UM12110

NXP EasyEVSE EV Charging Station Linux User Manual

Rev. 1.0 — 18 June 2024 User manual

Document information

Information	Content
Keywords	UM12110, EasyEVSE, EV, charging station, EasyEVSE charging station, electric vehicle
Abstract	The NXP EasyEVSE is a simulated electric vehicle supply equipment (EVSE) development platform based on the Microsoft Azure RTOS and IoT cloud services.

1 Introduction

NXP EasyEVSE is a simulated electric vehicle supply equipment (EVSE) development platform based on the NXP Linux BSP and Microsoft Azure IoT cloud services.

The platform enables engineers to develop an EVSE with a **mutually authenticated** connection with the Microsoft Azure IoT Central and between the EVSE and electric vehicle (EV). Once connected, developers can exchange telemetry and commands with the cloud. The platform also introduces functionality of ISO 15118 and SAE 1772 to manage state and signaling with the vehicle.

EasyEVSE vehicle charger hardware consists of six evaluation boards:

- i.MX 93 EVK (smart host controller) including IW612 Wi-Fi
- EVSE-SIG-BRD (HomePlug Green PHY powerline signal board)
- TWR-KM35Z75M (meter)
- OM-SE050ARD (security)
- OM27160B1 (near field communication)
- DY1212W-4856 or MX8-DSI-OLED1A (display)

In addition, EasyEVSE provides an electric vehicle simulator for demonstration consisting of two evaluation boards:

- i.MX 93 EVK (smart host controller)
- EVSE-SIG-BRD (powerline signal board)

This document describes the functionality and development process of each EasyEVSE building block with a dedicated chapter. For example, it provides the customers a starting point to develop their full-featured EV charger by including their own power delivery circuitry.

To facilitate the startup process with this platform, refer to the *NXP EasyEVSE EV Charging Station Development Platform Linux User Guide* (document UG10134). The user guide provides detailed instructions to get the EasyEVSE up and running.

2 Preparation

You must see the platform user guide before continuing with the user manual, because the user guide lists the:

- Required hardware
- Steps to set up, program, and assembly the platform
- Steps to create an Azure IoT Central application using a custom template (EasyEVSE Dashboard)
- Steps to connect the application with an Azure IoT Central application using x.509 certificates authentication-attestation
- Platform boot up process and basic operation

Note: *This document assumes that you are already familiar with the platform. Also, you must download and install the additional software required, as shown in the below section.*

Note: *The EasyEVSE and platform terms are used interchangeably in this document, and both refer to the NXP EasyEVSE MPU development platform.*

2.1 Get the required hardware

To get details on the required hardware and the instructions on how to assemble it, see the user guide document.

2.2 Get required software

Table 1 lists the required software to develop with the EasyEVSE platform. You may find software that already installed or downloaded for the user guide.

Table 1. Quick software download table

Required, recommended, or optional software	Where to obtain it?	Additional information or notes
EasyEVSE MPU – Peripheral board components	nxp.com/easyevse	-
EasyEVSE MPU – Host processor component	nxp-easyevse-mpu-manifest meta-nxp-easyevse-mpu nxp-easyevse-mpu	See the "Building the EasyEVSE service processing software" section of the *NXP EasyEVSE EV Charging Station Development Platform Linux User Guide* (document UG10134) for instructions on how to build.
EdgeLock SE05x Plug & Trust Middleware	Edgelock SE05x Plug & Trust Middleware (04.05.00)	-
MCUXpresso IDE	MCUXpresso Integrated Development Environment (IDE)	See the IDE installation for Windows OS in Section 10.2 of the *NXP EasyEVSE EV Charging Station Development Platform Linux User Guide* (document UG10134).
MCUXpresso SDK for TWR KM3x	MCUXpresso SDK Builder	Download and install the recommended SDK release version for the TWR-KM3x board variant. See the Appendix sections 10.3 to 10.4 of the *NXP EasyEVSE EV Charging Station Development Platform Linux User Guide* (document UG10134).
MCUXpresso SDK for EVSE-SIG-BRD	MCUXpresso SDK Builder	See chapter Software development from the *EVSE-SIG-BRD1X User Guide* (document UG10109) for instructions on how to build and program the EV and EVSE endpoints.

Table 1. Quick software download table*...continued*

Required, recommended, or optional software	Where to obtain it?	Additional information or notes
Lumissil firmware binaries (third-party)	-	Provided with an evaluation license, included in the resulting i.MX image build and used to demonstrate the capabilities of the development platform.
SEVENSTAX stack library and application source code (third-party)	-	Provided with an evaluation license, included in the resulting i.MX image build and used to demonstrate the capabilities of the development platform.
QT Development software (third-party)	Embedded Software Development Tools & Cross Platform IDE | Qt Creator	Download it to further develop or modify the Easy EVSE GUI. For more information, see Section 9.

2.3 EasyEVSE development environment

The EasyEVSE software development environment software is briefly described below.

2.3.1 EasyEVSE software

The EasyEVSE contains specific application software and miscellaneous tools that help to evaluate, run, and further develop a full-featured EVSE.

The components of the EasyEVSE software relevant for this document are listed below:

- **Sample peripheral software**
  - Meter project (TWRKM3x7x_EasyEVSE_Vx)
  - EVSE-SIG-BRD project (EVSE-SIG-BRD-SW-x.y.z)

  Note: *Ensure that you import and program the corresponding sample project variant for the selected KM3x board.*
- **Sample host processor source code**
  - EasyEVSE Yocto manifest
  - EasyEVSE Yocto meta layer containing recipes for building the image
  - EasyEVSE Linux userspace applications

  Note: *The `meta-nxp-easyevse-mpu` layer references third-party source code/binaries/libraries using during the build processing, coming from Lumissil and SEVENSTAX. The licensing terms for these can be found in the `PROJECT_LICENSE` file, along with the other components used in the build.*

2.3.2 MCUXpresso IDE and SDK

The MCUXpresso IDE is an Eclipse-based software IDE. The MCUXpresso SDK unites software enablement, tools, and middleware from NXP and enabling technology partners.

2.3.3 Lumissil firmware binaries

The Lumissil firmware binaries contain the necessary behavior for the HPGP CG5317 chip onboard the EVSE-SIG-BRD. The binaries are used for enabling the high-level communication channel between the EVSE and EV endpoints.

For more information about this subject, consult the associated IS32CG5317 literature.

2.3.4 SEVENSTAX stack library and application source code

SEVENSTAX implements the ISO15118 and additional V2G stack libraries, alongside with the basic application source code.

For more information regarding this subject, refer to Section 5.1.2 and the associated SEVENSTAX product page.

3 EasyEVSE

EasyEVSE is an electric vehicle charging station development platform with a mutually authenticated connection to the Microsoft Azure IoT Central. Similarly, the charger authenticates itself to the vehicle (depending on the communication standard used). This platform is a starting point to develop a customized EV charging solution, while addressing the current and increasing challenges in this sector. For example, standardization, security, monetization, user experience, safety, and security. NXP point of view for addressing electric vehicle charging is discussed in the Addressing Design Challenges for EV Charging Systems white paper.

Figure 1 shows that an i.MX 93 MPU, designated as host controller, manages the platform. The host controller manages various boards including the security board, metering board, NFC board, HomePlug Green PHY powerline communication board. It also supports the communication interfaces with a GUI, serial terminal, and an AzureIoT central instance. The host controller runs the i.MX Linux OS BSP and offers a comprehensive environment to develop the further designs and reduce the time to market.

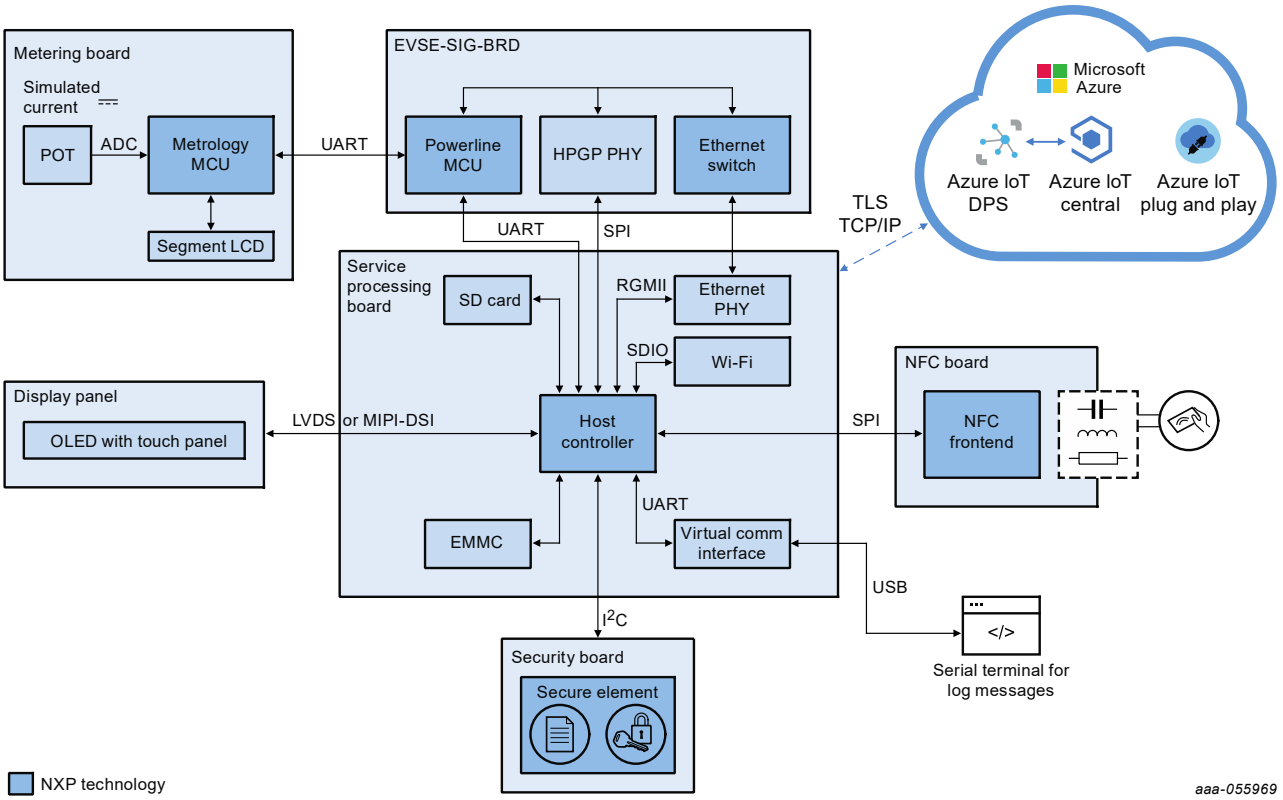


Figure 1. EasyEVSE block diagram

Security is another important aspect of the platform as it integrates the SE050 secure element to store device credentials and accelerate security operations. The NFC reader OM27160B1EVK can now read the Unique ID number or UID from NFC devices like MIFARE product-based cards for local user authentication.

Even though the TWR-KM35Z75M is used as a simulated metering interface of the EasyEVSE, it is not designed to interface with power mains. It simulates the current and voltage using the Meter library to calculate the power flow to an electric vehicle as the real application meter interface does. The EasyEVSE application calculates energy delivered based on the simulated power over time. The billing information is calculated using the energy delivered and tariff cost set in the cloud interface.

A rotary potentiometer simulates the current level through the meter, representing the instantaneous current delivered to the EV, although some other restrictions might apply.

3.1 SAE J1772

The SAE J1772 is a standard maintained by the Society of Automotive Engineers (SAE).

According to SAE Electric Vehicle and Plug-in Hybrid Electric Vehicle Conductive Charge Coupler J1772_201710, the SAE standard covers the general physical, electrical, functional, and performance requirements to facilitate conductive charging of EV/PHEV vehicles in North America. This document specifies a standardized conductive charging method for EV/PHEV and supply equipment vehicles. It includes operational requirements, and functional and dimensional requirements for the vehicle inlet and mating connector.

3.1.1 Charge state machine

A state machine, composed of six different states, controls the charge cycle of the vehicle.

Table 2. Charging states description

State	Pilot High	Pilot Low	Frequency	EV resistance	Description
STATE A	+12 V	N/A	DC	N/A	EV not connected
STATE B	+9 V	-12 V	1 kHz	2.74 kΩ	EV connected and ready
STATE C	+6 V	-12 V	1 kHz	882 Ω	EV charge
STATE D	+3 V	-12 V	1 kHz	246 Ω	EV charge vent requested
STATE E	0 V	0 V	N/A		Error
STATE F	N/A	-12 V	N/A		Error unknown

The charging states in Table 2 are general to any EVSE following the SAE J1772. In the EasyEVSE, the charging state machine is modeled as shown in Figure 2.

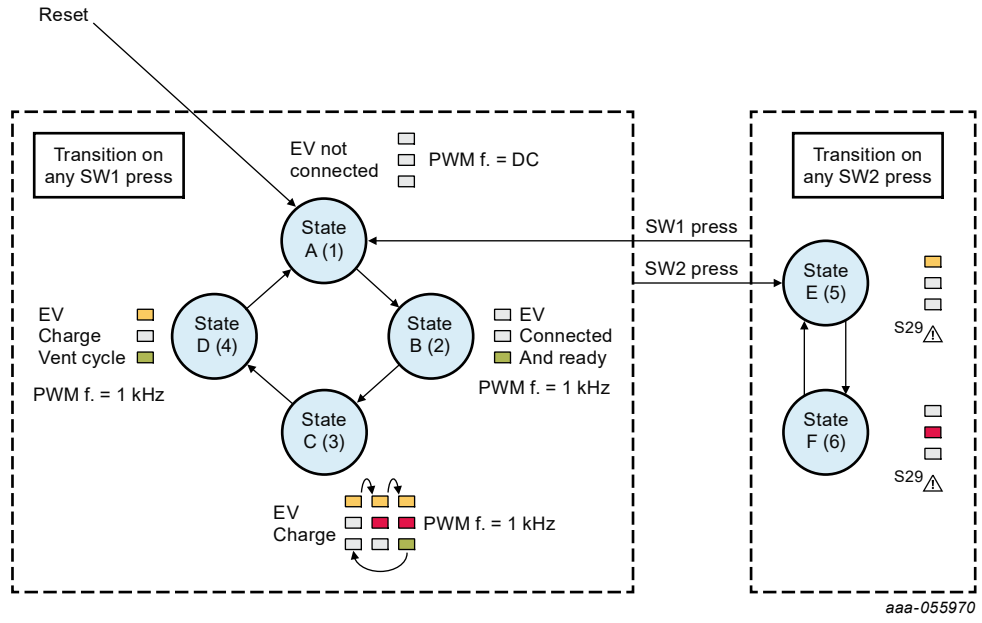


Figure 2. EasyEVSE charging state diagram

Although not shown in Figure 2, it is possible to force the charging state A or 1 by an UART message from the host controller. This is done if the *Terminate Charge Cycle* command is issued remotely from the dashboard of your Azure IoT Central application.

ISO15118 and J1772 software on the i.MX 93 platform implement the charge state machine. For details on the software design, see Section 5.2.

3.1.2 Control pilot

The control pilot signal is required to communicate the EVSE maximum current delivery capability as specified in IEC 61851-1. This signal is in the form of a PWM and the key parameter indicating the current limit is the duty cycle described in Table 3.

Table 3. EVSE control pilot signal

EVSE amp limit	Duty cycle	EVSE amp limit	Duty cycle
6 A	10 %	40 A	66 %
12 A	20 %	48 A	80 %
18 A	30 %	65 A	90 %
24 A	40 %	75 A	94 %
30 A	50 %	80 A	96 %

The EVSE indicates the EV that it is ready to supply energy by turning on the oscillator and providing the square wave signal. In each charging state, the EVSE may supply the pilot as a DC signal or as an oscillating signal.

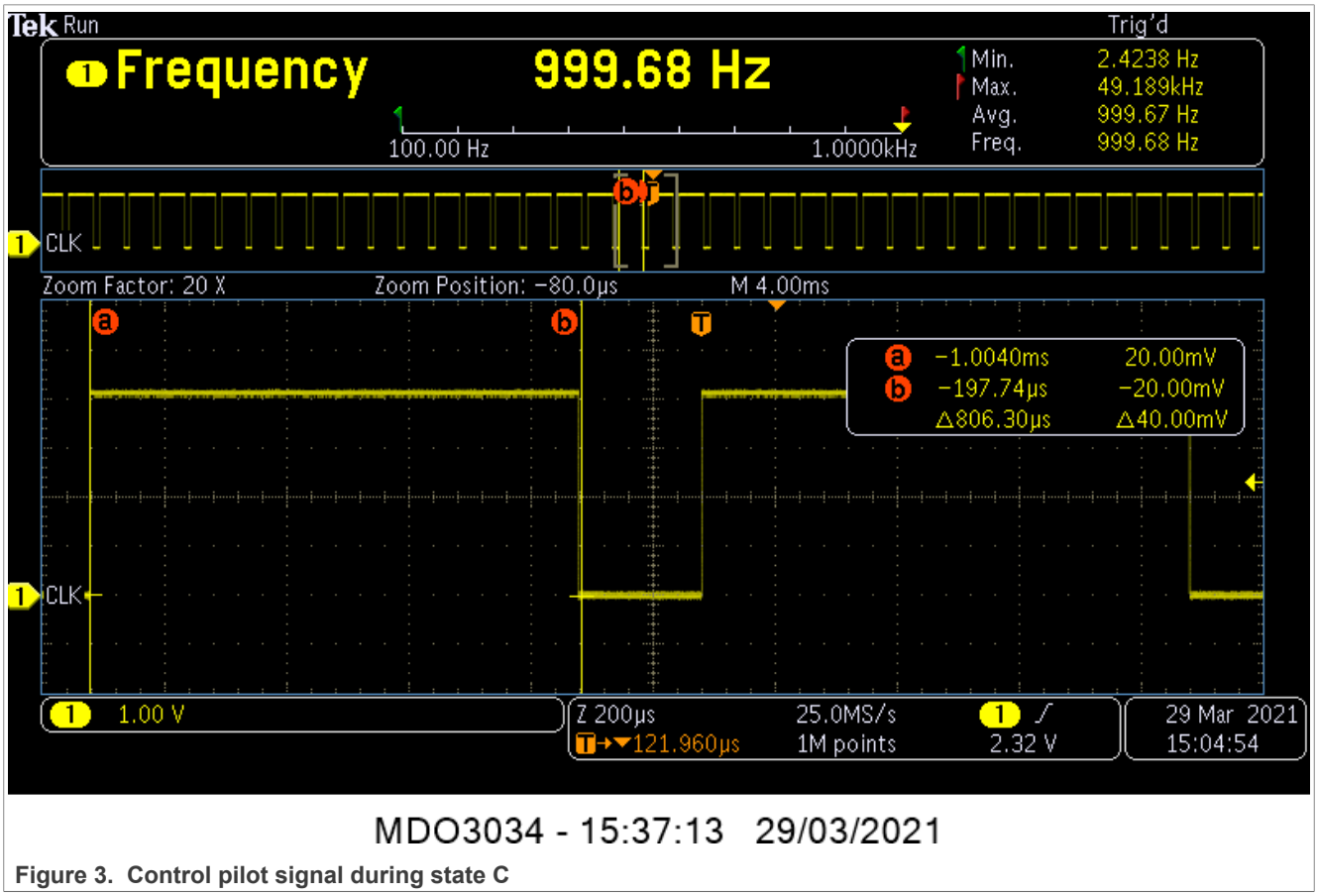


Figure 3. Control pilot signal during state C

The EVSE-SIG-BRD implements the control pilot signal. For details on the software design, see Section 6.2.2.

4 Host controller

This chapter introduces the EasyEVSE host controller and describes in detail which application components run, how they are designed, and communicate to each other.

For the EasyEVSE MPU solution, i.MX 93 EVK is used as host controller for both the EVSE and the EV sides.

4.1 EVSE software structure

Figure 4 outlines the EVSE host controller software structure.

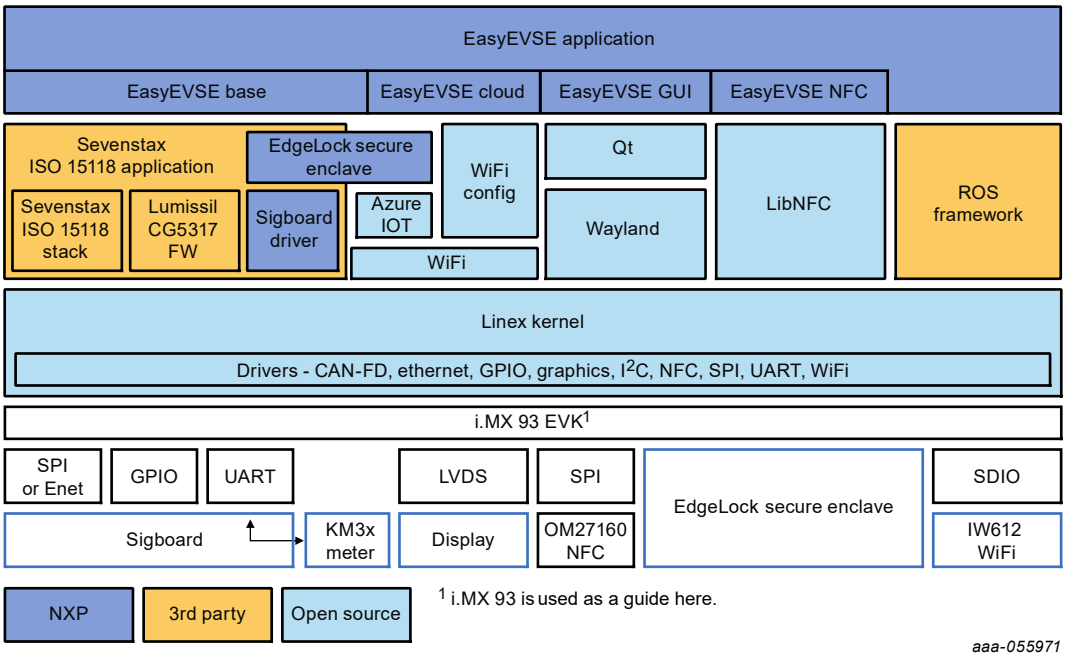


Figure 4. Host controller high-level architecture blocks

The EVSE host controller runs five application components and each of them implement specific tasks in a subscriber-publisher design. These components communicate with each other using the underlying ROS framework mechanisms, providing an easy way to transmit and notify when new data is available.

The five application components are as follows:

1. EasyEVSE Base – In charge of the communication with the EV side through the EVSE-SIG-BRD and handling of the charging process based on the information coming in from outside the application component scope. More information about this can be found in Section 6 and Section 5.1.2.
2. EasyEVSE Cloud – Responsible for establishing a connection and updating the device state in the Azure IoT Central instance. More information about this can be found in Section 8.
3. EasyEVSE GUI – Provides the local user interface and presents the status of the system through it. More information about this can be found in Section 9.
4. EasyEVSE NFC – Responsible for operations regarding the NFC peripheral. More information about this can be found in the Section 10.
5. Business logic client – Used for providing basic charger characteristics of the equipment that does not qualify for the other components. More information about this can be found in Section 4.3.

4.2 ROS framework

ROS is a collection of software libraries and tools used mainly for robotics applications. These releases target mainly the Ubuntu OS, but support is offered for the Yocto project as well, albeit limited. For more information, refer to https://www.ros.org/reps/rep-2000.html.

Note: *ROS is used in the document – it is referring to the ROS 2 version, not the ROS 1.*

The EasyEVSE MPU project is using the OpenEmbedded layer from the ROS repo, with some patches to have some basic support for running ROS applications. These can be found in the `recipes-ros` directory from the `meta-nxp-easyevse-mpu` layer.

Note: *At the moment, the only way to build ROS applications is only on the development machine. Building directly on the board with a flashed image is not validated.*

For more information and an introduction into how the ROS frameworks can be leveraged from a developer's perspective, refer to ROS documentation.

4.3 ROS node example: Business logic client

Looking from a high-level perspective, each EasyEVSE application can be split up in two distinct entities – the ROS part and the peripheral part.

- The ROS part is the first one started, and is in charge of handling the communication between the peripheral part and the other application components.
- The peripheral part does the useful work of interacting with a certain peripheral, be it Cloud, GUI, NFC, or Base. This does not contain any ROS adjacent code.

The business logic client is a simple node in charge of a certain set of parameters. Since it is not connected to any peripheral part, it only features a ROS part.

Note: *This is the template for how all the other application components are developed at the ROS part. Later chapters do not describe how the ROS part works in detail. The chapters only focus on the differences and the interface provided for communication with the other application components.*

The ROS part implementation itself borrows some concepts from the Publisher and Subscriber tutorial. Therefore, reading the tutorial is highly recommended before proceeding further in this chapter.

Here, the business logic client is the publisher for the instance of the data that is handled by it. Whereas the other application components interested in that data are the subscribers. Through callbacks, the subscribers are notified automatically when the publisher changes the values of the data they listen to, and can use that information later in their peripheral processing code.

Note: *A graphical representation of the data flow in this publisher-subscriber model can be found in Section 12.4.*

To have a clear 1:1 association for each data field and its publisher, they have been grouped in one or more message interfaces per the publisher who is responsible for updating them. These `*.msg` files can be found in the `src/interfaces/msg/`:

- CloudData – Contains data updated by a Cloud client.
- GeneralData – Contains data updated by a business logic client.
- GuiData – Contains data updated by a GUI client.
- MeterData – Contains meter data updated by a base client.
- NfcData – Contains data updated by an NFC client.
- StackData – Contains stack data updated by a base client.

Below there is a snippet of the contents of the `GeneralData.msg` interface file:

```
string      fw_vers
float64     lat
float64     lon
float64     alt
int16       temperature
string      evse_id
int16       evse_rating
```

As it can be seen, the interface file specifies both the type of the value and the name of the value transmitted. While using the interfaces, whenever any value is updated, the subscriber receives a notification for the whole group, and has to identify the changed value.

The values themselves are strictly tied to the charger characteristics, adhering to the rule of associating with a certain client. Their meanings are as follows:

- `fw_vers` – Represents the release version of the EasyEVSE MPU project.
- `lat` – Latitude value of the chargers' position in degrees (simulated).
- `lon` – Longitude value of the chargers' position in degrees (simulated).
- `alt` – Altitude value of the chargers' position in meters (simulated).
- `temperature` – Internal temperature of the charger enclosure in degrees Celsius (simulated).
- `evse_id` – Unique ID of the charger (simulated).
- `evse_rating` – Maximum current rating of the charger in amps (simulated).

5 ISO 15118

ISO 15118 is a standard defining the V2G communication between EVSE and the vehicle over various physical connectors across markets. It can be operated with plug & charge (PnC) and EIM to identify vehicles to charging network with TLS ≥1.2 security. The phase 2 NXP EasyEVSE EV charging station development platform combines basic J1772 AC charging with ISO 15118-2 AC HLC charging.

5.1 ISO 15118 overview

EasyEVSE uses the CG5317 HomePlug Green PHY transceiver from Lumissil on the EVSE-SIG-BRD component to manage the control pilot PLC.

The steps to enter ISO 15118 communication begin with the HPGP CP driving a 5 % PWM duty cycle as described in Section 3.1. The EVSE and EV agree on unique identifiers via SLAC and exchange IP address and port via the SECC discovery protocol (SDP). ISO 15118 is negotiated via the SupportedAppProtocol. The EVSE authenticates itself to the EV over TLS and communication begins. The ISO 15118 protocol initiates, monitors, re-negotiates, and terminates the charging session. ISO 15118 PLC is terminated by stopping the 5 % PWM signal.

5.1.1 Example charging sequence

Section 5.1.1 illustrates a simplified ISO 15118 AC charging sequence.

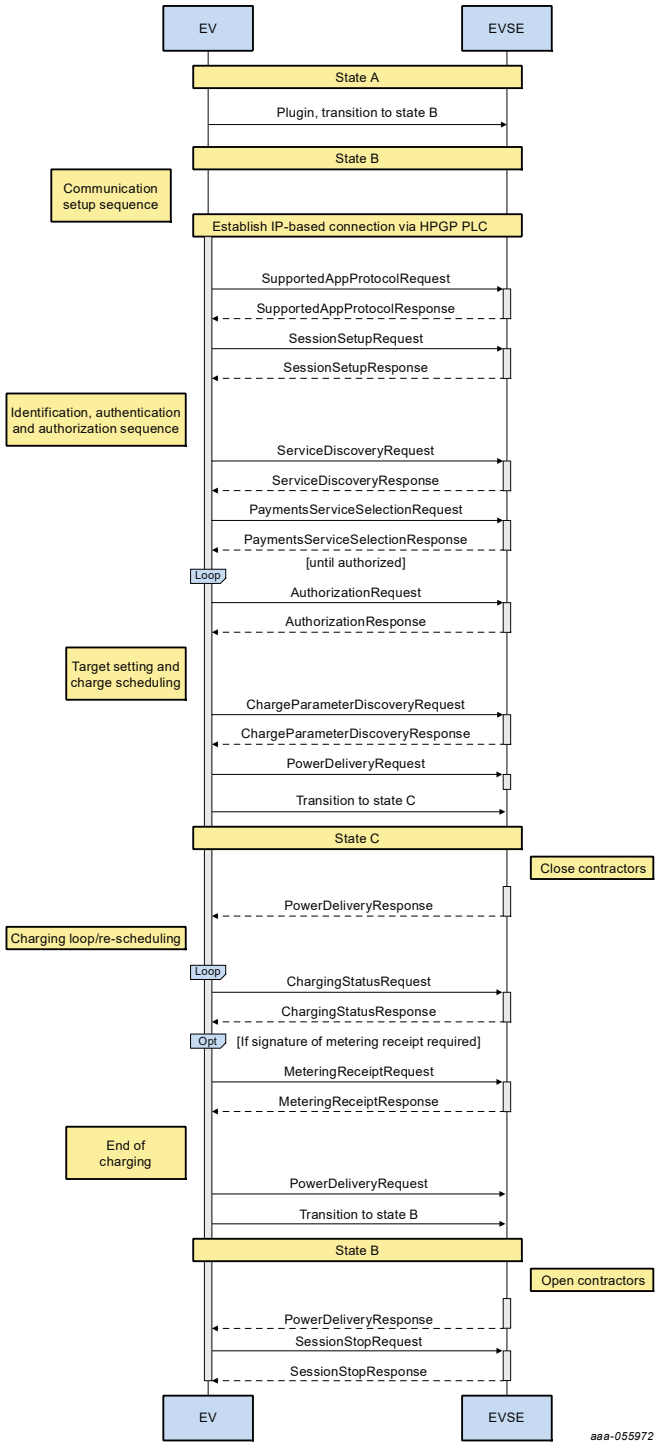


Figure 5. ISO 15118 AC charging sequence

5.1.2 SEVENSTAX

The EasyEVSE ISO 15118 PLC network stack is implemented by SEVENSTAX. For more information, see https://www.sevenstax.de/. The stack software is provided in the EasyEVSE distribution under an evaluation-only use license. Recipients can use the SEVENSTAX binary stack library and the SEVENSTAX V2G

application source code internally, but can not redistribute it in any form. The license can be found at https://www.nxp.com/LA_OPT_NXP_SW, standard license §2.2 applies. Customers are encouraged to contract directly with SEVENSTAX for EasyEVSE-compatible, redistributable software licensing.

The SEVENSTAX binary ISO 15118 stack library and V2G application source code are downloaded from NXP website during the Yocto Project build and available for immediate demonstration and development. For more information, refer to the `recipes-nxp-easyevse/sevenstax` files at https://github.com/nxp-imx-support/meta-nxp-easyevse-mpu/tree/release/easyevse-mpu-2.0. The V2G application Yocto build can be found in the `<Build Directory>/tmp/work/imx93evk_easyevse-poky-linux/sevenstax-v2g/02.00.00-r0/sevenstax-v2g-02.00.00` directory, including the SEVENSTAX libraries.

5.2 Basic charging to high-level communication charging (HLC-C)

ISO 15118 based charging control extends the IEC 61861-1 signaled charging. For AC charging, ISO 15118 allows to start charging based on IEC 61851-1 (BC) and switch to ISO 15118 based charging control (HLC-C) later.

This section introduces the transition of BC to HLC-C performed in application code. In the `stxV2GAppl_Tick` function, `eApplV2G_State` is a variable used to track the state transitions for establishing a SLAC protocol connection, progressing to establishment of ISO 15118 vehicle-to-grid (V2G) communication between the EVSE and EV. The initial charging mode is BC when charging starts. The `bApplV2G_SlacReady` function is used to indicate that the charging mode is BC or HLC-C. If `bApplV2G_SlacReady` is false that means the current charging mode is in BC. If `bApplV2G_SlacReady` is true, that means the current charging mode is in HLC-C. The `AC_DigCom_Req` is a digital communication request from EVSE side to EV side.

Note: *The application code does not support the transition of HLC-C to BC.*

5.2.1 EVSE process for BC to HLC-C

In `appl-v2g`, the `stxV2GAppl_Tick` function is executed periodically in the application. The transition of BC to HLC-C in the EVSE is shown in Figure 6.

After initialization, the `eApplV2G_State` variable remains in the `APPLV2G_STATE_SLAC_START_PRE` state until the transition from BC to HLC occurs.

When the EVSE detects the EV state change from basic signaling (BS) voltage state B (vehicle connected, not ready to charge) to state C (vehicle ready to charge), the relay is closed and BC starts. The simulation time for BC is set in the `APPLV2G_AC_CHARGING_DIGCOM_REQ_TIME` variable. When this time expires, the EVSE sets the PWM duty cycle to 5 % to initiate a digital communication request (`AC_DigCom_Req`). At this point, BC stops and the relay is opened.

The EVSE `eApplV2G_State` transitions to `APPLV2G_STATE_SLAC_START` and begins the SLAC setup protocol exchange with the EV. If a SLAC connection is established, the charging mode becomes HLC-C. However, if the SLAC connection cannot be established within a timeout, the charging mode returns to BC.

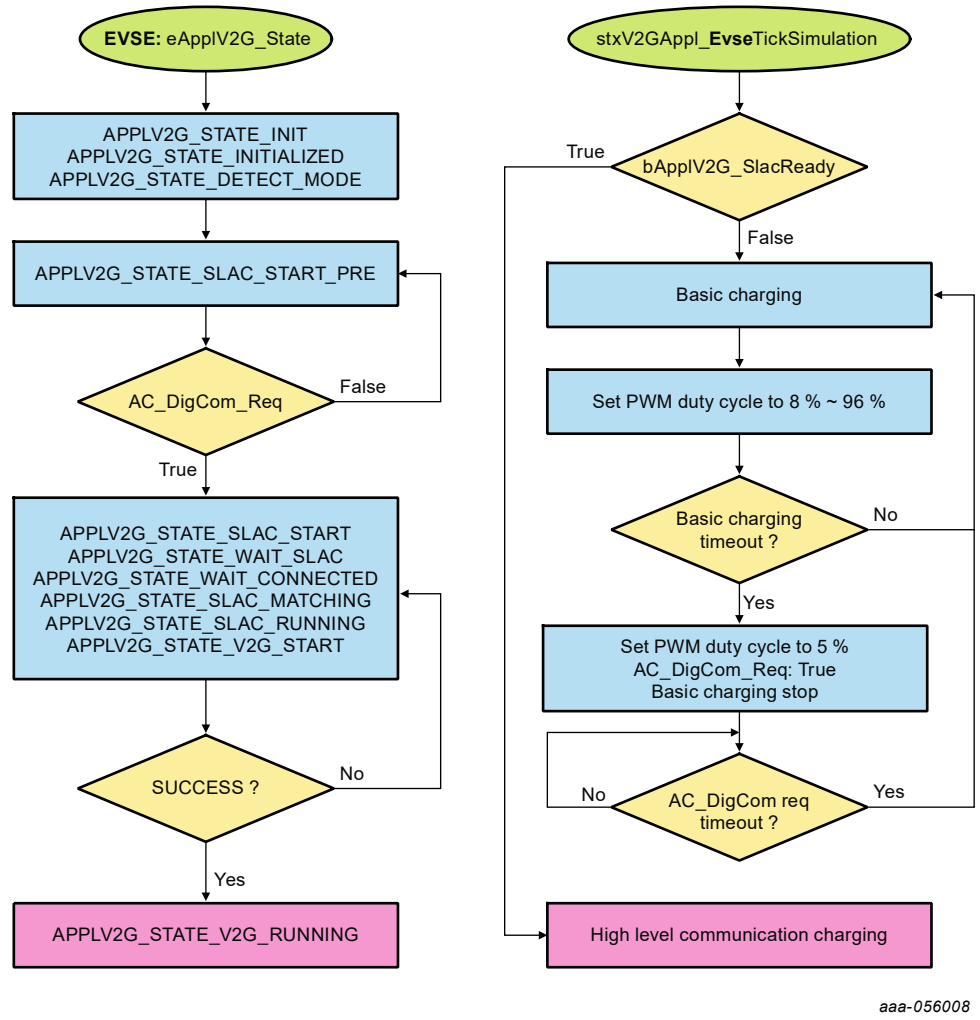


Figure 6. EVSE process for BC to HLC-C

5.2.2 EV process for BC to HLC-C

The process of BC to HLC-C at the EV side is shown in Figure 7.

After initialization, the `eApplV2G_State` variable remains in the `APPLV2G_STATE_SLAC_START_PRE` state until the duty cycle approaches 5 %. In BC mode, the EVSE sets the PWM duty cycle. If the PWM duty cycle is between 8 % to 96 %, the EV calculates the maximum current the EVSE can provide, then pulls down the CP voltage to 6 V (state C).

When the EVSE detects the CP state transitions to C, the relay is closed, and BC starts. When the EV detects the PWM duty cycle changes to 5 %, the EV recognizes digital communication is being requested and BC stops.

When the digital communication request is detected, the EV `eApplV2G_State` variable transitions to `APPLV2G_STATE_SLAC_START` and begins the SLAC setup protocol exchange with the EVSE. If a SLAC connection is established, the charging mode becomes HLC-C. However, if the SLAC connection cannot be established within a timeout, the charging mode returns to BC.

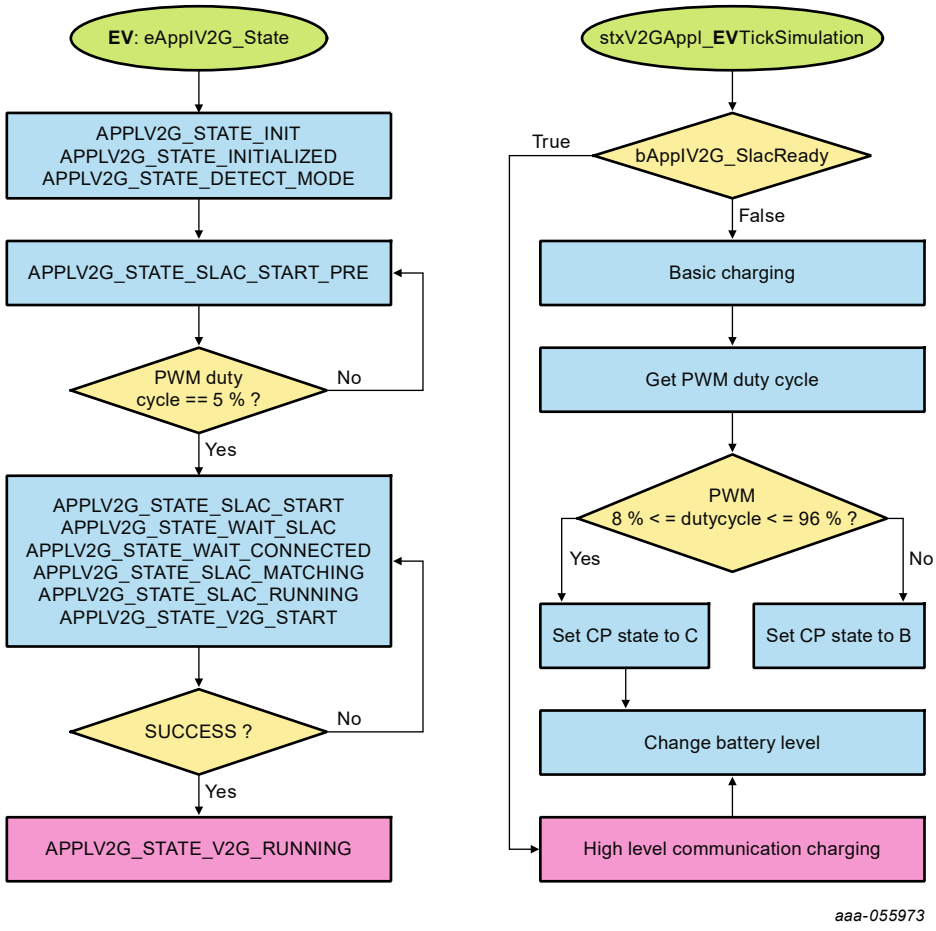


Figure 7. EV process for BC to HLC-C

6 EVSE-SIG-BRD

The EVSE-SIG-BRD is an add-on development board that supports EVSE or EV platform development when the main host of the system is on a separate processor development board, such as the NXP i.MX 93-EVK.

6.1 Introduction

EVSE-SIG-BRD is primarily designed to host a HomePlug Green PHY(HPGP) for ISO 15118-2/20 communication line, and J1772 PWM signaling for control pilot feature. It also supports proximity pilot, GFCI, and relay drive features. To support these hardware features, the circuit is built with power supply, MCUs, ASICs, HPGP, QSPI flash, and Ethernet switches interconnected to related hardware components of the board. The main host controller boards can be connected to EVSE-SIG-BRD through the host connector options.

6.2 Functional description

EVSE-SIG-BRD takes an LPC5536 as MCU controller to support the required local controller functions of the board. It can support the below list of features for EVSE or EV simulations of the EVSE-SIG-BRD:

- Generate pilot control PWM by using eFlexPWM and measure level by using ADC in the EVSE simulation mode of the EVSE-SIG-BRD.
- Measure the frequency and duty cycle of the control pilot signal in the EV simulation of the EVSE-SIG-BRD using a CTIMER module.
- Measure proximity pilot level by using ADC.
- GFCI fault detection driven by interrupt or GPIO level.
- Control relay ON/OFF GPIO function.
- Provide UART communication port between host controller and the EVSE-SIG-BRD MCU. EVSE-SIG-BRD performs the commands or requests from the host controller board. For example, in EVSE simulation mode, the MCU can set the control pilot state (high, low, or PWM) based on the UART request of the host controller. UART is the default communication channel for EVSE simulation of the board.

6.2.1 Proximity pilot

The proximity detection scheme is shown in Figure 8.

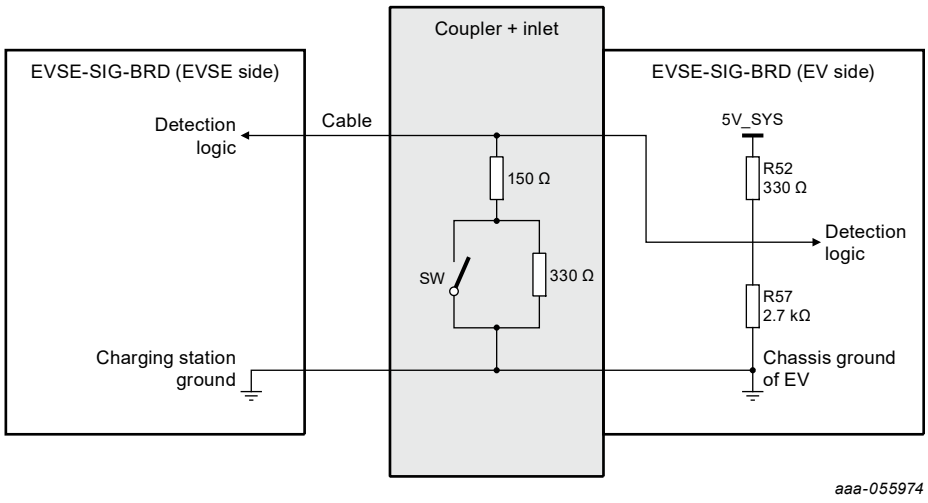


Figure 8. Proximity pilot detection logic

The voltage of the detection logic is used to distinguish the status of the proximity pilot. When the vehicle coupler connector is not connected to the vehicle inlet, the voltage of the detection logic keeps a fixed value at the EV side.

When the coupler is connected to the inlet and the latch release actuator switch is closed, the voltage at the point of detection logic drops to a lower one. When the latch SW is open, the detection voltage is changed again.

The proximity pilot circuit includes level sensing of the connector signal to trigger a wake-up to the EVSE or EV. The level of the proximity pilot signal is measured to determine the current state of proximity detection.

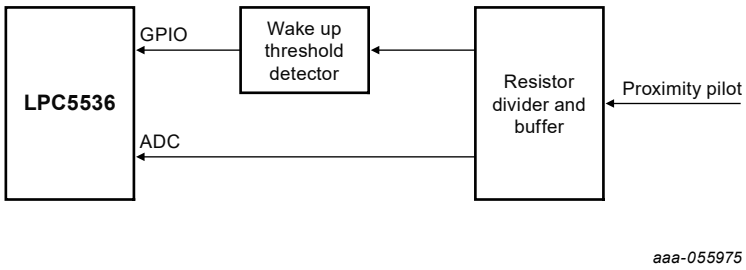


Figure 9. Proximity detection and wake-up

6.2.2 Control pilot

Control pilot of the EVSE-SIG-BRD includes generation of the J1772 PWM (IEC 61851) signal, amplification to +/-12 V, and detection of its changes of level due to connection of the charging cable to the EV. The control pilot also detects signals from the internal switching of charging states. An equivalent circuit is available at the EV side to measure the PWM ON time and to change switching levels to request the start/stop of charging sequence to the EVSE.

The 1 kHz PWM signal provides basic signaling between the EVSE and EV to indicate EV connect states, charging current capacity, and the charge start/stop requests. The PWM ON period can vary from 10 % to 96 % for basic signaling. PWM ON period of 3 % to 7 % is reserved to indicate high-level signaling using HPGP.

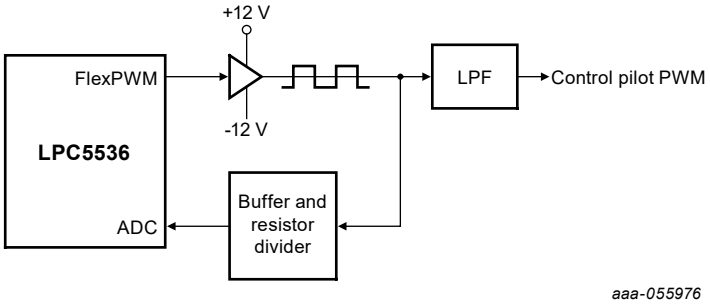


Figure 10. EVSE control pilot PWM

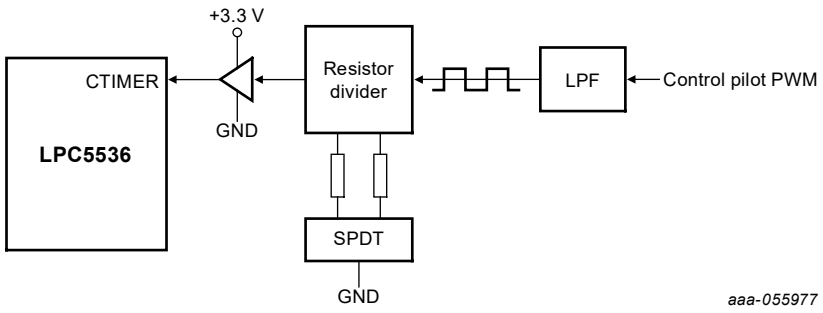


Figure 11. EV control pilot PWM

The HPGP transceiver provides high-level signaling over the same control pilot signal of the charging cable, that is, is super imposed onto the 5% J1772 PWM, and supports:

- Signal level attenuation characterization (SLAC)
- SECC discovery protocol (SDP)
- TCP/IP setup
- ISO 15118 communication sequences using orthogonal frequency division multiplexing (OFDM) with a carrier frequency of 2 MHz to 30 MHz
- A data rate of up to 10 Mbps can be achieved using the communication link between the EVSE and EV.

6.2.3 J1772 PWM

The control pilot circuit in EVSE-SIG-BRD has both the generation (EVSE side) and sense (EVSE and EV sides). The implementation is done to support both EVSE and vehicle interfaces. Figure 12 shows the PWM generation and sense scheme in the EVSE and EV sides.

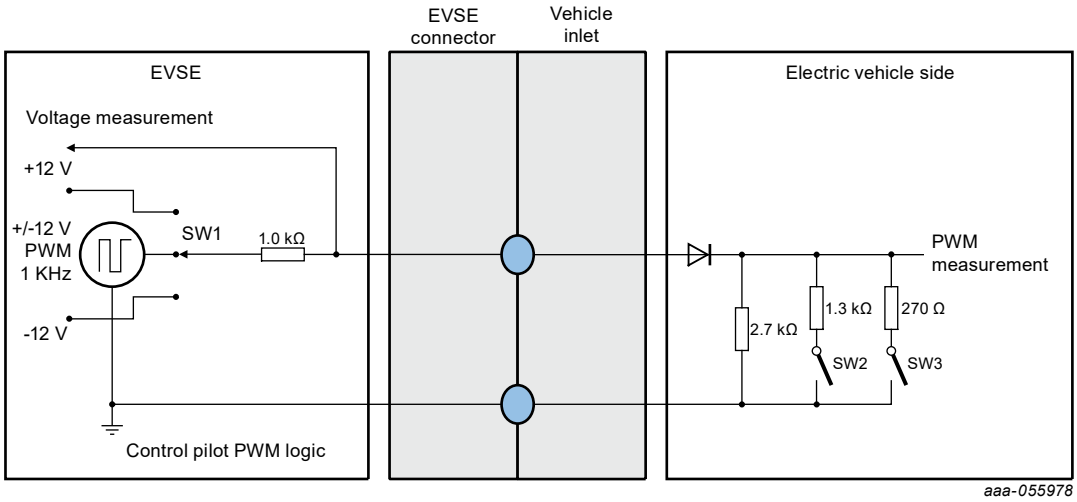


Figure 12. Control pilot PWM generation and sense

To simulate an electric vehicle side of the EVSE-SIG-BRD board, a pair of these boards are connected by using a suitable charging cable Control pilot wire.

A basic communication sequence between the EVSE and the EV is as follows:

Initially, if the EVSE can supply charge to the EV, it generates a +12 V at the CP pin. It waits for a vehicle to get connected by the charging cable. This state is termed as state 'A'.

When an EV is connected, the voltage level of the CP pin is nearly +9 V. This is measured both at the EVSE and the vehicle side. Individually, their detection logic can decide that these are connected. This state is termed as state 'B'.

Next, the vehicle can connect the internal resistance to further reduce the CP voltage level to about +6 V. This change in voltage level is again detected by EVSE and decides that the EV is ready for EVSE to start charging. Closing switch SW2 means that a vehicle indicates that it can charge in an un-ventilated area in the station. This state is termed as state 'C'.

The vehicle can otherwise connect to the internal switch SW3 to reduce the CP voltage level to about +3 V. EVSE detects this change in voltage level and decides that the EV is ready for EVSE to start charging. Closing switch SW3 means that the vehicle indicates that it can charge in a ventilated area in the station. This state is termed as state 'E'.

Note: *Instead of a recommended 270 Ω value, this resistance combination has been chosen for easy availability of components for reference design purposes only. Actual design must be done with 270 Ω value.*

After this, the EVSE can start the PWM signal with the duty cycle ranging from typically 5 % to 97 %. 5 % duty cycle has a special meaning that the EVSE wants to indicate to the EV that it can also support high-level-signaling using the HPGP CG5317. Other higher values of duty cycle indicate the basic-level-signaling only to indicate the charge current rating of the EVSE. The electric vehicle side of the EVSE-SIG-BRD measures this PWM frequency and duty cycle.

Note: *For details of the charge current encoded with PWM duty cycles, refer J1772.*

At the end of the charging, the EV can open SW2 and SW3. As a result, the CP voltage level comes back to +9 V, that is, state 'B'. The EVSE-SIG-BRD at the EVSE side continuously monitors this voltage level and changing to +9 V indicates that the PWM can be stopped, and the charging process can be stopped from the EVSE side.

If the charging cable is disconnected, then the voltage level of the CP pin must automatically come back to +12 V level, that is, state 'A'.

6.2.4 GFCI detection circuit

The ground-fault circuit interrupter (GFCI) is a fast-operating charging circuit breaker. There is a risk of electric shock due to leakage to the Earth in wet outdoor environments where the charging stations can be installed. In such cases, there is a difference between the phase versus the neutral currents through the conductors of the AC supply. It therefore becomes a mandatory requirement for the changing station to be equipped with the GFCI circuit. It is designed to trigger the generation circuit, which is sent to the relay drive circuit and also to the PIO/interrupt of the LPC5536 MCU. The trigger to the relay driver circuit enables the real-time response of the EVSE to disconnect the AC supply for user safety.

The block schematic in Figure 13 depicts an external GFCI sensor coil that is connected to the GFCI detection circuit on board. The coil interacts with the relay driver circuit and the LPC5536 MCU.

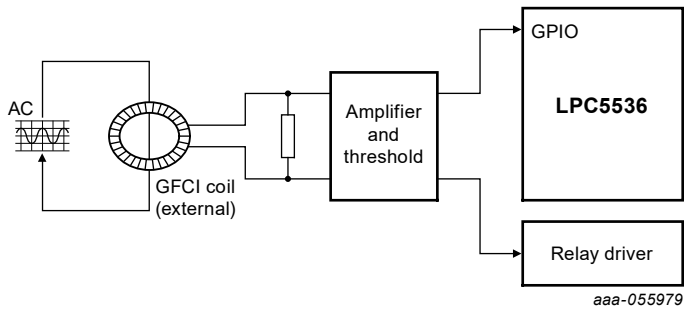


Figure 13. GFCI detection circuit

An external GFCI coil or current transformer (CT) is used as a sensor for ground fault. All the phase and neutral conductors of the AC are passed through the GFCI coil. The AC conductors act as a primary side of the transformer while the coil output acts as the secondary side. When there is no shock or leakage condition, the current passing through the phase and neutral conductors cancels each-other's induction. Therefore, there is no induced current at the GFCI coil ends. But when there is a shock condition, some amount of current is flown to the Earth surface and there is less current through the neutral return path conductor. This results in a little induced current between the GFCI coil ends.

The GFCI fault detection circuit output is fed to the relay driving circuit, which quickly disconnects the AC supply to prevent shock and damage.

GFCI software implementation is based on the NXP MCUXpresso SDK.

6.2.5 Relay driver

The relay driver circuit in EVSE-SIG-BRD mode can drive two DC coil relays to turn on or off single-phase to three-phase AC supplies to the EV through the charging cable. The external relays are hosted in the EVSE system and EVSE-SIG-BRD can drive them by a host controller command or when triggered by the GFCI circuit. An additional emergency stop push button is also supported.

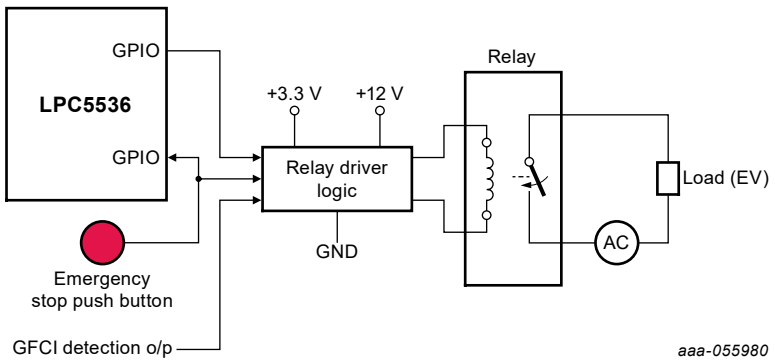


Figure 14. Relay driver circuit

The relay driver software implementation is based on the NXP MCUXpresso SDK.

6.2.6 UART bridge

The host UART serial interface can receive commands from the host controller to read and write parameters to the EVSE-SIG-BRD. There is a pre-defined set of commands supported at the UART interface. After receiving the commands from the interface, the LPC5536/LPC55S36 MCU on the board responds to the supported commands, as explained in Figure 15. The LPC5536/LPC55S36 MCU sends UNACK for the unsupported commands.

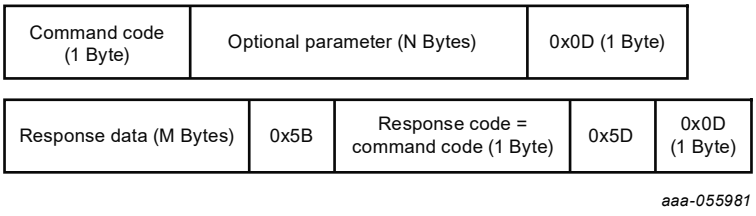


Figure 15. UART command response format

As shown in Figure 15, each command has a command code, an optional parameter, and a command end delimiter.

The host controller must wait until it gets the response of the last sent command, as the UART interface implementation of the EVSE-SIG-BRD1X software does not support a command queue. However, you can change the implementation to add support for a command queue.

Table 4 lists supported commands and their responses. Some of the commands are only applicable for the EVSE side of the software and the others are for the EV side of the software.

Table 4. Supported commands

Type/ Side	Command code	Command parameter	Response data	Resp. code	Description	Example
Read/ EVSE, EV	0x62 (alphabet 'b')	No parameter, 0 byte	PP state, 1 byte	'b'	Read proximity pilot state	Command = "b\r" Response = "0[b]\r"
Read/ EVSE	0x63 (alphabet 'c')	No parameter, 0 byte	CP state, 1 byte	'c'	Read control pilot state	Command = "c\r" Response = "0[c]\r"
Read/ EVSE	0x64 (alphabet 'd')	No parameter, 0 byte	GFCI state, 1 byte	'd'	Read GFCI state	Command = "d\r" Response = "0[d]\r"
Read/ EVSE	0x65 (alphabet 'e')	No parameter, 0 byte	ADC value, 5 characters	'e'	Read ADC value of control pilot	Command = "e\r" Response = "59354[e]\r" ***Note:*** *"59354" is the ADC value in character string format.*
Read/ EVSE	0x66 (alphabet 'f')	No parameter, 0 byte	ADC value, 5 characters	'f'	Read ADC value of proximity pilot	Command = "f\r" Response = "12345[f]\r" ***Note:*** *"12345" is the ADC value in character string format.*
Read/ EV	0x67 (alphabet 'g')	No parameter, 0 byte	PWM duty cycle value, 4 characters	'g'	Read PWM duty cycle in per mille (0-1000)	Command = "g\r" Response = "0500[g]\r" ***Note:*** *"0500" is the PWM ‰ value in character string format.*
Read/ EV	0x68 (alphabet 'h')	0 = Read resistor 270 Ω 1 = Read resistor 1.3 KΩ, 1 byte	Control pilot switch resistor values 0 = resistor not set 1 = resistor set	'h'	Read control pilot switch resistor values	Command = "h1\r" Response = "1[h]\r" ***Note:*** *Command byte 2 = '1' indicates CP resistor 1.3 kΩ state request. Response byte 1 = '1' means that this resistor is in ON state.*
Write/ EVSE	0x69 (alphabet 'i')	5 character string	None	'i'	Set PWM duty cycle in ‰	Command = "p00500\r" Response = "[i]\r" ***Note:*** *Command parameter "00500" to set a 50 % duty cycle.*
Write/ EVSE	0x6A (alphabet 'j')	No parameter, 0 byte	None	'j'	Close relay	Command = "j\r" Response = "[j]\r"
Write/ EVSE	0x6B (alphabet 'k')	No parameter, 0 byte	None	'k'	Open relay	Command = "k\r" Response = "[k]\r"
Write/ EV	0x73 (alphabet 's')	Byte1: '0' = Open 270 Ω resistor '1' = Close 270 Ω resistor	None	's'	Set control pilot switch resistors	Command = "s01\r" Response = "[s]\r" ***Note:*** *Command bytes 1, 2 are ASCII coded numbers.*

Table 4. Supported commands...*continued*

Type/ Side	Command code	Command parameter	Response data	Resp. code	Description	Example
		Byte2: '0' = Open 1.3K Ω resistor '1' = Close 1.3K Ω resistor, 2 bytes				***Note:*** *Command byte 2 = '0' means that 270 Ω resistor to be turned off.* ***Note:*** *Command byte 3 = '1' means that 1.3 KΩ resistor to be turned on.*

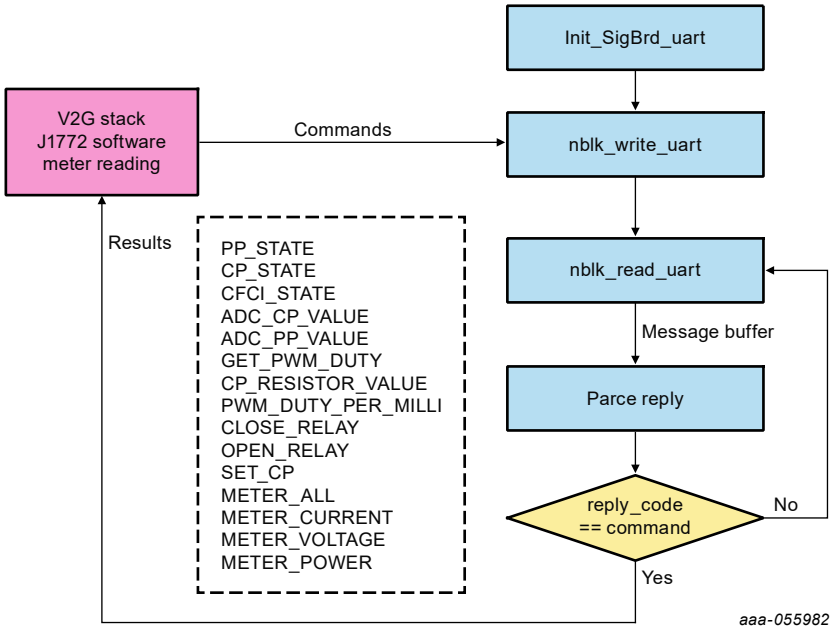


Figure 16. UART Bridge driver

The main flow of the UART Bridge driver is shown in Figure 16. V2G stack, J1772 software, and meter reading operation share one UART bridge bus. Based on the charging process, the host controller sent one certain command through non-block write, and get a response through non-block read. If the reply code of the message buffer is not equal to the command code sent, the driver continues monitoring another response until the expected reply code is got. The time-out value of non-block write and read in the driver is 20 ms by default.

When the host controller reads the meter, the EVSE-SIG-BRD transparently transmits the commands to the meter, and also transmits the reply value of the meter.

6.3 HomePlug Green PHY

The HomePlug Green PHY interface has been implemented using an ISSI CG5317 HPGP transceiver. It is HomePlug Green PHY compliant, HomePlug AV, and IEEE1901 ready supporting frequency band 2 MHz to 30 MHz. It has an internal AFE for the medium/line interface and SPI target and MII/RMII interfaces to an external host processor. The HomePlug Green PHY interface contains an internal processor. The HPGP can boot from either the external host processor or flash memory located on the EVSE-SIG-BRD. Figure 17 shows the design scheme with CG5317.

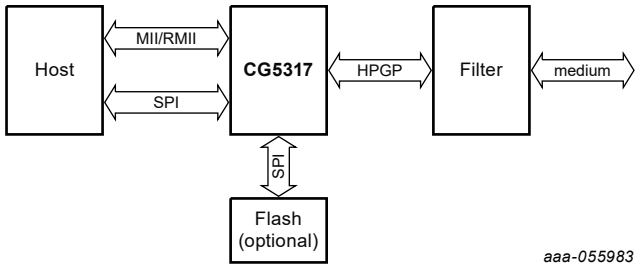


Figure 17. Design with CG5317

The on-board HPGP in the EVSE-SIG-BRD is accessible to the host processor by either SPI or Ethernet interfaces. These interfaces allow the host to:

- Boot
- Communicate
- Run control and management services with the HPGP

Note: *Customers are suggested to obtain the detailed and most up-to-date information from Lumissil directly through their website: https://www.lumissil.com/home.*

The host interface supports two interfaces that could be used in parallel – SPI and MII/RMII.

6.3.1 SPI slave interface

The SPI slave interface of CG5317 and an additional interrupt line be connected to an expansion connector on EVSE-SIG-BRD. The connectivity of the SPI interface is shown in Figure 18.

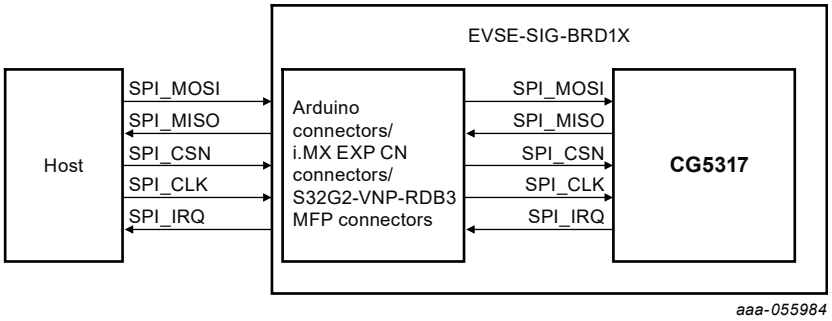


Figure 18. SPI connectivity between the CG5317 and the host

The SPI of CG5317 only supports mode 1 or mode 3, which can be selected by boot strap on EVSE-SIG-BRD. SPI mode 3 configurations in the device tree are decribed below:

```
spidev0: spi@0 {
      compatible = "lwn,bk4";
      reg = <0x0>;
      spi-cpha;
      spi-max-frequency = <8000000>;
      spi-cpol;
};
```

6.3.2 MII PHY interface

The CG5317 supports standard MII PHY and RMII PHY interfaces that cannot operate simultaneously. On EVSE-SIG-BRD, the MII PHY of CG5317 is connected to the SJA1110 switch. The RJ45 port of SJA1110 is

used to communicate with i.MX 93 EVK by default. The MII PHY interface is connected to the host controller, as shown in Figure 19.

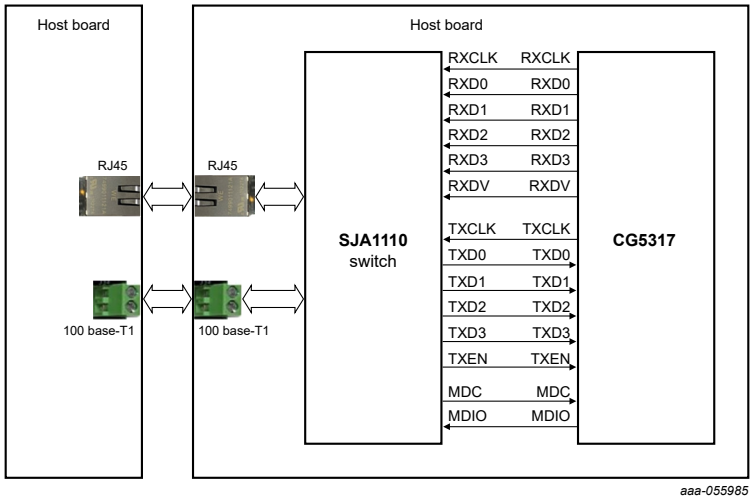


Figure 19. MII host interface of CG5317

6.3.3 Boot strapping CG5317

The CG5317 boot strap can be configured on EVSE-SIG-BRD. Table 5 lists the default configuration of the CG5317 boot strap.

Table 5. CG5317 boot strap on EVSE-SIG-BRD

Boot strap	STRAP 5	STRAP 4	STRAP 3	STRAP 2	STRAP 1	STRAP 0
Value	1	1	0	0	1	0
Description	PHY ADDRESS [0-2]			SPI MODE 3	Boot from host	Disable debugging

For boot strapping configuration, refer to *EVSE-SIG-BRD1X User Manual* (document UM12013).

Table 6. CG5317 boot strap configuration jumpers

Boot Strap name	Board Jumper positions	Description
STRAP[3-5] PHY ADDRESS[2-0]	J21 – Pins 1,2 short J19 – Pins 1,2 short J17 – Pins 1,2 short Default value "011"	This set of 3 pins defines the 3 LSBs of the PHY address, assigned for the MII port of CG5317
STRAP2 SPI_CLK_MODE	J18 – Pins 2,3 short Default value "0"	Controls the timing mode of the SPI bus based on the SPI_CLK polarity: Low – SPI mode 3, data is samples on clock falling edge High – SPI mode 1, data is sampled on the clock rising edge
STRAP1 BOOT_SRC	J20 – Pins 2,3 short	Controls the firmware load mode of CG5317 Low – Boot from Flash High – Boot from Host
STRAP0 UART DISABLE	J22 – Pins 1,2 short	Low – Enable debugging messages to UART

Table 6. CG5317 boot strap configuration jumpers*...continued*

Boot Strap name	Board Jumper positions	Description
		High – Disable debugging messages to UART

7 Security

The current cyber-threat landscape requires IoT devices to be properly provisioned and secured prior to deployment. Ensuring that hardware equipment is uniquely identified and managed throughout its lifecycle is mandatory for the protection of customers and industry verticals worldwide.

ISO 15118-2 mandates the use of TLS 1.2 for secure communication in V2G plug & charge. This ensures the confidentiality, integrity, and authenticity of data exchanged between electric vehicles and charging stations during the charging process.

Here is a breakdown of the main algorithms and their roles:

- Elliptic curve diffie-hellman ephemeral (ECDHE) is recommended for the key exchange process. ECDHE allows both parties to generate a shared secret key securely, which is used for encryption, without transmitting the key itself over the network.
- Elliptic curve digital signature algorithm (ECDSA) is used for the digital signing of messages. This ensures the authenticity and integrity of the messages exchanged between the EV and the charging station.
- ISO 15118-2 specifies the use of the AES for encrypting communication. Specifically, AES-128 in CBC mode or GCM for AES-128-GCM are mentioned. These algorithms are used to encrypt the data being exchanged to ensure confidentiality.
- Hash-based message authentication code (HMAC) combined with a secure hash algorithm (typically SHA-256 in the context of TLS 1.2) is used for message authentication and integrity checks. This ensures that the messages have not been altered in transit and are authentic.
- Random number generators (RNGs) are used for generating the unique random values used in the handshake process for secure session key establishment and for creating nonces.

In a V2G environment, particularly within the context of a CPO, the PKI tree is a hierarchical structure used to manage digital certificates and public-key encryption.

The PKI tree ensures secure communication between EVs, charging stations, and the backend systems of the CPO.

- Root certificate authority (Top level): At the top of the PKI tree is the root certificate authority (CA). This entity is trusted by all parties in the V2G ecosystem. It generates a root certificate used to sign the public keys of intermediate CAs. The Root CA's primary role is to establish trust within the network.
- Intermediate certificate authorities (Trust delegation): Below the root CA are one or two intermediate CAs. These CAs are certified by the root CA, and their role is to issue certificates to entities further down the tree. Intermediate CAs helps limit the exposure of the root CA and facilitates better management and revocation capabilities. ISO 15118 specifies that one or two sub-CAs are required to establish a chain of trust between a trust anchor (root) and the corresponding end-entity (leaf) certificate.
- CPO and charging station certificates (Operational level): At this level, certificates are issued to charge point operators and individual charging stations. These certificates enable secure communication between the charging stations and the vehicles, as well as between the charging stations and the CPO's backend systems. The CPO can have its own CA capabilities or use certificates issued by an intermediate CA.
- SECC certificate (End entity/leaf in the PKI tree): SECC certificates are used to authenticate the charging station (or the communication controller within it) to EVs during the V2G communication process. They ensure that the vehicle is connecting to a legitimate and secure charging station and enable encrypted communication between the EV and the charging station.
- Cloud certificate: End entity/leaf in the PKI tree: Cloud certificate is used for authenticating the device to the IoT hub. For more details, see https://learn.microsoft.com/en-us/azure/iot-hub/authenticate-authorize-x509.

Securing the private key and certificate of the SECC is crucial to maintaining V2G communication integrity and confidentiality, as outlined in ISO 15118. The security of these elements ensures that EV charging sessions are protected against unauthorized access, manipulation, and various cyber threats.

Similarly, securing IoT hub authentication credentials is crucial for data confidentiality and integrity between the device and Azure IoT Central application.

Secure elements such as SE050 are used to store and use SECC and CLOUD private keys and certificates. They provide a highly secure environment that prevents unauthorized access and use of the private key. SEs can perform cryptographic operations within the module, ensuring that the private key never leaves the device in plaintext.

SE050E supports the following cryptographic algorithms:

- ECC cryptographic support of extended set of ECC curves, including NIST (up to 521 bits key length), brainpool, twisted edwards, and montgomery
- RSA up to 4096 bits
- AES and 3DES encryption and decryption
- AES modes: CBC, CTR, ECB, CCM, GCM
- HMAC, CMAC, GMAC, SHA-1, SHA-224/256/384/512 operations
- HKDF, MIFARE KDF, PRF (TLS-PSK)
- TRNG compliant to NIST SP800-90B
- DRBG compliant to NIST SP800-90A
- Support of main TPM functionalities
- Support PKCS#11 specifications

EdgeLock SE050 supports a broad range of IoT security use cases:

- TLS connection
- Cloud onboarding
- Device-to-device authentication
- Device integrity protection
- Attestation
- Sensor data protection
- Wi-Fi credential protection
- Secure access to IoT services
- IoT device commissioning and personalization

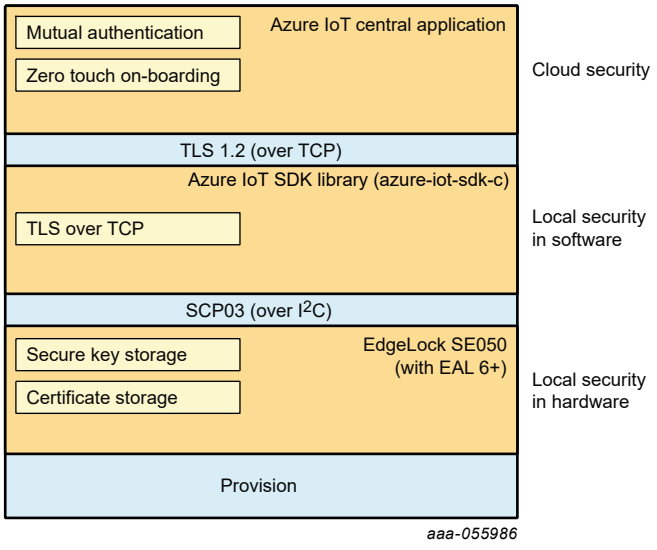


Figure 20. Security layers

For more information, see EdgeLock SE050: Plug and Trust Secure Element Family – Enhanced IoT security with high flexibility.

7.1 Credentials provisioning

This section presents two methods of provisioning the device:

- EdgeLock 2Go
- Manual

The EdgeLock 2Go provisioning method has the advantage of creating the credentials once and assigning them to multiple devices by using a device group. It also offers to set up the CN of the leaf certificates as a prefix, enabling users to have a unique CN per certificate. This is important when connecting to the Azure IoT Central application because it is a requirement from the cloud provider to have the `IOTCENTRAL_DEVICE_ID` from `~/cloud.conf` file the same as the CN of the X.509 certificate used when connecting.

7.1.1 EdgeLock 2Go provisioning

The provisioning process through EdgeLock 2GO is a one-time operation and must be executed using a single, common company account. For more details on creating a device group, secure objects, and provisioning the device with the credentials from the EdgeLock 2Go, see *EVSE-SIG-BRD1X User Guide* (document UG10109).

For more details on EdgeLock 2Go, see NXP website.

7.1.2 Manual provisioning

To provision the SECC credentials, run the `provisioning.sh` script that performs both online and offline provisioning.

```
# cd ~/res/se05x/
# ./provision.sh
```

For provisioning the cloud credentials, first run the script to create the credentials and then run the script for provisioning the SE050. For more details, refer to the *Plug & Trust MW Documentation* (document AN13030) that comes with the Plug&Trust MW package (se05x_mw_v04.05.00.zip).

Note: *If downloaded from nxp.com, the Plug & Trust MW Documentation may refer to an MW version newer than the one we are using (v04.05.00).*

The following commands are typed on the console of the i.MX 93 host processor used as the EVSE:

1. Generate the credentials.

```
root@imx93evk-easyevse:~# cd ~/.nxp-easyevse/cloud-manual-provisioning/provision
root@imx93evk-easyevse:~/.nxp-easyevse/cloud-manual-provisioning/provision# python3 GenerateAZURECredentials.py
##############################################################
#
#     SUBSYSTEM           : se05x
#     CONNECTION_TYPE     : t1oi2c
#     CONNECTION_PARAMETER : none
#
##############################################################
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
5001da2a7a048e8ff236
Generating Credentials Successfull
```

2. Inject the key and the certificate generated in the previous step into SE050.

```
root@imx93evk-easyevse:~/.nxp-easyevse/cloud-manual-provisioning/provision# python3 ResetAndUpdate_AZURE.py
#############################################################
#
#     SUBSYSTEM           : se05x
#     CONNECTION_TYPE     : t1oi2c
#     CONNECTION_PARAMETER : none
#
#############################################################
sss   :INFO :atr (Len=35)
      01 A0 00 00    03 96 04 03    E8 00 FE 02    0B 03 E8 00
      01 00 00 00    00 64 13 88    0A 00 65 53    45 30 35 31
      00 00 00
sss   :WARN :Communication channel is Plain.
sss   :WARN :!!!Not recommended for production use.!!!
5001da2a7a048e8ff236
key pair file: /home/root/.nxp-easyevse/cloud-manual-provisioning/provision/azure/377823485885254607041078_device_key.pem
Injecting ECC key pair at key ID: 0x83000001
Successfully Injected ECC key pair.
certificate file: /home/root/.nxp-easyevse/cloud-manual-provisioning/provision/azure/377823485885254607041078_device_certificate.cer
Injecting certificate at key ID: 0x83000002
Successfully Injected certificate.
Creating ECC Reference key from key ID: 0x83000001
Successfully Created reference key at: /home/root/.nxp-easyevse/cloud-manual-provisioning/provision/azure/377823485885254607041078_device_reference_key.pem
#############################################################
#                                                           #
#     Program Completed Successfully                        #
#                                                           #
#############################################################
```

By default, the reset option is commented. For resetting the SE050, uncomment the following line from `Provision/ResetAndUpdate_AZURE.py`.

```
35
36      # Reset the Secure Element
37      #reset(session_obj)
38
39      # Inject ECC key Pair
40      status = set_ecc_pair(session_obj, keypair_index_private, ecc_key_pair_file)
41      if status != apis.kStatus_SSS_Success:
42          return STATUS_FAIL
43
```

7.1.3 Verify credentials

The following Linux commands are typed on the console of the i.MX 93 host processor used as the EVSE:

1. Confirm that V2G keys and certificates are properly provisioned into the SE050 device.

```
# pkcs11-tool --module /usr/lib/libsss_pkcs11.so \
--list-objects2>/dev/null|grep"sss:80E0E0E" -A1 -B2
CertificateObject;type= X.509 cert
label: sss:80E0E0E3
subject: DN: C=FR, O=NXP, CN=V2GRootCA, DC=SCE/
emailAddress=marouene.boubakri@nxp.com
--
CertificateObject;type= X.509 cert
label: sss:80E0E0E2
subject: DN: C=FR, O=NXP, CN=V2GSubCA1, DC=SCE/
emailAddress=marouene.boubakri@nxp.com
--
CertificateObject;type= X.509 cert
label: sss:80E0E0E1
subject: DN: C=FR, O=NXP, CN=SECCCert, DC=SCE/
emailAddress=marouene.boubakri@nxp.com
--
PrivateKey Object; EC
label: sss:80E0E0E0
ID: e0e0e080
```

2. Ensure that the Azure cloud key and certificate have been correctly provisioned into the SE050 device.

```
# pkcs11-tool --module /usr/lib/libsss_pkcs11.so \
--list-objects 2>/dev/null | grep "sss:830000" -A1 -B2
CertificateObject; type = X.509cert
```

```
label: sss:83000002
subject: DN: CN=easyevsempu-0000000000b3be39-0001
--
PrivateKey Object; EC
label: sss:83000001
ID:01000083
--
EC_POINT: 044104f3c7fbc48f6094b346d22e408686b2528de40287247ec9710522ac7165
f6b0344b139239291e7579339ce6775161b0d8daa5cebc51e987e320997dbe3f3c8b65
EC_PARAMS: 06082a8648ce3d030107
label: sss:83000001
ID:01000083
```

In `~/cloud.conf`, the **`IOTCENTRAL_DEVICE_ID`** field is the CN from the output of the CLOUD certificate provisioned into the SE050 (in this example, from the output above `:CN=`**`easyevsempu-0000000000b3be39-0001`**). For more information, see *EVSE-SIG-BRD1X User Guide* (document UG10109).

7.1.4 Upload root CA to Azure IoT Central application

Section 8.1 presents how to select an enrollment and authentication-attestation method. In this document, we exemplify group enrollment using an x.509 certificate stored into SE.

A root certificate authority certificate is used as the basis of trust and therefore added to Azure IoT Central. The device certificates can then be used to connect via the enrollment group.

For EdgeLock 2Go provisioning method, the intermediate CA has to be downloaded from https://edgelock2go.com and uploaded to the Azure IoT Central as presented in the user guide.

For a manual provisioning method, the Root CA generated in the previous section can be found in the `~/.nxp-easyevse/cloud-manual-provisioning/provision/azure` directory and has to be uploaded to the Azure IoT Central. For more details on how to upload the certificate, see *EVSE-SIG-BRD1X User Guide* (document UG10109) and Generate root and device certificates.

7.2 Transport layer security (TLS)

TLS is an industry standard designed to provide identification, authentication, confidentiality, and integrity of the communication between two endpoints. Every TLS connection begins with a TLS handshake protocol that manages the cipher suite negotiation, the client, and server authentication, and the session key exchange. It consists of:

- The hello phase, where both parties negotiate the protocol version and cipher suite.
- The client and server key exchange phase.

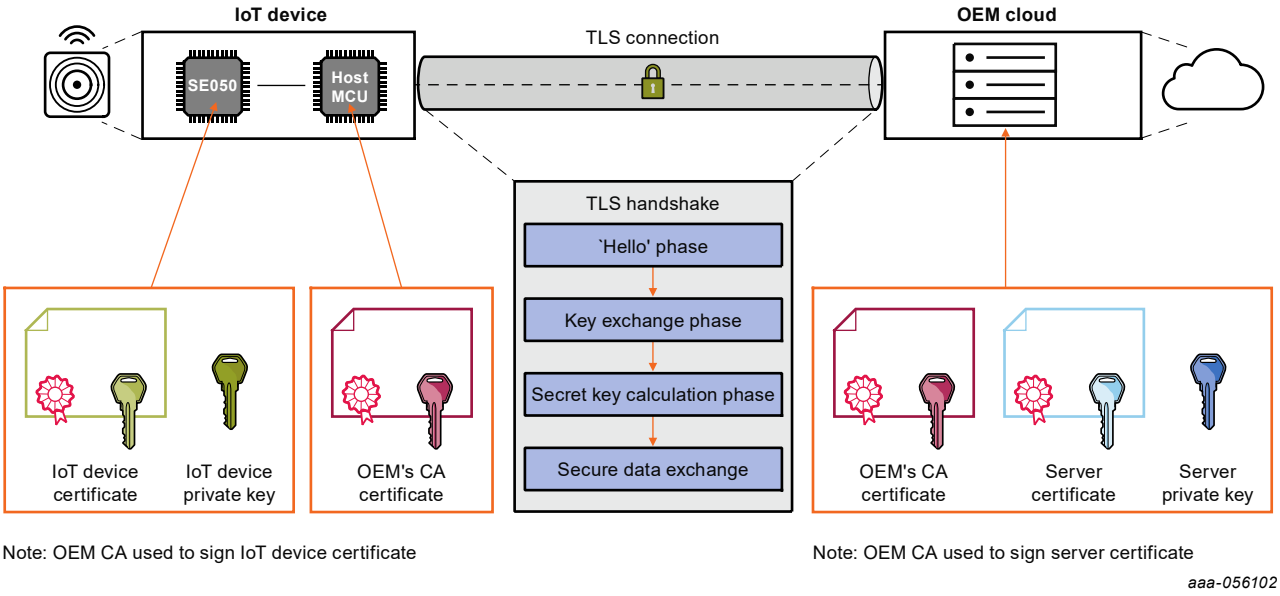


Figure 21. TLS handshake

- The session secret key calculation phase, where a pre-master secret and exchanged random values are used to calculate a session key for securing communication.

7.2.1 Hello phase

The TLS handshake begins by sending a *client_hello* message. The IoT device sends the *client_hello* message and includes its supported cipher suites. It comprises three distinct algorithms:

- The key exchange and authentication algorithm used during the handshake. For example, `TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256`.
- The encryption algorithm is used to encipher data. For example, `TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256`.
- The MAC algorithm is used to generate the message digest. For example, `TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256`.

In addition, the *client_hello* message also includes a random number. This random number must be requested to the EdgeLock SE05x security IC. The server responds with a *server_hello* message, which contains the cipher suite chosen, the session ID, and another random number.

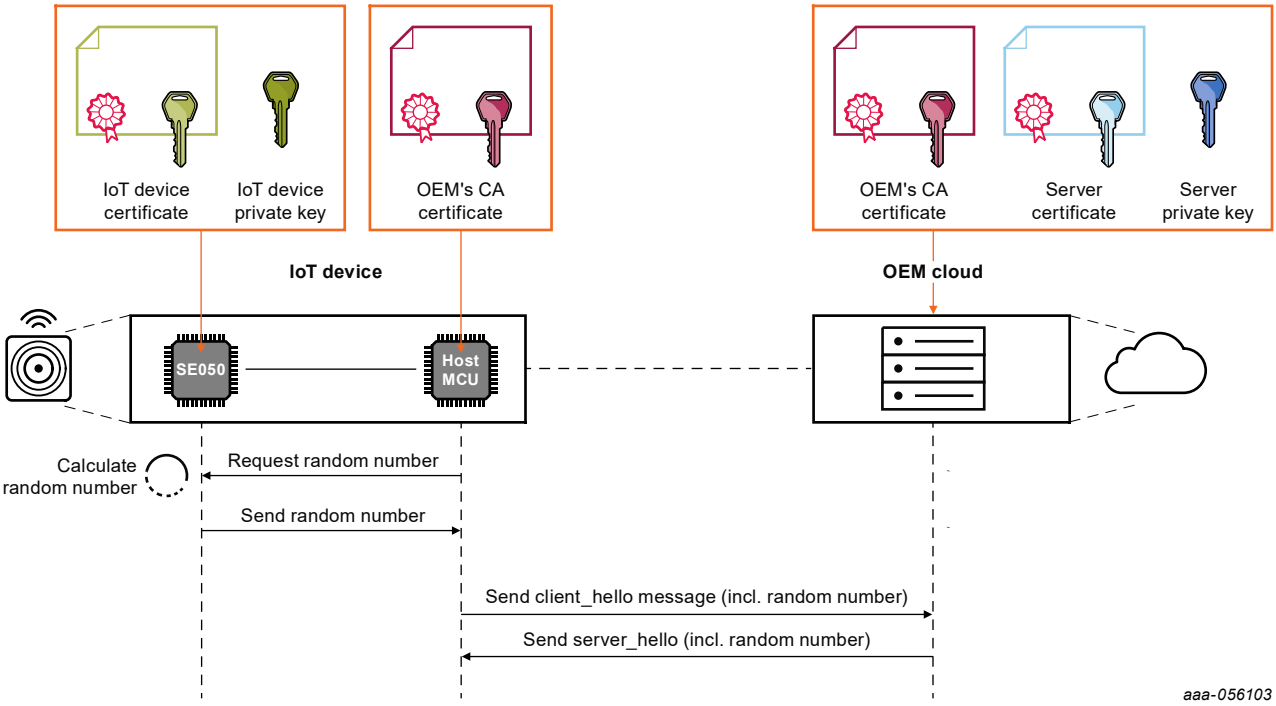


Figure 22. TLS handshake hello phase

7.2.2 Server key exchange phase

For the key exchange from the server side, the server sends:

- A *server_certificate* message, capable of carrying the whole server certificate chain (leaf certificate and CA certificate).
- A *serverKeyExchange* message, containing the ephemeral ECDH public key and a specification of the corresponding curve. These parameters are signed with ECDSA using the private key corresponding to the public key in the server certificate.
- A *client_certificate_request* message, which makes client authentication mandatory. This option is recommended to avoid unauthorized devices connecting to the IoT network.

The IoT device verifies the validity of the server certificate chain. It then uses the public key in the server certificate to verify the ECDSA signature of the parameters received in the serverKeyExchange message. EdgeLock SE05x is leveraged for verifying the ECDSA signature.

A valid signature proves the server identity.

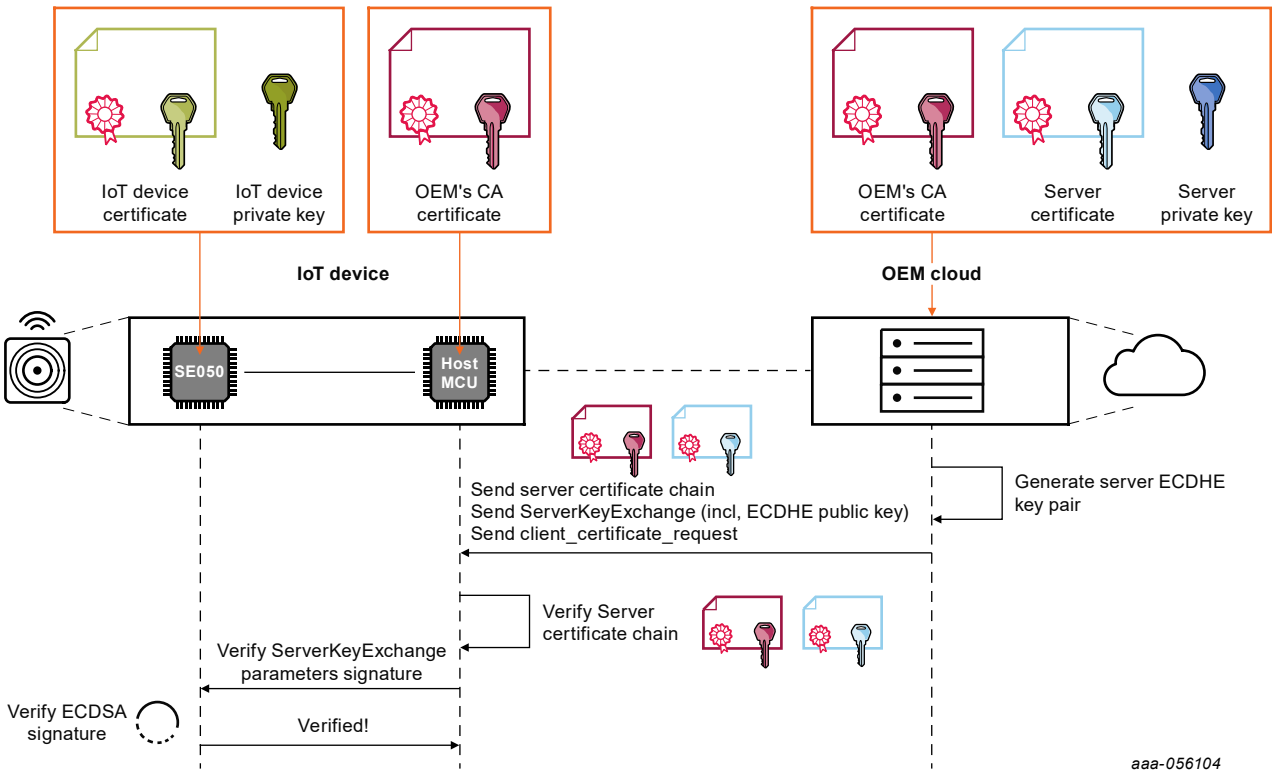


Figure 23. Server key exchange phase

7.2.3 Client key exchange phase

For the key exchange from the client side, the IoT device sends:

- A *client_certificate* message, capable of carrying the whole client certificate chain (leaf certificate and CA certificate).
- A proof of possession message, including a signature used to prove that the private key possesses the IoT device. The signature is performed in the EdgeLock SE05x using the IoT device-private key.
- A *client_key_exchange* message, including an ECDH key public generated on the same curve as the server ephemeral ECDH key. The ECDHE ephemeral keys are generated in the IoT device MCU.

The server verifies the IoT device certificate chain and uses the IoT device public key in the client certificate to verify the proof of possession.

By performing this operation, the server verifies that the private key possesses the IoT device corresponding to the public key in the client certificate.

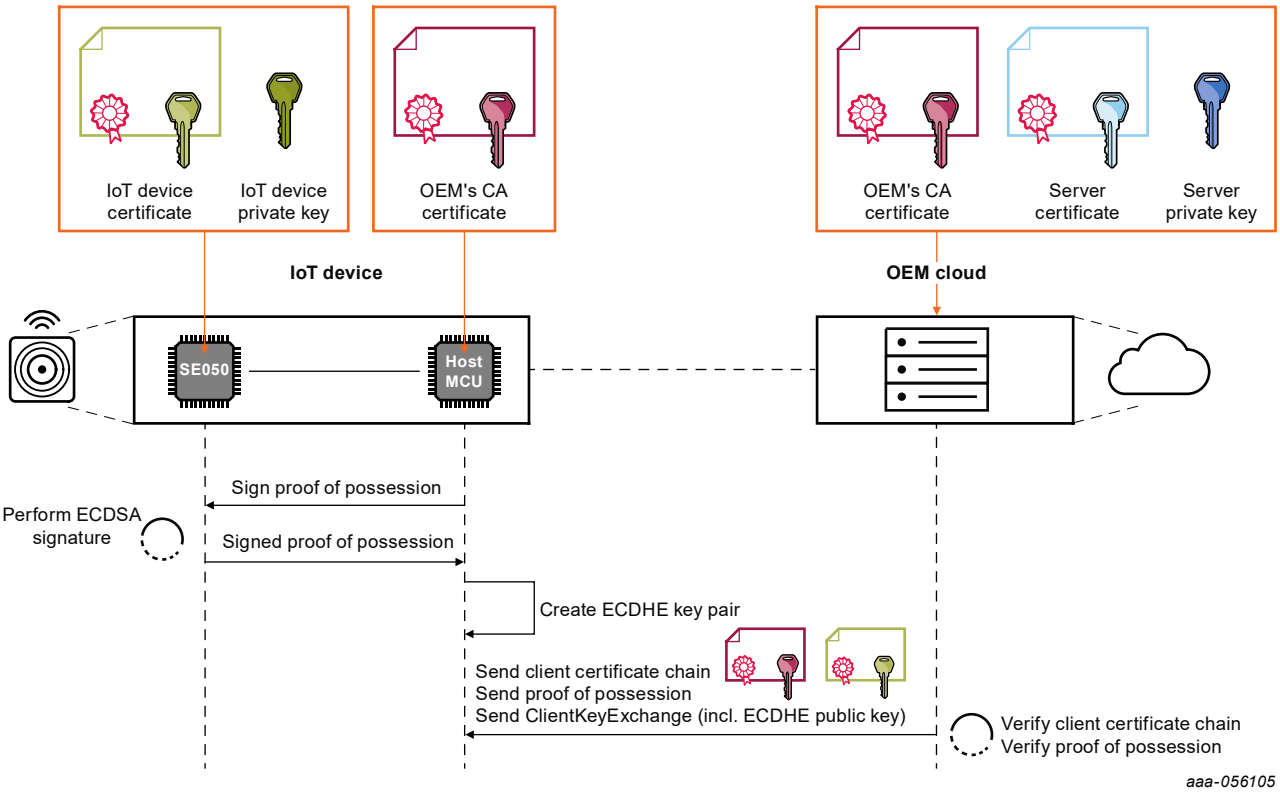


Figure 24. Client key exchange phase

7.2.4 Secret key calculation phase

Both client and server perform an ECDH operation. The result is used as an input to compute the pre-master secret. The EdgeLock SE05x is in charge of calculating both the pre-master and master secrets. The master secret is calculated using:

- The pre-master secret
- The client and server-random numbers
- An identifier label

Here, the shared secret key possesses both the IoT device and the cloud, and can start secure exchange of data using a symmetric cryptographic algorithm.

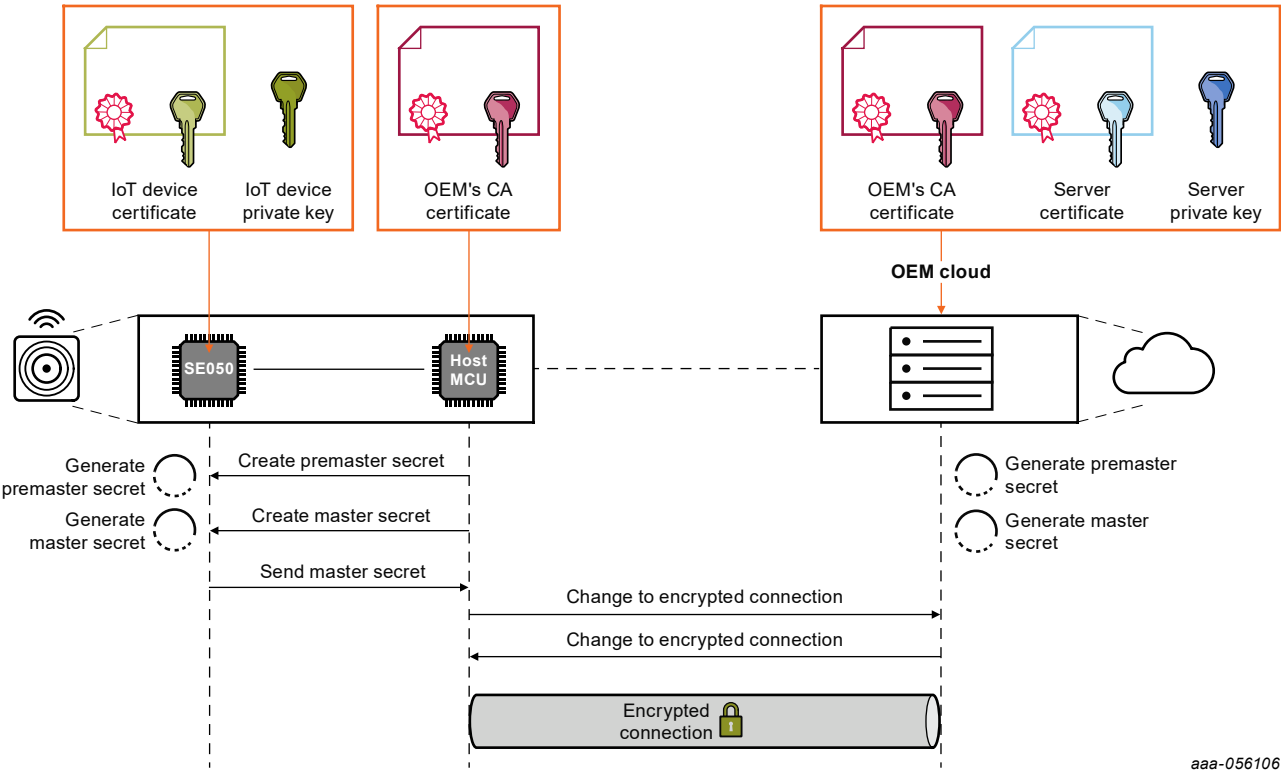


Figure 25. Secret key calculation phase

7.3 Architecture Overview

In the application, cryptographic operations may leverage two different PKCS11 tokens and/or slots because SE050 does not serve as cryptographic accelerators. It can only process a limited amount of data < ~1 kb. SE050 is primarily used for TLS authentication with long-life credentials stored within. To accommodate operations that exceed the SE's data handling capabilities or require faster processing, these tasks are offloaded to either a software (SW) PKCS11 provider or a more capable hardware (HW) token. Accordingly, two PKCS#11 slots are configured using p11-kit-proxy:

- SLOT1: Configured to execute ECDSA signing on the SE050 using the SECC private key.
- SLOT0: Configured to execute the remaining operations on the TEE.

If a single token can efficiently handle all operations required by the application, it is also possible to align both constants to the same token/slot.

Also, for instances requiring enhanced processing speed over security, the slot index can be configured to point to SoftHSMv2 at SLOT2.

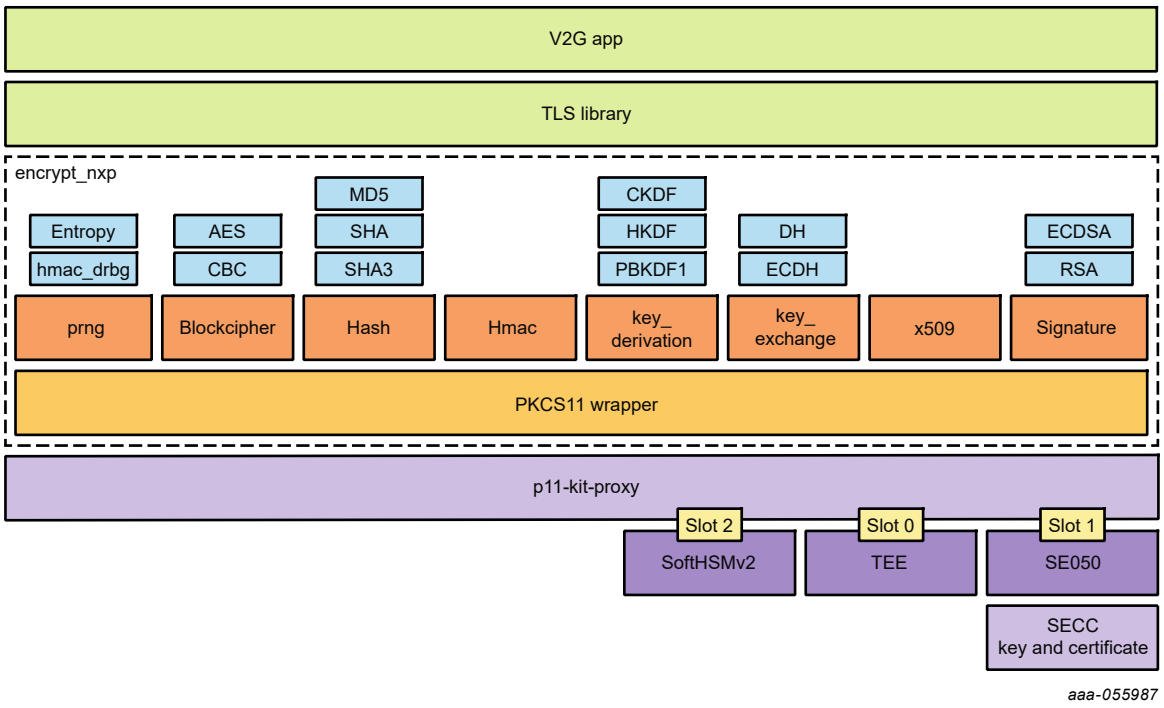


Figure 26. Architecture overview

7.4 EV/EVSE TLS handshake

In the context of replacing the SW crypto implementation with an HW token, make sure what security parameters must reside in an HW token and what can be visible to SW (the ISO15118-2 stack).

Table 7. EV/EVSE TLS handshake

Step	Security parameters	Location
The client sends a "Client Hello" message, which includes the TLS version. It supports a list of supported cipher suites and a random byte string.	TLS version, cipher suites, random byte string	SW can manage the random byte string, but, we can prefer an HW token for secure RNG. SLOT 0.
The server responds with a "Server Hello" message, selecting the TLS version and cipher suite, and sends its own random byte string.	TLS version, cipher suites, random byte string	Can be in SW but we can prefer an HW token for secure RNG. SLOT 0.
The server sends its certificate to the client (if it is a TLS server).	Digital certificate (includes the server's public key)	Digital certificates are public information. Can be managed by SW, stored in filesystem, can be also stored in HW token.
The server can send a "ServerKey Exchange" message, which contains cryptographic parameters necessary for the client to establish a pre-master secret.	Cryptographic parameters, for example, Ephemeral keypair for ECDH	Private key must be securely stored in a token. SW can manage the public key.
The server sends a "ServerHelloDone" message indicating that it is finished with handshake negotiation.	None	None

Table 7. EV/EVSE TLS handshake*...continued*

Step	Security parameters	Location
The client responds with a "ClientKey Exchange" message, which includes the pre-master secret encrypted with the server's public key.	The pre-master secret is highly sensitive as it's used to generate encryption keys.	Must be securely stored in a token.
The client can send its certificate if mutual authentication is required (for ISO15118-20).	Client's private key	Must be securely stored in a token.
Both the client and server send a "ChangeCipherSpec" message to signal that subsequent messages are encrypted.	None	None
Both client and server exchange "Finished" messages, verifying that the key exchange and authentication processes were successful.	Session keys derived from the pre-master secret.	Must be securely stored in a token.

7.5 Cloud TLS connection

When using DPS, two TLS connections are negotiated. Section 7.5 shows both connections. The "Client Hello" message is sent to the DPS endpoint and then to the Azure IoT Central application endpoint.

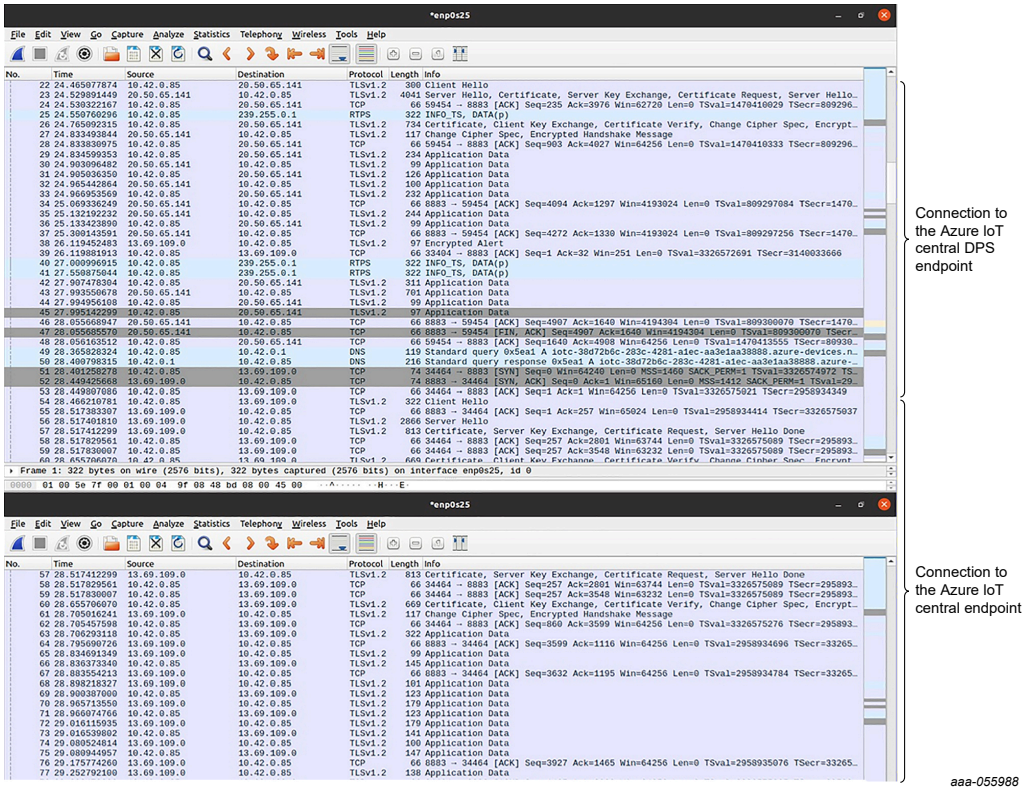


Figure 27. Wireshark output

8 IoT and connectivity

This chapter briefly explains the connection setup between EasyEVSE and the Azure IoT Central application. As a prerequisite, you must first deploy your Azure IoT Central application using either the "EasyEVSE Dashboard shareable link" template (which is included with the project) or your own version derived from EasyEVSE's device template (included as well in JSON format).

Note: *Auxiliary services are covered in the User Guide (Event hub, telemetry, optional cloud storage, cloud shell)*

Note: *Most security features can be independently tested, having no hard requirement on peripherals.*

Details about creating an Azure IoT Central application can be found in *EVSE-SIG-BRD1X User Guide* (document UG10109). Figure 28 describes the components necessary for successfully connecting the device to the Azure IoT Central application.

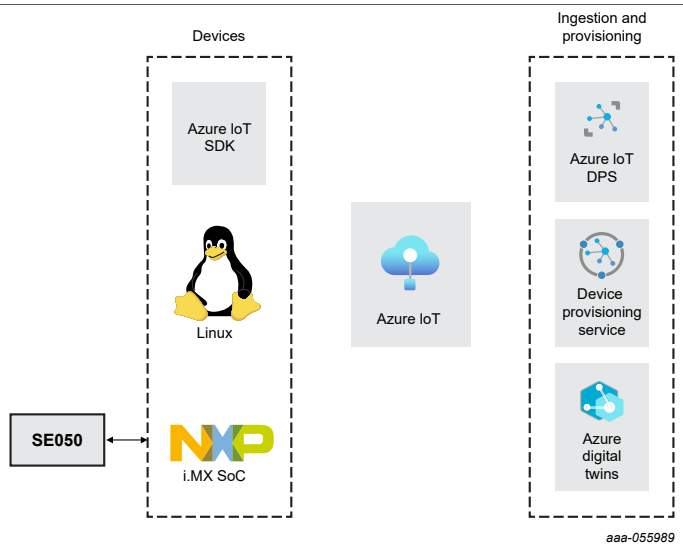


Figure 28. Simplified EasyEVSE reference architecture

Note: *Throughout this document, multiple pointers related to the cloud setup are directed toward the cloud service provider, or CSP (Microsoft Azure in this case).*

8.1 Connection setup

Azure IoT Central needs to enroll EasyEVSE as a device for telemetry, commands, and property updates. To achieve this, connect the device to your IoT Central application, as described in the user guide.

This section lists the steps to provision the EasyEVSE device and exchange data with the Azure DPS service.

8.1.1 Select enrollment and authentication-attestation method

Select an individual or a group enrollment. The EasyEVSE device supports both enrollment types. This configuration can only be made in the Azure IoT Central application.

- Group enrollment: It represents a group of devices that share a specific attestation mechanism. Group enrollments are recommended to enroll multiple devices with the same set of credentials. NXP recommends using an enrollment group for many devices, which share a desired initial configuration, or for devices going to the same tenant.
- Individual Enrollments: It represents an entry for a single device that may register with the DPS. Individual enrollments may use either X.509 certificates or SAS tokens (in a real or virtual TPM) as attestation

mechanisms. They are ideal to enroll a single device with its own credentials. NXP recommends using individual enrollments for devices that require unique initial configurations and for devices that can only use SAS tokens via TPM or virtual TPM as the attestation mechanism.

In this document, we exemplify group enrollment using the X.509 certificate stored into SE050.

Note: *X.509 certificate chain generation (including root, intermediate, device) via OpenSSL is comprehensively covered by Microsoft. For more details, see Tutorial: Provision multiple X.509 devices using enrollment groups. For more details about the enrollment options, see Manage device enrollments in the Azure portal.*

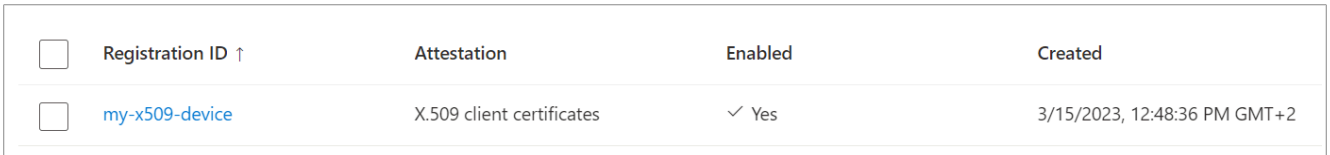


Figure 29. X.509 certificate upload

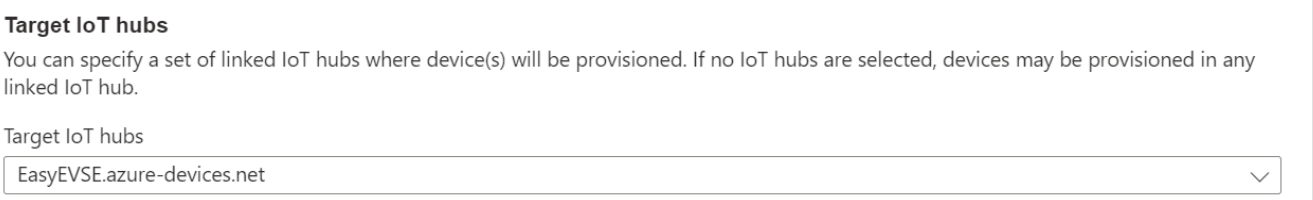


Figure 30. IoT hub target

Note: *Multiple IoT hubs can be linked to a single DPS service. For more details, see How to link and manage IoT hubs.*

Note: *If the device has not been enrolled before, DPS helps to enroll the device into your Azure IoT Central application. Once the operation is successful, the device can communicate with the Azure IoT Central application without relying on DPS.*

Note: *Note your unique scope ID, as it is necessary when connecting the device to the Azure IoT Central application. For more details, see IoT Hub Device Provisioning Service terminology.*

The X.509 certificate authentication-attestation method is the recommended method for production devices. It can be provisioned into the secure element SE050, providing increased security and ensuring mutual authentication.

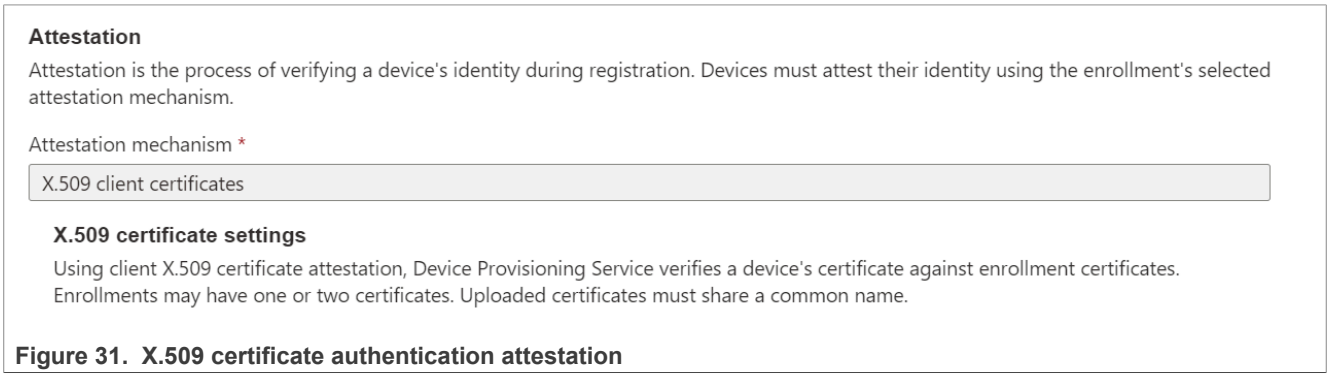


Figure 31. X.509 certificate authentication attestation

Note: *Detailed configuration steps for creating a group enrollment are available in the user guide.*

8.1.2 DPS onboarding

DPS is an automatic provisioning system for new devices. It eliminates human intervention, namely zero-touch provisioning. A complete description of the DPS, its benefits and use case scenarios can be found in the following article What it is azure IoT Hub Device Provisioning Service? at the Azure documentation.

Figure 32 outlines the role of X.509 certificates as part of the attestation and authentication inputs for the DPS.

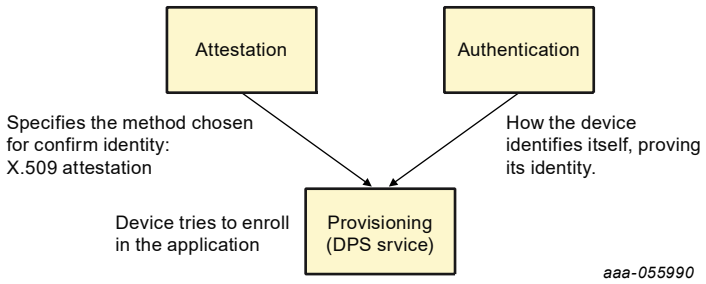


Figure 32. DPS inputs

By default, the EasyEVSE uses the Global device provisioning endpoint: `global.azure-devices-provisioning.net`.

Many devices and external entities foreign to the EasyEVSE that also require the DPS service can use the endpoint. To prevent the provisioning secrets from leaving the original endpoint, you can specify private endpoints. This is necessary because the Azure IoT hub conducts traffic load balancing when using the global DPS endpoint. For details, see the note in When to use Device Provisioning Service.

Figure 33 outlines the general DPS onboarding followed by devices when attempting to enroll into the Azure IoT Central application.

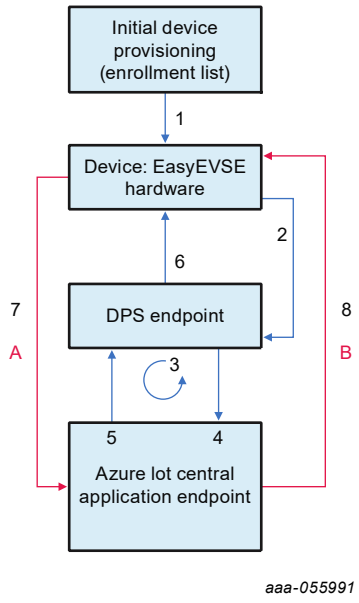


Figure 33. DPS onboarding

The following list explains Figure 33:

1. The device manufacturer adds the device registration information to the enrollment list in the Azure portal.
2. The device contacts the DPS endpoint. The device passes the identifying information to DPS to prove its identity.

3. DPS validates the identity of the device by validating the registration ID and key against the enrollment list entry using standard X.509 verification (X.509).
4. DPS registers the device with an IoT hub and populates the desired twin state of the device. For details on twin state, see Understand and use device twins in IoT Hub.
5. Azure IoT Central returns device ID information to DPS.
6. DPS returns the IoT Central connection information to the device. The device can now start sending data directly to the IoT Central application endpoint.
7. The device connects to the IoT Central application.
8. The device gets the desired state from its device twin in IoT Central.
9. Once it has been registered, The device directly connects to the IoT Central application.
10. The device gets the desired state from its device twin in IoT Central.

In Figure 33, steps A and B correspond to the case when the device does not need to contact the DPS anymore as it is already enrolled. Therefore, the connection scheme is simpler because the device cannot directly connect to the Azure IoT application host name endpoint also referred to as host name.

8.1.3 Connect state machine

Getting the EasyEVSE connected to the cloud takes several steps, as shown in Figure 34.

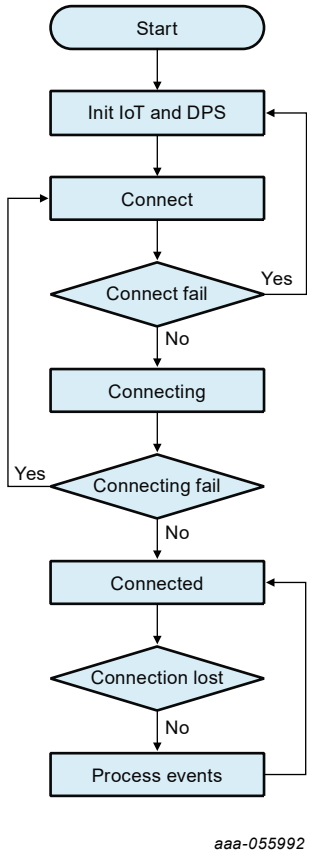


Figure 34. Connect state machine flowchart

If the device is set to enroll to the Azure IoT Central application (`IOTCENTRAL_DEVICE_SECURITY_TYPE` is set to DPS in ~/`cloud.conf`), the IoT connection settings are initialized as well as the DPS.

After the initialization is complete and successful, the actual connection of the EasyEVSE to the Azure IoT Central application begins. A robust state machine is implemented to manage device connection to your Azure IoT Central application or to the DPS service when enrolling the device.

After the initialization completes and the state machine sets into the connected state, the device can exchange bidirectional communication with the Azure IoT Central application. This involves IoT functions such as telemetry send, direct method, device twin, and so on.

8.1.4 Connected state

Reaching the connected state means that the EasyEVSE is connected to the Azure IoT Central application. The EasyEVSE remains connected unless a disconnect event is set.

Note: *The connected state is the most important state of the state machine as it manages bi-directional communication between the EasyEVSE and the Azure Iot Central application.*

The following is a list of the functions performed in the connected state.

- Direct method
- Device twin
  - Request device twin properties (already request properties when executing this state)
  - Receive/get device twin properties
  - Receive/get desired device twin properties
  - Report/send device twin properties
- Telemetry sent
  For better comprehension, you can associate the direct method, device twin, and telemetry functions to the options in the EasyEVSE dashboard.
  - Direct method is associated with any command sent to the EasyEVSE. See the Commands tab:
    - Terminate charge
  - Device twin is associated with any property that is updated to the EasyEVSE. See the Properties tab:
    - Grid Power Limit
    - Tariff Cost
    - Tariff Rate
  - Telemetry send is associated with the EasyEVSE data sent to the dashboard. See the View tab:
    - All the variables sent are shown in the telemetry send section of the EasyEVSE dashboard under the *View* tab.

8.2 Implementation details

The device cloud app functions as a C socket client that operates on the MPU side. It receives telemetry data from the server and forwards it to the cloud application in Azure IoT Central. The app also receives parameters and commands from the cloud application to pass on to the server, allowing the device to be remotely controlled and configured. It uses the Azure IoT C SDK and Libraries to establish communication with the Azure IoT Central application.

The device cloud app expects that a file named ~/`cloud.conf` exists and that it is populated with the following information:

`IOTCENTRAL_DEVICE_SECURITY_TYPE=DPS`

`IOTCENTRAL_DEVICE_ID=`<DEVICE_ID is the CN of the CLOUD certificate stored in SE050E>

`IOTCENTRAL_SCOPE_ID=`<SCOPE_ID assigned to the device provisioning service>

`IOTCENTRAL_CERT_ID=`<CERT ID of the cloud certificate stored in the SE050E>

`IOTCENTRAL_KEY_ID=`<KEY ID of the cloud key stored in the SE050>

`IOTCENTRAL_MODEL_ID=`<DTMI of the device template created in Azure IoT Central>

The ID scope can be found in the Azure IoT Central application as presented in Figure 35.

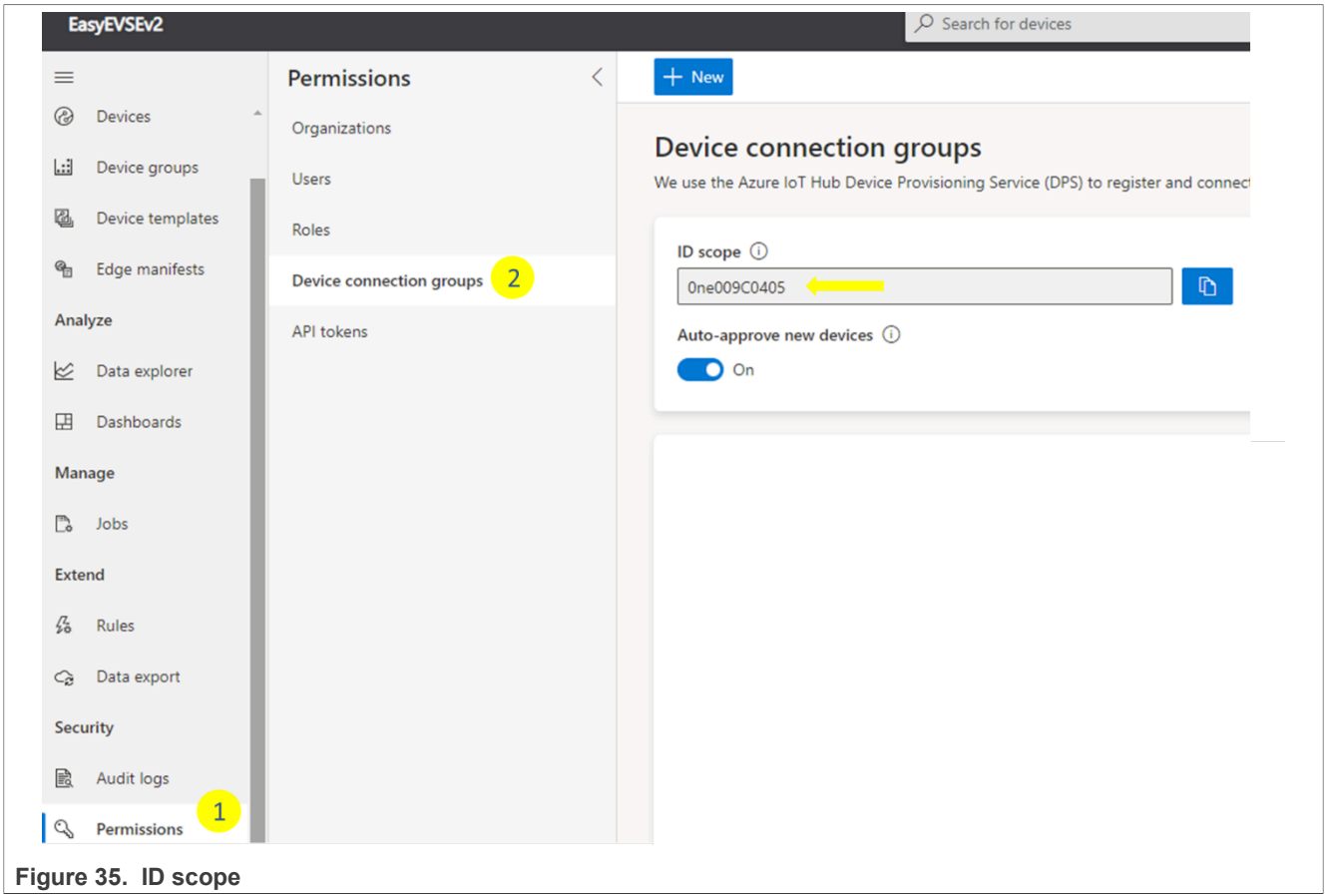


Figure 35. ID scope

The "IOTCENTRAL_DEVICE_ID" is the CN of the certificate provisioned into the SE050. More details can be found in the user guide.

The "IOTCENTRAL_MODEL_ID" corresponds to the IoT application dashboard and can be found in the Azure IoT Central application. For more details, see device templates.

The application can be separated into four main parts, as can be seen in Figure 36. The first part gets the information from ~/`cloud.conf` to provision the device and to generate a connection string if it is the first execution. After that, it uses the generated connection string for the next connections. After this authentication step, the app initializes the library, retrieves the device twin from the cloud and configures callbacks for future properties updates and for the command handle. The device twin is an individual JSON file for one device that keeps some information from it, including the desired properties configured in the dashboard by the user. The second part initializes the client socket. The third part is the main loop responsible for pushing and receiving data from and to the cloud. The fourth part is the "parallel" and non-deterministic: the callbacks that can be triggered at any moment.

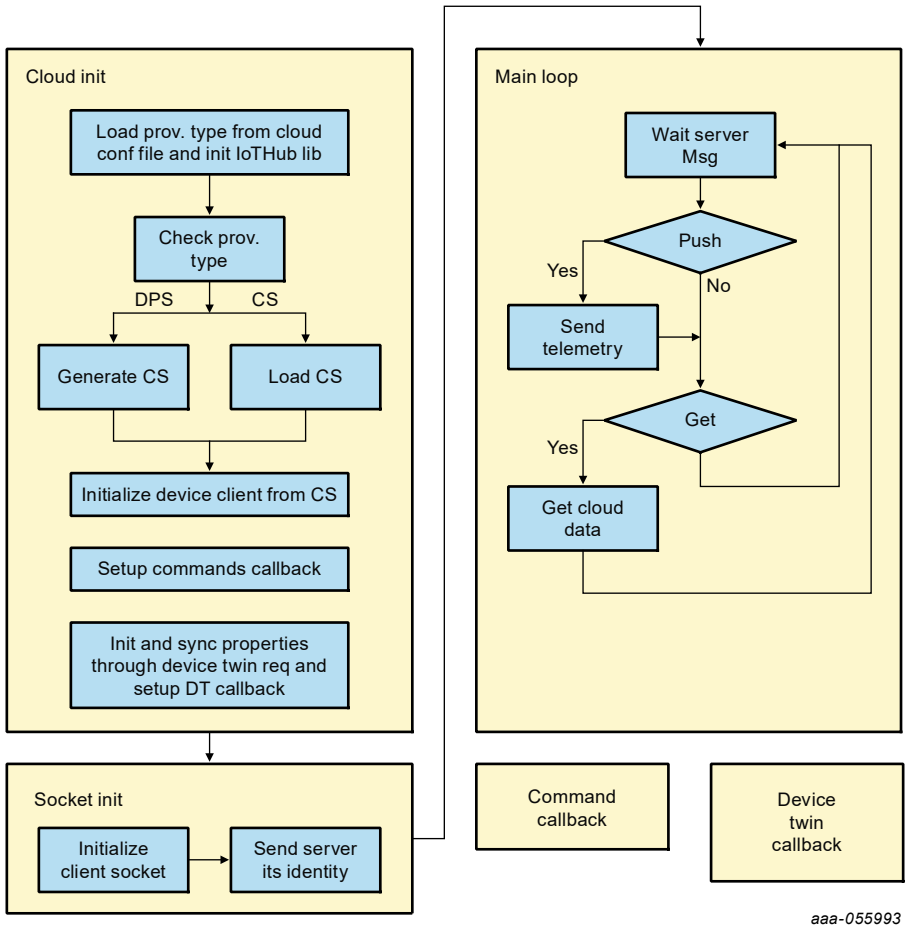


Figure 36. Cloud communication flow

The following snippet shows the code from the main routine where the three main parts are highlighted.

```
int main(int argc, char *argv[])
{
struct tm dt;
int retCode;

// setup signal handling
HandleSignal(SIGUSR1, CloudDeinit);

// prepare the logging file structure based on evse.conf file parsed by server
if (argc == 2)
{
logLevel = atoi(argv[1]);
if (logLevel > NONE)
{
PrepareLoggingEnv(identity);
}
}

printf("[CLOUD] Starting Cloud Client\n");

retCode = InitCloud();
if (retCode == -1)
{
```

```
CloudDeinit(RUNTIME_ISSUE);
return retCode;
}

endpointFd = InitSocket(identity);
if (endpointFd == -1)
{
CloudDeinit(RUNTIME_ISSUE);
return endpointFd;
}

retCode = SendIdentity(endpointFd, identity);
if (retCode == -1)
{
CloudDeinit(RUNTIME_ISSUE);
return retCode;
}

UpdateCycle(endpointFd, identity);

CloudDeinit(RUNTIME_ISSUE);

return 0;
}
```

The following snippet shows the process of retrieving the variable from *~/cloud.conf* to define if it generates the CS using DPS or uses the previously generated one. In this part, `IoTHub_Init()` is also called to initialize the IoT hub device library and `IoTHubDeviceClient_CreateFromConnectionString()` specifying the CS and the communication protocol (MQTT).

```
int InitCloud()
{

    // Initializes the IoT Hub Client System.
    char errorBuffer[ERRNO_MAX_SIZE];
    int ret;
    protocol = MQTT_Protocol;
    char *reported_properties_message = NULL;

    // Load the type of provisioning
    LoadVariable(g_securityTypeEnvironmentVariable,
 credentials.provisioningType, sizeof(credentials.provisioningType));
    printf("[CLOUD] [PROV TYPE] %s\n", credentials.provisioningType);

    ret = IoTHub_Init();
    if (ret != 0)
    (…)
    if (strcmp(credentials.provisioningType, "DPS") == 0)
    {
        prov_transport = Prov_Device_MQTT_Protocol;
        ret = InitDPS();
        if (ret == -1)
        {
            return ret;
        }
        GenerateCS();
    }
    else if (strcmp(credentials.provisioningType, "connectionString") == 0)
    {
```

```
        // Load the CS
        LoadVariable(g_deviceCSEnvironmentVariable,
credentials.connectionString, sizeof(credentials.connectionString));
        printf("[CLOUD] [cs] %ld %s\n", strlen(credentials.connectionString),
credentials.connectionString);
    }
    else
    {
(…)
        return -1;
    }
    device_handle =
IoTHubDeviceClient_CreateFromConnectionString(credentials.connectionString,
protocol);
```

Following the connection, a callback is configured report connection events. If a connection is established successfully or not and a callback for receiving and handling remote method invocation from the cloud, aka, commands from the cloud.

```
    // Setting connection status callback to get indication of connection to
iothub
    ret = IoTHubDeviceClient_SetConnectionStatusCallback(device_handle,
connection_status_callback, NULL);
    if (ret != IOTHUB_CLIENT_OK)
    {
        (..)
    }

    // Set method invocation callback
    ret = IoTHubDeviceClient_SetDeviceMethodCallback(device_handle,
deviceMethodCallback, NULL);
    if (ret != IOTHUB_CLIENT_OK)
    {
        (..)
    }
```

Details about starting the EVSE Server application can be found in the user guide.

Figure 37 shows the output of the cloud application upon successful connection.

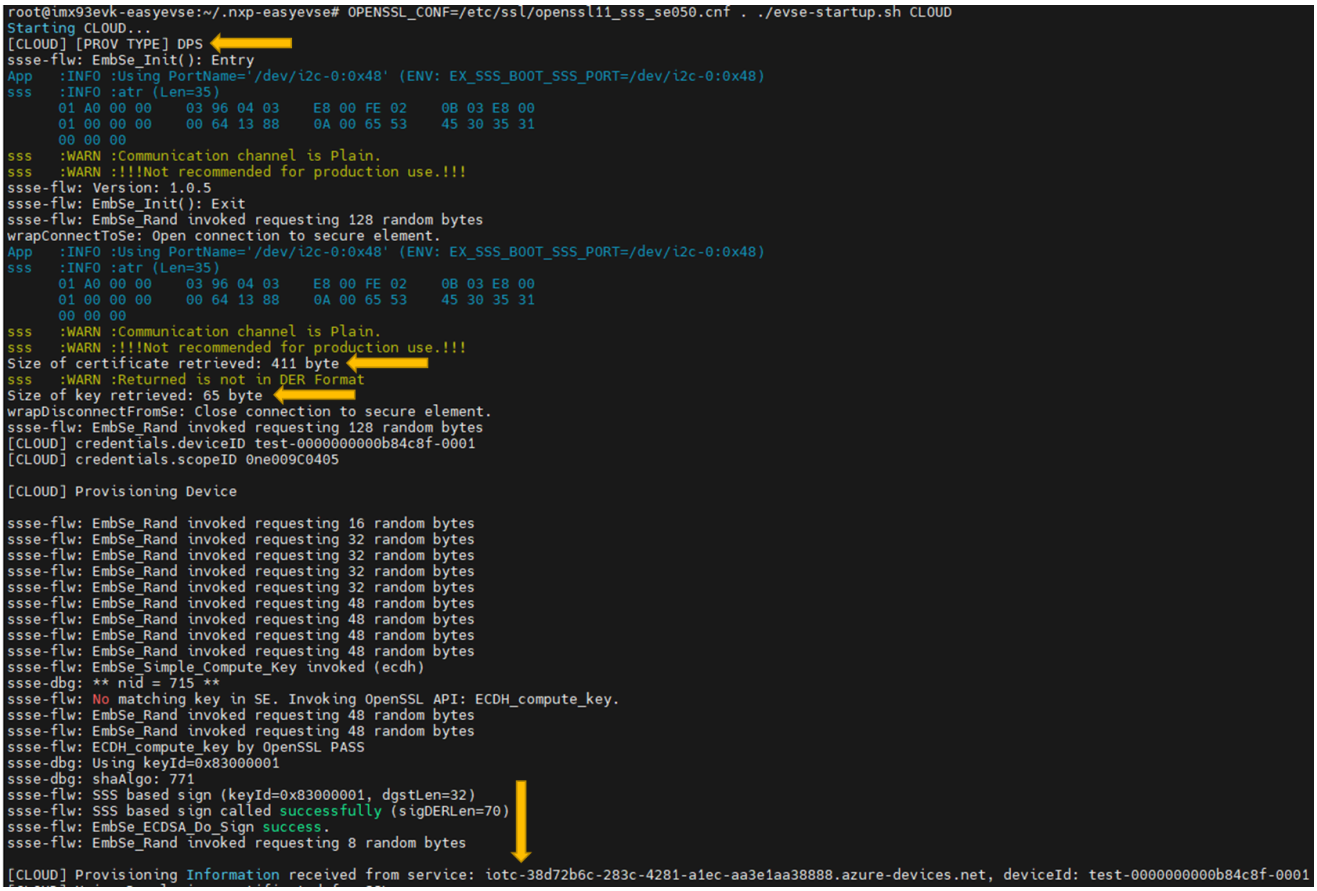


Figure 37. Successful connection output

When the connection is successful, the `~/cloud.conf` file gets automatically updated. The `IOTCENTRAL_DEVICE_SECURITY_TYPE` is set to `connectionString` instead of DPS and the `IOTCENTRAL_DEVICE_HUB_URI` is updated with the connection string received.

Figure 38. Cloud configuration file after successfully connecting

Properties are used for configuring devices remotely by defining a set-point. In the EVSE solution, there are three **properties: *Grid Power Limit*, *Tariff Cost* and *Tariff Rate*** that are configurable in the dashboard by the user. The user can configure it before the device is provisioned or after it is connected. The device is updated irrespective of the time. In the following code snippet, two structs are initialized. One is intended to store the desired properties (the values that come from the cloud). The other one stores the reported properties (the values that were updated in the device due to a desired property change in the cloud). The latter one is used to report the changes in the cloud, so the user can ensure that the changes were propagated.

For retrieving the current desired properties already set on the client, or to check if they are set, `IoTHubDeviceClient_GetTwinAsync` is called. This function has an "asyns" behavior, in the sense that it answers immediately, and it only has the purpose to issue a get twin requisition and register a callback to wait for it. Therefore, there is a polling while waiting for the device twin to be checked.

After this polling is released, `UpdateSerializeReportedMessage` is called to assign the new values to the reported properties and to wrap it in a JSON format, which is sent back to the cloud through `IoTHubDeviceClient_SendReportedState`. Finally, `IoTHubDeviceClient_SetDeviceTwinCallback` sets a callback that handles future properties updates.

```
    // Init desired and reported properties equally
    desired_properties.grid_pwr_lim = 0;
    desired_properties.tariff_cost = 0;
    desired_properties.tariff_rate = 0;

    reported_properties.grid_pwr_lim = 0;
    reported_properties.tariff_cost = 0;
    reported_properties.tariff_rate = 0;

    // Retrieve the device twin and have the values for properties
    (void)IoTHubDeviceClient_GetTwinAsync(device_handle,
getCompleteDeviceTwinOnDemandCallback, NULL);
(..)
    // Wait until the twin is received and the desired properties updated
successfully
    while (!twin_updated);

    reported_properties_message = UpdateSerializeReportedMessage();
(..)
    printf("[CLOUD] Reporting: %s\n", reported_properties_message);
    // Send reported properties
    ret = IoTHubDeviceClient_SendReportedState(device_handle, (const unsigned
char *)reported_properties_message, strlen(reported_properties_message),
reportedStateCallback, NULL);
(..)
    // Subscribe to desired properties update notifications
    ret = IoTHubDeviceClient_SetDeviceTwinCallback(device_handle,
deviceTwinCallback, NULL);
(..)
```

The next routines are calls to `InitSocket()` and `SendIdentity()`, which purpose to initialize the client socket and to inform the server socket about its identity. This code is common for all socket clients and is not addressed here.

The following snippet is the main cycle implementation where the code waits on server communication. It starts initializing an int vector "flags" that is sent by the server. The first position is intended to ask the client for getting cloud data and send to server and the second position flag is to ask the client to send telemetry data. First, the client waits on `ReadMessage()` until the server sends it a message. Once it receives a message, it calls `ParseJSONMessage` (also a common function) to parse the message and to populate the flags.

Before moving to the next part, one thing is important to be understood regarding the server vs client vs cloud communication. The server has full control of when it wants to push and to get data. That means that when a command or property is received from the cloud, it is not propagated immediately to server. It is propagated only when the server asks for reading the updates. In addition, it is also important to understand that the cloud only has the power of stopping the charge, while the device is responsible for starting it. Therefore, the cloud sets `chg_stop`, while the server cleans `chg_stop`.

Because of this architecture, there is a risk that the cloud issues a command to set `chg_stop` to '1' and the client socket receives a server message setting it to '0' before the server reads the updates. In this case, we cannot have the cloud command propagated in this specific scenario and the consistency cannot be guaranteed.

So, if it is for pushing data to cloud, first `UpdateChargeStatus()` parses the message and checks the value of `chg_stop` from the server payload. If `chg_stop` is '0', it first checks if the previous state of `chg_stop` is '1' to update the variable that is sent as `chg_stop` to the server when it requests a read. Otherwise, if the previous state is '0' and the variable that is sent as `chg_stop` to the server is '1', it means that the server still did not issue a read message and it is not updated yet. The whole logic is not implemented only on `UpdateChargeStatus()`, it is also partially implemented on the method callback.

After this trick part, `FilterTelemetry()` is used to filter the package that comes from the server and to select only what needs to be sent.

If it is the first connection, then the following **telemetry data** is sent:

- EVSE ID, FW version, lon, lat, alt, EVSE limit, battery level, and charging status

From the second connection on:

- Vehicle ID, vehicle auth, battery capacity, current, power, voltage, temperature, charge rate, time remaining and charge cost, battery level, and charging status.

Then `FilterTelemetry()` returns a string in JSON format that is used to create a message by `IoTHubMessage_CreateFromString()`. Finally, the message is sent using `IoTHubDeviceClient_SendEventAsync()`, which is an async function in which `send_confirm_callback()` is registered to confirm the transfer event. Then, the telemetry message is deallocated and the push flag is cleaned.

The next and last part of the main loop consists of sending data to the server.

First, `SerializeCloudData()` is called to create a JSON format string storing:

- The three properties: Grid power limit, tariff cost, and tariff rate.
- The `chg_stop` value, which is the value changed by the remote command.
- The client identity, so the server knows who is sending the message.

Then, the message is sent using the common function `SendMessage()`.

```
void UpdateCycle(int serv_fd, EndPoint_t clientType)
{
    char messageBuffer[CLOUD_MESSAGE_BUFFER_SIZE];
    int flags[] = {0, 0}; // first - read flag, second - modify flag
    int retCode;
    IOTHUB_MESSAGE_HANDLE message_handle;

    while (1)
    {
        // clear messageBuffer before read
        memset(messageBuffer, 0, sizeof(messageBuffer));

        // read the message from server
        retCode = ReadMessage(serv_fd, messageBuffer, sizeof(messageBuffer),
 clientType);

        // if read failed, return from UpdateCyle
        if (retCode == -1)
        {
            return;
        }

        // Retrieve the flags value
        ParseJSONMessage(messageBuffer, flags, identity);

        if (flags[PUSH_DATA])
```

```
        {
            // Update charging status
            printf("[CLOUD] Send data to cloud\n");
            printf("[CLOUD] %s\n", messageBuffer);
            UpdateChargeStatus(messageBuffer);
            char *telemetry = FilterTelemetry(messageBuffer);
            printf("[CLOUD] %s\n", telemetry);
            message_handle = IoTHubMessage_CreateFromString(telemetry);
            IoTHubDeviceClient_SendEventAsync(device_handle, message_handle,
 send_confirm_callback, NULL);
            free(telemetry);
            flags[PUSH_DATA] = 0;
        }

        if (flags[GET_DATA])
        {

            char *message = SerializeCloudData();
            retCode = SendMessage(serv_fd, message, strlen(message),
 clientType);
            free(message);
            if (retCode == -1)
            {

                return;
            }
            flags[GET_DATA] = 0;
        }
    }
}
```

8.3 EasyEVSE dashboard

The following are the menu options in the EasyEVSE dashboard. For details on how to operate the EVSE dashboard, see the user guide.

- **Manage device tab**: This tab is used to update the following options:
  - Grid Power Limit
  - Tariff Cost
  - Tariff Rate

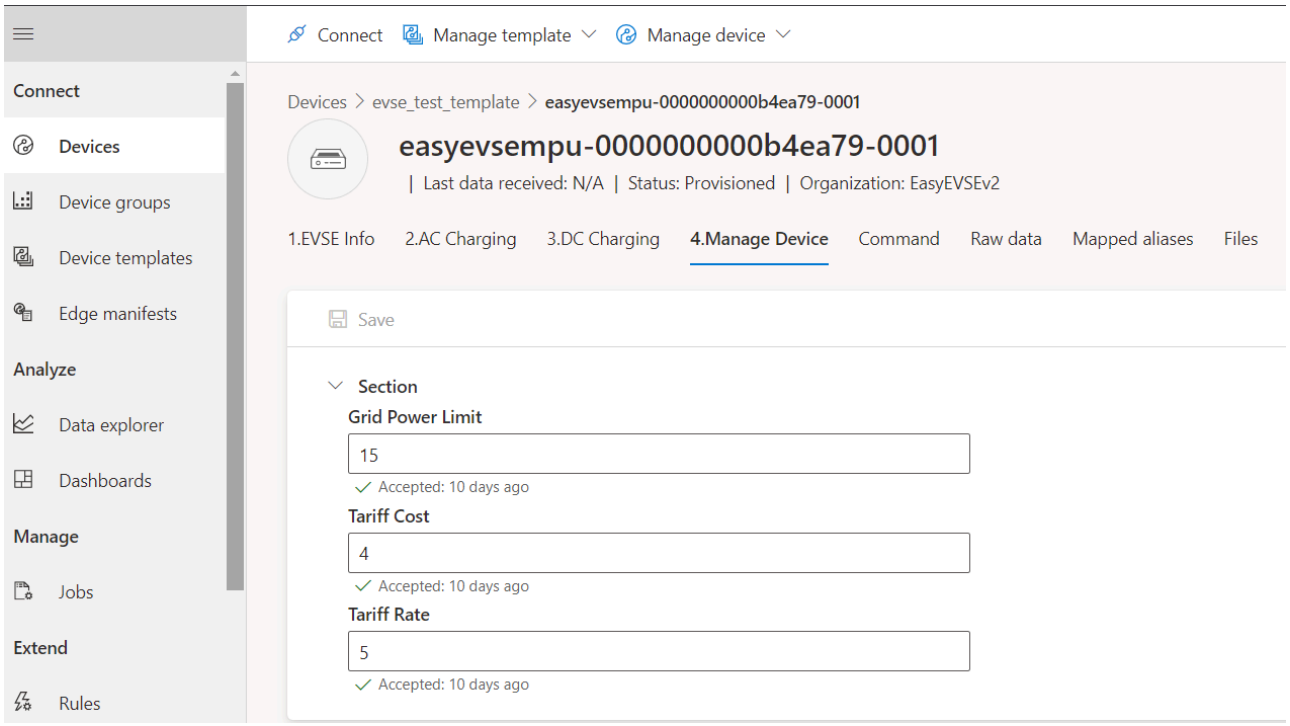


Figure 39. Manage Device

- **Command tab**: This tab is used to send the Terminate Charge Cycle command.
  Note: *If a board is connected, the terminate command also sets the meter board to State A.*

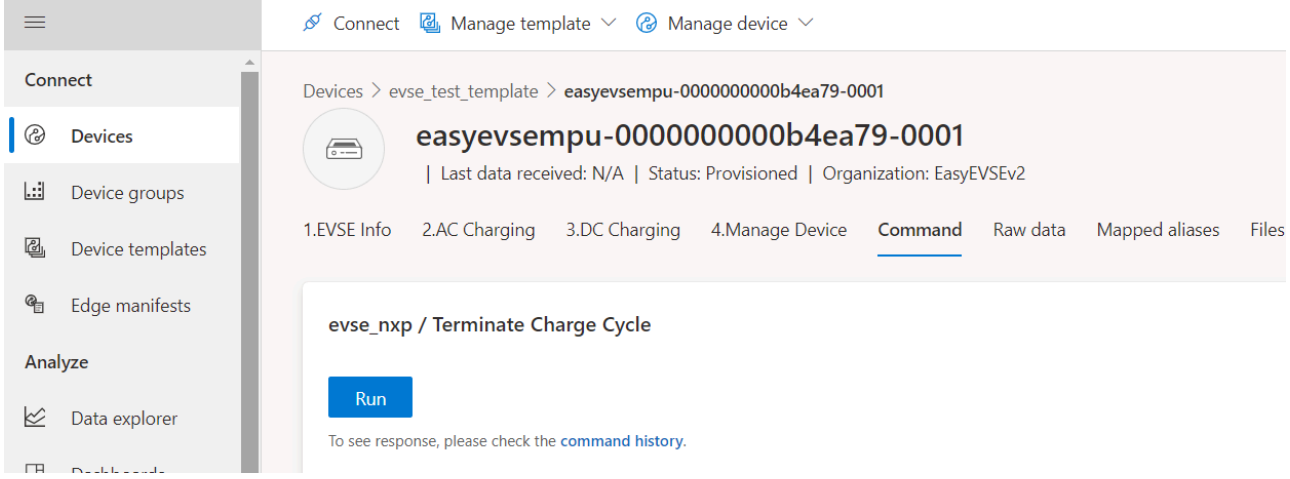


Figure 40. Terminate charge cycle

- **1.EVSE Info tab**: This tab is used to open the main dashboard view.

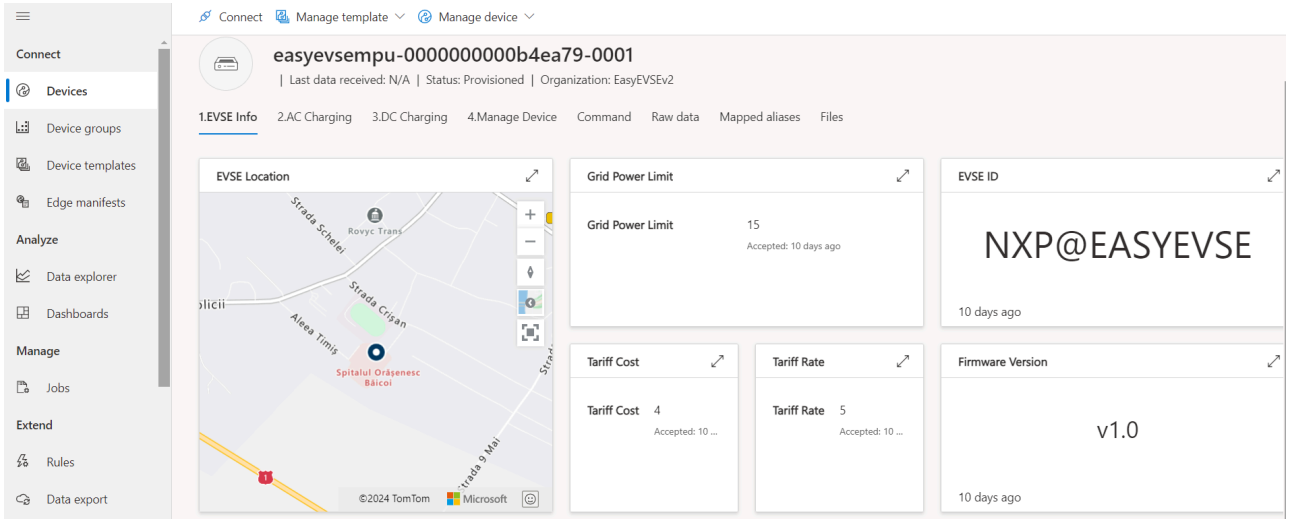


Figure 41. EVSE Info

- **2.AC Charging tab**: Shows AC charging session details such as vehicle, protocol, cost, and energy.

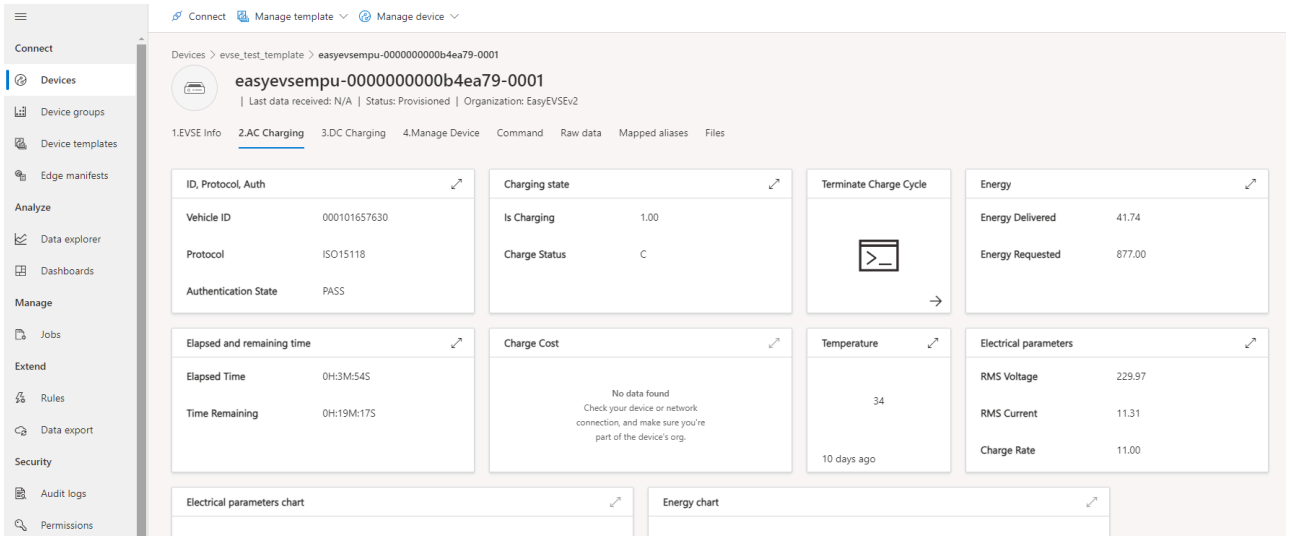


Figure 42. AC Charging

- **3.DC Charging tab**: Shows DC charging details.
  Note: *EasyEVSE v2.0 does not implement DC charging.*

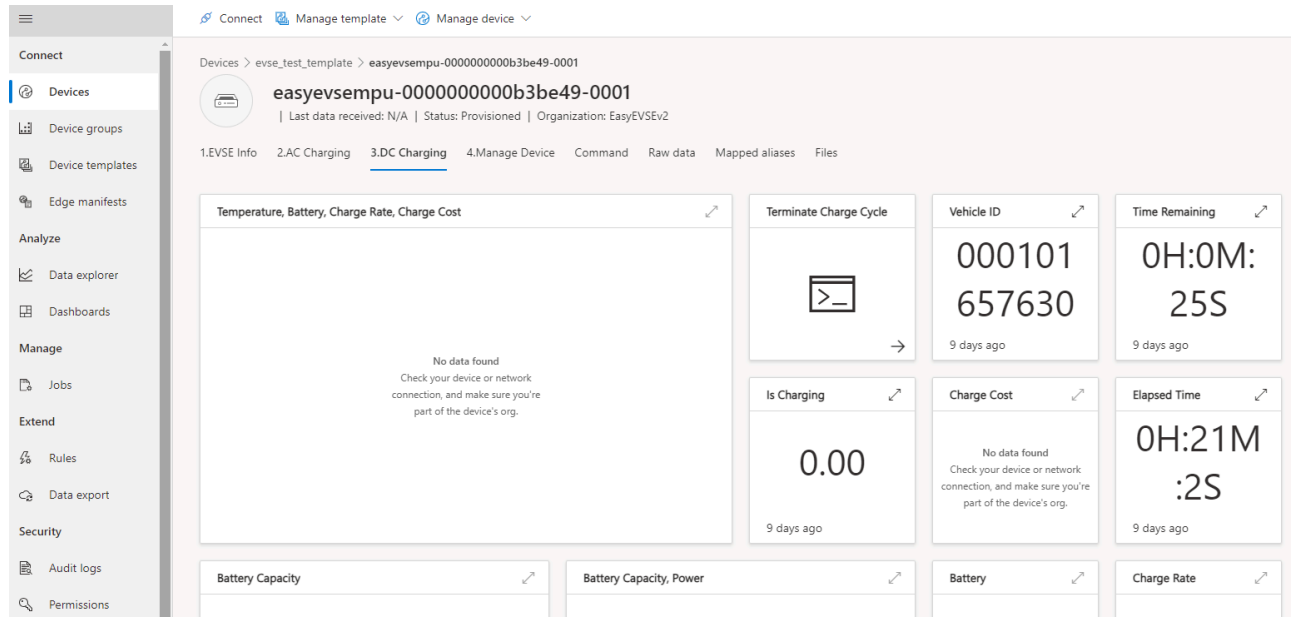


Figure 43. DC Charging

9 GUI

A simple GUI is implemented to read data from cloud, SEVENSTAX, meter, and NFC through the ROS2 library. Figure 44 shows all the user menus of the EasyEVSE GUI. The GUI is developed using the Qt6 framework and Qt creator as IDE.

Note: *The goal of this chapter is to describe only the information presented by the GUI. Therefore, the chapter does not focus on the designing of the GUI from scratch.*

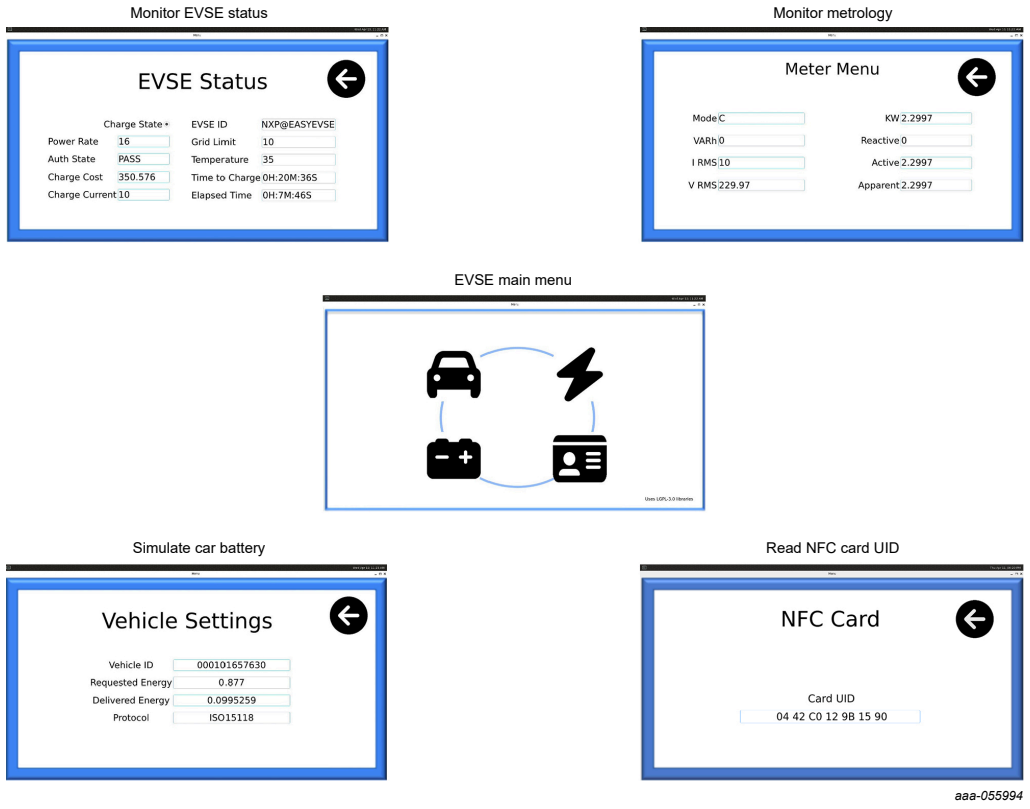


Figure 44. GUI menus

9.1 GUI design overview

The application was developed starting with generating a new Qt widget application. The `mainwindow.ui` is the UI file containing the implementation of the graphical elements. The main window is a QWidget object. To create multiple pages, one for each screen, a QStackedWidget is used.

9.2 Project structure

The files in the GUI demo are as below:

- `mainwindow.ui`: The UI file in XML format that is used to create the window-based application layout and functionalities.
- `mainwindow.cpp`: The file containing the main window class with the logic of the application and all the events.
- `mainwindow.h`: The header file of the application.
- `guiNode.cpp`: New file similar to the default `main.cpp` file in the Qt project with functions of ROS2 to communicate with general/cloud/stack/meter/nfc modules through publisher/subscription.

In addition, other files are further added:

- `images`: The directory where all the images used in the application are stored.
- `resource.qrc`: An XML-based file format that lists files used in the application. For details regarding the resource system mechanism, see https://doc.qt.io/qt-6/resources.html.

Note: *Because the released EasyEVSE GUI demo is not a standard Qt project with *.pro file, the user can develop their GUI in the Qt creator and then port to a proper path of the EasyEVSE GUI demo.*

9.3 Functionalities

The demo application is based on a publisher-subscription communication over ROS2. This chapter focuses on the content and operation of the GUI client side. It also presents the logic of the application, how the communication is initiated, and other functionalities such as data exchange and data manipulation.

9.4 Data sending

First, the `guiNode.cpp` creates. It initializes the data structure and all the data that are published through the publisher for other nodes reading.

```
interfaces::msg::CloudData cloud_data;
publisher_ = this->create_publisher<interfaces::msg::GuiData>("gui_data", 10);
void init_gui_data()
{
gui_data.user_stop_req = false;
cloud_data.grid_pwr_lim = 32.0;
cloud_data.tariff_cost = 0.0;
cloud_data.tariff_rate = 0.0;
cloud_data.grid_stop_req = false;
general_data.fw_vers = "0";
general_data.lat = 0.0;
general_data.lon = 0.0;
general_data.alt = 0.0;
general_data.temperature = 40;
general_data.evse_id = "0";
general_data.evse_rating = 0;
stack_data.evcc_id = "0";
stack_data.vehicle_auth = "fail";
stack_data.energy_requested = 0;
stack_data.chg_rate = 0;
stack_data.chg_cost = 0;
stack_data.chg_elapsed_time = "00H:00M:00S";
stack_data.chg_remaining_time = "N/A";
stack_data.chg_state = "A";
stack_data.energy_delivered = 0.0;
stack_data.protocol = "none";
stack_data.charging = false;
meter_data.current = 1.1;
meter_data.voltage = 2.2;
meter_data.power = 3.3;
nfc_data.nfc_id = "0";
}
publisher_->publish(gui_data);
```

9.5 Data reading

Assuming all the related nodes are already created, the GUI node continuously updates data from the other nodes.

```
interfaces::msg::GeneralData general_data;
interfaces::msg::MeterData meter_data;
interfaces::msg::NfcData nfc_data;
interfaces::msg::StackData stack_data;
cloud_data_subscription_ = this-
>create_subscription<interfaces::msg::CloudData>(
"cloud_data", 10, std::bind(&GUINode::cloud_data_callback, this, _1));
general_data_subscription_ = this-
>create_subscription<interfaces::msg::GeneralData>(
"general_data", 10, std::bind(&GUINode::general_data_callback, this, _1));
stack_data_subscription_ = this-
>create_subscription<interfaces::msg::StackData>(
"stack_data", 10, std::bind(&GUINode::stack_data_callback, this, _1));
meter_data_subscription_ = this-
>create_subscription<interfaces::msg::MeterData>(
"meter_data", 10, std::bind(&GUINode::meter_data_callback, this, _1));
nfc_data_subscription_ = this->create_subscription<interfaces::msg::NfcData>(
"nfc_data", 10, std::bind(&GUINode::nfc_data_callback, this, _1));
```

Above subscriptions are the interfaces that are used by the GUI demo. If new data is published, the callback function is called and the `guiNode` reads the messages and update them into both the data structure and the GUI itself.

9.6 Logging

To debug with logs, call the below function with the proper format. Take printing NFC ID as an example:

```
RCLCPP_INFO(this->get_logger(), "NFC ID: '%s'", msg->nfc_id.c_str());
```

9.7 Transition between pages

For this application, a stack of five widgets was used, each of them representing a page in the application. The main page is displayed when the application starts and it contains four buttons, one for each of the following menus: NFC menu, EVSE menu, Vehicle Settings menu, and Meter menu. The user can push one of them and the associated page is displayed. From the selected menu, the user can return to the main menu by pushing the arrow from the upper right-hand corner. Table 8 describes the map.

Table 8. The map between icons and page names

Icons	Page names
Car	EVSE Status
Lightning	Meter Menu
Battery	Vehicle Settings
Certificate	NFC Card

For example, from the EasyEVSE Menu, the user can select Lightning Button and that displays the Meter Menu. To return, the user can select the Back arrow button and the Main Menu is displayed.

To create this flow, when pushing a button, the clicked() signal is emitted and the expected widget becomes visible. Below is an example to show how pushing the back button makes the main menu visible (index 0 is associated with the Main Menu).

```
void MainWindow::on_button_back_1_clicked()
{
ui->stackedWidget->setCurrentIndex(0);
}
```

9.8 GUI design

The GUI is designed in the Qt creator but need to port the necessary code files to `nxp-easyevse-mpu/src/easyevse/src/gui directory`. All the files except `guiNode.cpp` are copied from the Qt creator project. Also, `guiNode.cpp` shares some code with `main.cpp` of the Qt creator project in the main functions.

9.9 Main menu page

This section describes how the main menu is made and the purpose of it.

The main menu is displayed when the application starts. It contains four buttons with intuitive images. By selecting one button at a time, the user can access the other menus.

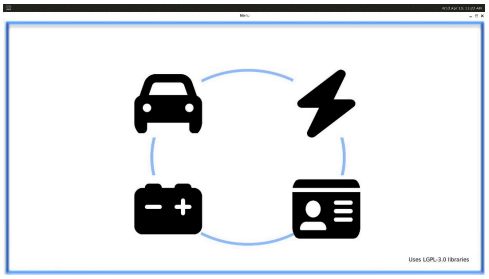


Figure 45. Main menu with the four buttons

To select the menu, in the Qt creator open the `mainwindow.ui` and the Qt designer displays the proper page.

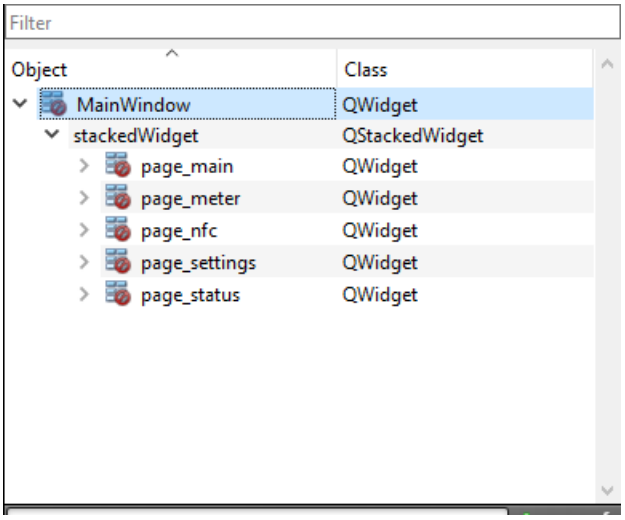


Figure 46. Qt designer widget description

9.10 NFC Card page

This section describes how the NFC Card page is made and the purpose of it.

This page displays the NFC ID in the designated field once an NFC card is read.

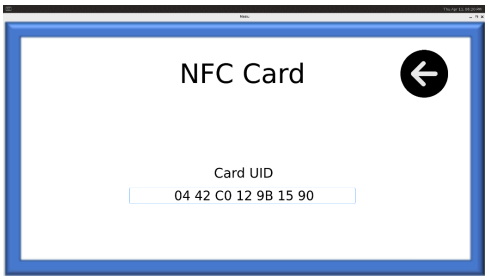


Figure 47. NFC Card page

The NFC node sends the data through the publisher and then the GUI node updates data through the subscription. Finally, the card ID in the GUI is set as below (see `file src/gui/guiNode.cpp`):

```
w->ui->lineEdit_Card_UID->setText(QString("%2").arg(msg->nfc_id.c_str()));
```

Each time another NFC card is used, the field instantly display the new ID.

9.11 EVSE Status page

This section describes how the EVSE Status page is made and the purpose of it.

Using this page, the user can find details related to the EV charge state, the ID of the EVSE, time, charging cost, temperature, and so on. All the information is received from the multiple subscriptions, then parsed and displayed.

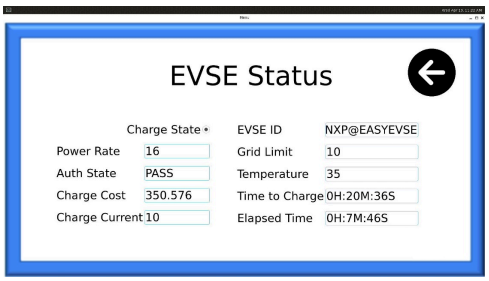


Figure 48. EVSE Status page

Table 9. Data Source of EVSE Status

Component name	Source node	Comment
Charge_State	SEVENSTAX	stack_data.charging
EVSE_ID	BUSINESS_LOGIC	general_data.evse_id = "NXP@EASYEVSE";
Power_Rate	BUSINESS_LOGIC	general_data.evse_rating = MAX_EVSE_CURRENT;
Grid_Limit	CLOUD	cloud_data.grid_pwr_lim = reported_properties->grid_ pwr_lim;
Auth_State	SEVENSTAX	stack_data.vehicle_auth
Temperature	BUSINESS_LOGIC	general_data.temperature

Table 9. Data Source of EVSE Status*...continued*

Component name	Source node	Comment
Charge_Cost	SEVENSTAX + CLOUD	stack_data.chg_cost = stack_data.energy_delivered * cloud_data.tariff_cost;
Charge_Current	SEVENSTAX	stack_data.chg_rate
Time2Charge	SEVENSTAX	stack_data.chg_remaining_time
Elapsed_Time	SEVENSTAX	stack_data.chg_elapsed_time

9.12 Vehicle Settings page

This section describes how the Vehicle Settings page is made and the purpose of it.

In this page, the user can find the following elements: Vehicle ID, Requested/Delivered Energy with unit kWh, and Protocol (Base or ISO15118). All the data are aligned with SEVENSTAX and the protocol implies the changes from basic charging to high-level charging.

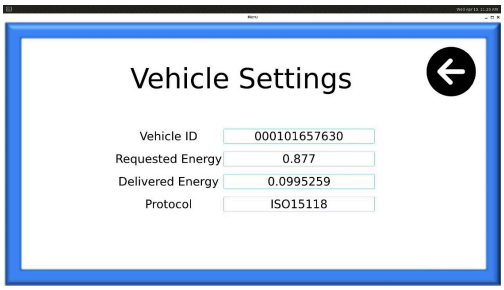


Figure 49. Vehicle Settings page

Table 10. Data source of Vehicle Settings

Component name	Source node	Comment
Vehicle_ID	SEVENSTAX	stack_data. evcc_id
Requested_Energy	SEVENSTAX	stack_data.energy_requested/1000.0
Delivered_Energy	SEVENSTAX	stack_data.energy_delivered/1000.0
Protocol	SEVENSTAX	stack_data.protocol

9.13 Meter Menu page

This section describes how the Meter Menu is made and the purpose of it.

The Meter Menu page shows the information about voltage, current, and power. All the data are aligned with SEVENSTAX and meter with the UART bridge. The meaning of the charging mode is aligned with ISO15118 and mode C means charging normally.

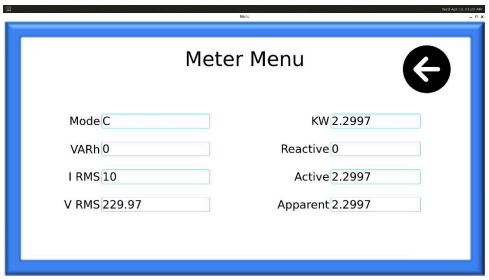


Figure 50. Meter Menu page

Table 11. Data source of Meter Menu

Component name	Source node	Comment
Mode	SEVENSTAX	stack_data.chg_state
KW	SEVENSTAX	meter_data.power /1000.0
VARh	SEVENSTAX	0
I RMS	SEVENSTAX	meter_data.voltage
V RMS	SEVENSTAX	meter_data.current
Reactive	SEVENSTAX	0
Active	SEVENSTAX	meter_data.power /1000.0
Apparent	SEVENSTAX	meter_data.power /1000.0

10 Near-field communications (NFC)

This chapter describes the addition and implementation of the NXP public NFC reader library to the host controller. The library is software, written in the C language, as is integrated into the host controller project to manage the PN7160 NFC multiprotocol reader.

NFC enables an EVSE user or local operator to authenticate with a single tap using a card, key fob, cell phone, and so on. In this EasyEVSE platform, the Unique ID (UID) from an NFC device such as a MIFARE product-based card is read and displayed locally (serial terminal and GUI) and in the cloud (the Azure IoT Central application).

Even though your NFC IC is provisioned with a UID number, this number cannot be the basis of authentication. For example, an attacker could supplant a device of yours using its UID.

Note: *In a real application, the UID is not considered secure and we must rely on encrypted and authenticated data stored on the card or key fob.*

The host controller configures the NFC PN7160 reader in a discovery loop mode. When performing discovery, the reader scans for any NFC devices. If a device is discovered (detected on the field of the reader antenna), the reader triggers an interrupt to the host controller. Then the host controller requests the basic card information including the UID. This communication is done in plaintext.

The EasyEVSE platform simply reads card information as a demonstration of the NFC functionality. Therefore, the NFC authentication and data encryption and any specific development is out of the scope of this document. For details on how to implement NFC features, contact your local Field Applications Engineer (FAE).

10.1 PN7160 plug and play NFC controller

The PN7160 is an NFC plug and play controller with integrated firmware and NFC controller interface (NCI). The NCI provides users a logical interface, which can be used with different physical transports, such as SPI and I2C.

For more technical details about the chip, see NFC Plug and Play Controller with Integrated Firmware and NCI Interface.

10.2 NXP NFC kernel driver

The nxpnfc kernel driver is used to communicate with the PN7160 NFC controller. The driver offers communication to the NFC controller connected over either I2C or SPI physical interface. In our case, the SPI interface is used.

When loaded to the kernel, this driver exposes the interface to the NFC controller through the device node named `/dev/nxpnfc`.

In the final image, the NFC driver is added and integrated into the Linux kernel through patches located in the `meta-nxp-easyevse/recipes-kernel/linux/files` directory. These patches are added to the `linux-imx` recipe.

These patches automatically port the PN7160 library in Linux, according to *PN7160 Linux porting guide* (document AN13287).

The following steps are performed:

- Replacing the existing NFC drivers implementations from the kernel directory `drivers/nfc` with the cloned `nxpnfc` repository.
- Including the driver to the kernel and making it loaded during device boot by adding into the device tree `imx93-11x11-evk-evse.dts` the appropriate configuration.

```
&lpspi6 {
```

```
...
status = "okay";

nxpnfc@0 {
compatible = "nxp,nxpnfc";
reg = <0>;
nxp,nxpnfc-irq = <&gpio2 22 GPIO_ACTIVE_HIGH>;
nxp,nxpnfc-ven = <&gpio2 23 GPIO_ACTIVE_HIGH>;
nxp,nxpnfc-fw-dwnld = <&gpio2 24 GPIO_ACTIVE_HIGH>;
spi-max-frequency = <7000000>;
};
};
```

- Including the targeted driver (I2C and SPI version) to the build, as built-in.

```
do_copy_defconfig:append () {
echo "CONFIG_NXP_NFC_I2C=y" >>  ${B}/.config
echo "CONFIG_NXP_NFC_SPI=y" >>  ${B}/.config
echo "CONFIG_NXP_NFC_RECOVERY=y" >>  ${B}/.config
}
```

10.3 PN71xx NFC library

The NFC Linux `libnfc-nci` stack is implemented for NCI-based NXP NFC controllers (PN71xx) and consists in a library running in the user space. It is available in the following repository: https://github.com/NXPNFCLinux/linux_libnfc-nci

The vertical structure of the NXP NFC library is classified into the following layers described in Figure 51.

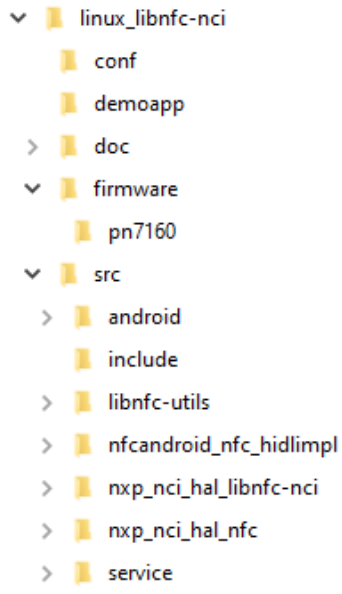


Figure 51. NXP NFC library

The following list explains Figure 51:

- conf: Contains files that allow configuring the library at runtime. There are defining tags, which impact library behavior.
- demo: Here can be found the main function of the application.
- firmware: Firmware library for PN7160

- src: API implementation

The NFC library is added to the image using the `libnfceasyevse.bb` recipe, located in the `meta-nxp-easyevse/recipes-nxp-easyevse/libnfceasyevse` directory. The library is fetched from the GitHub repository, placed under the `tmp/work/ armv8a-poky-linux/libnfceasyevse` directory, then compiled and deployed to the target's rootfs.

10.4 NFC library operation theory

The following section describes the principles of the NFC library's functionality. Figure 52 covers the main steps for initializing the NFC library and how it works in polling loop mode.

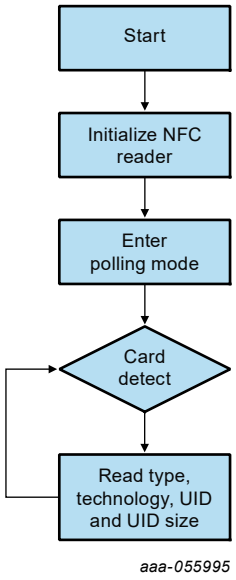


Figure 52. NFC library

To initialize the NFC library, follow the steps below:

1. Create mutexes and threads for NFC library.
2. Initialize data parameter structures.
3. Initialize the protocol abstraction layer.
4. PN7160 enters the polling loop and waits for a card detection.
5. On-card detection determines a single or multi card in the field.
6. Select each card individually.
7. Now, perform the following:
   - Identify card type
   - Identify card technology
   - Identify the UID of the card
   - Identify size of UID

10.5 Application executing

Every time a card is presented in the PN7160 antenna field, the chip raises an IRQ on every card detect.

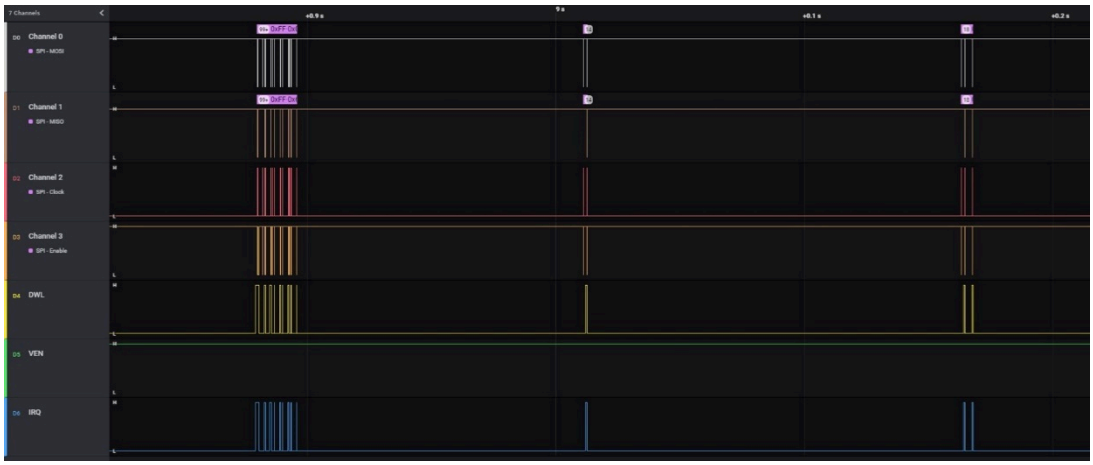


Figure 53. Signal capture of NFC communication interface activity upon reading an NFC card

This signals an SPI transfer from the host controller to determine if a valid card has been detected.

10.6 NFC client implementation

Similar to the other client applications, the NFC one is divided into two parts: the ROS and the peripheral processing side.

The ROS part (`nfcNode.cpp`) starts first and establishes the communication mechanism with the other clients through the framework, after which it starts the peripheral part as a thread.

The peripheral part of the NFC client application (`nfc_api.c`) implements the basic discovery loop behavior, similar to how the NFC library demo does it. This reads the UID of the NFC card whenever it is presented and transmits that to the ROS client.

The data published by the NFC client can be found in the `NfcNode.msg` file:

```
string      nfc_id
```

the `nfc_id` contains the UID of the last scanned NFC device.

11 Meter

This chapter introduces the basic electrical meter principles using the NXP Kinetis KM3x MCUs and meter libraries. Also, the section describes the implementation of the EasyEVSE meter block. For detailed information on how to develop a meter application, see the metering application notes and design reference guides listed in Reference documentation.

11.1 Introduction

The goal of a digital electric meter is to mathematically compute billing information with high accuracy. A digital electric meter must calculate the charge for consuming power from the electrical grid; and non-billing information for grid balancing and further analysis.

The computation of meter quantities requires periodic sample of instantaneous current and voltage (denoted as i(t) and u(t), respectively) from circuits that condition the mains AC or electrical grid.

The mains AC can be arranged as three-phase four-wire, two-phase two-wire, or single- phase one-wire topologies. Figure 54 shows a diagram of an electric meter computing billing and non-billing quantities from a three-phase mains AC.

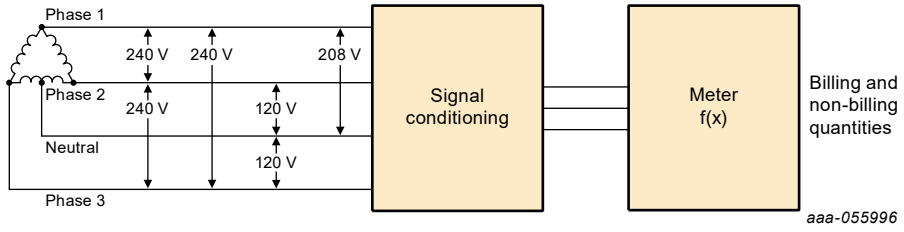


Figure 54. Example of a meter with three phases four wire AC

An EVSE requires a high-accuracy meter interface to measure the power delivered to the vehicle. For example, if high accuracy is not achieved, it represents substantial amounts of lost revenue or an overcharge to the customer.

The NXP Kinetis KM metrology MCUs are a specialized solution that fits high-accuracy EVSE metering requirements. This MCU series is based on the Arm Cortex-M0+ and integrates a powerful and dedicated AFE to achieve high-resolution sampling. It also features a specialized MMAU support for 32-bit and 64-bit math, enabling fast execution of metering algorithms.

Table 12 outlines some general electric meter quantities and highlights, which are commonly referred as billing quantities.

Table 12. General electric meter quantities

Quantity	Unit/symbol	Description	Billing
Active energy	Wh	Measured in the unit of watt hours (Wh). Represents the electrical energy produced, flowing or supplied by an electric circuit during a time interval.	✓
Reactive energy	Varh	Measured in the unit of volt-ampere-reactive hours (VARh).	✓
Active power	Wh/P	Measured in watts (W). Expressed as the product of the voltage and the in-phase component of the alternating current.	✓
Reactive power	VAR/Q	Expressed in volt-amperes-reactive (VAR) is the product of the voltage and current and the sine of the phase angle between them.	✓
Apparent power	VA/S	Measured in the units of volt-amperes (VA). The total power in an AC circuit (both absorbed and dissipated) is referred to as the total apparent power (S).	

Table 12. General electric meter quantities*...continued*

Quantity	Unit/symbol	Description	Billing
RMS voltage	URMS	Measurement of the magnitude of the alternating voltage.	
RMS current	IRMS	Measurement of the magnitude of the alternating current.	

The computation of meter quantities can be done either in the time or in the frequency domain. For the time domain, there is a filter-based algorithm and conversely a Fast Fourier Transform (FFT) algorithm for the frequency domain.

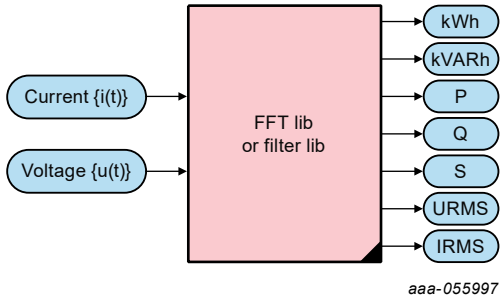


Figure 55. Implementation of FFT or filter library

NXP provides libraries for both algorithms. Both share the requirement of instantaneous voltage and current samples to be provided at constant sampling intervals, to calculate billing and non-billing quantities. However, the libraries are different.

If you want harmonic information, use the FFT library.

- Only the FFT library supports harmonic information.

The FFT library requires the signal frequency, so the FFT needs more hardware resources.

- FFT needs one CMP module and at least one timer channel.
- Use these resources to measure the signal frequency.
- Filter library does not use the frequency information. FFT uses more RAM resources.
- FFT uses a double buffer. One is used for calculation, and the other for saving sampled values. Therefore, FFT uses more RAM.

Figure 56 shows an example representation of a single-phase meter.

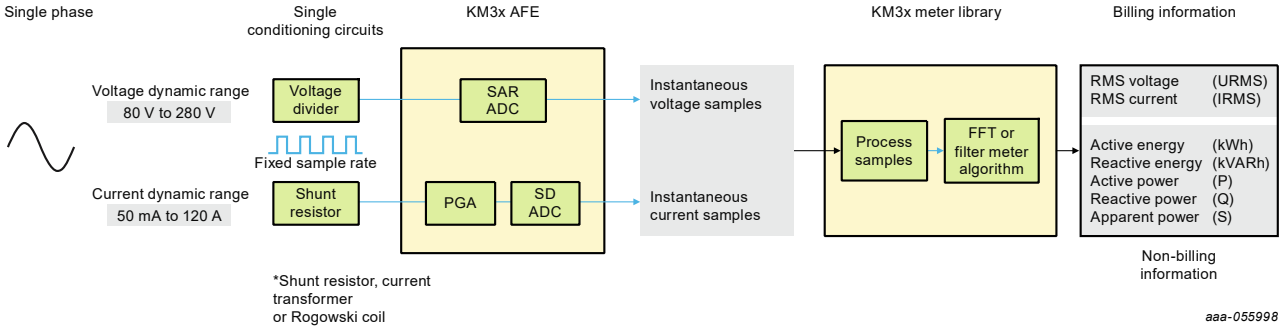


Figure 56. Single phase meter simplified example

11.2 Application meter vs simulated meter

This section lists some significant differences between a real application meter and the simulated meter, implemented in the EasyEVSE demo platform.

The EasyEVSE meter with the TWR-KM3x board is a simulated meter. The meter constantly samples a rotary potentiometer to calculate meter quantities and performs additional tasks as described in Section 11.3. A real application meter also performs the same or similar computations. However, the difference between both is that a real meter is capable to be connected to the AC mains using dedicated AC conditioning circuits.

Warning: *The TWR-KM3x board is not designed to support direct connection with the AC mains. Therefore, attempting to connect with AC mains could result in a personal safety hazard or damage to the board. To develop a real meter, see the standalone electricity meter reference designs, which are designed to connect directly to mains.*

From a software perspective, both application and simulated meters share the implementation of metering libraries and several functionalities such as LCD driving. Therefore, the simulated meter, using the TWR-KM3x is a good point to start your application meter.

11.2.1 Topology

The topology of a meter is designed on the number of AC mains phases. For measuring the current of a single phase, there is one sigma-delta (SD) ADC channel plus another one for the neutral line. Therefore, a three-phase AC mains requires four SD channels to measure AC current. Furthermore, this topology implements all the available SD ADC channels on the KM3x family (up to four). To measure the phase and neutral voltage, the SAR ADC channels can be used with ease because v(t) has a low dynamic range. Typically, the range is from 80 V to 280 V.

A three-phase meter implemented on a Kinetis-M device is based on the signals dynamic range analysis. The metering current signal is typically from 50 mA to 120 A. A precise and linear ADC with a wide dynamic range, typically 24-bit, must digitize the current.

Therefore, SD ADC is the ideal solution to solve current dynamic range requirements. The voltage dynamic range is approximately 60 times smaller than the current dynamic range. The use of a hi-resolution SAR ADC can solve the voltage requirements.

Even though the EasyEVSE meter is not connected to the AC mains, you can say that it implements a single-phase topology, with no measure of the neutral line. It only samples a rotary potentiometer, which is wired to an SAR ADC channel of the KM3x MCU.

The filter-based library implemented only calculates meter quantities for a single-phase current and voltage. However, no SD ADC channels are implemented. Also, the neutral line calculations are not performed. For your application, you must consider the number of phases required. Both filter based and FFT meter libraries support 1, 2 or 3 phases. See Reference documentation and choose the appropriate design reference manual.

11.2.2 Analog circuits

For demonstration, the instantaneous current or i(t) delivered to the EV is acquired from a single SAR ADC channel. This channel samples the voltage at the rotary potentiometer of the TWR-KM3x board. Then, the sampled voltage generates input variables for the filter-based metering library, and therefore obtain meter quantities.

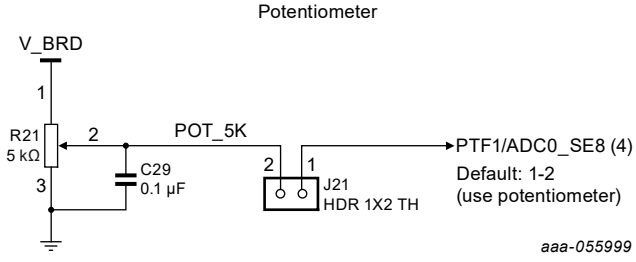


Figure 57. EasyEVSE sampling potentiometer voltage

However, for a real application, see the analog circuits in the design reference manual.

11.2.3 Meter library configuration

The meter project is based on the TWR-KM3xZ75 MCUXpresso SDK example: `meterlib1ph_test`. The project implements the filter-based metering library. The coefficients of the library must be tuned according to the specific application. The project simply uses the default coefficients.

meterlib1ph_cfg.h

```
/*****************************************************************************
 * General parameters and scaling coefficients
 *****************************************************************************/
#define POWER_METER              1PH  /*!< Power meter topology                        */
#define CURRENT_SENSOR   PROPORTIONAL  /*!< Current sensor output characteristic       */
#define LIBRARY_PREFIX       METERLIB  /*!< Library prefix; high-performance library   */
#define I_MAX                141.421  /*!< Maximal current I-peak in amperes          */
#define U_MAX                350.000  /*!< Maximal voltage U-peak in volts            */
#define F_NOM                     50  /*!< Nominal frequency in Hz                    */
#define COUNTER_RES            10000  /*!< Resolution of energy counters in inc/kWh   */
#define IMP_PER_KWH            50000  /*!< Impulses per kWh                           */
#define IMP_PER_KVARH          50000  /*!< Impulses per kVARh                         */
#define DECIM_FACTOR               2  /*!< Auxiliary calculations decimation factor   */
#define KWH_CALC_FREQ       1200.000  /*!< Sample frequency in Hz                     */
#define KVARH_CALC_FREQ     1200.000  /*!< Sample frequency in Hz                     */
/*****************************************************************************
```

Figure 58. Meter project filter-based library with default parameters

To adapt the library for your application, see Section 4.2 "Configuration tool" in *Filter-Based Algorithm for Metering Applications* (document: AN4265).

11.2.4 UART isolation board

When communicating with other ICs, application meters implement an isolation barrier. For this evaluation platform, no isolation layer is implemented. Also, if power is applied to the IO of the i.MX 93, it could fail to start. This issue can occur with the KM3x and i.MX 93 UART communication line. To include UART isolation between KM3x and i.MX 93 in a real application, see the ADM3251E evaluation board from Analog Devices.

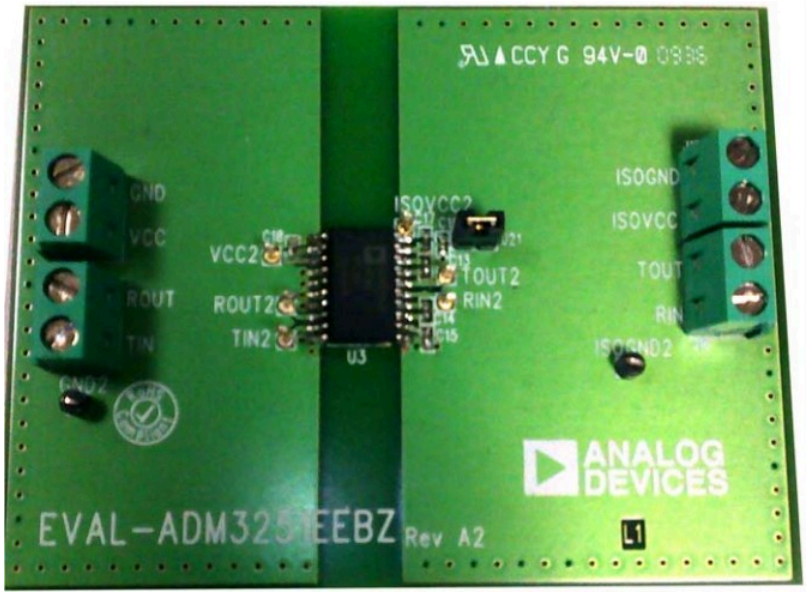


Figure 59. UART isolation board

Product page: EVAL-ADM3251E

User guide: UG-124

11.3 Meter project

The meter project implements the filter-based metrology library to convert the voltage read from the TWR-KM3x board-rotary potentiometer into the electrical meter quantities related to the charge of the EV. It reports the data onto the board LCD and establishes a serial communication channel with the EVSE-SIG-BRD.

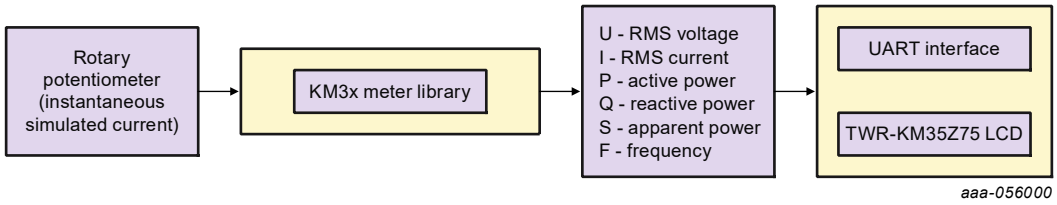


Figure 60. Meter project

The meter project performs three main tasks:

- Run meter: To calculate meter quantities.
- Display data on LCD: Presenting instantaneous current and charging state information.
- Communication with EVSE-SIG-BRD: Replying to meter data request commands.

Figure 61 shows four general flowcharts corresponding to each task that the meter project performs. Each task is described in the following subsections.

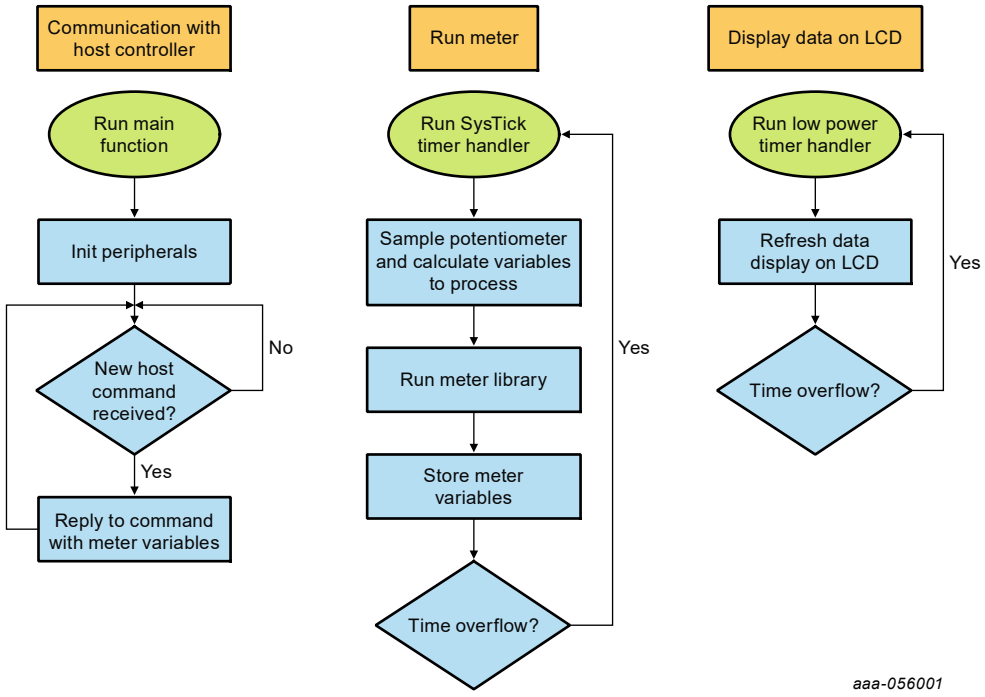


Figure 61. Meter tasks

The meter project runs on TWR-KM3x devices. It is implemented on bare-metal, uses a filter-based metering library and it is based on the `meterlibFFT1ph_test` MCUXpresso SDK example.

Note: *This project is intended to serve you as a demonstration application, giving the basics to develop your own meter application.*

11.3.1 Run meter

This section describes the basic operation of the meter software.

11.3.1.1 SysTick timer handler

The SysTick timer handler routine is triggered every 10 ms to perform the following actions:

- Sample the rotary potentiometer. The sampling is done by pulling the conversion-complete flag of the SAR ADC channel and then reading the conversion result value.
- Convert the 16-bit ADC result to be part of the input variables of the meter library.
- Trigger a new ADC conversion by software.
- Run the meter library to obtain the meter quantities.

```
 * Handler routine used to obtain set value of R21 potentiometer, scale to max current
 * and then use this value when running the metrology library
 * This occurs every 10mS
 */
void SysTick_Handler(void)
{
    /* Obtain latest ADC value */
    while (0U == (kADC16_ChannelConversionDoneFlag &
                  ADC16_GetChannelStatusFlags(DEMO_ADC16_BASEADDR, DEMO_ADC16_CHANNEL_GROUP)))
    {
    }
    tmp16 = ADC16_GetChannelConversionValue(DEMO_ADC16_BASEADDR, DEMO_ADC16_CHANNEL_GROUP);
    /* scaling amps to max 32 */
    tmp32 = (3200 * tmp16) / 4095;
    Pot_Value = (((float)tmp16) / 4095) * 32;
    RMS_Voltage = Pot_Value * 10.0;

    /* start another conversion */
    ADC16_SetChannelConfig(DEMO_ADC16_BASEADDR, DEMO_ADC16_CHANNEL_GROUP, &adc16ChannelConfigStruct);

    Run_MeterLib();
}
```

Figure 62. SysTick handler

11.3.1.2 Potentiometer sample

The TWR-KM3x board-rotary potentiometer R21 is used to simulate the instantaneous current being drawn from the vehicle:

- Clock-wise adjustment increases the current up to a maximum of 32 Amps.
- Counter clockwise reduces the simulated current down to 0 Amps.

To represent the maximum current input of 32 Amps, the voltage sampled from the potentiometer is acquired by the tmp16 uint16_t variable. This variable is used in the calculations of:

- Instantaneous current displayed on the TWR-KM3x LCD.
- RMS voltage.
- Simulated current waveform.

11.3.1.3 Processing variables with the meter library

In the `Run_MeterLib()` function, current and voltage sine wave forms are simulated using the potentiometer value and other variables and parameters.

```
static void Run_MeterLib(void)
{
    /* calculate phase voltage and phase current waveforms                    */
    time = time+(1.0/KWH_CALC_FREQ);
    // simulated calculated voltage waveform
    u24_sample = FRAC24(((sin(2*_PI*50.0*time+U_ANGLE)*230.0*sqrt(2)+0.0)/U_MAX));

    /* simulated calculated current waveform */
    /* Calculate current sample based on Pot reading */
    i24_sample = FRAC24(((sin(2*_PI*50.0*time+I_SHIFT)*(Pot_Value)*sqrt(2)+0.0)/I_MAX));

    METERLIB1PH_ProcSamples(&mlib,u24_sample,i24_sample,&shift);
    METERLIB1PH_CalcWattHours(&mlib,&wh_cnt,METERLIB_KWH_PR(IMP_PER_KWH));

    /* functions below might be called less frequently - please refer to      */
    /* KWH_CALC_FREQ, KVARH_CALC_FREQ and DECIM_FACTOR constants              */

    if (!(cycle % (int)(KWH_CALC_FREQ/KVARH_CALC_FREQ)))
    {
      METERLIB1PH_CalcVarHours (&mlib,&varh_cnt,METERLIB_KVARH_PR(IMP_PER_KVARH));
    }

    if (!(cycle % DECIM_FACTOR))
    {
      METERLIB1PH_CalcAuxiliary(&mlib);
    }

    METERLIB1PH_ReadResults (&mlib,&U_RMS,&I_RMS,&P,&Q,&S);

        cycle++;
}
```

Figure 63. Meter library implementation

The meter quantities are then stored in global variables, ready to be sent to the host controller data request. For details on the use of the filter-based library, see the *Filter-Based Algorithm for Metering Applications* (document: AN4265).

Finally, the meter library calculates the meter quantities and stores them in global variables.

11.3.2 Communication with the host controller

The EVSE-SIG-BRD and the meter board have a UART communication channel running at 115200 baud. The host controller sends a command to request meter data or to change the current charging state. Figure 64 shows a snapshot of the function `Process_HostCommand()`, which continuously polls for commands.

```
static void Process_HostCommand(void)
{
      char ch_in;
      uint8_t loop;

      loop = 1;
      while(loop)
      {
       ch_in = GETCHAR();

        if ((ch_in<'0') || (ch_in>'F'))
        {
          /* command received is invalid, restart while loop immediately*/
          continue;
        }

        switch (ch_in)
           {
           case '0':
               PRINTF("%3.2f[1]%3.2f[2]%4.2f[3]%1.1f[4]\r", I_RMS, U_RMS, P, Status_Index);
               loop = 0;
               break;
           case '1':
               PRINTF("%3.2f[1]\r", I_RMS);
               loop = 0;
               break;
           case '2':
               PRINTF("%3.2f[2]\r", U_RMS);
               loop = 0;
               break;
           case '3':
               PRINTF("%4.2f[3]\r", P);
               loop = 0;
               break;
           case '4':
               PRINTF("%1.1f[4]\r", Status_Index);
               loop = 0;
               break;
           case 'A':
               strcpy(Status_String, "STATEA");
               Status_Index = 1.0;
               PWM_OnOFF(0, 53);
               GPIO_PortSet(BOARD_INITPINS_LED_ORANGE_GPIO, BOARD_INITPINS_LED_ORANGE_GPIO_PIN_MASK);
               GPIO_PortSet(BOARD_INITPINS_LED_RED_GPIO, BOARD_INITPINS_LED_RED_GPIO_PIN_MASK);
```

Figure 64. Function Process_HostCommand()

For example, if the host controller sends a '0' char, the meter responds by sending I_RMS, U_RMS, and P and the current charging state, represented by numbers. Figure 65 shows an example of the physical link of the host and meter boards.

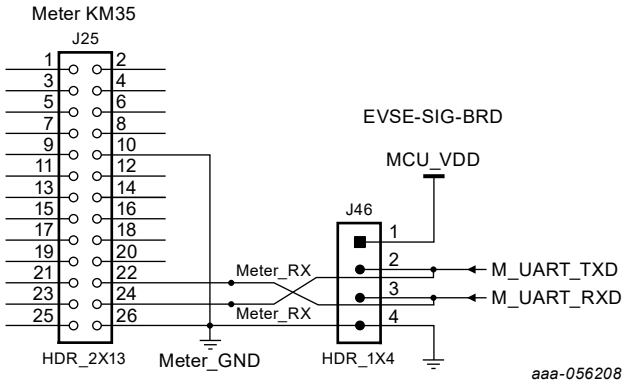


Figure 65. Link of the host and meter boards

11.3.3 Displaying data on LCD

The TWR-KM3x uses an S-Tek GDH-1247WP display to represent instantaneous current, controlled by the potentiometer.

Note: *The meter project does not display any meter quantity calculated by the meter library.*

Figure 66. Meter LCD

The LCD data is refreshed under a periodic rate of the KM3x low-power timer.

```
405 void LPTMR0_IRQHandler(void)
406 {
407     LPTMR_ClearStatusFlags(LPTMR0, kLPTMR_TimerCompareFlag);
408
409     tmpnum = (uint8_t)(tmp32/1000);
410     tmp32 = tmp32 % 1000;
411     SLCD_Engine_Show_Num(&slcd_engine, tmpnum, NUM_POS5, 1);
412
413     tmpnum = (uint8_t)(tmp32/100);
414     tmp32 = tmp32 % 100;
415     SLCD_Engine_Show_Num(&slcd_engine, tmpnum, NUM_POS6, 1);
416
417     tmpnum = (uint8_t)(tmp32/10);
418     tmp32 = tmp32 % 10;
419     SLCD_Engine_Show_Num(&slcd_engine, tmpnum, NUM_POS7, 1);
420
421     tmpnum = (uint8_t)(tmp32);
422     SLCD_Engine_Show_Num(&slcd_engine, tmpnum, NUM_POS8, 1);
423
424     tmpnum = Status_Index / 1;
425     SLCD_Engine_Show_Num(&slcd_engine, tmpnum, NUM_POS15, 1);
426
427     if ((Error_Counter==1) || (Error_Counter==2))
428     {
429         SLCD_Engine_Show_Icon(&slcd_engine, ICON_S29, 1);
430     }
431     SLCD_Engine_Show_Icon(&slcd_engine, ICON_L1, 1);
```

Figure 67. Function LPTMR0_IRQHandler()

12 Appendix

This appendix describes the pin tables, meter, and LCD information.

12.1 TWR-KM3x pin table

The meter project pin table is in the *.mex file. You can open the pin mux table using the Pins Config Tool.

To open the pin mux table, perform the following steps:

1. Select the project.
2. Click the config tools icon.
3. Select **Open Pins**.

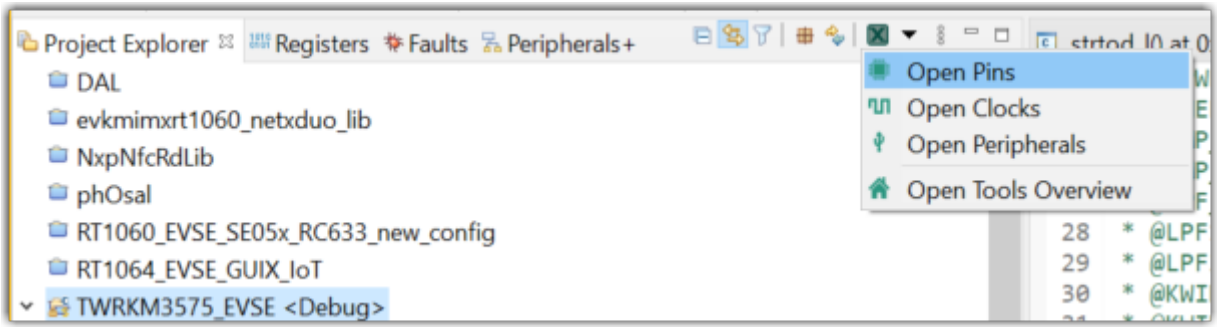


Figure 68. Open Pins tool

4. You can see the pins mux table in the Pins tool.

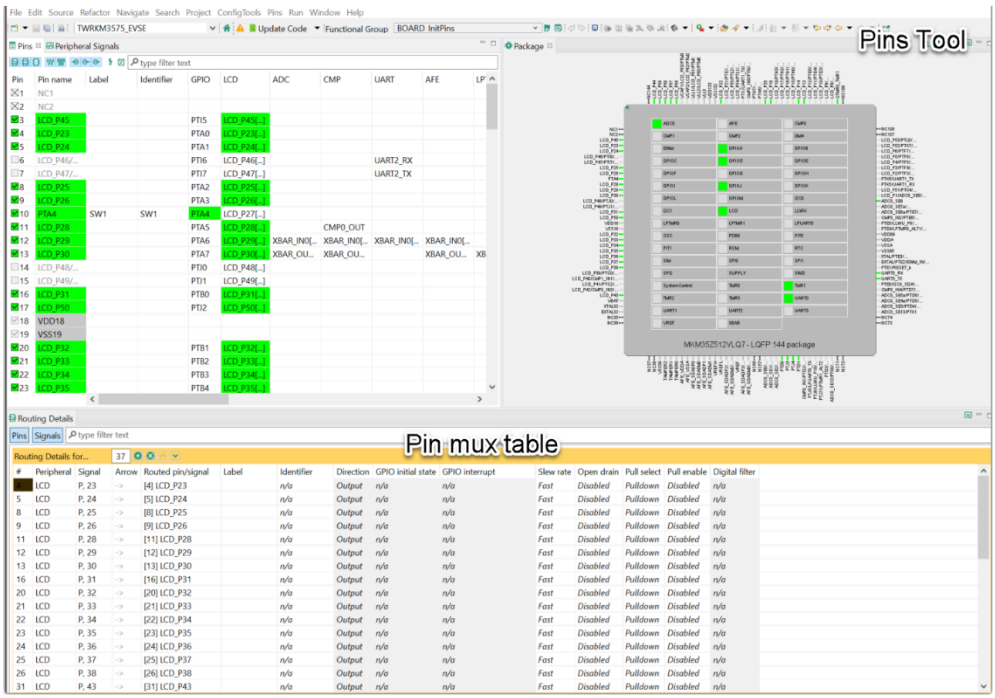


Figure 69. Pins table

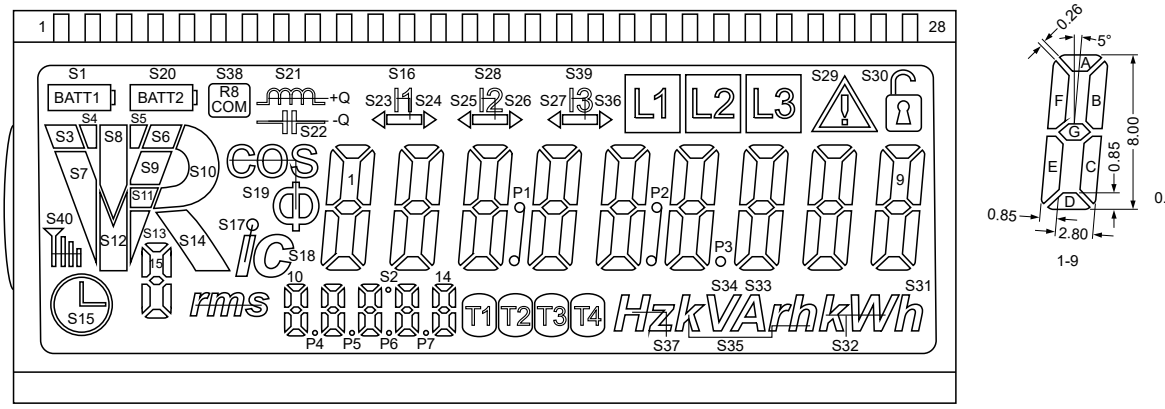
5. To export the pin table in a separate output file, follow the steps in the "Exporting the Pins table" section in the *MCUXpresso Config Tools User's Guide (IDE)* (document MCUXIDECTUG).

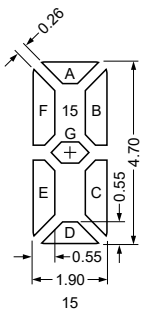
12.2 Meter LCD information

The following subsections describe the specific meter board LCD screen visual components and schematic.

12.2.1 LCD layout

Figure 70 graphically shows the layout of the meter LCD screen and a matrix of the pins enabing each individual element.





PIN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
COM1	P7	P6	P5	P4	S18	15D	S40	S3	S28	S26	S27	S39	S36	L1	L2	L3	S29	S30	S2	S15	COM1	\	\	\	\	\	\	\
COM2	14D	13D	12D	11D	10D	15E	S7	S4	S25	1A	2A	3A	4A	5A	6A	7A	8A	9A	P3	rms	\	COM2	\	\	\	\	\	\
COM3	14E	13E	12E	11E	10E	15C	S12	S8	S24	1F	2F	3F	4F	5F	6F	7F	8F	9F	S31	T1	\	\	COM3	\	\	\	\	\
COM4	14C	13C	12C	11C	10C	15G	S13	S5	S16	1B	2B	3B	4B	5B	6B	7B	8B	9B	S32	T2	\	\	\	COM4	\	\	\	\
COM5	14G	13G	12G	11G	10G	15F	S11	S9	S23	1G	2G	3G	4G	5G	6G	7G	8G	9G	S33	T3	\	\	\	\	COM5	\	\	\
COM6	14F	13F	12F	11F	10F	15B	S10	S6	S22	1E	2E	3E	4E	5E	6E	7E	8E	9E	S34	T4	\	\	\	\	\	COM6	\	\
COM7	14B	13B	12B	11B	10B	15A	S14	S1	S21	1C	2C	3C	4C	5C	6C	7C	8C	9C	P2	S37	\	\	\	\	\	\	COM7	\
COM8	14A	13A	12A	11A	10A	S17	S19	S20	S38	1D	2D	3D	4D	5D	6D	7C	8D	9D	P1	S35	\	\	\	\	\	\	\	COM8

aaa-056004

Figure 70. LCD layout

Table 13. List of specialized icons in the meter LCD screen

ICON_L1	ICON_P7	ICON_S14	ICON_S28
ICON_L2	ICON_S1	ICON_S15	ICON_S29
ICON_L3	ICON_S2	ICON_S16	ICON_S30
ICON_T1	ICON_S3	ICON_S17	ICON_S31
ICON_T2	ICON_S4	ICON_S18	ICON_S32
ICON_T3	ICON_S5	ICON_S19	ICON_S33
ICON_T4	ICON_S6	ICON_S20	ICON_S34
ICON_RMS	ICON_S7	ICON_S21	ICON_S35
ICON_P1	ICON_S8	ICON_S22	ICON_S36
ICON_P2	ICON_S9	ICON_S23	ICON_S37
ICON_P3	ICON_S10	ICON_S24	ICON_S38
ICON_P4	ICON_S11	ICON_S25	ICON_S39
ICON_P5	ICON_S12	ICON_S26	ICON_S40
ICON_P6	ICON_S13	ICON_S27	

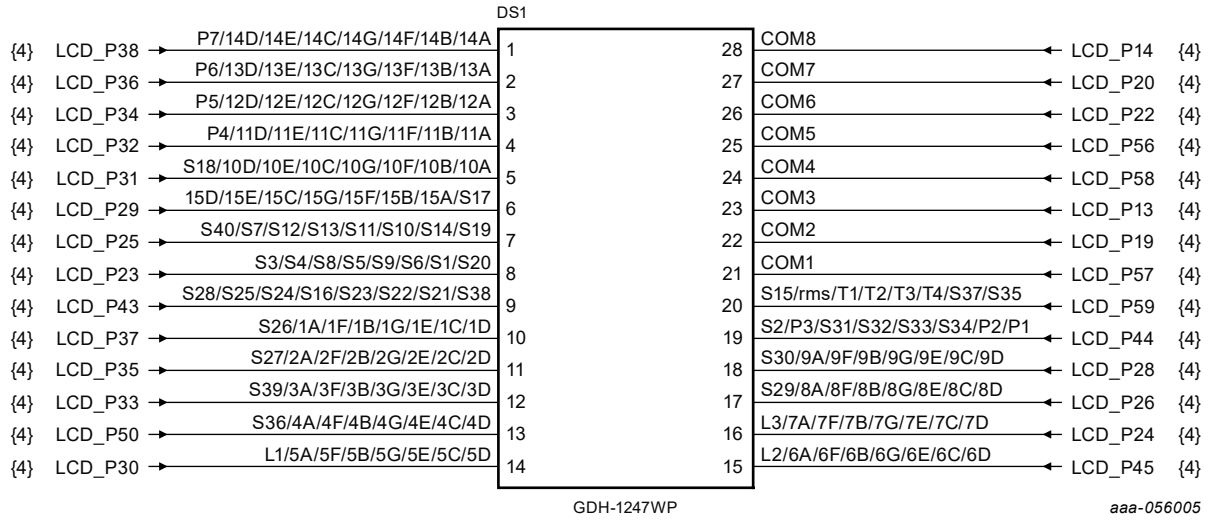


Figure 71. LCD schematic

12.3 Device programming

The following subsections describe the tools and means for programming the flash storage of EasyEVSE component boards.

12.3.1 UUU i.MX 93 programming

The build image could be flashed to the i.MX board using the UUU tool that can be downloaded from the following link: https://github.com/nxp-imx/mfgtools/releases.

12.3.2 Using blhost flash tool

In the case that the MCU LPC55 firmware on the EVSE-SIG-BRD needs to be updated, the following ISP method is provided, using blhost flash programming tool, which is part of the NXP Secure Provisioning SDK (SPSDK).

The SPSDK can be installed by cloning the repository https://github.com/nxp-mcuxpresso/spsdk/tree/master or by using the Python pip installer inside a Virtual Environment (venv):

```
C:\Users\nxabcdef>python3 -m venv venv
C:\Users\nxabcfdef>venv\Scripts\activate.bat
(venv) C:\Users\nxabcdef>pip install spsdk
```

The EVSE-SIG-BRD requires firmware versions 1.1.5 for EVSE and 1.1.1 for EV.

If the EVSE-SIG-BRD came as part of an EVSE kit, then the appropriate images must already be programmed. Otherwise, you must perform the steps below to check and update the firmware as needed.

You can check the firmware version of an EVSE-SIG-BRD by connecting to it in two different ways:

- From your host computer through a Serial to USB adapter; then open a serial console to the appropriate serial interface.
- From the UART console of the appropriate i.MX 93 board: In this situation, the EVSE-SIG-BRD is seen as the serial device /dev/ttyLP2. Then open a serial console to device /dev/ttyLP2 from within the i.MX 93 serial console.

Communication parameters with EVSE-SIG-BRD in either case are 115200 8N1.

You can check the EVSE-SIG-BRD firmware version with the command 'v':

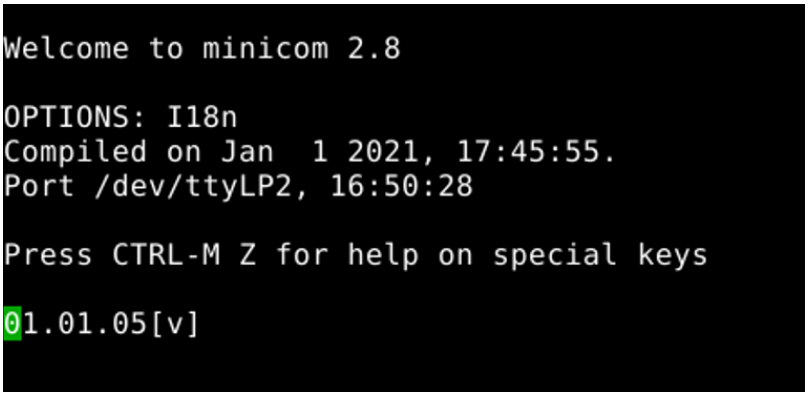


Figure 72. EVSE console

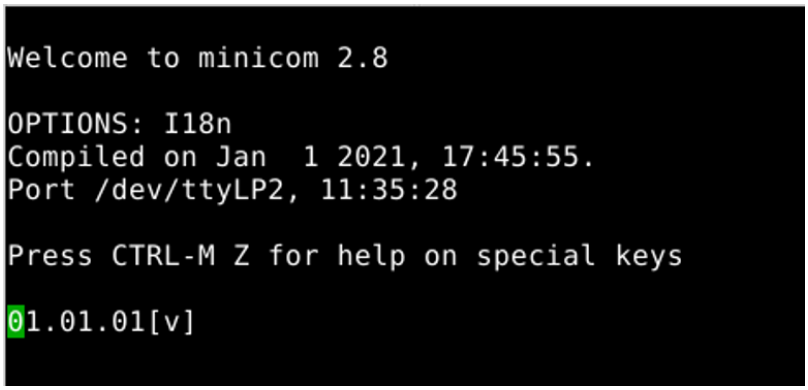


Figure 73. EV console

To program a new firmware on the EVSE-SIG-BRD, perform the following steps:

1. Open Jumper J29 to enter ISP boot mode.
2. Connect host UART of EVSE-SIG-BRD (J44) to PC USB through a serial port to USB adapter. If you are using an FTDI programmer, make sure it is a 3.3 V device (for example, TTL-232R-3V3-WE) and you connect TXD to HOST_UART_RXD, RXD to HOST_UART_TXD, and one of the GND pins.

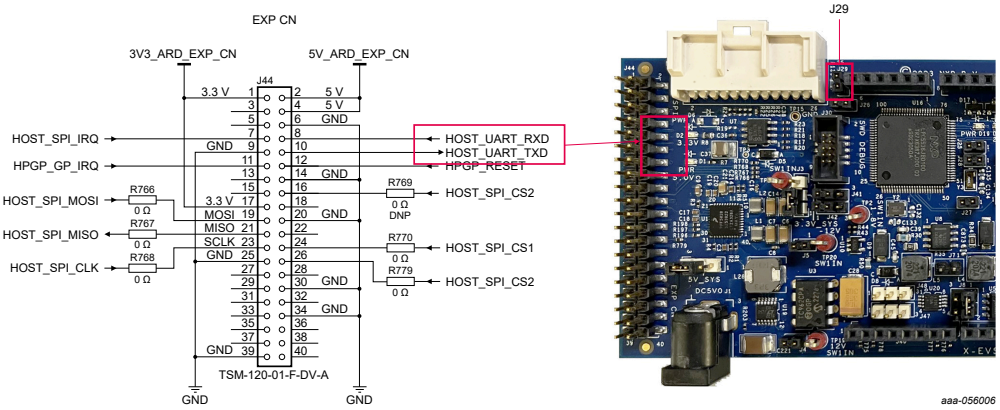


Figure 74. EVSE-SIG-BRD pin connections necessary to program the LPC55 firmware

Note: *Be sure that the active environment is active (venv).*

3. Discover the serial COM port using the `nxpdevscan` program.

```
(venv) C:\Users\nxa11506\venv\Scripts>nxpdevscan.exe
-------- Connected NXP SDIO Devices --------

-------- Connected NXP USB Devices --------

-------- Connected NXP UART Devices --------

Port: COM11
Type: mboot device

-------- Connected NXP SIO Devices --------
```

Figure 75. Using nxpdevscan to find the correct serial port

4. Erase the old firmware with the blhost command `flash-erase-all`.
5. Specify the firmware file from browse.
6. Flash the binary using the `write-memory` command to the flash address 0x00.

```
(venv) C:\Users\aaaaaaaa\venv\Scripts>blhost.exe -v -p COM11 flash-erase-all
INFO:spsdk.mboot.mcuboot:Connect: identifier='uart', device=COM11
INFO:spsdk.mboot.mcuboot:CMD: FlashEraseAll(mem_id=0)
INFO:spsdk.mboot.mcuboot:CMD: Status: 0 (0x0) Success.
Response status = 0 (0x0) Success.
INFO:spsdk.mboot.mcuboot:Closing: identifier='uart', device=COM11
(venv) C:\Users\nxa11506\venv\Scripts>blhost.exe -v -p COM11 write-memory 0 "C:\Users\nxa11506\Downloads\evsesigbrd
_sw_v1.1.5.bin"
INFO:spsdk.mboot.mcuboot:Connect: identifier='uart', device=COM11
INFO:spsdk.mboot.mcuboot:CMD: WriteMemory(address=0x00000000, length=197120, mem_id=0)
INFO:spsdk.mboot.mcuboot:CMD: GetProperty(MaxPacketSize, index=0)
INFO:spsdk.mboot.mcuboot:CMD: Status: 0 (0x0) Success.
INFO:spsdk.mboot.mcuboot:CMD: Max Packet Size = 256
INFO:spsdk.mboot.mcuboot:CMD: Status: 0 (0x0) Success.
INFO:spsdk.mboot.mcuboot:CMD: Successfully Send 197120 out of 197120 Bytes
Response status = 0 (0x0) Success.
Response word 1 = 197120 (0x30200)
INFO:spsdk.mboot.mcuboot:Closing: identifier='uart', device=COM11
```

Figure 76. Expected output of the execution of the write-memory program

7. Close the J29 when the flash process is complete and reset the board.

12.3.3 Using MCUXpresso GUI flash tool

For demonstration purposes, the below figures show the i.MX RT106x programmed with the SE050 VCOM binary. However, you can use this tool for other MCU targets.

1. Select the MCUXpresso project for the target to program.
2. Click the GUI Flash Tool icon.

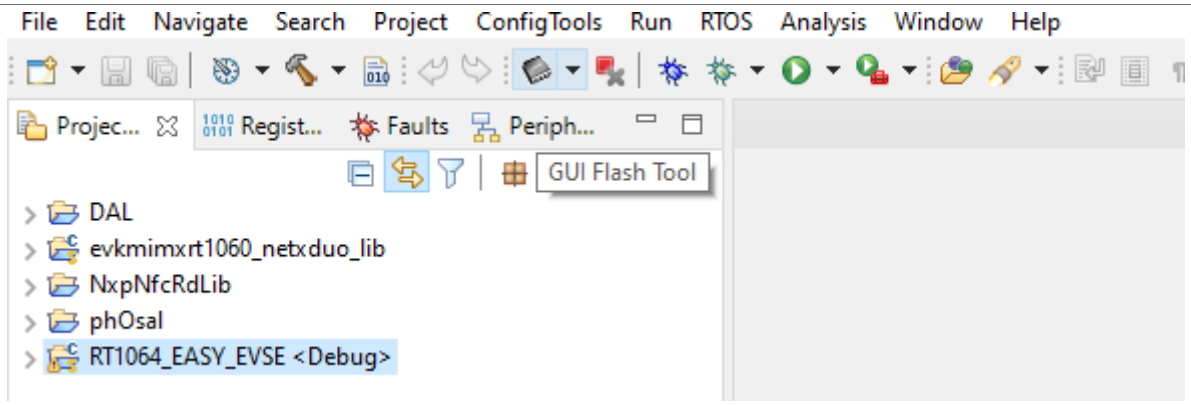


Figure 77. GUI Flash Tool icon

3. The IDE discovers the debugger.

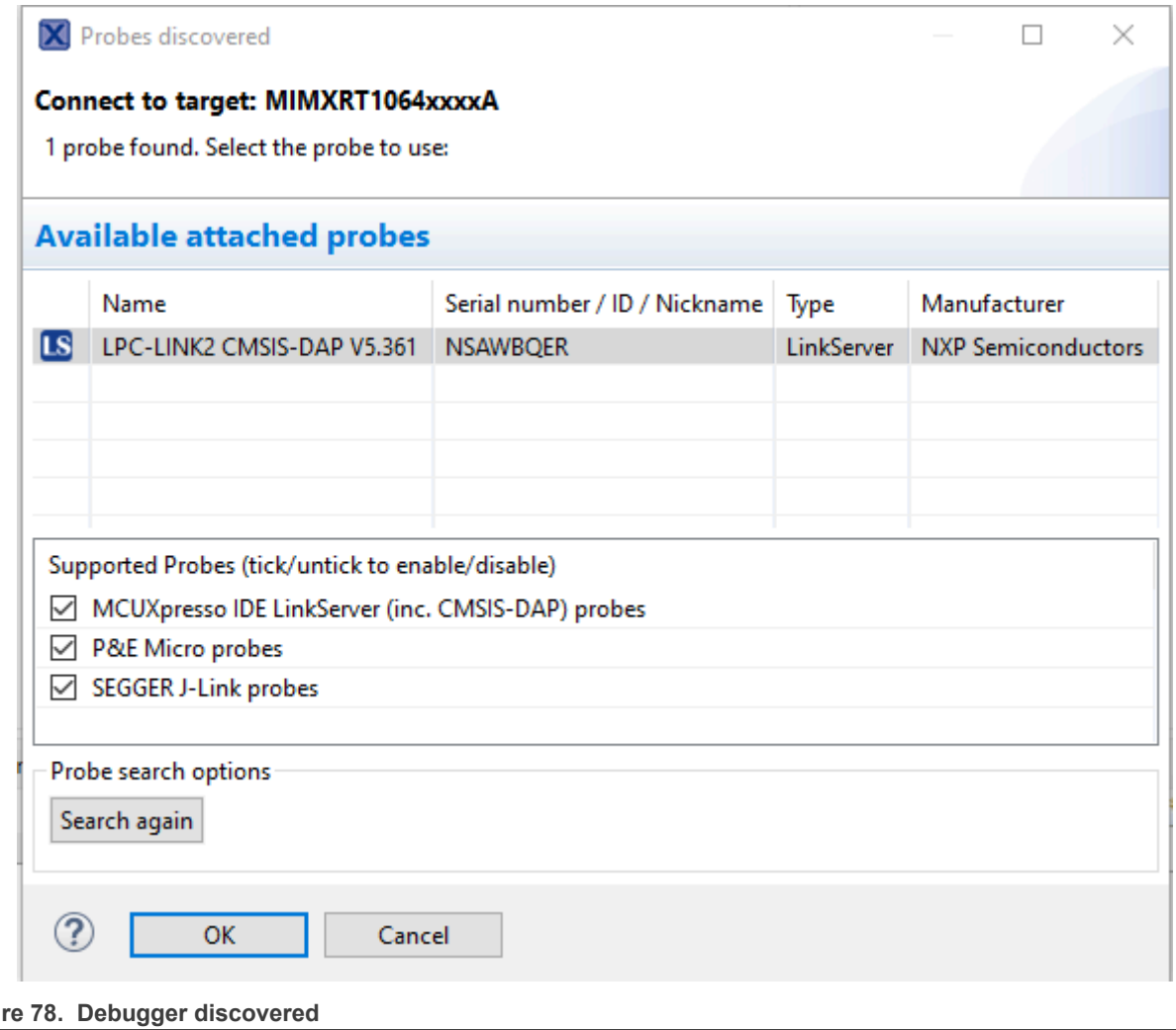


Figure 78. Debugger discovered

4. Specify the **File to program** and click **Run**. In this example, it is a binary file.

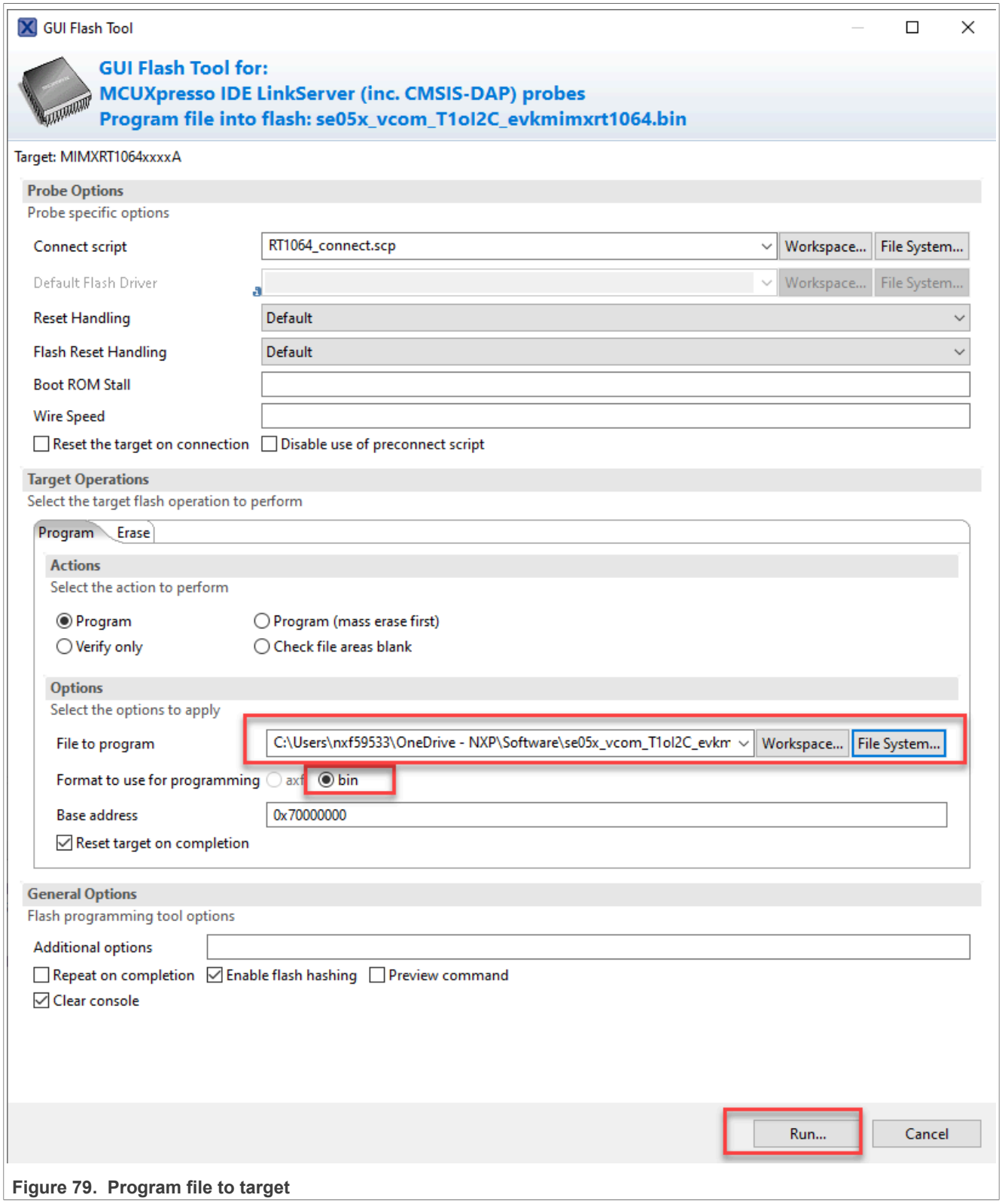


Figure 79. Program file to target

5. Verify the successful program operation.

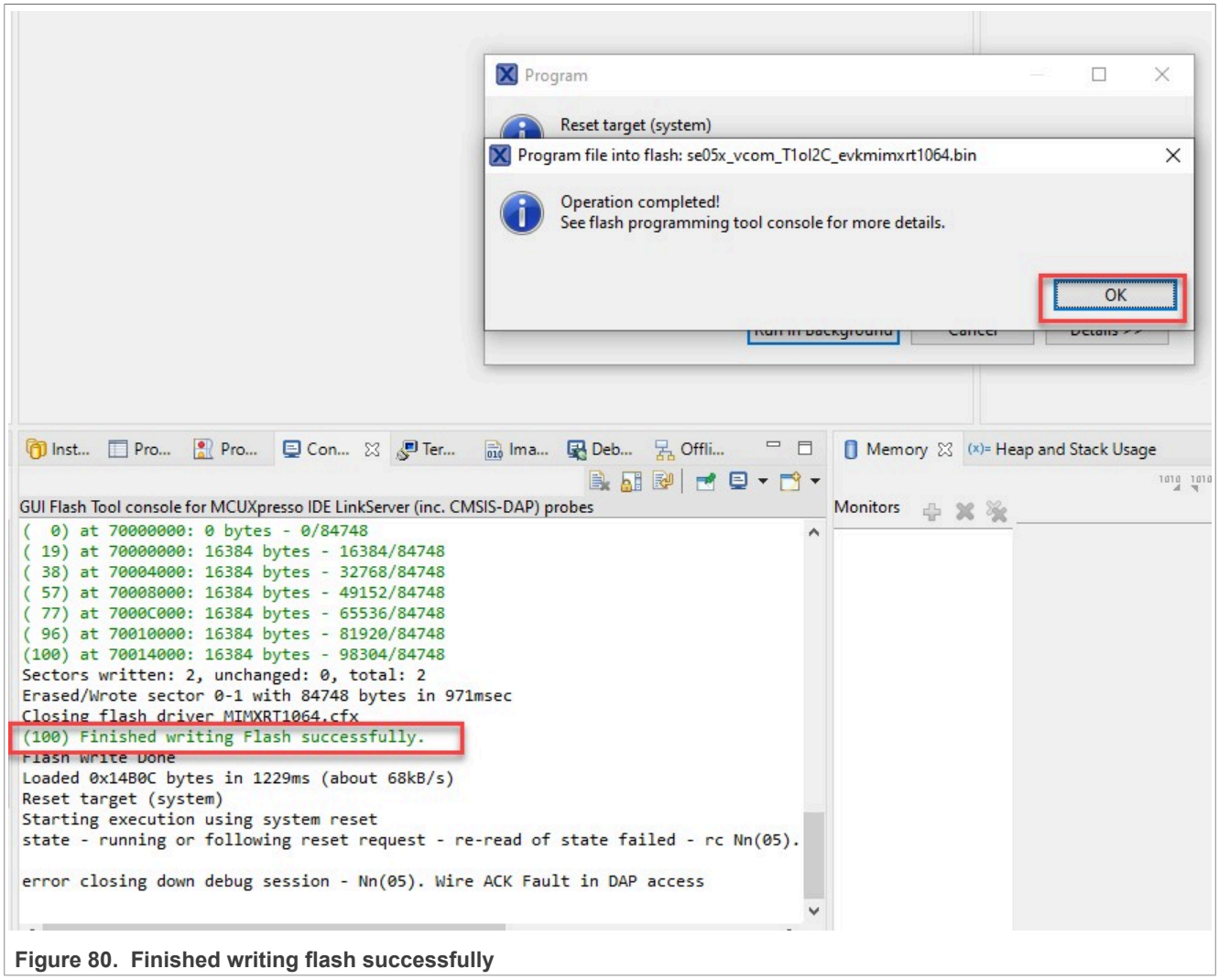


Figure 80. Finished writing flash successfully

6. Make sure that the target has the proper settings to boot the application. For example, the i.MX RT106x must be set to boot from internal flash memory.
   For further details on the GUI Flash tool, see the *MCUXpresso IDE - User Guide* (document MCUXPRESSO-UG).

12.4 EasyEVSE ROS clients data flow

Figure 81 shows the data flow between the EasyEVSE clients, with the publisher and subscribers presented in a graphical way.

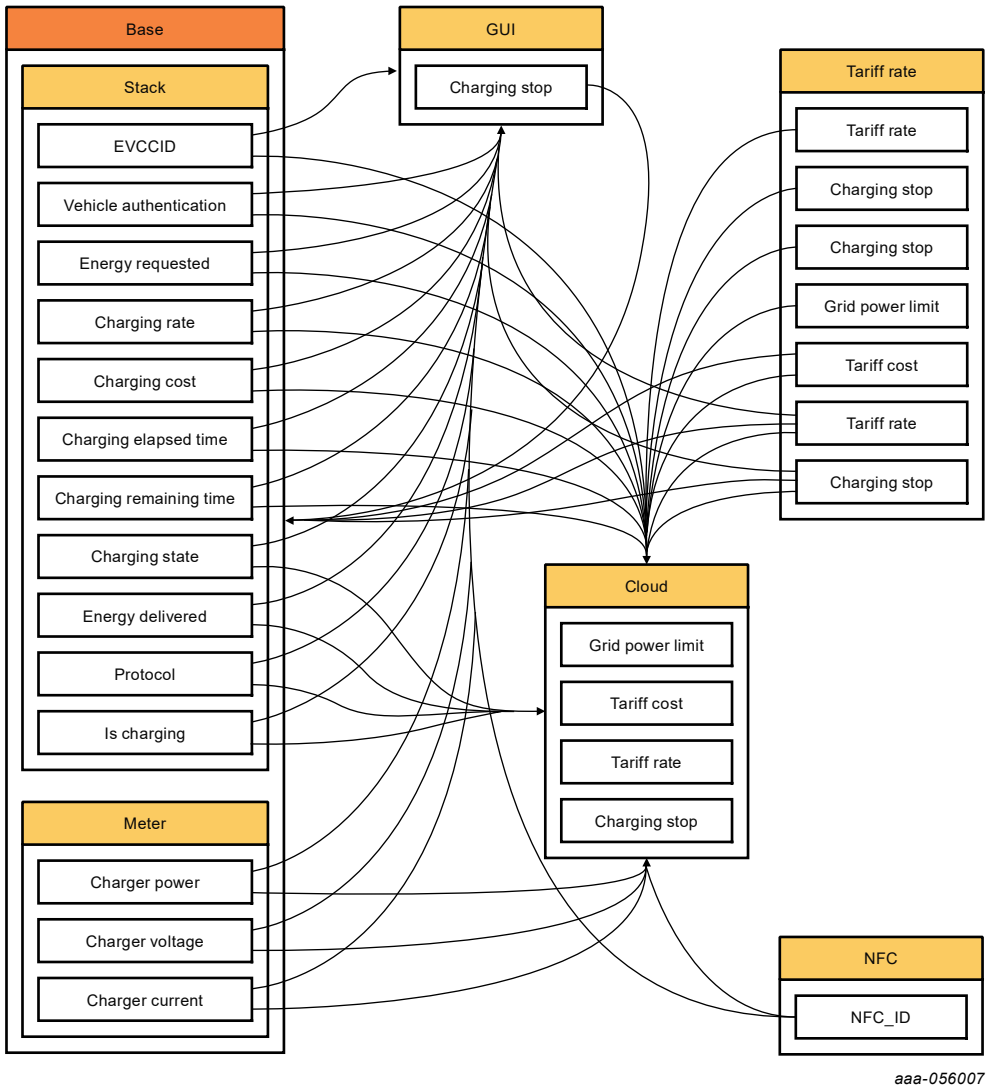


Figure 81. EasyEVSE data flow

12.5 References

- *Kinetis-M Three-Phase Power Meter Reference Design* (document: DRM147)
- *Kinetis-M One-Phase Power Meter Reference Design* (document: DRM143)
- *Low-Power Real-Time Algorithm for Metering Applications* (document: AN13259)
- *Filter-based Algorithm for Metering Applications* (document: AN4265)
- *Application Hints For Using Freescale Metering Libraries in Three-Phase Power Meters* (document: AN5007)
- *J1772 SAE standard* (https://www.sae.org/standards/content/j1772_201710/)
- *NXP EV Charging Whitepaper* (document EV-CHARGINGWP)
- *Smart Appliance Solutions Training – Cloud Solutions* (https://www.nxp.com/design/training/smart-appliance-solutions-training-cloud-solutions:TIP-SMART-APPLIANCE-SOLUTIONS-CLOUD-SOLUTIONS)
- *Azure RTOS ThreadX documentation* (https://docs.microsoft.com/en-us/azure/rtos/threadx/)
- *Azure IoT documentation* (https://docs.microsoft.com/en-us/azure/iot-fundamentals/)
- *MCUXpresso Config Tools User's Guide (IDE)* (document MCUXIDECTUG)

12.5.1 Reference designs

- *KM35Z512 based One-Phase Smart Power Meter Reference Design* (document: AN12837)
- *Kinetis-M One-Phase Power Meter Reference Design* (document: DRM143)
- *Kinetis-M Two-Phase Power Meter Reference Design* (document: DRM149)
- *Kinetis-M Three-Phase Power Meter Reference Design* (document: DRM147)

12.5.2 Meter library

- *Filter-Based Algorithm for Metering Applications* (document: AN4265)
- *FFT-Based Algorithm for Metering Applications* (document: AN4255)
- *Low-Power Real-Time Algorithm for Metering Applications* (document: AN13259)

12.5.3 Calibration

See the documents listed in Section 12.5.1 and Section 12.5.2.

12.5.4 Total harmonic distortion (THD)

The FFT-based metering library is used to calculate THD for voltage and current signals.

- *One-Phase Power Meter Reference Design* (document: DRM163)
- *Two-Phase Power Meter Reference Design* (document: DRM149)
- *FFT-based Metering Library for Metering Application* (document: AN4255)

13 Acronyms

Table 14 lists acronyms used in this document.

Table 14. Acronyms and definition

Acronyms	Definition
ROS	Robot operating system
V2G	Vehicle to grid
EVSE	Electric vehicle supply equipment
EIM	External identification means
HLC	High-level control
PLC	Powerline communication
SLAC	Signal level attenuation characterization
SDP	SECC discovery protocol
BC	Basic charging
HLC-C	High-level communication control
ECDHE	Elliptic curve diffie-hellman ephemeral
ECDSA	Elliptic curve digital signature algorithm
CBC	Cipher block chaining
GCM	Galois/counter mode
AES	Advanced encryption standard
CPO	Charge point operator
PKI	Public key infrastructure
SECC	Supply equipment communication controller
TLS	Transport layer security
ECDH	Elliptic-curve diffie-hellman
TEE	Trusted execution environment
DPS	Device provisioning service
CS	Command string
GUI	Graphical user interface
AFE	Analog front end
MMAU	Memory mapped arithmetic unit

14 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

15 Revision history

Table 15 summarizes the revisions to this document.

Table 15. Revision history

Document ID	Release date	Description
UM12110 v.1.0	18 June 2024	Initial public release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

1 Introduction 2
2 Preparation 3
2.1 Get the required hardware 3
2.2 Get required software 3
2.3 EasyEVSE development environment 4
2.3.1 EasyEVSE software 4
2.3.2 MCUXpresso IDE and SDK 4
2.3.3 Lumissil firmware binaries 4
2.3.4 SEVENSTAX stack library and application source code 5
3 EasyEVSE 6
3.1 SAE J1772 7
3.1.1 Charge state machine 7
3.1.2 Control pilot 8
4 Host controller 9
4.1 EVSE software structure 9
4.2 ROS framework 10
4.3 ROS node example: Business logic client 10
5 ISO 15118 12
5.1 ISO 15118 overview 12
5.1.1 Example charging sequence 12
5.1.2 SEVENSTAX 13
5.2 Basic charging to high-level communication charging (HLC-C) 14
5.2.1 EVSE process for BC to HLC-C 14
5.2.2 EV process for BC to HLC-C 15
6 EVSE-SIG-BRD 17
6.1 Introduction 17
6.2 Functional description 17
6.2.1 Proximity pilot 17
6.2.2 Control pilot 18
6.2.3 J1772 PWM 19
6.2.4 GFCI detection circuit 20
6.2.5 Relay driver 21
6.2.6 UART bridge 21
6.3 HomePlug Green PHY 23
6.3.1 SPI slave interface 24
6.3.2 MII PHY interface 24
6.3.3 Boot strapping CG5317 25
7 Security 27
7.1 Credentials provisioning 29
7.1.1 EdgeLock 2Go provisioning 29
7.1.2 Manual provisioning 29
7.1.3 Verify credentials 30
7.1.4 Upload root CA to Azure IoT Central application 31
7.2 Transport layer security (TLS) 31
7.2.1 Hello phase 32
7.2.2 Server key exchange phase 33
7.2.3 Client key exchange phase 34
7.2.4 Secret key calculation phase 35
7.3 Architecture Overview 36
7.4 EV/EVSE TLS handshake 37
7.5 Cloud TLS connection 38
8 IoT and connectivity 39
8.1 Connection setup 39
8.1.1 Select enrollment and authentication-attestation method 39
8.1.2 DPS onboarding 41
8.1.3 Connect state machine 42
8.1.4 Connected state 43
8.2 Implementation details 43
8.3 EasyEVSE dashboard 51
9 GUI 55
9.1 GUI design overview 55
9.2 Project structure 55
9.3 Functionalities 56
9.4 Data sending 56
9.5 Data reading 57
9.6 Logging 57
9.7 Transition between pages 57
9.8 GUI design 58
9.9 Main menu page 58
9.10 NFC Card page 59
9.11 EVSE Status page 59
9.12 Vehicle Settings page 60
9.13 Meter Menu page 60
10 Near-field communications (NFC) 62
10.1 PN7160 plug and play NFC controller 62
10.2 NXP NFC kernel driver 62
10.3 PN71xx NFC library 63
10.4 NFC library operation theory 64
10.5 Application executing 64
10.6 NFC client implementation 65
11 Meter 66
11.1 Introduction 66
11.2 Application meter vs simulated meter 67
11.2.1 Topology 68
11.2.2 Analog circuits 68
11.2.3 Meter library configuration 69
11.2.4 UART isolation board 69
11.3 Meter project 70
11.3.1 Run meter 71
11.3.1.1 SysTick timer handler 71
11.3.1.2 Potentiometer sample 72
11.3.1.3 Processing variables with the meter library 72
11.3.2 Communication with the host controller 73
11.3.3 Displaying data on LCD 74
12 Appendix 76
12.1 TWR-KM3x pin table 76
12.2 Meter LCD information 77
12.2.1 LCD layout 77
12.3 Device programming 78
12.3.1 UUU i.MX 93 programming 78
12.3.2 Using blhost flash tool 78
12.3.3 Using MCUXpresso GUI flash tool 80
12.4 EasyEVSE ROS clients data flow 83
12.5 References 84
12.5.1 Reference designs 85
12.5.2 Meter library 85
12.5.3 Calibration 85
12.5.4 Total harmonic distortion (THD) 85

13 Acronyms 86
14 Note about the source code in the document 87
15 Revision history 88
Legal information 89