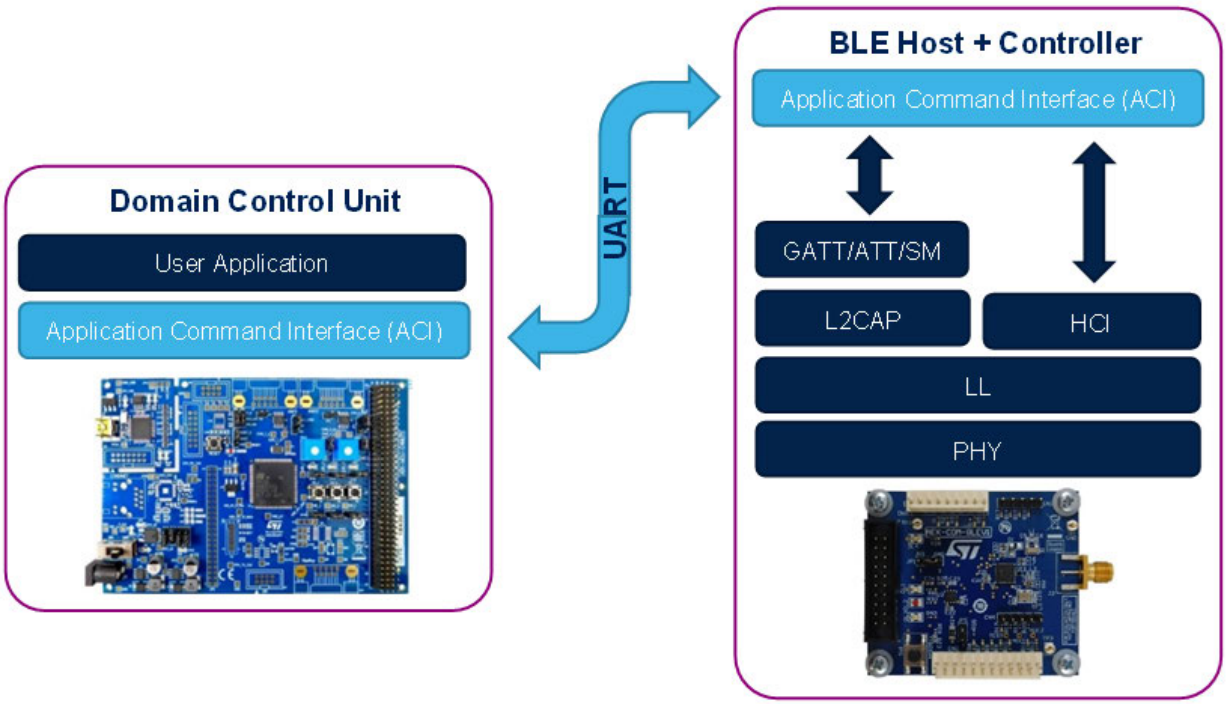
Figure 62. AEK-COM-BLEV1 functional board



In the AEK-COM-BLEV1 functional board, the host and the controller are in the same chip. To allow an external device (external domain control unit) to access the Bluetooth® Low Energy host and controller, an application command interface (ACI) is required.

User applications (programs that run on the external device) run the same ACI to communicate with the Bluetooth® Low Energy host-controller device via the UART protocol.

Figure 63. **Bluetooth® Low Energy architecture of the power liftgate**



When sending an ACI command to the Bluetooth® Low Energy architecture host-controller, the command must be formatted as described in the official Bluetooth® Low Energy architecture specification, volume 2, part E, section 5.

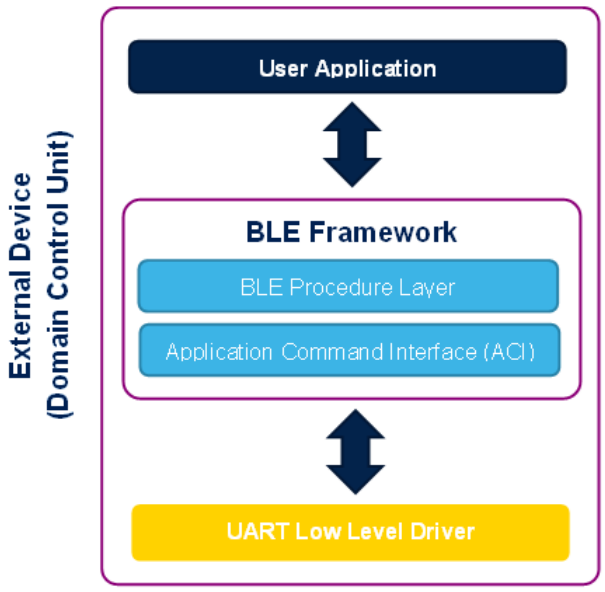
The ACI interface also supports the HCI commands. If a command is received, the ACI checks whether the command is for the host or for the controller. If the command is an HCI command (a command for the controller), the ACI directly forwards it to the controller.

This implementation has two advantages:

- the host can control the LL/PHY without using the HCI commands, thus improving performance
- user applications can still test the controller individually or set up some low-level hardware parameters with the HCI commands, without going through the host

The AEK-COM-BLEV1 functional board has a corresponding software component in the AutoDevKit software library. An external device (domain control unit) drives the board. This device is responsible for both the user application and the Bluetooth® Low Energy framework.

Figure 64. **Software architecture of the domain control unit**



The Bluetooth® Low Energy framework consists of two layers:

- the Bluetooth® Low Energy procedure layer is responsible for defining the high-level procedures such as advertising, connection to the client/server, disconnection from the client/server, attribute value updating, etc.
- the ACI layer is responsible for:
  - the conversion of the high-level procedure (generated by the Bluetooth® Low Energy procedure level) into low-level ACI commands (with a specific format described in the official Bluetooth® Low Energy specification, volume 2, part E, section 5) sent to the Bluetooth® Low Energy host-controller via UART
  - the conversion of the low-level ACI data/event packet received from the Bluetooth® Low Energy host-controller via UART into the high-level procedure

The Bluetooth® Low Energy framework communicates with the user application that runs the high-level procedures (initialization of the Bluetooth® Low Energy device, connection to the client/server, advertising, etc.) and with the UART low-level driver. These are ready-to use SPC5-STUDIO low-level drivers for the selected microcontroller.

For further details on the ACI, refer to the Bluetooth® Low Energy specification, volume 2, part E, section 5.

B.4 AEK-COM-BLEV1 - Bluetooth® Low Energy procedure layer

The Bluetooth® Low Energy procedure layer defines the following high-level procedures:

- Initialize the Bluetooth® Low Energy host/controller device
- Add a new service to a GATT server profile
- Add a new characteristic to a service
- Security configuration
- Start Bluetooth® Low Energy advertising
- Start Bluetooth® Low Energy general discovery
- Start Bluetooth® Low Energy limited discovery
- Start Bluetooth® Low Energy general connection
- Terminate connection
- Service discovery
- Characteristic discovery
- Update characteristic value
- Read characteristic value

These procedures are related to a list of ACI commands that are described in the official Bluetooth® Low Energy specification.

They allow the user application to connect to a Bluetooth® Low Energy host/controller device.

B.4.1 Initialize a Bluetooth® Low Energy host/controller device

It allows a device initialization by selecting:

- role
- address type (public, random) and value
- Tx power level
- power mode
- device name
- device address

The C function declaration is:

```
uint8_t BLE_initDevice(
AEK_COM_BLEV1_COMPONENT component,
uint8_t role,
uint8_t device_name_len,
uint8_t *device_name,
uint8_t privacy,
uint8_t address_type,
uint8_t server_address_len,
uint8_t *server_address,
uint8_t high_power);
```

The following table details the C function declaration elements.

Table 1. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
role	Allowed roles: • GAP_PERIPHERAL_ROLE • GAP_BROADCASTER_ROLE • GAP_CENTRAL_ROLE • GAP_OBSERVER_ROLE
device_name_len	Length of the device name
device_name	Device name pointer
privacy	Used to specify if the privacy is enabled or disabled: • GAP_PRIVACY_DISABLED • GAP_PRIVACY_ENABLED
address_type	Used to specify which address is used as the identity address: • PUBLIC_ADDR • RANDOM_ADDR • RESOLVABLE_ADDR • NO_RANDOM_ADDR
server_address_len	Used to specify the server address length, if needed
high_power	Used to enable/disable the high-power mode: • HIGH_POWER_NONE • From 0 to 7 POWER LEVEL

Refer to the official Bluetooth® Low Energy specification, volume 2, part E, section 5 for further details.

B.4.2 Add a new service to a GATT profile

This function llows adding a service by selecting:

- UUID type (16 or 128 bits)
- service type (primary or secondary)
- set the max. number of records

The C function declaration is:

```
uint8_t BLE_Addservice(
AEK_COM_BLEV1_COMPONENT component,
service_t svr);
```

The following table details the C function declaration elements.

Table 2. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the board) allocated→AEK_COM_BLEV1_DEV0
svr	Service data

A `service_t` data type is defined by:

- `uint8_t service_uuid[16];`
  - service UUID value
- `uint8_t service_uuid_type;`
  - service UUID type: 16 or 128 bits
- `uint8_t service_type;`
  - service type: `PRIMARY_SERVICE` or `SECONDARY_SERVICE`
- `uint8_t max_attr_records;`
  - max. value of attribute records
- `uint16_t serviceHandle;`
  - service handle
- `chars_t chars[MAX_NUM_CHAR];`
  - array of characteristics
- `uint8_t n_chars;`
  - number of characteristics of this service

B.4.3 Add a new characteristic to a GATT service

For each service, it allows adding a characteristic and the related descriptors by selecting:

- UUID type (16 or 128 bits)
- properties
- security permissions
- variable length or not
- length
- GATT event mask
- encryption key size

The C function declaration is:

```
uint8_t BLE_AddChar(
AEK_COM_BLEV1_COMPONENT component,
service_t svr,
chars_t chr);
```

The following table details the C function declaration elements.

Table 3. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
svr	Service data
chr	Characteristic data

A chars_t data type is defined by:

- uint8_t charUuid[16];
  service UUID value
- uint8_t charUuidType;
  characteristic UUID type: 16(UUID_TYPE_16) or 128 bit (UUID_TYPE_128)
- uint16_t charValueLen;
  characteristic value length
- uint8_t charProperties;
  characteristic properties:
  - CHAR_PROP_NONE
  - CHAR_PROP_BROADCAST
  - CHAR_PROP_READ
  - CHAR_PROP_WRITE_WITHOUT_RESPONSE
  - CHAR_PROP_WRITE
  - CHAR_PROP_NOTIFY
  - CHAR_PROP_INDICATE
  - CHAR_PROP_SIGNED_WRITE
  - CHAR_PROP_EXT
- uint8_t secPermissions;
  security permission flags:
  - ATTR_PERMISSION_NONE
  - AUTHEN_READ
  - AUTHOR_READ
  - ENCRY_READ
  - AUTHEN_WRITE
  - AUTHOR_WRITE
  - ENCRY_WRITE
- uint8_t gattEvtMask;
  GATT event mask flags:
  - GATT_DONT_NOTIFY_EVENTS
  - GATT_NOTIFY_ATTRIBUTE_WRITE
  - GATT_NOTIFY_WRITE_REQ_AND_WAIT_FOR_APPL_RESP
  - GATT_NOTIFY_READ_REQ_AND_WAIT_FOR_APPL_RESP
- uint8_t encryKeySize;
  minimum encryption key size required to read characteristic (from 0 to 16)
- uint8_t isVariable;
  used to specify if the characteristic has a fixed (FIXED_LENGTH) or variable (VARIABLE_LENGTH) length
- uint16_t charHandle;
  characteristic handle
- uint8_t arrayCharValue[MAX_LEN_CHAR_VALUE];
  array characteristic value read o written by services

B.4.4 Security configuration

It allows performing the security operations (send slave security request, send pairing request, etc.).
The C function declaration is:

```
uint8_t BLE_securityConfiguration(
AEK_COM_BLEV1_COMPONENT component,
uint8_t setClearSecDatabase,
uint8_t IOCapability,
uint8_t bonding_mode,
uint8_t MITMprotection,
uint8_t secConnections,
uint8_t keypressNotification,
uint8_t identifyAddressType,
uint8_t useFixedPin,
uint8_t* fixedPin,
uint8_t encryptionKeySizeMin,
uint8_t encryptionKeySizeMax)
```

The following table details the C function declaration elements.

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
setClearSecDatabase	Used to clear the security database • NO_CLEAR_DATABASE • CLEAR_DATABASE
IOCapability	Used to set IO Capability of BLE Host/Controller device • IO_CAP_DISPLAY_ONLY • IO_CAP_DISPLAY_YES_NO • IO_CAP_KEYBOARD_ONLY • IO_CAP_NO_INPUT_NO_OUTPUT • IO_CAP_KEYBOARD_DISPLAY • NO_IO_CAPABILITY
bonding_mode	Used to enable the bonding mode • BONDING_NOT_REQUIRED • BONDING_REQUIRED
MITMprotection	Used to enable the MITM mode • MITM_PROTECTION_NOT_REQUIRED • MITM_PROTECTION_REQUIRED
secConnections	Used to enable Bluetooth® Low Energy secure connection support • SC_IS_NOT_SUPPORTED • SC_IS_SUPPORTED • SC_IS_MANDATORY
keypressNotification	Used to enable the key pressed notification support • KEY_PRESSED_IS_NOT_SUPPORTED • KEY_PRESSED_IS_SUPPORTED
identifyAddressType	Used to identify the address type specification • PUBLIC_ADDR • RANDOM_ADDR • RESOLVABLE_ADDR • NO_RANDOM_ADDR
useFixedPin	To identify the use of the fixed pin • USE_FIXED_PIN_FOR_PAIRING • NOT_USE_FIXED_PIN_FOR_PAIRING

Element	Description
	-
fixedPin	Fixed pin used for paring if MITM is enabled
encryptionKeySizeMin	Minimum encryption size used for pairing
encryptionKeySizeMax	Maximum encryption size used for pairing

B.4.5 Start Bluetooth® Low Energy advertising

It allows placing a peripheral device in advertising mode by selecting:

- discoverable mode (limited, non-discoverable and general discoverable)
- type (ADV_IND, ADV_SCAN_IND, ADV_NONCONN_IND)
- set local name and type (complete or short)
- advertising intervals (min. and max.)
- policy:
  - allows a scan request from any device, allows a connect request from any device
  - allows a scan request from the whitelist only, allows a connect request from any device
  - allows a scan request from any device, allows a connect request from the whitelist only

The C function declaration is:

```
uint8_t BLE_Advertising(
AEK_COM_BLEV1_COMPONENT component,
uint8_t AdvType,
uint16_t AdvIntervMin,
uint16_t AdvIntervMax,
uint8_t OwnAddrType,
uint8_t AdvFilterPolicy,
uint8_t ServiceUUIDLen,
uint8_t* ServiceUUIDList,
uint16_t SlaveConnIntervMin,
uint16_t SlaveConnIntervMax,
uint8_t advType)
```

The following table details the C function declaration elements.

Table 4. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
AdvType	Used to configure the advertising type • ADV_IND • ADV_DIRECT_IND • ADV_SCAN_IND • ADV_NONCONN_IND
AdvIntervMin	Minimum value of the advertising interval
AdvIntervMax	Maximum value of the advertising interval
OwnAddrType	Address type • PUBLIC_ADDR • RANDOM_ADDR • RESOLVABLE_ADDR • NO_RANDOM_ADDR
AdvFilterPolicy	Used to enable/disable the whitelist • NO_WHITE_LIST_USE • WHITE_LIST_USE

Element	Description
ServiceUUIDLen	Length of the UUID service list
ServiceUUIDList	UUID service list
SlaveConnIntervMin	Minimum value of the connection interval
SlaveConnIntervMax	Maximum value of the connection interval
advType	Used to set the short or complete local name visualization • AD_TYPE_COMPLETE_LOCAL_NAME • AD_TYPE_SHORTED_LOCAL_NAME

B.4.6 Start Bluetooth® Low Energy general discovery

If the device has a central role, this function sets the device in the general scanning mode.
The C function declaration is:

```
uint8_t BLE_General_Discovery(
AEK_COM_BLEV1_COMPONENT component,
uint16_t LE_scan_interval,
uint16_t LE_scan_window,
uint8_t address_type,
uint8_t filter_duplicates);
```

The following table details the C function declaration elements.

Table 5. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
LE_scan_interval	Time interval from the moment the controller starts its last scan until it begins the subsequent scan on the primary advertising physical channel
LE_scan_window	Scanning duration
address_type	Identify the address type specification • PUBLIC_ADDR • RANDOM_ADDR • RESOLVABLE_ADDR • NO_RANDOM_ADDR
filter_duplicates	Used to enable or disable the duplicate filters

B.4.7 Start Bluetooth® Low Energy general connection

If the device has a central role, this function sets the device in the general connection mode.
The C function declaration is:

```
uint8_t BLE_Start_general_connection_establish_proc(
AEK_COM_BLEV1_COMPONENT component,
uint8_t LE_scan_type,
uint16_t LE_scan_interval,
uint16_t LE_scan_window,
uint8_t address_type,
uint8_t scanning_filter_policy,
uint8_t filter_duplicates)
```

The following table details the C function declaration elements.

Table 6. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
LE_scan_type	Passive or active scanning
LE_scan_interval	Time interval from the moment the controller starts its last scan until it begins the subsequent scan on the primary advertising physical channel
LE_scan_window	Scanning duration
address_type	Identify the address type specification • PUBLIC_ADDR • RANDOM_ADDR • RESOLVABLE_ADDR • NO_RANDOM_ADDR
Scanning_filter_policy	Used to set the scanning filter policy
filter_duplicates	Used to enable or disable the duplicate filters

B.4.8 Start Bluetooth® Low Energy limited discovery

If the device has a central role, this function sets the device in the limited scanning mode.
The C function declaration is:

```
uint8_t BLE_Limited_Discovery(
AEK_COM_BLEV1_COMPONENT component,
uint16_t LE_scan_interval,
uint16_t LE_scan_window,
uint8_t address_type
uint8_t filter_duplicates);
```

The following table details the C function declaration elements.

Table 7. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
LE_scan_interval	Time interval from the moment the controller starts its last scan until it begins the subsequent scan on the primary advertising physical channel
LE_scan_window	Scanning duration
address_type	Identify the address type specification • PUBLIC_ADDR • RANDOM_ADDR • RESOLVABLE_ADDR • NO_RANDOM_ADDR
filter_duplicates	Used to enable or disable the duplicate filters

B.4.9 Terminate connection

If the device has a central role, this function terminates the available connections.
The C function declaration is:

```
uint8_t BLE_Disconnect(
AEK_COM_BLEV1_COMPONENT component,
uint8_t reason);
```

The following table details the C function declaration elements.

Table 8. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
reason	The reason to end the connection • AUTHENTICATION_FAILURE • REMOTE_USER_TERMINATED_CONNECTION • REMOTE_DEV_TERMINATED_LOW_RESOURCES • REMOTE_DEV_TERMINATED_POWER_OFF • UNSUPPORTED_REMOTE_FEATURE • UNACCETTABLE_CONNECTION_PARAM

B.4.10 Service discovery

If the device has a central role, this function allows discovering all the available connections.
The C function declaration is:

```
uint8_t BLE_ServiceDiscovery(
AEK_COM_BLEV1_COMPONENT component);
```

The following table details the C function declaration elements.

Table 9. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0

B.4.11 Characteristics discovery

If the device has a central role, this function allows discovering all the characteristics of the available connections.
The C function declaration is:

```
uint8_t BLE_CharDiscovery(
AEK_COM_BLEV1_COMPONENT component);
```

The following table details the C function declaration elements.

Table 10. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0

B.4.12 Update characteristic value

This function updates the value (or array values) of specific characteristics.
The C function declaration is:

```
uint8_t BLE_updateCharValue(
AEK_COM_BLEV1_COMPONENT component,
service_t svr,
chars_t chr,
uint8_t *arrayValue);
```

The following table details the C function declaration elements.

Table 11. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0

Element	Description
svr	Service handle
chr	Characteristics handle
arrayValue	Array of values to write in characteristic handle

B.4.13 Read characteristic value

This function reads the value (or array values) of specific characteristics.
The C function declaration is:

```
uint8_t BLE_read_charValue(
AEK_COM_BLEV1_COMPONENT component,
service_t svr,
chars_t chr,
uint8_t *readCharValue);
```

The following table details the C function declaration elements.

Table 12. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0
svr	Service handle
chr	Characteristics handle
readCharValue	Pointer to the array of values read from the characteristic handle

B.4.14 Bluetooth® Low Energy UART communication

The Bluetooth® Low Energy UART communication is initialized in the external device using the following API:

```
SERIAL_init_Communication(AEK_COM_BLEV1_DEV0);
```

The following table details the C function declaration elements.

Table 13. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0

Note: The UART baud rate is equal to 115200 bps.

B.4.15 Bluetooth® Low Energy reset host/controller

The Bluetooth® Low Energy host/controller can be reset by using the following API:

```
_BLENRG_reset(AEK_COM_BLEV1_DEV0);
```

The following table details the C function declaration elements.

Table 14. C function details

Element	Description
component	The AEK_COM_BLEV1 device (the allocated board) →AEK_COM_BLEV1_DEV0

B.4.16 Bluetooth® Low Energy decode message

To decode a message sent by the Bluetooth® Low Energy host/controller, invoke the following API:

```
void BLE_Decode_Message_Process(void);
```

From a scheduler, you can call this function in your main application after the Bluetooth® Low Energy reset and Bluetooth® Low Energy serial initialization communication.

B.5 AEK-COM-BLEV1 sample application

In this example, we create an application to configure the AEK-COM-BLEV1 as a Bluetooth® Low Energy host controller device with a peripheral role server. We suppose that the Bluetooth® Low Energy host controller communicates with the AEK-MCU-C4MLIT1 via the UART protocol.

Step 1. Create a new SPC5-STUDIO application for the SPC58EC series microcontroller and add the following components:

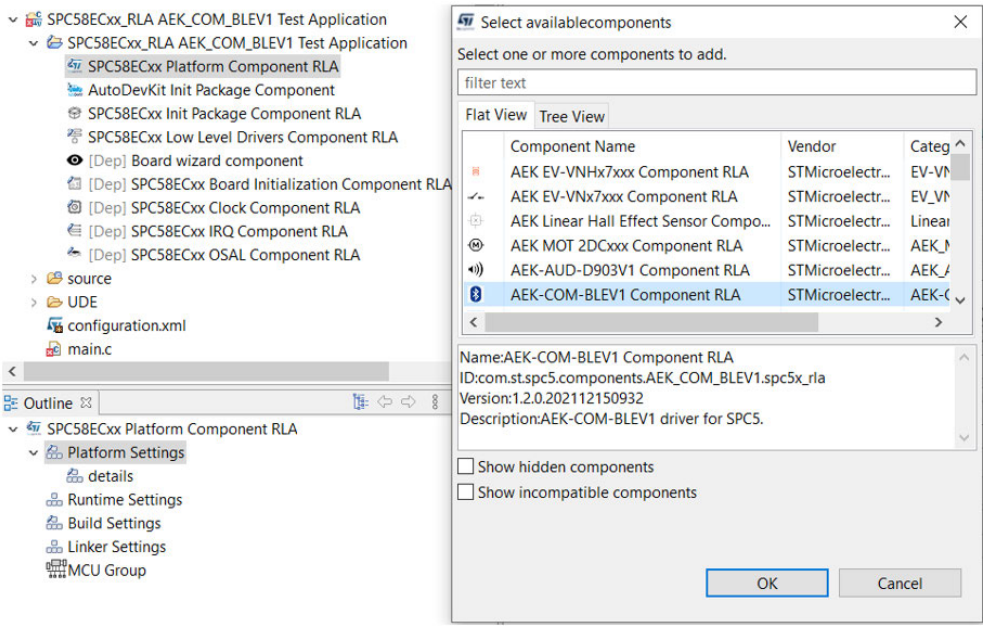
- `SPC58ECxx Init Package Component RLA`
- `SPC58ECxx Low Level Drivers Component RLA`

Add these components immediately, or the remaining ones are not visible.

Step 2. Add the following additional components:

- `AutoDevKit Init Package Component`
- `SPC58ECxx Platform Component RLA`
- `AEK-COM-BLEV1 Component RLA`

Figure 65. AutoDevKit: adding the AEK-COM-BLEV1 component



Step 3. Select the [**AEK-COM-BLEV1 Component RLA**] to open the [**Application Configuration**] window.

Step 4. Click on [**+**] to add a new element to the board list.

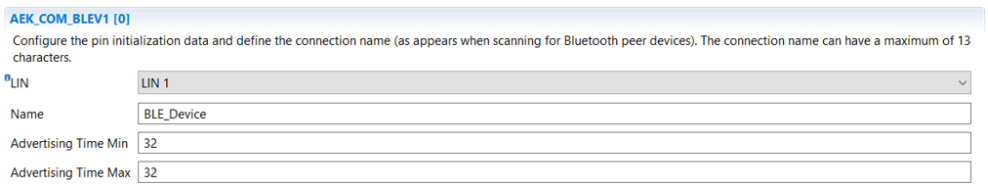
Figure 66. AutoDevKit: configuring the AEK-COM-BLEV1 component (1 of 2)

Step 5. Double-click on the newly added element to configure the board.

Step 6. Select the LIN for the UART communication.

Step 7. Configure the Bluetooth® Low Energy device name and advertising interval values.

Figure 67. AutoDevKit: configuring the AEK-COM-BLEV1 component (2 of 2)



Step 8. Click the [**Allocation**] button below the AEK-COM-BLEV1 list. Then, click [**OK**] in the confirmation window.

This operation delegates the automatic pin allocation to the AutoDevKit.

Step 9. Generate and build the application using the appropriate icons in SPC5-STUDIO.

The project folder is then populated with new files, including the main.c and the components folder with the AEK-COM-BLEV1 drivers.

Step 10. Open the main.c file and include AEK_COM_BLE_MID.h and AEK_COM_BLEV1.h header files.

Step 11. Place the following functions inside the `main()` function.

```
#include "components.h"
#include <string.h>
#include "AEK_COM_BLE_MID.h"
#include "AEK_COM_BLEV1.h"

/*
 *  Custom Service UUID setup
 */
#define CUSTOM_SERVICE_UUID(uuid_struct)
COPY_UUID_128(uuid_struct,0x36,0xA7,0x2F,0x40, 0x0C,0x86, 0x11,0xEC, 0xA3,0xEA,
0x08,0x00,0x20,0x0C,0x9A,0x66)
/*
 *  Custom Characteristic UUID setup
 */
#define CUSTOM_UUID(uuid_struct)
COPY_UUID_128(uuid_struct,0x29,0x3A,0xF2,0x50, 0x0C,0x87, 0x11,0xEC, 0xA3,0xEA,
0x08,0x00,0x20,0x0C,0x9A,0x66)
/*
 *  Custom Characteristic UUID setup
 */
#define CUSTOM1_UUID(uuid_struct)
COPY_UUID_128(uuid_struct,0x21,0x3B,0xF2,0x51, 0x0C,0x88, 0x11,0xEC, 0xA3,0xEA,
0x08,0x00,0x20,0x0C,0x9A,0x66)

/*
 * Application entry point.
 */
int main(void) {

    /* Initialization of all the imported components in the order specified in
    * the application wizard. The function is generated automatically.
    */
    componentsInit();

    /* Enable Interrupts
     */
    irqIsrEnable();

    /*Reset the Bluetooth Device
     */
    _BLENRG_reset(AEK_COM_BLEV1_DEV0);

    /*
     * Peripheral Service address
     */
    uint8_t SERVER_BDADDR[] = {0x02, 0x80, 0xE1, 0x00, 0x09, 0xE2};
    /*
     * Fixed Pin used
     */
    uint8_t Fixed_Pin[] = {0x00, 0x00, 0x04, 0xD2};
    /*
     * Local variables
     */
    uint8_t uuid[16];
    uint8_t valueToWrite[] = {0x00};
    uint8_t valueToRead[4];
    service_t custom_service;
    chars_t custom_char;
    chars_t custom_char1;

    /*
     * Device Name Configuration
     */
    uint8_t name[] = {'B','L','U','E','T','H','N','R','G'};
    /*
     *  Serial Initialization
     */
    SERIAL_init_Communication(AEK_COM_BLEV1_DEV0);
```

```
    /*
     * Debug Led
     */
    pal_lld_setpad(PORT_F,PF_LED1);

    /*
     * Init device
     */
    BLE_initDevice(AEK_COM_BLEV1_DEV0, GAP_PERIPHERAL_ROLE, sizeof(name)/
sizeof(uint8_t) , name, GAP_PRIVACY_ENABLED, PUBLIC_ADDR, sizeof(SERVER_BDADDR) /
sizeof(uint8_t), SERVER_BDADDR, 4);

    /*
     * Adding a Service
     */
    CUSTOM_SERVICE_UUID(uuid);
    custom_service.service_uuid_type = UUID_TYPE_128;
    memcpy(custom_service.service_uuid,uuid,16);
    custom_service.service_type = PRIMARY_SERVICE;
    custom_service.max_attr_records = 5;
    BLE_Addservice(AEK_COM_BLEV1_DEV0, custom_service);

    /*
     * Adding a Characteristic
     */
    CUSTOM1_UUID(uuid);
    memcpy(custom_char1.charUuid,uuid,16);
    custom_char1.charUuidType = UUID_TYPE_128;
    custom_char1.charValueLen = 1;
    custom_char1.encryKeySize = 7;
    custom_char1.gattEvtMask = GATT_NOTIFY_ATTRIBUTE_WRITE;
    custom_char1.charProperties = CHAR_PROP_WRITE;
    custom_char1.secPermissions = ATTR_PERMISSION_NONE;
    BLE_AddChar(AEK_COM_BLEV1_DEV0, custom_service, custom_char1);

    /*
     * Adding a Characteristic
     */
    CUSTOM_UUID(uuid);
    memcpy(custom_char.charUuid,uuid,16);
    custom_char.charUuidType = UUID_TYPE_128;
    custom_char.charValueLen = 1;
    custom_char.encryKeySize = 7;
    custom_char.gattEvtMask = GATT_NOTIFY_ATTRIBUTE_WRITE;
    custom_char.charProperties = CHAR_PROP_READ;
    custom_char.secPermissions = ATTR_PERMISSION_NONE;
    BLE_AddChar(AEK_COM_BLEV1_DEV0, custom_service, custom_char);

    /*
     * Adding security information
     */
    BLE_securityConfiguration(AEK_COM_BLEV1_DEV0, NO_CLEAR_DATABASE,
NO_IO_CAPABILITY, BONDING_NOT_REQUIRED, MITM_PROTECTION_REQUIRED, SC_IS_SUPPORTED,
KEY_PRESSED_IS_SUPPORTED, PUBLIC_ADDR, USE_FIXED_PIN_FOR_PAIRING, Fixed_Pin, 7, 16);

    while(1){
        /*
         * Peripheral Advertising
         */

if(BLE_getStatus(AEK_COM_BLEV1_DEV0)==NOT_CONNECTED_NOT_PAIRED_NOT_DISCOVERABLE){
                uint8_t service_list[] = {0x00};
                BLE_Advertising(AEK_COM_BLEV1_DEV0, ADV_IND, 0x020, 0x0900,
PUBLIC_ADDR, NO_WHITE_LIST_USE, 0, service_list, 0, 0, AD_TYPE_COMPLETE_LOCAL_NAME);
        }
        /*
         * Reading and Sending
         */
        if((osalThreadGetMilliseconds()%500)==0){
```

```
if(BLE_getStatus(AEK_COM_BLEV1_DEV0)==CONNECTED_NOT_PAIRED_NOT_DISCOVERABLE ||
BLE_getStatus(AEK_COM_BLEV1_DEV0)==CONNECTED_PAIRED_NOT_DISCOVERABLE){
                BLE_updateCharValue(AEK_COM_BLEV1_DEV0, custom_service,
custom_char, valueToWrite);
                BLE_read_charValue(AEK_COM_BLEV1_DEV0,custom_service, custom_char1,
valueToRead);
                valueToWrite[0]+=1;
            }
        }
        /*
         * Decoding Received Messages
         */
        BLE_Decode_Message_Process();
    }
}
```

Step 12. Save, generate, and compile the application.

Step 13. Open the [**Board View Editor**] provided by the AutoDevKit.
This editor offers a graphical point-to-point guide on how to wire the boards.

Step 14. Connect the AEK-MCU-C4MLIT1 to a USB port on your PC, using a mini-USB to USB cable.

Step 15. Launch SPC5-UDESTK-SW. Then, open the debug.wsx file in the AEK-MCU-C4MLIT1– application UDE folder.

Step 16. Run and debug your code.

Appendix C AEK Controller and Trunk Lift app

C.1 AEK Controller app

The trend for increasingly connected cars and the pervasiveness of powerful mobile phones are orienting the automotive industry towards keyless applications such as keyless entry or keyless engine start.

Several wireless technologies, including Bluetooth® Low Energy and near-filed communication (NFC), support this trend.

In line with this trend, the AutoDevKit ecosystem has been extended to include the AEK Controller app.

This app is paired with the AEK-COM-BLEV1. It is a sort of Bluetooth® Low Energy remote control for all the demonstrators developed with the AutoDevKit.

Figure 68. AEK Controller logo

The first demonstrator included in the AEK Controller app is the power liftgate application.

You can download this app from Google Play.

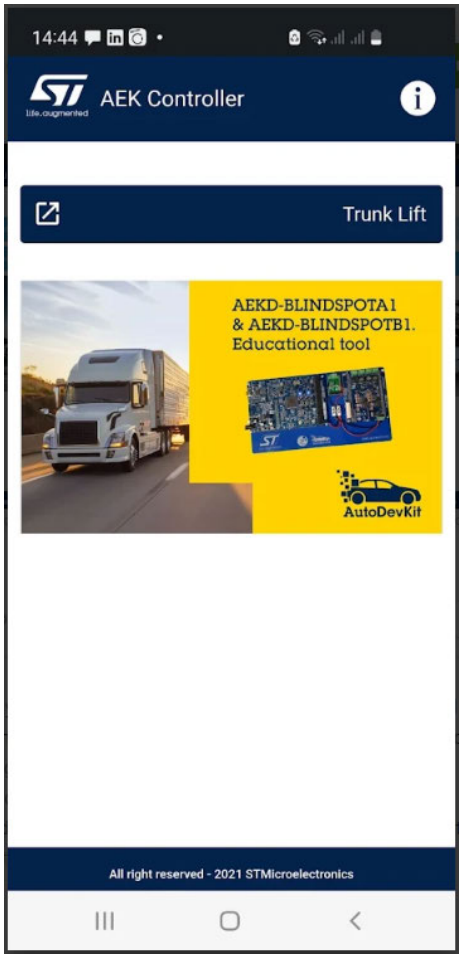
The Trunk Lift app is the portion of the AEK Controller dedicated to the power liftgate application.

C.2 AEK Trunk Lift app

The Trunk Lift app offers the possibility to control remotely the opening/closing of the AutoDevKit power liftgate. The Trunk Lift app matches the following requirements:

- each power liftgate system can accept only one Trunk Lift app as master
- only the Trunk Lift app configured as a master can associate a secondary Trunk Lift app installed on another mobile to the power liftgate
- the communication between the power liftgate and the Trunk Lift app is validated by exchanging encrypted tokens
- while connected, the power liftgate can interact only with one Trunk Lift app at a time

Figure 69. **App startup**



Note: Master key (or primary key) indicates that the app is installed on the primary mobile. Secondary key indicates that the app is installed on a secondary mobile.

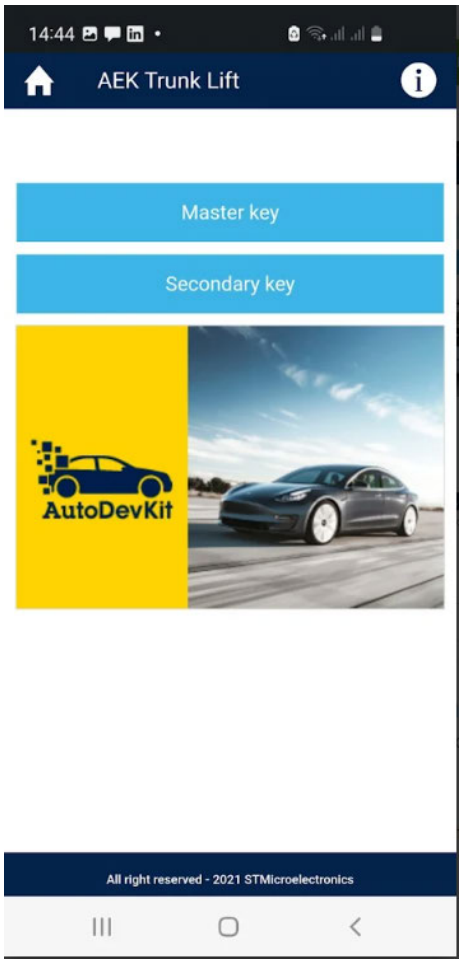
C.2.1 Login as master key

Step 1. Tap on the label to open the app.

Figure 70. **App label**

Step 2. Tap on [**Master Key**] to enter the master key configuration.

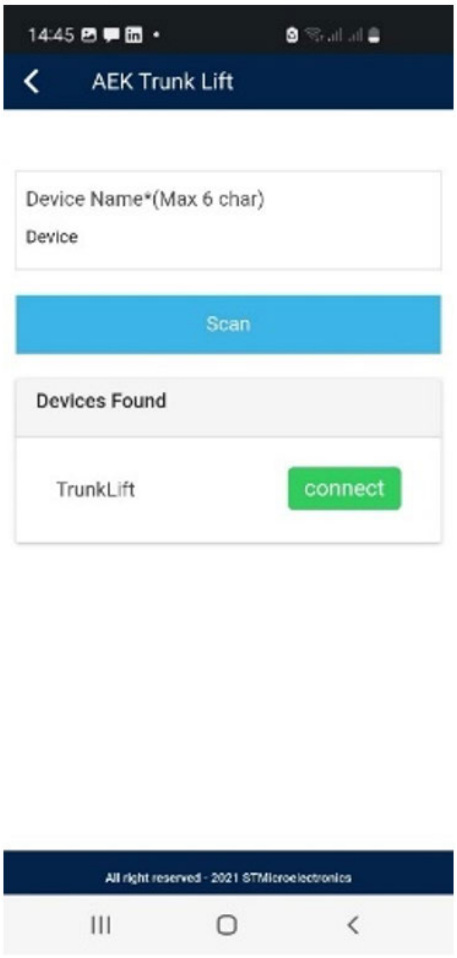
Figure 71. Master and secondary role selection



Step 3. Activate geolocation (mandatory for Bluetooth® Low Energy) and Bluetooth® Low Energy in your smartphone.

Step 4. On the displayed screen, select the [**Scan**] button to scan for all the BlueNRG devices.
The devices found are shown in the [**Device Found**] section.

Figure 72. Establishing a Bluetooth® Low Energy connection



Step 5. Add a device name in the [**Device Name**] text box.
This is a mandatory field, which accepts a maximum of six alphanumeric symbols.

Step 6. Tap on the [**Connect**] button near the device found.
The app tries to connect with the power liftgate, sending an encrypted token message. If the connection is successful, this token is stored in the power liftgate EEPROM. The token is used to validate the subsequent connection requests that the app sends.

C.2.1.1 Encrypted token

The message exchange between the power liftgate and the Trunk Lift app is based on an encrypted token. The encryption is generated as follows:

Step 1. Generate four random digits between 0 and 9.

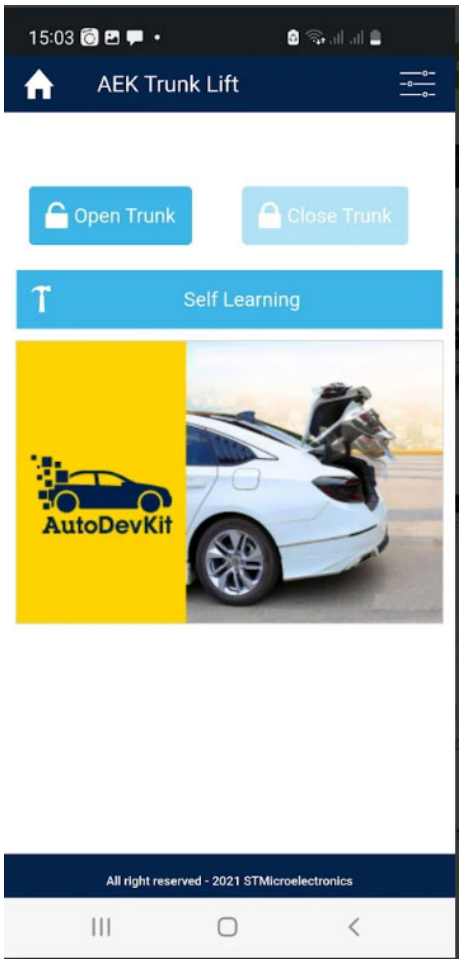
Step 2. Apply the following formula to each generated random digit:
$Number\ encrypted = ((random\ \ Number)\ XOR\ \ 170) XOR\ 187$

Step 3. Convert each of the six characters of the device name in an integer. Then, apply the same formula described before:
$Number\ encrypted = ((charIntegerCode)\ XOR\ \ 170) XOR\ 187$
The 10 encrypted numbers, obtained in steps 2-3, are used in the communication message to validate the connection between the power liftgate and the Trunk Lift app.

C.2.2 Trunk lift app - main view

When the connection between the app and the power liftgate is successfully established, the main view is displayed. This view is the same for the master and the secondary keys.

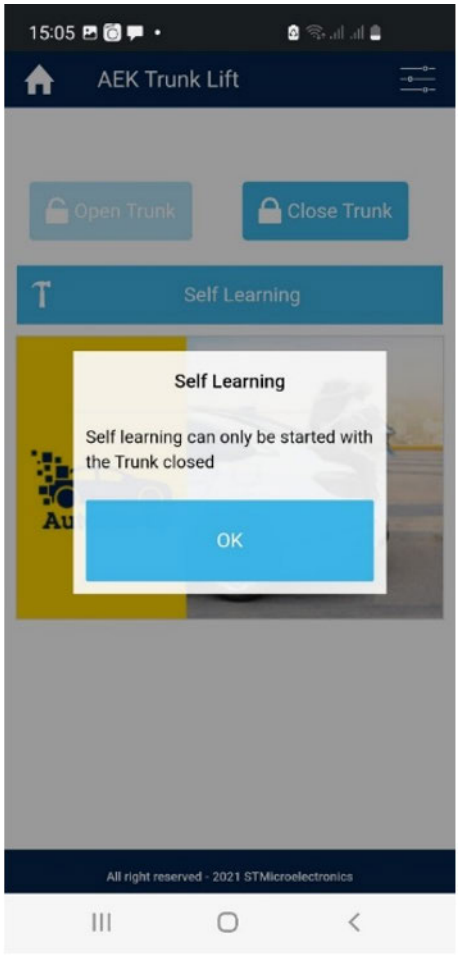
Figure 73. Main view options



According to the state of the power liftgate, the buttons for opening and closing the trunk are enabled or disabled.

The [**Self-Learning**] button is selectable only when the trunk lift is closed. It is used to calibrate some electrical parameters according to the system tear and wear.

Figure 74. **Self-learning: blocked execution**



C.2.3 Adding a secondary key

Once connected to the system, the master key can create a secondary key. The trunk lift app installed on a different mobile phone can become this secondary key. To create the secondary key, follow the procedure below.

Step 1. Tap on [**Add Secondary Key**].

Figure 75. Adding a secondary key

Step 2. Add the device name.

Step 3. Tap [**Add Device**].

The master app sends a message with an encrypted token to the power lift gate. The OTP code contained in this message is stored in the memory of the power lift gate for 20 seconds. The system uses the OTP code to identify and validate the connection that the secondary key requires.

Important: *If the connection request does not reach the power lift gate within the following 20 seconds, the generated OTP code is discarded.*

The master key app shows an OTP code. The connection with the power lift gate is closed.

Figure 76. OTP in the master key app

Step 4. Open the trunk lift app in the secondary key mobile phone and tap on the [**Secondary key**] label.

Step 5. Copy the OTP code generated from the master key in the text box above the [**Scan**] button.

Figure 77. Typing the OTP in the secondary key

Step 6. Press the [**Scan**] button to scan all the BlueNRG devices.
The devices found are shown in the [**Device Found**] section.

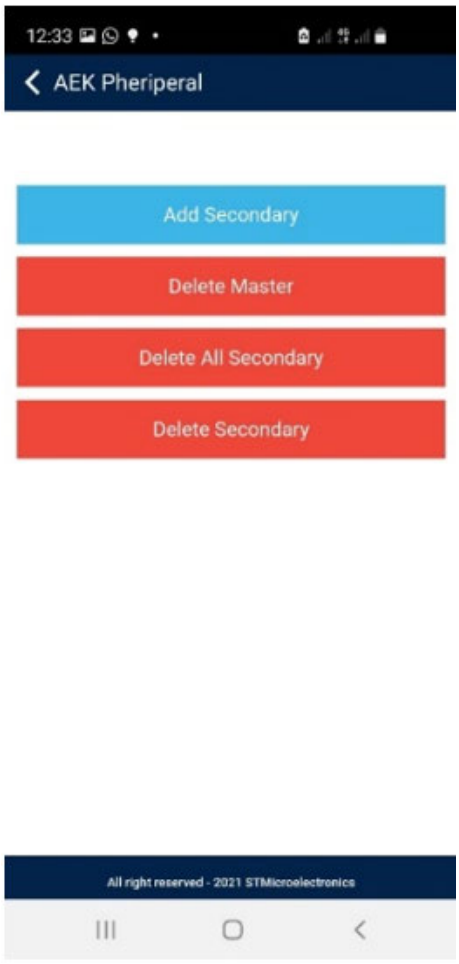
Step 7. Tap on [**Connect**] to complete the operation.

C.2.4 Deleting a registered app

The master app can:

- delete all the secondary keys
- delete only a specific secondary key. When the [**Delete Secondary**] button is pressed, you can select the secondary key to remove
- delete the master, that is, remove all the key stored in the power liftgate

Figure 78. **Master app capabilities**



A secondary app can only request for itself to be removed from the power liftgate.

Appendix D References

1. https://it.mathworks.com/help/physmod/sps/ref/dcmotor.html
2. https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed& section=SimulinkModeling
3. Thesis: Marco Mannino, Development of a Trunk Control System using AutoDevKit - An innovative foot detection feature based on Time of Flight principle.
4. https://s3.amazonaws/actuonix/Actuonix+L12+Datasheet.pdf
5. https://it.mathworks.com/products/embedded-coder.html
6. https://it.mathworks.com/products/matlab.html
7. https://it.mathworks.com/products/simulink.html
8. https://it.mathworks.com/products/stateflow.html
9. https://www.autodesk.it/products/fusion-360/personal

Revision history

Table 15. **Document revision history**

Date	Revision	Changes
25-May-2022	1	Initial release.

Contents

1 **Getting started** 2
1.1 Overview 2
1.2 Power liftgate features 3
1.3 Safety mechanisms 3
2 **Hardware architecture** 4
2.1 AEK boards in the power liftgate system 4
3 **Software architecture** 7
3.1 Power liftgate application 7
3.1.1 Gesture recognition algorithm with ToF sensors 8
3.1.2 Acoustic alerting 9
3.1.3 Mini-infotainment subsystem 10
3.1.4 LED light visual alerting 11
3.1.5 Bluetooth® Low Energy communication subsystem 11
3.1.6 AEK controller app 15
4 **Power liftgate design** 18
5 **Model-based design overview** 19
5.1 Modeling fundamentals 19
5.2 Modeling toolchain 20
6 **Open software architecture** 21
7 **Power liftgate model** 22
7.1 Plant 22
7.2 Control algorithm 24
7.2.1 External command control block 25
7.2.2 Trunk locking control block 25
7.2.3 Trunk lift control block 26
7.2.4 Fault detection control block 27
7.2.5 Obstacle detection control block 28
7.2.6 Self-calibration control block 28
7.2.7 Motion detection control block 29
8 **Model-in-the-loop (MIL) validation** 31
9 **Code generation** 32
10 **Software-in-the-loop (SIL) validation** 33
11 **Basic software integration** 34
11.1 ADC filtering function 36

12 System validation. 38
13 Power lifgate waveforms 41
13.1 Closing waveform 41
13.2 Opening waveform 42
13.3 Object detection during the closing waveform 42
13.4 NFC closing waveform 43
Appendix A NFC technology with AutoDevKit and X-NUCLEO-NFC06A1 45
A.1 X-NUCLEO-NFC06A1 board overview 45
A.2 X-NUCLEO-NFC06 in the AutoDevKit ecosystem 46
A.2.1 Hardware connection 46
A.2.2 AutoDevKit software library for the X-NUCLEO-NFC06A1 47
A.2.3 How to create an application with AEK-COM-NFC06A1 48
A.2.4 Available demos for AEK-COM-NFC06A1 52
A.3 Access key to open/close the car trunk system 53
Appendix B How to use Bluetooth® Low Energy with AutoDevKit and AEK-COM-BLEV1 54
B.1 Power liftgate Bluetooth® Low Energy overview 54
B.2 Bluetooth® Low Energy stack fundamentals 54
B.2.1 Generic access profile (GAP) 55
B.2.2 Generic attribute profile (GATT) 55
B.2.3 Attribute protocol (ATT) 57
B.2.4 Security manager (SM) 58
B.2.5 L2CAP 58
B.2.6 Host controller interface (HCI) 58
B.2.7 Link layer (LL) 58
B.2.8 Physical layer (PHY) 58
B.2.9 Application framework layer (APP) 58
B.3 AEK-COM-BLEV1 - Bluetooth® Low Energy hardware architecture 58
B.4 AEK-COM-BLEV1 - Bluetooth® Low Energy procedure layer 61
B.4.1 Initialize a Bluetooth® Low Energy host/controller device 62
B.4.2 Add a new service to a GATT profile 63
B.4.3 Add a new characteristic to a GATT service 63
B.4.4 Security configuration 65
B.4.5 Start Bluetooth® Low Energy advertising 66
B.4.6 Start Bluetooth® Low Energy general discovery 67
B.4.7 Start Bluetooth® Low Energy general connection 67
B.4.8 Start Bluetooth® Low Energy limited discovery 68

B.4.9 Terminate connection . . . 68
B.4.10 Service discovery . . . 69
B.4.11 Characteristics discovery . . . 69
B.4.12 Update characteristic value . . . 69
B.4.13 Read characteristic value . . . 70
B.4.14 Bluetooth® Low Energy UART communication . . . 70
B.4.15 Bluetooth® Low Energy reset host/controller . . . 70
B.4.16 Bluetooth® Low Energy decode message . . . 71
B.5 AEK-COM-BLEV1 sample application . . . 71
Appendix C AEK Controller and Trunk Lift app . . . 76
C.1 AEK Controller app . . . 76
C.2 AEK Trunk Lift app . . . 76
C.2.1 Login as master key . . . 77
C.2.2 Trunk lift app - main view . . . 80
C.2.3 Adding a secondary key . . . 81
C.2.4 Deleting a registered app . . . 84
Appendix D References . . . 86
Revision history . . . 87
List of tables . . . 91
List of figures . . . 92

List of tables

Table 1. C function details . . . 62
Table 2. C function details . . . 63
Table 3. C function details . . . 64
Table 4. C function details . . . 66
Table 5. C function details . . . 67
Table 6. C function details . . . 68
Table 7. C function details . . . 68
Table 8. C function details . . . 69
Table 9. C function details . . . 69
Table 10. C function details . . . 69
Table 11. C function details . . . 69
Table 12. C function details . . . 70
Table 13. C function details . . . 70
Table 14. C function details . . . 70
Table 15. Document revision history . . . 87

List of figures

Figure 1. Power liftgate: use case . . . 2
Figure 2. Power liftgate opening . . . 3
Figure 3. Power liftgate architecture . . . 4
Figure 4. Power liftgate system . . . 6
Figure 5. Software architecture layers . . . 7
Figure 6. Accepted foot sequence to open/close the trunk . . . 9
Figure 7. Acoustic alert . . . 10
Figure 8. LCD display . . . 10
Figure 9. Printing “closed” message . . . 11
Figure 10. LED alerting code . . . 11
Figure 11. Block diagram of the token key algorithm . . . 13
Figure 12. Block diagram of the master token privileges . . . 14
Figure 13. Block diagram of the secondary token privileges . . . 15
Figure 14. App main screen: master and secondary view . . . 16
Figure 15. App input commands . . . 17
Figure 16. Power liftgate system . . . 18
Figure 17. Model-based design workflow . . . 19
Figure 18. Diagram of the open software architecture . . . 21
Figure 19. ACTUONIX linear actuator . . . 22
Figure 20. DC motor: mathematical model . . . 23
Figure 21. DC motor look-up table . . . 24
Figure 22. FSM of the external command control block . . . 25
Figure 23. FSM of the trunk locking control block . . . 26
Figure 24. FSM of the trunk lift control block . . . 27
Figure 25. FSM of the fault detection control block . . . 27
Figure 26. FSM of the obstacle detection control block . . . 28
Figure 27. FSM of the self-calibration control block . . . 29
Figure 28. FSM of the motion detection control block . . . 30
Figure 29. Closed loop system . . . 31
Figure 30. Basic software integration . . . 34
Figure 31. Obstacle detection code . . . 35
Figure 32. Command and self-calibration code . . . 35
Figure 33. EEPROM code . . . 35
Figure 34. Position feedback code . . . 36
Figure 35. Digital filtering code . . . 36
Figure 36. Trunk opening options: gesture, app, NFC . . . 38
Figure 37. Power lift gate fully open . . . 39
Figure 38. Vehicle motion emulation . . . 39
Figure 39. Obstacle detection . . . 40
Figure 40. Trunk closing via the fail-safe button . . . 41
Figure 41. Zoom of the PWM signals . . . 42
Figure 42. Trunk opening via the fail-safe button . . . 42
Figure 43. Object detection . . . 43
Figure 44. Trunk closing via NFC . . . 44
Figure 45. Automotive applications for NFC . . . 45
Figure 46. X-NUCLEO-NFC06A1 component placement . . . 46
Figure 47. X-NUCLEO-NFC06A1 and SPC582B-DIS connection . . . 47
Figure 48. Wire connection . . . 47
Figure 49. Software stack . . . 48
Figure 50. ST25R3916 driver files . . . 48
Figure 51. Adding components . . . 49
Figure 52. SPI configuration . . . 50
Figure 53. Application configuration . . . 50

Figure 54. PinMap editor . . . 51
Figure 55. Configuration check . . . 51
Figure 56. Code debugging . . . 52
Figure 57. Source folder files . . . 53
Figure 58. Bluetooth® Low Energy architecture in the power liftgate . . . 54
Figure 59. Bluetooth® Low Energy architecture . . . 55
Figure 60. GATT attribute format . . . 56
Figure 61. Architecture of the GATT server profile . . . 57
Figure 62. AEK-COM-BLEV1 functional board . . . 59
Figure 63. Bluetooth® Low Energy architecture of the power liftgate . . . 60
Figure 64. Software architecture of the domain control unit . . . 61
Figure 65. AutoDevKit: adding the AEK-COM-BLEV1 component . . . 71
Figure 66. AutoDevKit: configuring the AEK-COM-BLEV1 component (1 of 2) . . . 72
Figure 67. AutoDevKit: configuring the AEK-COM-BLEV1 component (2 of 2) . . . 72
Figure 68. AEK Controller logo . . . 76
Figure 69. App startup . . . 77
Figure 70. App label . . . 77
Figure 71. Master and secondary role selection . . . 78
Figure 72. Establishing a Bluetooth® Low Energy connection . . . 79
Figure 73. Main view options . . . 80
Figure 74. Self-learning: blocked execution . . . 81
Figure 75. Adding a secondary key . . . 82
Figure 76. OTP in the master key app . . . 83
Figure 77. Typing the OTP in the secondary key . . . 84
Figure 78. Master app capabilities . . . 85