AWS Libraries for S32K3

AWS Libraries for S32K3 User Manual

Rev. 0 — 3 October 2023 User manual

1 Introduction

This User Manual describes *AWS Libraries for S32K3*.

This package provides support for secure connection for S32K3 devices to AWS Cloud services like *AWS IoT Core* or to more powerful devices running *AWS IoT Greengrass*.

The *AWS Libraries for S32K3* product integrates the following open-source FreeRTOS LTS libraries:

- coreMQTT
- coreMQTT Agent
- coreJSON
- corePKCS11
- Backoff Algorithm
- OTA Agent
- Device Shadow
- Device Defender

This release is compatible with *S32K3 RTD AUTOSAR 4.4 Version 2.0.1*, *FreeRTOS for S32K3 2.0.1 HF01*, and *TCPIP_STACK for S32K3 version 1.0.1 HF1*.

1.1 Overview

With the automotive industry heading towards software defined vehicles, OEMs are seeking ways to integrate cloud connectivity with edge devices in vehicles, aiming to enhance both vehicle performance and customer experience utilizing the advancements in IoT, AI and data analytics.

However, this shift to connectivity has its own challenges:

- **Secure Connectivity:** Challenges in securely connecting and managing a multitude of devices. Due to the sensitive nature of the data, the data is prone to cyber threats comprising driver safety.
- **Data Management:** Handling a vast amount of data generated by IoT devices can be daunting.

Prebuilt integrations like *AWS Libraries for S32K3* (hardware and the cloud providers) address the problem of connectivity and security challenges that automakers face when building applications, thereby accelerating their digital transformation initiatives by unlocking the value of vehicle data and leveraging edge-to-cloud services.

AWS Libraries for S32K3:

The *AWS Libraries for S32K3* product provides support for secure connection of the S32K3 family of devices to AWS Cloud services or to more powerful edge devices running AWS IoT Greengrass.

S32K3:

The 32-bit S32K3 AEC-Q100 qualified MCUs are a scalable family of Arm® Cortex®-M7-based microcontrollers in single, dual, and lockstep core configurations, supporting up to ASIL D functional safety for automotive and industrial applications.

S32K3 MCUs feature hardware security engine (HSE) with NXP firmware, support for firmware over-the-air (FOTA) updates and free ISO 26262 compliant Real-Time Drivers (RTD) for AUTOSAR® and non-AUTOSAR.

1.2 High level architecture

The *AWS Libraries for S32K3* product and its surroundings are depicted in Figure 1.

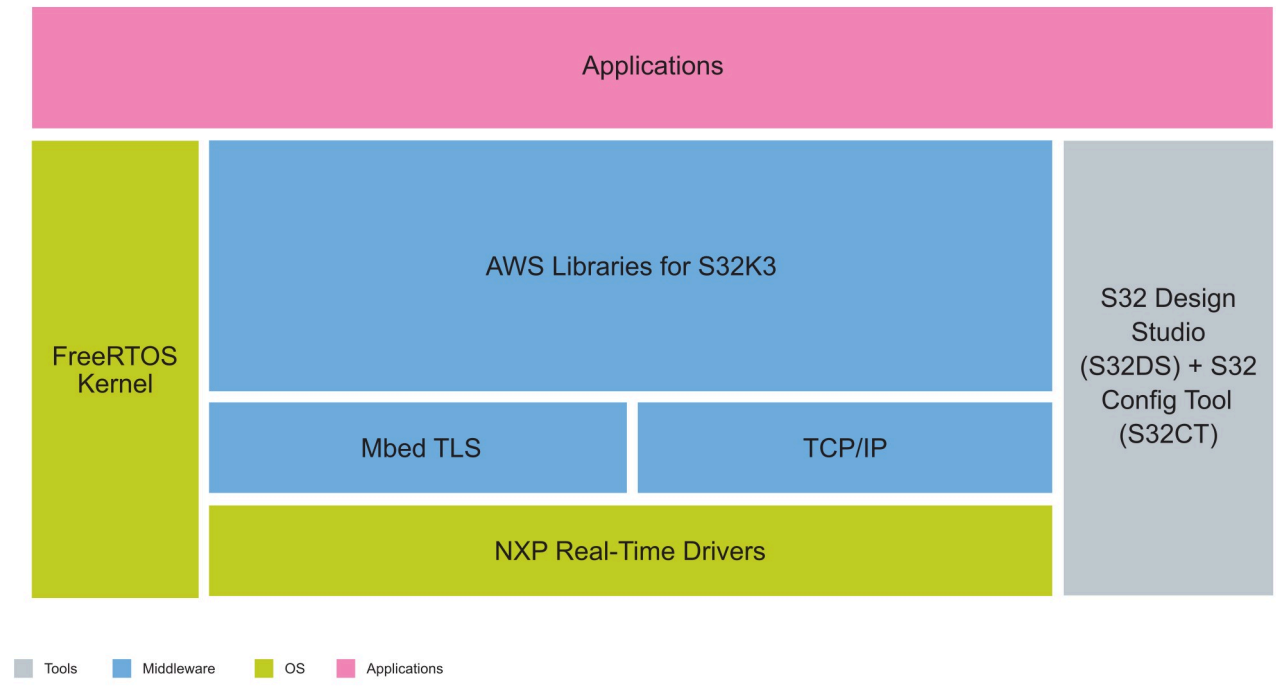

Figure 1. NXP software ecosystem

In terms of software ecosystem, the *AWS Libraries for S32K3* product is running on top of other existing NXP software:

- Real-Time Drivers
- TCP/IP stack
- FreeRTOS
- Mbed TLS (delivered as part of AWS Libraries for S32K3)

By using *AWS Libraries for S32K3* and its dependencies, integrated with NXP's IDE S32 Design Studio, users can build applications connecting to AWS Cloud services.

The internal architecture of *AWS Libraries for S32K3* is presented in Figure 2.

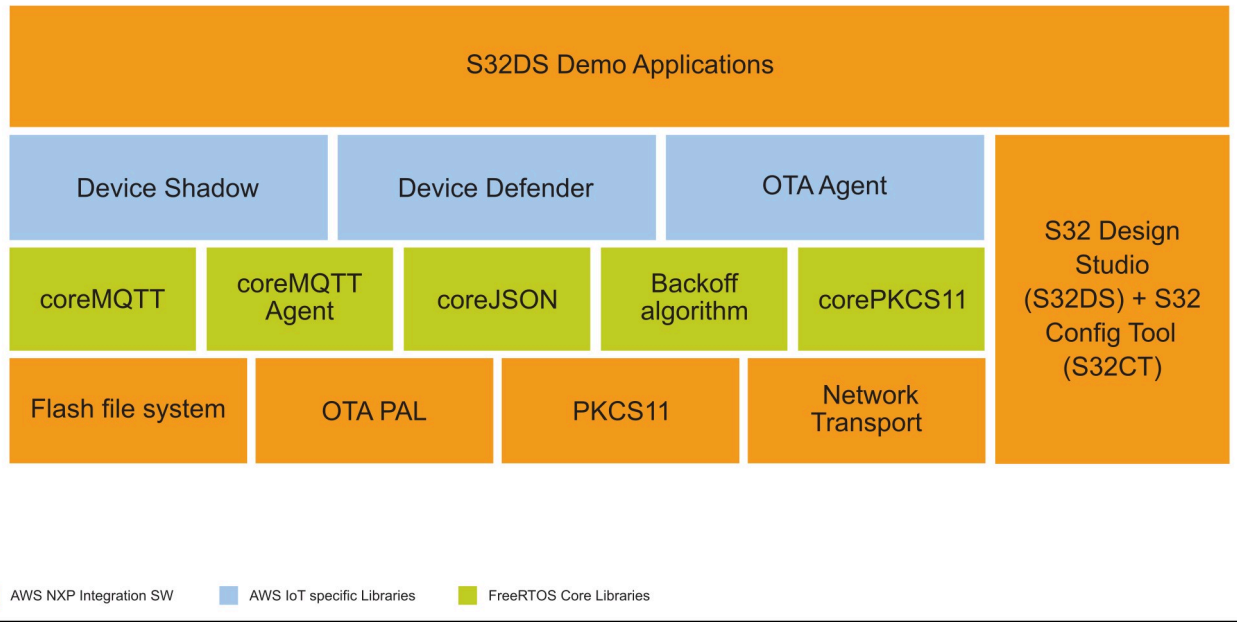

Figure 2. AWS libraries for S32K3 architecture

The most important functionalities provided by individual *AWS Libraries for S32K3* components are:

coreMQTT

- Implements the client side of the MQTT protocol.
- Provides a high-level API to connect to an MQTT broker, subscribe or unsubscribe to a topic, publish a message to a topic and receive incoming messages.
- Fully synchronous API, to allow applications to completely manage their concurrency and multi-threading method.
- Exposes a low-level serializer/de-serializer API.

coreMQTT Agent

- High level API that adds thread safety to the coreMQTT library.
- Allows the user to create a dedicated MQTT agent task that manages an MQTT connection in the background and doesn't need any intervention from other tasks.

coreJSON

- Parser that supports key lookups while also strictly enforcing the ECMA-404 JSON standard.

corePKCS11

- Implements a subset of the PKCS #11 API required to establish a secure connection to AWS IoT.
- Verification of the contents of a message.
- Signing messages.
- Management of certificates and keys.
- Generation of random numbers.

Backoff Algorithm

- Utility library to space out repeated retransmissions of the same block of data, to avoid network congestion.

- Calculates backoff period for retrying network operations (like failed network connection with server) using an exponential backoff with jitter algorithm.

OTA Agent

- Update device firmware.
- Register for notifications or poll for new update requests that are available.
- Receive, parse and validate the update request.
- Download and verify the file according to the information in the update request.
- Run a self-test before activating the received update to ensure the functional validity of the update.
- Update the status of the device.

Device Shadow

- Send commands to the AWS IoT Device Shadow service over MQTT to query the latest known device state, or to change the state

Device Defender

- Send security metrics from the device to AWS IoT Device Defender.
- API to compose and recognize the MQTT topic strings used by AWS IoT Device Defender.

From the libraries presented above, the *corePKCS11*, *coreMQTT* and *OTA Agent* libraries require porting for the NXP microcontrollers. Details on how this porting is implemented can be found in Section 4.

These libraries can be configured by using the *S32 Configuration Tool* embedded in *S32 Design Studio*.

For easy development startup, several *S32 Design Studio* examples are provided, presented in Section 6.

2 Hardware and software prerequisites

2.1 Hardware prerequisites

This guide and software examples assume using **S32K3X4EVB-T172 Evaluation and Development Board (EVB)** for general purpose industrial and automotive applications. In the case of the usage of a different EVB, compare the hardware differences and modify the configuration appropriately.

Based on the 32-bit Arm Cortex-M7 S32K3 MCU in a 172 HDQFP package, the S32K3X4EVB-T172 offers dual cores configured in lockstep mode, ASIL D safety capable hardware, HSE security engine, OTA support, advanced connectivity and low power.

The S32K3X4EVB-T172 offers a standard-based form factor compatible with the Arduino® UNO pin layout, providing a broad range of expansion board options for quick application prototyping and demonstration.

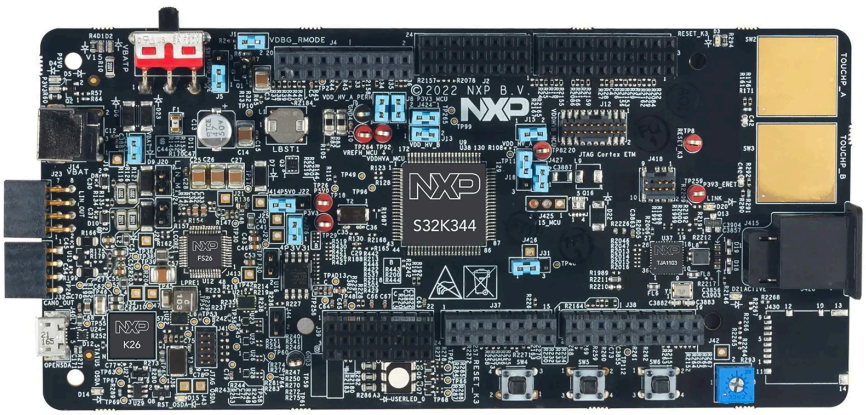

Figure 3. S32K3X4EVB-T172

Additional information about the EVB can be found in the HW User Manual (sign in on the NXP website required).

Visit S32K3X4EVB-T172 web page for general information including features, block diagrams and design resources and consult the Getting Started with the S32K3X4EVB-T172 Evaluation Board.

The **+12 V power supply** and **micro USB cable** are not part of the package, so they must be provided by the user. The +12 V connector is a center-positive barrel type with outer 5.5 mm and inner 2.1 mm diameter.

Additionally, all demos require internet connection and in order to test them, an Ethernet Media Converter is required since S32K3X4EVB-T172 board contains automotive 100BaseT1 Ethernet Physical Layer (PHY). For example, the NXP RDDRONE-T1ADAPT can be used to convert to the standard 100BASE-TX interface.

2.2 Software prerequisites

The following NXP Software products must be installed.

Table 1. Software prerequisites

Name	Version	Description
S32 Design Studio for S32 Platform	3.4 Update 3	Complimentary integrated development environment (IDE) for automotive and ultra-reliable Arm-based microcontrollers and processors that enable editing, compiling and debugging of designs
S32K3 Real-Time Drivers	2.0.1	Drivers set supporting real-time software on AUTOSAR and non-AUTOSAR applications targeting Arm Cortex-M cores and ISO 26262 compliance for all software layers
FreeRTOS for S32K3	2.0.1 HF01	The implementation of the FreeRTOS v10.4.6 kernel version for the S32K3 microcontrollers family
SW32K3 TCPIP Stack	1.0.1 HF01	Software library that implements a port of the lwIP stack for the S32K3 microcontrollers family
S32K3 AWS IoT Core	1.0.0	The FreeRTOS LTS libraries port for S32K3, adding AWS Cloud connectivity to the S32K344 platform (the current product)

3 Download and install software

Information on how to get started with using **AWS Libraries for S32K3** on the **S32K3X4EVB-T172 Evaluation and Development Board (EVB)**, including download locations, installation steps and running of the examples, can be found here.

4 AWS libraries integration

4.1 coreMQTT

The coreMQTT library is a client implementation of the MQTT standard. The MQTT standard provides a lightweight publish/subscribe messaging protocol that runs on top of TCP/IP and is often used in Machine to Machine (M2M) and Internet of Things (IoT) use cases.

The library provides a high-level API to connect to an MQTT broker, subscribe or unsubscribe to a topic, publish a message to a topic and receive incoming messages. The library also exposes a low-level serializer/deserializer API. This low-level API handles formatting and parsing messages, leaving the application full, zero-overhead control over the network connection to the MQTT broker.

The library is decoupled from the underlying network drivers through a two-function send and receive transport interface. The application writer can select an existing transport interface or implement their own, as appropriate for their application. NXP reuses the FreeRTOS-Plus network transport implementation based on Mbed TLS from here.

This layering can be observed in the picture below:

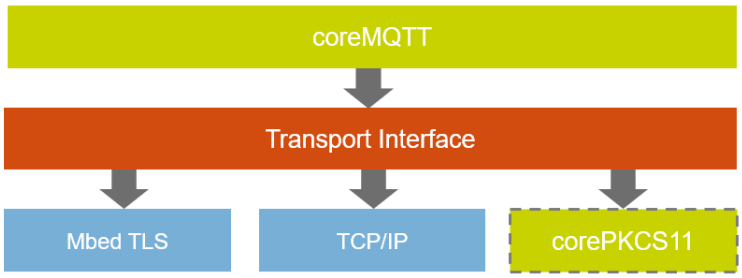

Figure 4. coreMQTT layering

- **coreMQTT** - client implementation of the MQTT standard.
- **Transport Interface** - coreMQTT has no dependency on any particular TCP/IP stack. Therefore, in order to use it, a Transport Interface structure containing function pointers and context data is required to send and receive data on a single network connection.
- **Mbed TLS** - TLS library used for creating and exchanging data via a secure connection.
- **TCP/IP** - light-weight implementation of the TCP/IP protocol suite.
- **corePKCS11** - PKCS11 implementation used for handling and using the device certificate and private key during the TLS connection (used only for the Mbed TLS + PKCS11 transport interface).

NXP provides two transport interfaces, from which only one can be selected at a time and where the transport interface selection depends on application requirements.

- Mbed TLS – uses Mbed TLS for the transport protocol and device certificate and key as plaintext buffers;
- Mbed TLS + PKCS11 – uses Mbed TLS for the transport protocol and device certificate and key as PKCS11 objects.

4.2 corePKCS11

The Public Key Cryptography Standard #11 defines a platform-independent API to manage and use cryptographic tokens. PKCS #11 refers to the API defined by the standard and to the standard itself. The PKCS #11 cryptographic API abstracts key storage, get/set properties for cryptographic objects, and session semantics. It's widely used for manipulating common cryptographic objects, and it's important because the

functions it specifies allow application software to use, create, modify, and delete cryptographic objects, without ever exposing those objects to the application's memory. For example, FreeRTOS AWS reference integrations use a small subset of the PKCS #11 API to access the secret (private) key necessary to create a network connection that is authenticated and secured by the Transport Layer Security (TLS) protocol without the application ever 'seeing' the key.

The corePKCS11 library contains a software-based mock implementation of the PKCS #11 interface (API) that uses the cryptographic functionality provided by Mbed TLS. This software mock was replaced by NXP with a port over the NXP Mbed TLS library integrated with NXP (`core_pkcs11_mbedtls_hse.c`).

This layer further calls either Mbed TLS APIs (when the required functionality is already implemented) or functions from the extended corePKCS11 PAL (a Peripheral Abstraction Layer provided by AWS to help the vendors port the corePKCS11 Mbed TLS mock implementation - and extended by NXP to allow integration with HSE features).

The extended corePKCS11 PAL makes use of:

- Mbed TLS API: for key management, sign/verify operations and random number generation
- Flash file system: for certificate management.

This layering can be observed in the picture below:

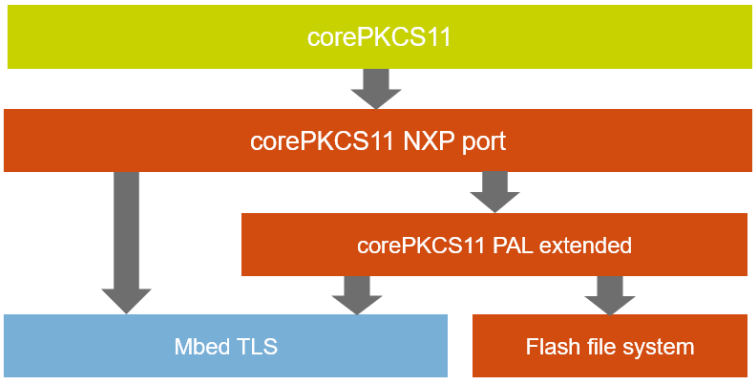

Figure 5. corePKCS11 layering

4.2.1 Device provisioning

Provisioning the device with credentials is showcased in the *aws_pkcs11_mqtt_s32k344* demo. In addition to the standard MQTT example, this example uses the *corePKCS11* library to provision the device with the client certificate and private key. The *corePKCS11* library is integrated with the S32K3 hardware security engine (HSE), as shown in Section Section 4.2.

An additional macro was added to the configuration in `demo_config.h`:

```
#define democonfigPROVISION_DEVICE
```

This macro must be defined to enable provisioning of the device with the client certificate and private key defined below. After running the example once, the macro can be undefined, as the certificate will be stored in FLASH and the private key in the HSE.

The provisioning is implemented in the `vDevModeKeyProvisioning()` function implemented in `src/aws_dev_mode_key_provisioning.c`. This function is invoked by the `InitTask()` located in `src/main.c`, if the `democonfigPROVISION_DEVICE` macro is defined.

4.2.1.1 Provisioning private key

This is an overview of the steps done in the example for provisioning the device with a private key using the *corePKCS11* library:

1. Initialize the PKCS11 module - `xInitializePkcs11Token()`
2. Initialize a PKCS11 session - `xInitializePkcs11Session()`
3. Get PKCS11 function list - `C_GetFunctionList()`
4. Fill in private key template (`CK_ATTRIBUTE`). Important elements in the template:
   - Key class (`CKA_CLASS`) - must be `CKO_PRIVATE_KEY`
   - Key type (`CKA_KEY_TYPE`) - supported `CKK_EC/CKK_RSA`
   - Key label (`CKA_LABEL`) - string helping with key identification
   - Token (`CKA_TOKEN`) - must be `xTrue`
   - Sign (`CKA_SIGN`) - must be `xTrue`
   - (**only for EC type of keys**) EC parameters (`CKA_EC_PARAMS`), Value (`CKA_VALUE`)
     - If the key is provided in PEM/DER format, the parameters can be obtained using Mbed TLS by parsing the key using `mbedtls_pk_parse_key()` and:
       - using `mbedtls_mpi_write_binary()` for obtaining the `d` parameter (for `CKA_VALUE`)
       - prepending `\x06\x08` to the Mbed TLS curve macro - e.g. `MBEDTLS_OID_EC_GRP_SECP256R1` (for `CKA_EC_PARAMS`)
   - (**only for RSA type of keys**) Modulus (`CKA_MODULUS`), private exponent (`CKA_PRIVATE_EXPONENT`), public exponent (`CKA_PUBLIC_EXPONENT`)
     - If the key is provided in PK format, the parameters can be obtained using Mbed TLS by parsing the key using `mbedtls_pk_parse_key()` and:
       - using `mbedtls_rsa_export_raw()` for obtaining the modulus, `d` and `e` (for `CKA_MODULUS`, `CKA_PRIVATE_EXPONENT`, `CKA_PUBLIC_EXPONENT`)
5. Create the PKCS11 object - `C_CreateObject()`
   - This call will lead to the `PKCS11_NXP_PAL_SaveKey()` function being invoked, which will call `KeyStoreMgmt_ImportKey()`. The latter function is implemented as part of the Mbed TLS integration with HSE and will lead to the key being loaded inside the HSE, with a handle being retured for further usage.

A reference implementation of these steps can be found in the `xProvisionPrivateKey()` function.

4.2.1.2 Provisioning certificate

This is an overview of the steps done in the example for provisioning the device with a private key using the *corePKCS11* library:

1. Initialize the PKCS11 module - `xInitializePkcs11Token()`
2. Initialize a PKCS11 session - `xInitializePkcs11Session()`
3. Get PKCS11 function list - `C_GetFunctionList()`
4. Fill in client certificate key template (`CK_ATTRIBUTE`). Important elements in the template:
   - Object class (`CKA_CLASS`) - must be `CKO_CERTIFICATE`
   - Certificate type (`CKA_CERTIFICATE_TYPE`) - must be `CKC_X_509`
   - Certificate label (`CKA_LABEL`) - string helping with certificate identification
   - Token (`CKA_TOKEN`) - must be `xTrue`
   - Sign (`CKA_SIGN`) - must be `xTrue`

- Subject (`CKA_SUBJECT`) - string used as certificate subject, currently not used
- Value (`CKA_VALUE`) - certificate in DER format
  - If the certificate is provided in PEM format, it can be converted using the `convert_pem_to_der()` function implemented in `<AWS Libraries source code location>\libraries\FreeRTOS\corePKCS11\source\dependency\3rdparty\mbedtls_utils\mbedtls_utils.c`.

5. Create the PKCS11 object - `C_CreateObject()`
   - This call will lead to the `PKCS11_PAL_SaveObject()` function being invoked. This function will call the flash file wrapper function `flash_SaveFile()` to further store the certificate in flash.

A reference implementation of these steps can be found in the `xProvisionCertificate()` function.

4.3 Over the air (OTA) updates

The AWS IoT Over-the-air (OTA) update library enables you to manage the notification, download, and verification of firmware updates for FreeRTOS devices using HTTP or MQTT as the protocol. By using the OTA Agent library, you can logically separate firmware updates and the application running on your devices. The OTA Agent can share a network connection with the application. By sharing a network connection, you can potentially save a significant amount of RAM.

In addition, the OTA Agent library lets you define application-specific logic for testing, committing, or rolling back a firmware update.

This library's APIs provide these major functions:

- Register for notifications or poll for new update requests that are available.
- Receive, parse and validate the update request.
- Download and verify the file according to the information in the update request.
- Run a self-test before activating the received update to ensure the functional validity of the update.
- Update the status of the device.

Porting of the OTA Agent library is done by implementing a set of platform abstraction layer (PAL) functions (`aws_iot_ota_pal.c`).

The current NXP OTA PAL implementation is using the flash file system for storing the new image in the inactive partition, Mbed TLS for verifying the signed image, the HSE IP driver for switching the active partition and the Power IP driver for triggering the reset.

The layering can be observed in the picture below:

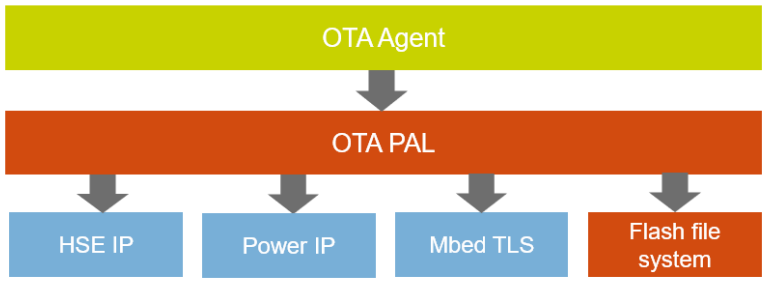

Figure 6. OTA layering

- **OTA Agent** - library enabling management of notifications of newly available updates, downloading of the updates, and performing cryptographic verification of the firmware updates.
- **OTA PAL** - peripheral abstraction layer defined by AWS and implemented by NXP to store incoming blocks of a new image in flash, verify it and switch execution to the new image.
- **HSE IP** - RTD driver used for switching execution to passive partition in the OTA PAL implementation.

- **Power IP** - RTD driver used for resetting the device in the OTA PAL implementation.
- **Mbed TLS** - TLS library used for verifying the signed image.
- **Flash file system** - simple flash file system used for storing the new image and reading the code signing certificate.

The process of an OTA update is the following and is depicted below.

1. The *OTA Agent* requests a job document
2. The *OTA Update Manager* sends a job document containing information regarding the update job
3. The *OTA Agent* parses the job document and further invokes `prvPAL_CreateFileForRx()` *(OTA PAL)*
4. The *OTA PAL* uses the flash file system to erase the passive blocks
5. The *OTA Agent* requests a file block
6. The *OTA Update Manager* sends the first file block
7. The *OTA Agent* receives the file block and invokes `prvPAL_WriteBlock()` *(OTA PAL)*
8. The *OTA PAL* uses the flash file system write the file block in the passive blocks area

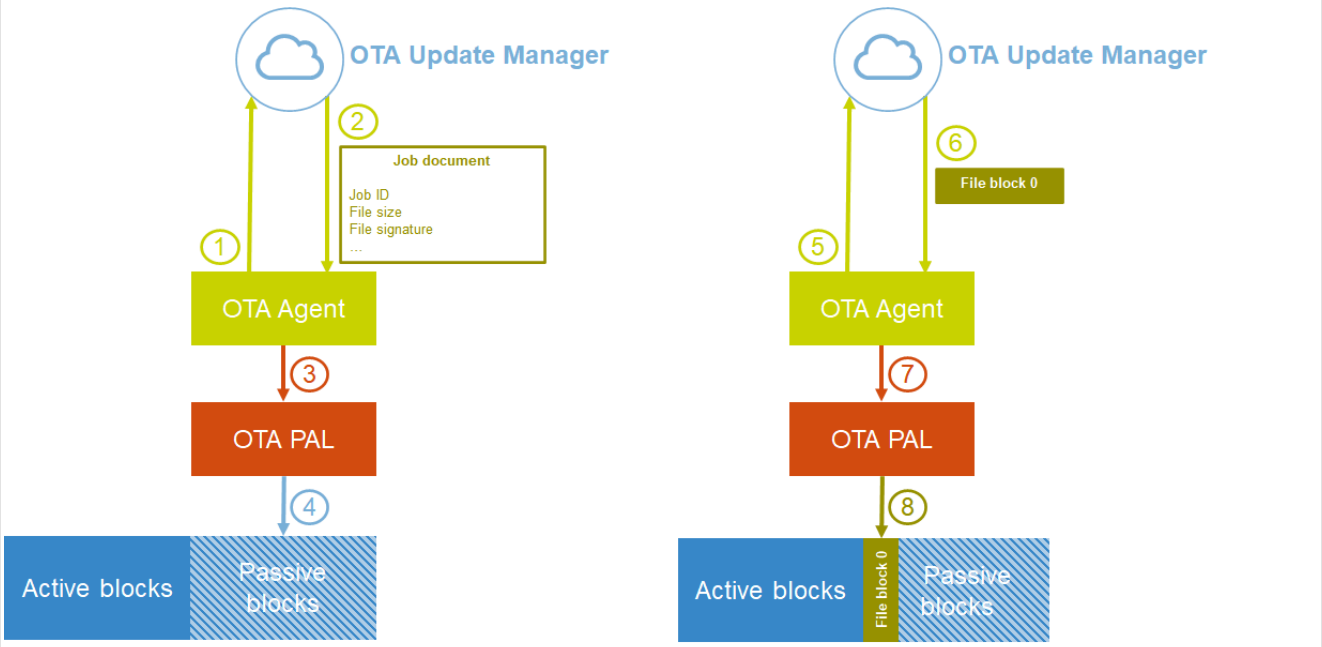

Figure 7. OTA update flow 1-8

Steps 5-8 are repeated until the full image is downloaded to the passive partition.

9. The *OTA Agent* invokes the *OTA PAL* to verify the signature of the downloaded image (via *Mbed TLS*)
10. The *OTA Agent* invokes the OTA complete callback, which by default invokes `prvPAL_ActivateNewImage` *(OTA PAL)*
11. The *OTA PAL* uses the *HSE IP* to invoke the Activate Passive Block service
12. The *OTA PAL* uses the *Power IP* to trigger reset

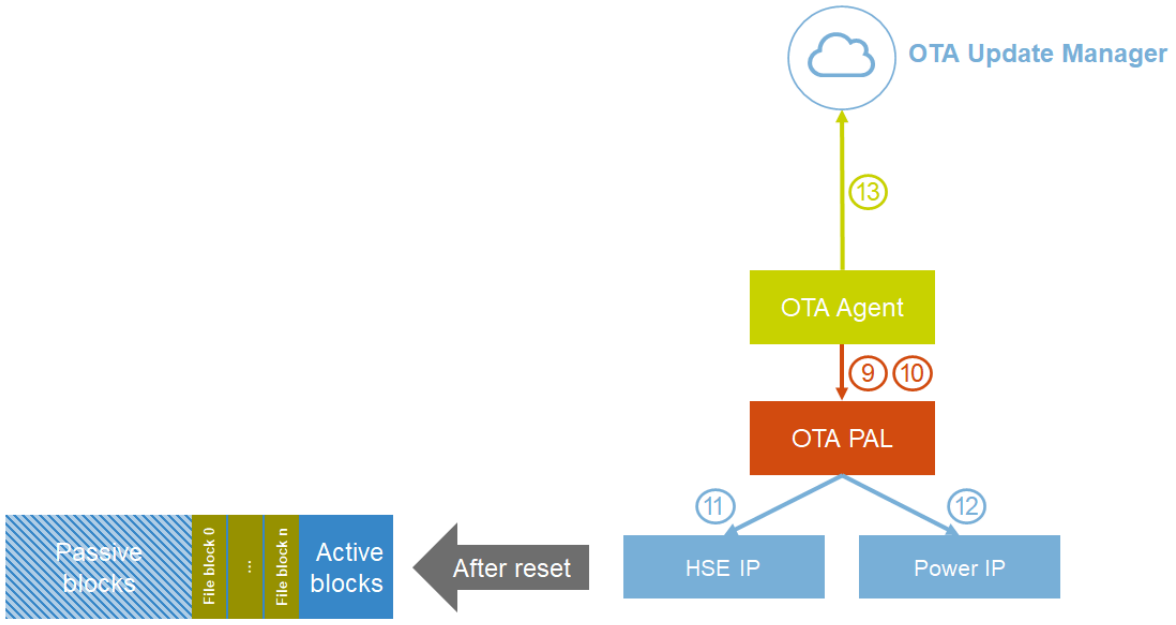

Figure 8. OTA update flow 9-13

After successful boot, the status will be sent to the *OTA Update Manager*

The current implementation of OTA requires the image to be signed. In order to achieve this, a code signing certificate must be generated.

For obtaining the code signing certificate, the following steps can be followed (commands can be run in Windows command line or any other command line with *openssl* installed):

1. Generate a private key

```
openssl ecparam -name prime256v1 -genkey -noout -out private-key.pem
```

2. Generate the corresponding public key

```
openssl ec -in private-key.pem -pubout -out public-key.pem
```

3. Create a self-signed certificate

```
openssl req -new -x509 -key private-key.pem -out cert.pem -days 360
```

For the configuration of the OTA Job in the IoT Console, the necessary steps can be found here.

For the code signing profile you will need to create a profile (one time only); you can select *Windows Simulator* as *Device hardware platform*. You will need to upload the previously generated private key and certificate for the *Certificate body* and *Certificate private key*. For the *Path name of code signing certificate on device*, you can input `Code_Sign_Certificate.dat` - this will only be used if you select the code signing certificate input type as *From file* in S32CT.

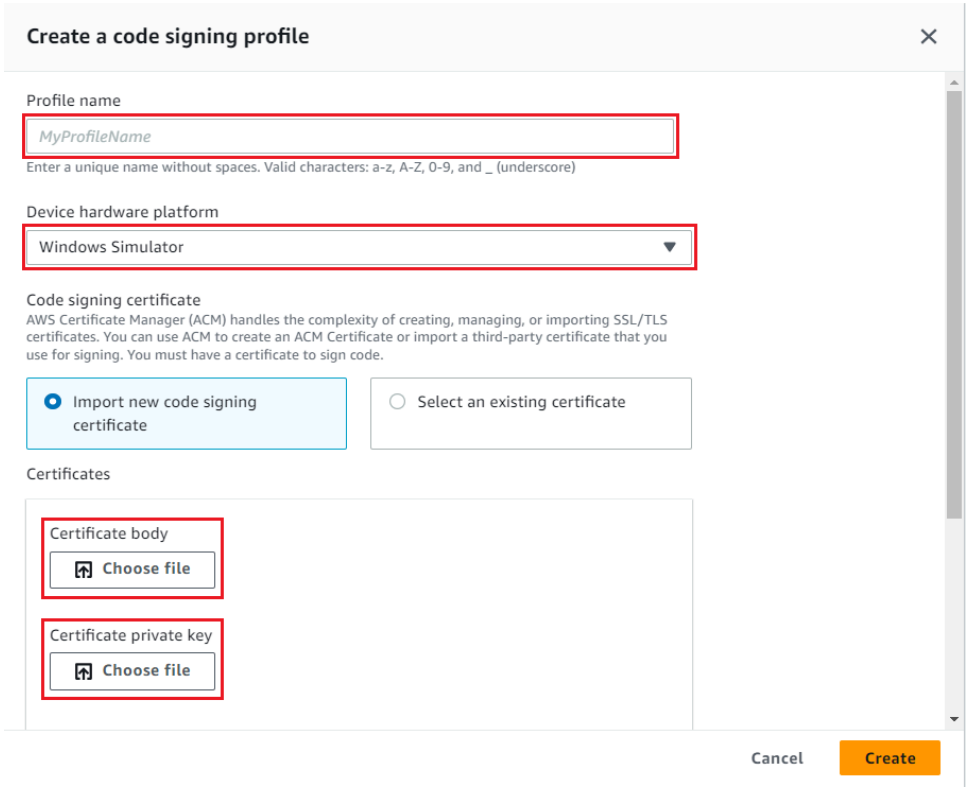

Figure 9. Creating code signing profile

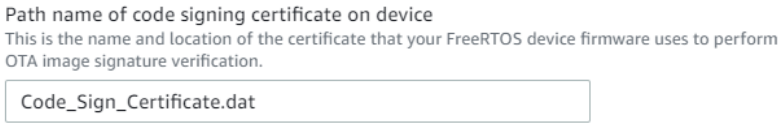

Figure 10. Code signing certificate path

The *Path to firmware image on device* should match the file name of the file from the file system table (configured in the S32CT AWS IoT Core component).

Please note that if you select in S32CT for *Code signing certificate input type* the *From file* option, you will have to extend the demo to handle the `Code_Sign_Certificate.dat` file (it will not be initialized by default). The easier way is to use the *Static (C macro)* option, for which you will need to paste in the *Code signing certificate value* field the contents of the certificate file generated above. This input must be in PEM format - it should match the regular expression:

```
 -----BEGIN CERTIFICATE-----(\n|\r|\r\n)([0-9a-zA-Z\+\/=]{64}(\n|\r|\r
\n))*([0-9a-zA-Z\+\/=]{1,63}(\n|\r|\r\n))?
 -----END CERTIFICATE-----
```

5 Configuration

For easy configuration of the *AWS Libraries for S32K3*, an S32CT configuration component is provided. This component allows enabling/disabling of the libraries (to enable the user to select only the ones needed by the application), configuring of logging options and adding files to the file system.

The first two tabs (*FreeRTOS Core Libraries* and *AWS IoT specific Libraries*) allow enabling and disabling of specific libraries.

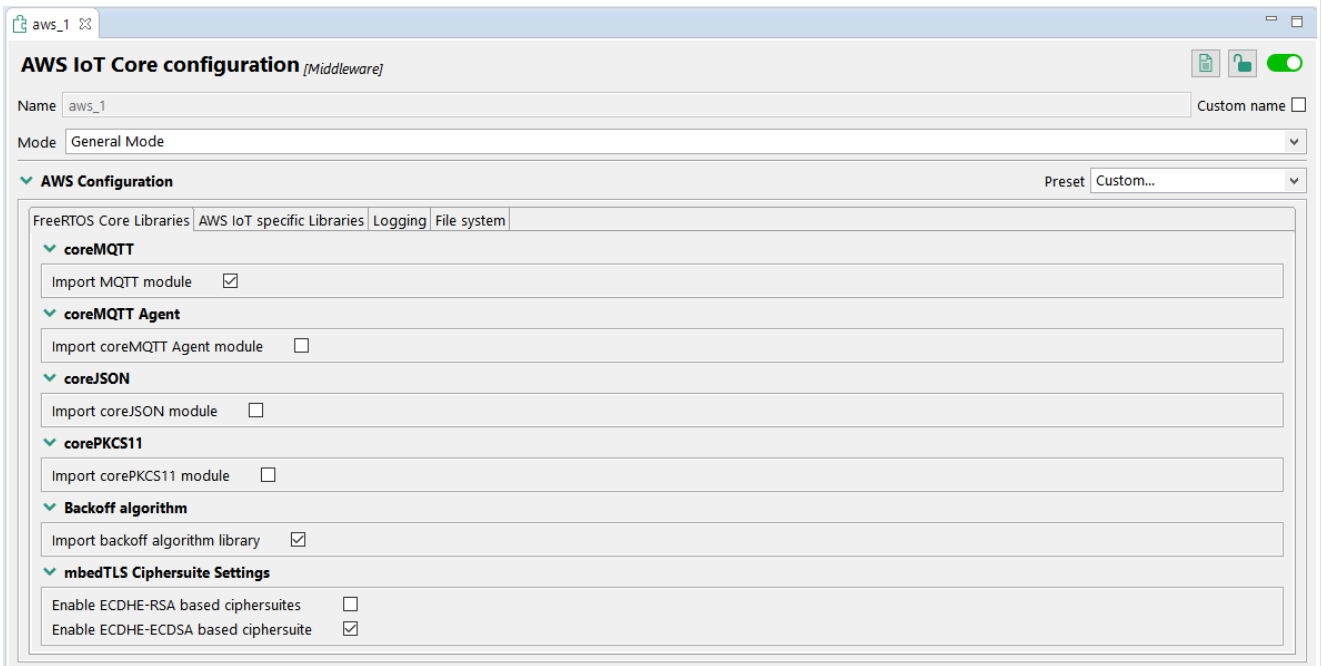

Figure 11. FreeRTOS core libraries

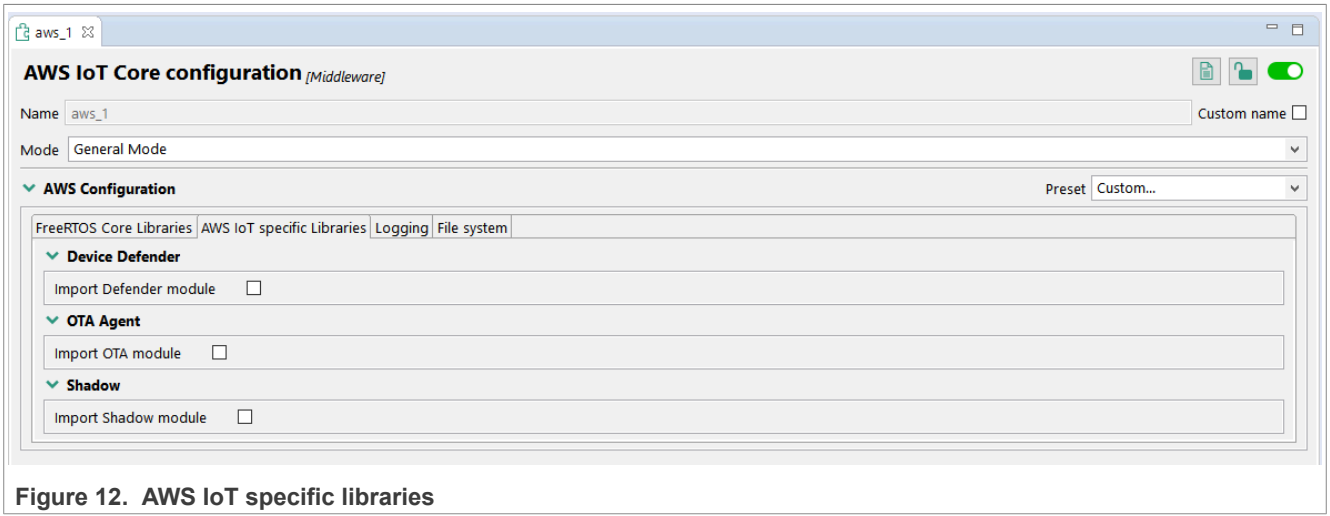

Figure 12. AWS IoT specific libraries

The *Logging* tab allows the user to choose only the necessary logs for the application.

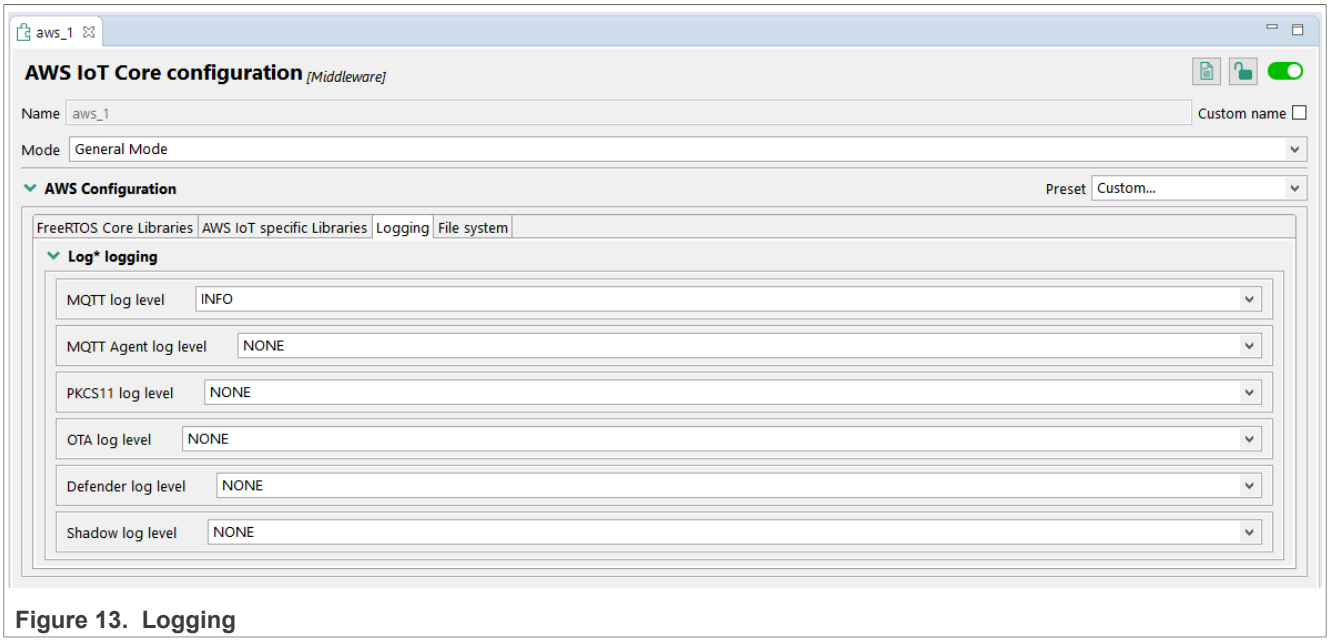

Figure 13. Logging

The *File system* tab allows configuration of the files implemented in the file system. Some files will be added automatically based on the libraries enabled (e.g. device certificate, `flash_image.bin` if OTA is enabled). The user can also include additional files required by the application.

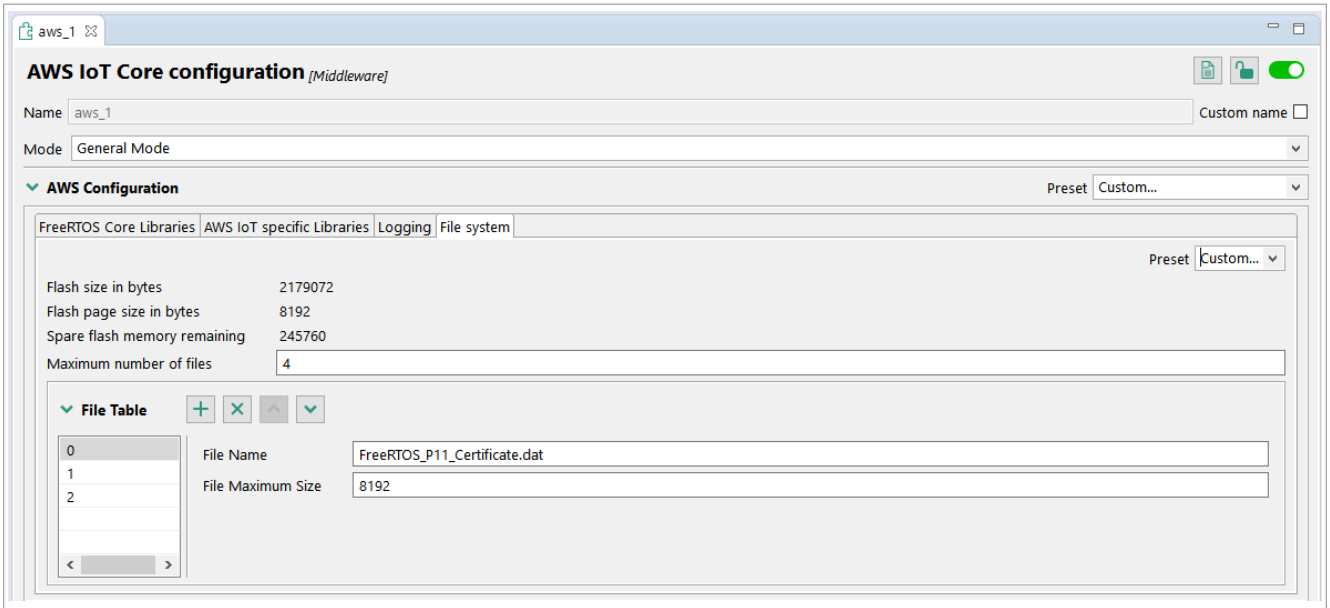

Figure 14. File system

6 Software examples

The *AWS Libraries for S32K3* product offers the following S32 Design Studio projects as examples:

1. **aws_mqtt_s32k344**
   This example is used to create an MQTT connection to your AWS IoT account. The device will publish MQTT topic messages to the AWS endpoint and is able to receive publish messages from the server to which will reply with acknowledgement messages.
   This example project was created based on the MQTT_Mutual_Auth demo which is part of FreeRTOS-Plus.
2. **aws_pkcs11_mqtt_s32k344**
   In addition to the standard MQTT example, this example uses corePKCS11 to provision the device with the client certificate and private key.
   This example project was created based on the corePKCS11_MQTT_Mutual_Auth demo which is part of FreeRTOS-Plus.
3. **aws_ota_s32k344**
   This example is used to showcase the capabilities of over-the-air update in AWS IoT Core.
   The example starts the OTA Agent and listens for incoming updates requests from the AWS server. Once the update starts, a new firmware image will be downloaded to flash and the device will restart, loading the new image.
   This example project was created based on the Ota demo which is part of FreeRTOS-Plus.
4. **aws_defender_s32k344**
   This demo creates a single application task that demonstrates how to collect metrics, construct a device defender report in JSON format, and submit it to the AWS IoT Device Defender service through a secure MQTT connection to the AWS IoT MQTT Broker. The demo includes the standard networking metrics as well as custom metrics. For custom metrics, the demo includes:
   - A metric named *task_numbers* which is a list of FreeRTOS task IDs. The type of this metric is *list of numbers*.
   - A metric named *stack_high_water_mark* which is the stack high watermark for the demo application task. The type of this metric is *number*.

   This example project was created based on the Device_Defender demo which is part of FreeRTOS-Plus.
5. **aws_shadow_s32k344**
   This demo shows how to use the AWS IoT Device Shadow library to connect to the AWS Device Shadow service. It uses the *coreMQTT* library to establish an MQTT connection with TLS (Mutual Authentication) to the AWS IoT MQTT Broker and the coreJSON library parser to parse shadow documents received from the AWS Shadow service. The demo shows basic shadow operations, such as how to update a shadow document and how to delete a shadow document. The demo also shows how to register a callback function with the coreMQTT library to handle messages like the shadow /update and /update/delta messages that are sent from the AWS IoT Device Shadow service.
   This example project was created based on the Device_Shadow demo which is part of FreeRTOS-Plus.
6. **aws_gg_demo_s32k344**
   This project is showcasing the Greengrass capability by connecting S32K344 running *AWS Libraries for S32K3* (representing a *Greengrass device*) to S32G2 RDB2 running GoldVIP (representing the *Greengrass core*) and publishing diagnostics to a SiteWise dashboard. It requires extra prerequisites, which are mentioned in the `description.txt` file.

For details regarding running the demos please consult the `description.txt` file located in the root of each of the examples.

7 Cloud applications

7.1 Capabilities

By extending the S32K3 family features with the AWS Cloud services, the following capabilities are enabled:
Cloud Services - Secure data communication

- Send messages to AWS cloud via MQTT
- S32K3 specific vehicle data such as battery parameters, sensor details, motor conditions, diagnostic data, external acoustic sound and network performance details
- Leverage use cases such as Predictive Maintenance, Advanced Vehicle Diagnostics, Fleet Management, etc…

Firmware Over-the-Air (FOTA) update

- Receive firmware update for S32K3 via MQTT by using AWS OTA agent
- Upgrade vehicle SW with S32K3 A/B firmware swap

Enhance security

- Ensure secure cloud communication by suing AWS secure sockets
- Additionally, collect connection metrics from devices and report data to AWS cloud to identify anomalies using AWS Device Defender

Remote ECU state monitoring (Digital Twin)

- Save device/application state in AWS device shadow, so that it can be retrieved/updated by cloud applications while the device is offline, mirroring ECU state in the cloud with latest device information

Gateway Communication

- Communication with gateway/edge ECU (S32G) using AWS Greengrass

7.2 Applications

Based on the capabilities metioned above, different cloud applications can be enabled:

Cloud processing

Leveraging existing AWS cloud services, part of the processing can be offloaded in the cloud.

As an example, an **Alexa in the Cloud** application could be enabled - obtaining audio input from the S32K3 and sending it in the Cloud to be processed by AVS (Alexa Voice Services Integration for IoT Core). This way, the compute and memory intensive audio workloads could be offloaded.

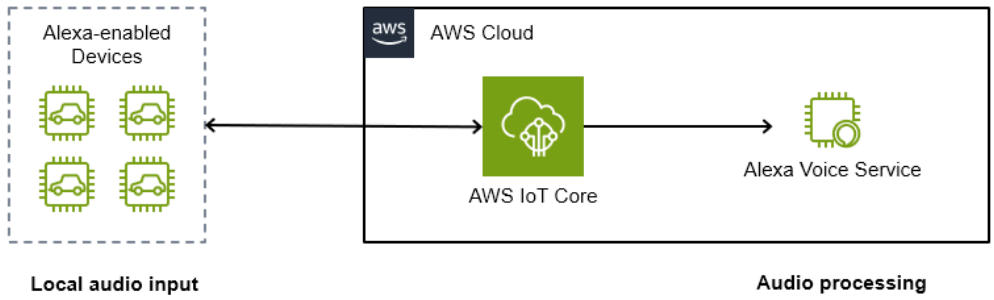

Figure 15. Alexa in the cloud

AI/ML

AWS includes also Machine Learning enabling services, like **Amazon SageMaker**. It enables building and training machine learning models, as well as running inference in the Cloud.

By using this type of services, an application like user profile detection based on speed with Machine Learning in the Cloud could be enabled.

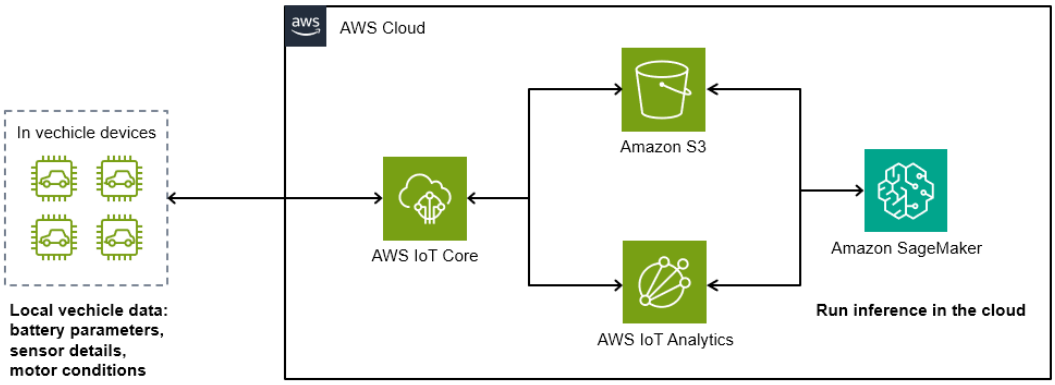

Figure 16. Machine learning in the cloud

8 Abbreviations

Table 2. Abbreviations

Acronym	Description
AEC	Automotive Electronics Council
AI	Artificial Intelligence
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
AWS	Amazon Web Services
ECMA	European Computer Manufacturers Association
ECU	Electronic Control Unit
EVB	Evaluation Board
GG	Greengrass
HDQFP	High Density Quad Flat Package
HSE	Hardware Security Engine
HTTP	Hypertext Transfer Protocol
HW	Hardware
IDE	Integrated development environment
IoT	Internet of Things
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JTAG	Joint Test Access Group
LTS	Long Term Support
MCU	Microcontroller Unit
MQTT	Message Queuing Telemetry Transport
OEM	Original Equipment Manufacturer
OTA	Over-the-air
PEM	Privacy-Enhanced Mail
PKCS	Public-Key Cryptography Standards
RAM	Random Access Memory
RTD	Real-Time Drivers
S32CT	S32 Configuration Tool
S32DS	S32 Design Studio
SW	Software
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
USB	Universal Serial Bus

9 Revision history

This table summarizes the revisions to this document.

Table 3. Revision history

Revision	Date	Description
0	10/2023	Initial release

10 Legal information

10.1 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.

10.2 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.

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 http://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.

Suitability for use in automotive applications — This NXP product has been qualified for use in automotive applications. If this product is used by customer in the development of, or for incorporation into, products or services (a) used in safety critical applications or (b) in which failure could lead to death, personal injury, or severe physical or environmental damage (such products and services hereinafter referred to as "Critical Applications"), then customer makes the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, safety, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. As such, customer assumes all risk related to use of any products in Critical Applications and NXP and its suppliers shall not be liable for any such use by customer. Accordingly, customer will indemnify and hold NXP harmless from any claims, liabilities, damages and associated costs and expenses (including attorneys' fees) that NXP may incur related to customer's incorporation of any product in a Critical Application.

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.

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.

10.3 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

For more information, please visit: http://www.nxp.com

Date of release: 3 October 2023
Document identifier: UM11974