1 Introduction

1.1 Overview

The S32G Vehicle Integration Platform (GoldVIP) is a reference software platform for a vehicle service-oriented gateway running on NXP vehicle network processors like the S32G274A.

GoldVIP addresses key automotive market trends and use cases:

- Vehicle Networking
- Service-oriented Architecture (SoA)
- Data Analytics
- Over-the-Air (OTA) Services
- Virtualization & Isolation
- Network Security and Security Services

The platform integrates use cases from the three major vehicle gateway types that will be detailed in the next chapters:

- Cloud Edge Gateway
- Ethernet Gateway
- CAN Gateway

1.2 High Level Architecture
2 Hardware and Software Prerequisites

This chapter describes the hardware and software prerequisites required to use the GoldVIP product.

2.1 Hardware prerequisites

The GoldVIP product has been developed for these S32G hardware platforms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>S32G GoldBox</td>
<td><a href="https://www.nxp.com/GoldBox">https://www.nxp.com/GoldBox</a></td>
</tr>
<tr>
<td>Wiring harness for CAN</td>
<td>Delivered with the GoldBox</td>
</tr>
<tr>
<td>USB to Eth dongle (needed for ETH use cases)</td>
<td>N/A</td>
</tr>
<tr>
<td>USB to Wi-Fi dongle (optional for Wi-Fi connectivity)</td>
<td>N/A</td>
</tr>
<tr>
<td>SD-card of at least 8GB (16GB or more is recommended)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
## 2.2 Software prerequisites

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux BSP</td>
<td>32.0</td>
<td>Linux Board Support Package contains ARM Trusted Firmware and Linux.</td>
</tr>
<tr>
<td>AWS IoT Greengrass</td>
<td>2.6.0</td>
<td>AWS IoT Greengrass V2 software</td>
</tr>
<tr>
<td>Elektrobit ACG</td>
<td>8.8.3</td>
<td>Elektrobit tresos Autocore Generic for S32G</td>
</tr>
<tr>
<td>PFE Firmware</td>
<td>1.2.0</td>
<td>Packet Forwarding Engine (PFE) Firmware</td>
</tr>
<tr>
<td>PFE driver (Cortex-M7)*</td>
<td>0.9.7</td>
<td>Packet Forwarding Engine (PFE) Driver for Arm Cortex-M7</td>
</tr>
<tr>
<td>LLCE Firmware and Driver*</td>
<td>1.0.4</td>
<td>Low Latency Communication Engine (LLCE) Firmware</td>
</tr>
<tr>
<td>SJA1110 SDK</td>
<td>1.0.0</td>
<td>SJA1110 SDK</td>
</tr>
<tr>
<td>GoldVIP SJA1110 customization</td>
<td>1.4.0</td>
<td>SJA1110 firmware customization environment</td>
</tr>
<tr>
<td>GoldVIP LLCE customization</td>
<td>1.4.0</td>
<td>LLCE firmware customization environment</td>
</tr>
<tr>
<td>AWS IoT FleetWise</td>
<td>0.1.2</td>
<td>AWS IoT FleetWise software</td>
</tr>
<tr>
<td>IPCF</td>
<td>4.5.0</td>
<td>IPCF Shared Memory Driver</td>
</tr>
<tr>
<td>RTD</td>
<td>3.0.2 HF1</td>
<td>Real-Time Drivers</td>
</tr>
<tr>
<td>Platform Software Integration</td>
<td>2022.06</td>
<td>Platform Software Integration reference examples for S32G</td>
</tr>
<tr>
<td>SAF</td>
<td>1.0.1</td>
<td>S32 Safety Software Framework</td>
</tr>
<tr>
<td>HSE Firmware</td>
<td>1.0.5</td>
<td>Hardware Security Engine Firmware</td>
</tr>
<tr>
<td>eiQ Auto</td>
<td>3.3.0</td>
<td>eiQ Auto Deep Learning toolkit</td>
</tr>
</tbody>
</table>

Software is available [here](#).

**Note:** Items marked with `*` are bundled as part of the GoldVIP Gateway tresos project.
3 Host system

3.1 Hardware

3.1.1 PC Ethernet Ports
A host PC with two dedicated Ethernet ports is required in order to run the Ethernet Gateway examples.

3.2 Host Environment

3.2.1 Host system
In order to evaluate all the GoldVIP use cases, it is recommended to use a Windows Machine running an Ubuntu Virtual Machine.

3.2.2 Docker container
This Docker image recipe is based on a reference Ubuntu image from https://hub.docker.com/_/ubuntu and can be deployed on the host PC running Ubuntu 18.04.
4 GoldVIP Quick Start

This guide contains the steps to quickly deploy GoldVIP images, setup the board connectivity and try out some vehicle gateway use cases.

4.1 Setup cable connections

To locate the connectors and ports from the board check the figures from the appendix or check S32G-VNP-RDB2 Quick Start Guide from the board's box.

1. Connect the power cable
2. Connect the board's UART0 serial port to PC using the USB cable from the box.

4.2 Deploy GoldVIP images

Download GoldVIP binary images from here.

4.2.1 Deploy GoldVIP SD-card image

1. Write the GoldVIP .sdcard image (fsl-image-goldvip-s32g274ardb2.sdcard) on the SD-card:
   - On Linux host:
     - Check the device name under which your SD-card is installed on PC so that you won't overwrite another disk. Use $ cat /proc/partitions command before and after inserting SD-card, see which new sd* disk appears (e.g., /dev/sdb) and use its name in the next step command.
     - Write fsl-image-goldvip on SD-card plugged into the Linux host PC, e.g.:
       ```
       $ sudo dd if=fsl-image-goldvip-s32g274ardb2.sdcard of=/dev/sdb bs=1M status=progress && sync
       ```
   - On Windows host, one can use Win32DiskImager in order to write the .sdcard file to the SD-card.
2. Set the board to boot from SD (SW4.7 to ON, and SW10 to ON-OFF). An image describing the DIP switches positions on the RDB board can be found in the appendix.
3. Plug in the SD-card, and power on the board and it will boot Arm Trusted Firmware and Linux from SD-card.

Note: The SD-card partition table is altered during the first boot, taking 5 seconds at maximum, expanding existing partitions and creating new ones in order to use the SD-card to its full capacity.

4.2.2 Deploy images to Flash

The real time bootloader runs on Cortex-M7-0 and is loaded from QSPI NOR flash. It's main function is to load ARM Trusted Firmware on Cortex-A53 cores and the real-time applications on Cortex-M7 cores.

To write boot-loader in NOR flash, first boot from SD-card using the above SD-card image, stop in u-boot console when prompted, and run the following command, which writes all the images in flash:

```
run write_goldvip_images
```
This operation shall take at most 20 seconds. It is also possible to update images in flash individually:

1. Update boot-loader image:
   
   run write_bootloader

2. Update Arm Trusted Firmware image:
   
   run write_atf

3. Update GoldVIP Gateway binary:
   
   run write_gateway_app

After all the binaries are written, power off the board, configure the DIP switches for NOR Flash Boot mode (set SW4.7 to OFF) and then power on the board.

**Note:** It is recommended to setup SW12 to ON-ON position in order to have the SJA telemetry application loaded to the board by Linux.
5 Docker

5.1 Building GoldVIP Docker image

A GoldVIP Docker image recipe is provided in the layout under the docker directory.

1. Install Docker on your PC - complete guide for Ubuntu is here.
2. Build the image using BuildKit from Docker (requires minimum 18.09 Docker release).

Open Linux command line in the GoldVIP install directory:

```
$ cd docker
$ sudo chmod +x create_image.sh
$ ./create_image.sh
```

**Note:** The fetching of some packages might fail if your network uses an HTTP proxy. In such case, provide the HTTP proxy address by adding the following parameter to the above command: `--build-arg http_proxy=<HTTP_PROXY_ADDRESS>`. This will add the newly created image to your local Docker registry, docker-goldvip:<Release_Version>. To see all available Docker images in your environment, use the command:

```
$ sudo docker images
```

**Note:** for more detailed information about adding files/packages into the image, please see the official documentation: https://docs.docker.com.

5.2 Deploy GoldVIP Docker image

After following the steps from the Building GoldVIP Docker image chapter, execute the following commands:

1. Run a container based on the GoldVIP Docker image:

```
```

This command will start a container with a bash console. To spawn another console of the running container run:

```
$ sudo docker exec -it goldvip /bin/bash
```

2. To login to the board's Dom0 Linux console, from Docker container run this command (default user is "root"):

```
$ sudo expect /home/vip/device_utils.exp /dev/ttyUSB0
```

**Note:** The number of UART device (e.g., ttyUSB0) may be different on your PC. See which device is created after plugging the serial USB cable by running:

```
$ ls /dev | grep ttyUSB
```

Now you can start running the use cases detailed in each gateway chapter.
5.3 Running the GoldVIP Graphical User Interface (GUI) in Docker container

The release Docker container provides GoldVIP Graphical User Interface for user that want to run Ethernet, CAN and Cloud scripts via GUI. After deploying a container derived from GoldVIP Docker image, perform the following steps:

1. Start nodejs service to run GUI application:

```
$ sudo /home/vip/goldvip/nodejs_server_start.sh &
```

2. Start a browser from your Linux machine and navigate to the GoldVIP dashboard by typing in the address bar:

```
http://localhost/nxpdemo
```

3. Use the default credentials <'1'/'1'> to sign in the dashboard.

Note:

- The board must have an ethernet connection and be on the same network as the host machine.
- Please go to the browser settings to enable Pop-ups and redirects. For steps on how to enable them, please follow this guide.
- For cloud telemetry use cases, exporting variable environments AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY in docker eliminates the need to enter text boxes for AWS credentials.
6 Boot Flow

6.1 Bootloader

The GoldVIP makes use of the Platform Software Integration Bootloader in order to start the platform. Several hardware resources (clocks, pins, DMA, QSPI, HSE key catalog) are used by the Bootloader. The boot flow can be described as follows, assuming the images are written into flash and QSPI boot has been enabled via boot switches configuration, (see Deploy images to Flash chapter):

First time boot only:

1. BootROM configures the QSPI interface, initializes the SRAM, starts the HSE and the Cortex-M7 core via Image Vector Table (IVT) configuration;
2. Bootloader initializes the aforementioned hardware resources;
3. A safety startup check is performed using the SAF (S32G Safety Software Framework) functionality for the resources used in the boot flow;
4. The bootloader configures the Secure Boot for its own image via HSE Secure Memory Region (SMR) configuration;
5. A functional reset is issued.

Next boot procedures:

1. BootROM configures the QSPI interface, initializes the SRAM and starts the HSE;
2. HSE starts the Bootloader via the previously mentioned SMR configuration;
3. Bootloader initializes the hardware resources;
4. A safety startup check is performed using the SAF functionality for the resources used in the boot flow;
5. The application images (ARM Trusted Firmware for A53_0 and Can-GW for Cortex-M7) are loaded sequentially from QSPI Flash using DMA;
6. The used hardware resources are then de-initialized;
7. The XRDC (eXtended Resource Domain Controller) is configured, allowing separation between Real Time and Linux domains;
8. Bootloader then starts the A53_0 core and jumps to the Cortex-M7 application.

The Bootloader tresos workspace and source code can be found in the GoldVIP installer in the <GoldVIP_install_path>/configuration/bootloader directory. More details on the Bootloader functionality can be found in the Bootloader User Manual from the Platform Software Integration installer.

Note: The SAF is a premium product, integrated in the delivered binary from the GoldVIP installer. It is possible to reconfigure the Bootloader without having access to the SAF, by removing the configurations from the Bootloader EB tresos project. The following steps must be performed in order to remove the SAF functionality:

1. Load the Bootloader EB tresos project into workspace, and for all the missing plugins choose Remove all faulty Modules;
2. Disable the Safety Bootloader Enable field in the Bootloader plugin configuration;
3. Go to the Bootloader workspace and delete the output folder;
4. Generate and compile the project, following the steps from the Bootloader User Manual, delivered in the Platform Integration Installer.
7 Cloud Edge Gateway

7.1 Introduction

The Cloud Edge Gateway use case demonstrates telemetry to the cloud using AWS IoT Greengrass. Telemetry statistics are fetched from the device, calculated and sent to the cloud counterpart of the application. Statistics received in the cloud are then displayed into user-friendly graphs. The statistics include but may not be limited to: Networking accelerator usage statistics, Real-time cores load, Domain-0 VCPU load and Domain-0 Memory utilization statistics.

The following architecture was employed:

The following access policies to hardware resources are applicable for the virtual machines (Domain-0 and v2xdomu) described in Xen chapter:

- Domain-0 has access to all the hardware resources in the system.
- v2xdomu has access to limited resources which are virtualized by Xen.

Telemetry data is collected from Domain-0 and passed to v2xdomu through a TCP client-server communication. The Domain-0 v2xbr is a virtual switch with no outbound physical interface attached to it. This connection is used to pass telemetry data from Domain-0 to v2xdomu and vice-versa, without outside interference or snooping possibility. Any change in the configuration (update of the telemetry interval from the cloud) is communicated from cloud to the Dom0 TCP Server, via the TCP client available on v2xdomu. This ensures that the system resources are protected from outside interference. Data is prepared for fetching in the Domain-0 at any given time via a telemetry service (see /etc/init.d/telemetry) which is started at boot time.

7.2 Prerequisites

- AWS account with SSO enabled. Follow the steps in this guide to enable SSO: https://docs.aws.amazon.com/singlesignon/latest/userguide/getting-started.html
  Enabling SSO will grant you access to the SSO console. SSO is also required to use the AWS IoT SiteWise Dashboard.
• S32G board with the GoldVIP image deployed and the GMAC port connected to the internet.

Note: AWS IoT Greengrass uses ports 8883 and 8443. As a security measure, restrictive environments might limit inbound and outbound traffic to a small range of TCP ports, which might not include these ports. Therefore the provisioning script (described in chapter Connecting the board to AWS) changes these ports to 443. To use the default ports (8883 and 8443) use the arguments --mqtt-port and --http-port from the provisioning script.

7.3 AWS IAM Permissions

A policy needs to be set for the AWS IAM user. This will allow the user to create the necessary resources needed for the telemetry use case:

```
{
"Version": "2012-10-17",
"Statement": [
  {
"Effect": "Allow",
"Action": [
"cloudformation:*",
"cloudwatch:*",
"iot:*",
"lambda:*",
"logs:*",
"s3:*",
"greengrass:*",
"sns:*",
"iotsitewise:*",
"iam:*",
"sso:*",
"sso-directory:*",
"serverlessrepo:*"
],
"Resource": "*"
}
]
}
```


7.4 Supported Regions

The Telemetry Stack is limited to the AWS regions which support AWS IoT SiteWise. Here is the list of supported regions: [https://docs.aws.amazon.com/general/latest/gr/iot-sitewise.html](https://docs.aws.amazon.com/general/latest/gr/iot-sitewise.html)

Select in the AWS Console the region you desire from the list of supported regions.

7.5 Deployment of the Telemetry Stack in AWS

1. Go to the AWS Serverless Application Repository (SAR) console: [https://console.aws.amazon.com/serverlessrepo/](https://console.aws.amazon.com/serverlessrepo/)
2. Go to Available applications tab; then to Public applications and search for goldvip.
3. Check Show apps that create custom IAM roles or resource policies to see the application.
4. Click on nxp-goldvip-telemetry. You can modify the application parameters. TelemetryTopic should be unique. CoreThingName must be unique, however if the default value is used the stack name will be attached to the core thing name to ensure that it is unique. Pay attention to the application name (stack name). This parameter will be needed when deploying the stack on the S32G board.
5. Check I acknowledge that this app creates custom IAM roles.
6. Click Deploy. The deployment will take a few minutes. You will be redirected to another page. The name of the stack is on the top of the page, starting with serverlessrepo-, if you changed the application name you will need this name in the next step. You can go to the Deployments tab and see the status of the deployment. Wait for the status to change from Create in progress to Create complete. Note: you may need to refresh the page to see the status change.

This AWS CloudFormation stack creates on your account:

- A AWS IoT Greengrass V2 telemetry component, this is a python function which runs on v2xdomu and sends data to AWS IoT Core. The provisioning script described in chapter Connecting the board to AWS creates a AWS IoT Greengrass V2 continuous deployment which will run the telemetry component on your board.
- A AWS IoT SiteWise Portal with multiple Dashboards; after the board is connected to AWS a live visual representation of the telemetry data received via the AWS IoT Greengrass V2 component is displayed in these.

7.6 SJA1110 Telemetry Setup

Steps needed to enable SJA1110 telemetry:

1. Connect the SJA1110 to the internet using the P4 ethernet port on the board (See Appendix A). The SJA1110 application and v2xdomu will need to be connected to the same local network.
2. Connect the GMAC0 port to the same network as the SJA1110.
3. Make sure that SW12 is set to ON-ON position. If not, set it to ON-ON and reboot the board.
4. Run the provisioning script (described in chapter Connecting the board to AWS) with the --setup-devices option.

Notes:

- You can connect the GMAC0 port to P2A or P2B to access the internet through the SJA1110 switch, but if this type of connection is used, the SJA fast path cannot be used any longer.
- Setting SW12 to ON-OFF will prevent the SJA1110 application to be loaded, and the default SJA1110 firmware will run instead.
- To restart the SJA1110 telemetry after a reboot rerun the provisioning script with the --setup-devices option, as described in chapter Configure AWS IoT Greengrass after reboot.

Chapter SJA1110 Telemetry Application contains more details about the SJA1110 application.

7.7 Connecting the board to AWS

1. Log into the v2xdomu virtual machine using the command: `xl console v2xdomu`
2. Configure environment variables for AWS IoT Greengrass provisioning script:

   $ export AWS_ACCESS_KEY_ID=<access key id>
   $ export AWS_SECRET_ACCESS_KEY=<secret access key>

One way of obtaining your AWS credentials is the following:

   From the AWS SSO console select your account and retrieve the environment variables by clicking on Command line or programmatic access. From section macOS and Linux copy the variables and paste them on your board. Use Option 1: set the AWS credentials as environment variables.

Please check the AWS documentation for additional information: https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-envvars.html

Notes:

• IAM credentials should never be used on a device in production scenario.
• These variables are temporary and are erased at reboot.

4. Run the AWS IoT Greengrass provisioning script on your board:

   $ python3 ~/cloud-gw/greengrass_provision.py \
     --stack-name <stack-name> \
     --region-name <region-name> \
     --setup-devices

Where <stack-name> is the name of the deployed stack prepended with serverlessrepo-. If you did not change the application name you do not need to specify this parameter. In <region-name> put the region in which you have deployed the Telemetry Stack. --setup-devices starts the sja provisioning script. This will setup the network interface, start the AWS IoT Greengrass V2 Nucleus, and create a AWS IoT Greengrass V2 continuous deployment, which will run the telemetry component created by the Telemetry Stack.

Note: the provisioning script will try to setup the internet connection using the eth0 network interface (attached to GMAC0) by default.
To get more details about the script parameters use:

   $ python3 ~/cloud-gw/greengrass_provision.py -h

The board is now connected to your AWS account and it will begin to send telemetry data.

In some cases, DHCP client is running for each of the PFE interfaces (PFE0 and PFE2), hence 2.5 Mbps spikes can be observed in the AWS IoT SiteWise dashboard. To close the DHCP client, it is necessary to run the command killall udhcpc in the Dom0 console. This will close the DHCP client and the spikes will no longer be observed in the dashboard.

Note: The AWS IoT Greengrass V2 Nucleus does not start automatically after a reboot. The network configuration and time are not persistent between reboots. Please check Configure AWS IoT Greengrass after reboot for further information.

7.8 Accessing the AWS IoT SiteWise dashboard

1. Go to the AWS IoT SiteWise console: https://console.aws.amazon.com/iotsitewise/
2. Click on Portals from the list on the left.
3. Click on the name of your portal, it starts with SitewisePortal_serverlessrepo.
4. Click on Assign administrators
5. Add your account and any other you want to have access to the AWS IoT SiteWise Dashboard.
6. Click Assign administrators.
7. Click on the Portal's Url (or Link).
8. Close the Getting started pop up window.
9. Click on one of the dashboards to visualize the telemetry.
You will now see the live telemetry data from your board.

7.9 GoldVIP telemetry dashboards

After accessing the SiteWise portal, several dashboards will be shown. The GoldVIP telemetry application creates the following dashboards by default:

1. **Core Loads Dashboard**: contains one widget per virtual CPU running in Domain-0 and one widget for all the Cortex-M7 MCUs core loads.
2. **SJA Dashboard 1-3**: contains widgets containing packet statistics per each of the SJA ports available on the S32G.
3. **SOC Dashboard**: contains widgets dedicated to SOC specific statistics: PFE traffic (both RX and TX), SOC temperature sensors values and Domain-0 memory load.
4. **Machine Learning Dashboard**: contains widgets dedicated to the predictive maintenance application, it shows the prediction results and execution durations.

7.10 Testing the Telemetry Application

1. Log into the Domain-0 virtual machine using CTRL+].
2. Simulate core load:
   - Execute a computationally intensive task to generate CPU load:
     ```
     $ dd if=/dev/zero of=/dev/null &
     ```
     This process will be assigned to one of the available cores and will run in the background. An increase of 25% on the core load shall be observed in the AWS console, per each of the started processes.
   - Kill all cpu loading processes:
     ```
     $ killall dd
     ```

7.11 Deleting the Telemetry Application

1. Go to AWS Cloudformation: https://console.aws.amazon.com/cloudformation/
2. Select your stack and delete it.

7.12 Configure AWS IoT Greengrass after reboot

The AWS IoT Greengrass V2 Nucleus starts automatically when the v2xdomu VM is booted up. However, the network configuration is not persistent between reboots, so it must be recreated in order to configure a specific network interface to be used for internet connection. Also, the SJA1110 client device needs to be provisioned after every reboot. To configure the network and provision the client devices and provision the client devices:
- Log into the v2xdomu virtual machine using the command:
  
  $ xl console v2xdomu

- The provision script can be used again to configure the network interface that will be used by AWS IoT Greengrass:

  $ python3 ~/cloud-gw/greengrass_provision.py --no-deploy \ 
  --stack-name <stack-name> \ 
  --region-name <region-name> \ 
  --netif <net-dev> --setup-devices

  Where `<net-dev>` is the network interface that shall be configured. When the flag `--no-deploy` is set, the script will not create a AWS IoT Greengrass deployment, it will just start the AWS IoT Greengrass V2 Nucleus. Adding the `--setup-devices` parameter will start the provisioning of the client devices. To setup the client devices you will also need to specify `<stack-name>` and `<region-name>`, otherwise these two are not required.

- Alternatively, other commands could be used:
  
  Acquire an IP address, by running the DHCP client:

  $ udhcpc -i <net-dev>

  Synchronise date and time (restart ntpd):

  $ killall ntpd && ntpd -gq

  Restart the AWS IoT Greengrass V2 Nucleus:

  $ service greengrass restart

7.13 Sending custom data to AWS IoT MQTT client

You can use the Cloud Gateway to send custom data to AWS.

To send data from your custom application, connect with a socket to `localhost:51001` and send a message with the following format:

```json
{
    "app_data": <payload>,
    "mqtt_topic_suffix": <topic>
}
```

Where `payload` can either be a payload of any format, or a list of payloads. `mqtt_topic_suffix` is optional, and if it is not specified, then the default application data MQTT topic will be used: `<TelemetryTopic>/app_data`. `TelemetryTopic` was specified in the deployment of the Telemetry Stack in AWS.

Run the following command on dom0 to send data to AWS through the telemetry service:

```bash
$ echo '"
    "app_data": ["payload", "other payload"],
    "mqtt_topic_suffix": "example/topic/suffix""
"' | nc -c -v 127.0.0.1 51001
```

In the AWS IoT MQTT client subscribe to the topic `<TelemetryTopic>/example/topic/suffix` before running the command to see the incoming example messages.
The telemetry server is a local server hosted on the v2xdomu VM that can display real-time system telemetry data, such as: A53 CPU loads, Cortex-M7 CPU loads, PFE traffic, Domain-0 Memory, and temperature. It is an alternative for the cloud-gw telemetry dashboards.

The telemetry server starts automatically when the v2xdomu VM boots up. In order to access the local telemetry dashboard, the v2xdomu VM must be connected to the same local network as your workstation. The dashboard can be accessed from any browser by typing in the address bar the following URL: <v2xdomu ip address>:5000/system_telemetry

By default the charts display live telemetry in a time window of 5 minutes. To change this enter the desired time window in seconds in the input box at the top of the webpage and click on the Set time window button.

Chapter Testing the telemetry application describes how to increase the A53 core loads in order to showcase the live telemetry data.
9 SJA1110 Telemetry Application

The SJA1110 telemetry application runs on the SJA1110 switch's Cortex-M7 core. It collects data from the switch and sends it to AWS via the Greengrass core on v2xdomu virtual machine. SJA1110 and the Greengrass core communicate through the local network.

The SJA1110 application consists of two binaries: the firmware `sja1110_uc.bin` and the static configuration `sja1110_switch.bin`. These are uploaded when the board boots. The binaries are located in the default firmware directory, usually `/lib/firmware`. The SJA1110 application restarts at every reboot.

When the application first connects to the network it will acquire an IP address. You will need to provision the SJA1110 application after every reboot of the board (and also if you reset the SJA application manually), this is done by running the Greengrass provisioning script (described in chapter [Connecting the board to AWS](#)) with the `--setup-devices` option. You can also use `--no-deploy` if you already ran the provisioning script before rebooting the board.

The provisioning script will send through the subnet the required information and credentials: AWS endpoint, thing name, topic, v2xdomu virtual machine ip, the certificates of the sja thing, and the certificate authority of the Greengrass core. Next the SJA1110 application will begin sending telemetry data to the cloud through the greengrass core on v2xdomu virtual machine.

9.1 Data flow

SJA1110 telemetry is sent to the cloud in a separate json payload than the other cloud telemetry.

**MQTT topic used:** `s32g/sja/switch/<stack-name>`

**MQTT payload structure:**

```json
{
  s0 : {
    p0 : {
      DropEventsCounter : "<value>,",
      IngressPktsCounter : "<value>,",
      EgressPktsCounter : "<value>,",
      DropEventsDelta : "<value>,",
      IngressPktsDelta : "<value>,",
      EgressPktsDelta : "<value>,",
      link: "<value>
    },
    p1 : {
      ...
    },
    ...
  }
}
```

Where `s0` denotes switch 0 (the only switch on RDB2), and `p<x>` denotes ports 0 through 10.

Messages will arrive at one second interval. The counter fields show the total number of packets for each category since the SJA1110 application started running. The Delta fields show the number of packets for each category since the last message (in other words,
the number of packets in the last second). Link is either 0 or 1, and represents the link status. All values are integer.

The SJA1110 telemetry will appear in three AWS IoT Sitewise Dashboards named SJA1110 Dashboard 1/2/3.
AWS IoT FleetWise is a service that makes it easy for Automotive OEMs to collect, store, organize, and monitor data from vehicles at scale. AWS IoT FleetWise Edge Agent provides C++ libraries that allow you to run the application on your vehicle. Full product documentation is available here: https://github.com/aws/aws-iot-fleetwise-edge/tree/v0.1.2

In GoldVIP, AWS IoT FleetWise Edge Agent runs on the Dom0 virtual machine and collects data from the FlexCAN0 interface available in Linux as CAN0. The system architecture is shown in the following picture:

The main software blocks used in the example are the following:

- **CAN simulator**, which is used to simulate Engine's ECU OBD data on the FlexCAN1 interface (available as can1 in the Linux rootfs). The simulator is wrapped with a SysVinit service which starts the CAN simulator at startup.
- The **GoldVIP Real Time application**, which configures the LLCE to allow diagnostic messages to be sent between FlexCAN interfaces without any M7 core application.
- **AWS IoT FleetWise Edge Agent**, used to request data from the system via the FlexCAN0 interface. When the engine is running on the device, requests will be sent via the can0 interface. These requests will be forwarded by the LLCE to the CAN simulator. The CAN simulator will then reply with simulated engine OBD data. For easier control of the Edge Agent, a wrapper SysVinit service is delivered by default in GoldVIP.
- **AWS IoT FleetWise**, service which runs in the AWS cloud. Data sent by the **AWS IoT FleetWise Edge Agent** can be captured and visualized directly in the cloud.

### 10.1 Hardware setup

In order to connect the CAN simulator to the AWS IoT FleetWise Edge Agent, perform the following steps:
1. Connect the CAN wires as described in the CAN gateway chapter.
2. Connect the S32G board to the internet via a wired or Wi-Fi connection of your choice. By default, the scripts assumes that a connection is active on the GMAC port (P3A).
3. Deploy the GoldVIP images, and set the board to boot from QSPI. Traffic is intended to be forwarded by the LLCE from one FlexCAN interface to the other, hence it’s mandatory that the CAN gateway application is running on the Cortex M7 core.

10.2 Preparing the use case

Several steps need to be performed before running the use case. First time setup:

1. **Launch an Amazon EC2 instance in the AWS cloud and connect to it:**
   a. Launch an EC2 Graviton instance with administrator permissions: Launch CloudFormation Template. You can choose any type of machine from the template.
   b. Enter the Name of an existing SSH key pair in your account from here.
      i. Do not include the file suffix .pem.
      ii. If you do not have an SSH key pair, you will need to create one and download the corresponding .pem file. Be sure to update the file permissions, if you are using a Linux machine: $ chmod 400 <PATH_TO_PEM>. You will need this file to connect securely to the development machine.
   c. Select the checkbox next to 'I acknowledge that AWS CloudFormation might create IAM resources with custom names.'
   d. Choose Create stack.
   e. Wait until the status of the Stack is CREATE_COMPLETE; this can take up to five minutes.
   f. Select the Outputs tab, copy the EC2 IP address, and connect via SSH from your local machine to the development machine:

   ```
   $ ssh -i <PATH_TO_PEM> ubuntu@<EC2_IP_ADDRESS>
   ```

   You are now connected to the EC2 instance. Save the IP address in a file, as you will need it for some of the next steps.

2. **Setup the EC2 instance:**
   a. Clone the AWS IoT FleetWise repo:

   ```
   $ git clone --branch v0.1.2 https://github.com/aws/aws-iot-fleetwise-edge.git &&
   cd aws-iot-fleetwise-edge
   ```

   b. Install dependencies on the EC2 machine. Depending on the machine type you have chosen, this step might take long.

   ```
   $ sudo -H ./tools/install-deps-native.sh
   ```

   c. Install dependencies of the AWS IoT FleetWise on the EC2 machine:

   ```
   $ cd ~/aws-iot-fleetwise-edge/tools/cloud &&
   sudo -H ./install-deps.sh
   ```

   d. Add the IoT FleetWise functionality to the AWS CLI:

   ```
   $ aws configure add-model --service-model file://iotfleetwise-2021-06-17.json
   ```
3. **Generate the certificates which are needed on the S32G:**
   a. Create an empty directory:
   ```bash
   $ mkdir ~/goldvip-deploy && cd ~/goldvip-deploy
   ```
   b. Provision an AWS IoT Thing with credentials. It's possible to modify the vehicle id with a name of your choice, however, this needs to be taken into account at the next steps:
   ```bash
   $ ~/aws-iot-fleetwise-edge/tools/provision.sh \
   --vehicle-id fwdemo-s32g \
   --private-key-outfile ~/goldvip-deploy/private-key.key \
   --endpoint-url-outfile endpoint.txt \
   --vehicle-id-outfile vehicle-id.txt \
   --certificate-pem-outfile ~/goldvip-deploy/certificate.pem
   ```
   **Note:** To create AWS IoT things in Europe (Frankfurt), configure `--region` to `eu-central-1` in the call to `provision.sh`.
   c. Configure the AWS IoT FleetWise Edge Agent:
   ```bash
   $ ~/aws-iot-fleetwise-edge/tools/configure-fwe.sh \
   --input-config-file ~/aws-iot-fleetwise-edge/configuration/static-config.json \
   --vehicle-id `cat vehicle-id.txt` \
   --endpoint-url `cat endpoint.txt` --can-bus0 can0 \
   --output-config-file ~/goldvip-deploy/goldvip-config.json
   ```
   d. Create an archive with the contents of the folder:
   ```bash
   $ tar -cf deploy.tar *
   ```
   You will need to deploy this archive on the S32G board.

4. **Deploy the certificates to the S32G:**
   a. Open a command line or a terminal on your **local machine** and run the following command:
   ```bash
   $ scp -i <PATH_TO_PEM> ubuntu@<EC2_IP_ADDRESS>:~/goldvip-deploy/deploy.tar .
   ```
   b. Power up the S32G board and connect it to the internet via the GMAC Port (P3A) or Wi-Fi. Log in to Dom0 and get its IP from command line:
   ```bash
   $ ifconfig xenbr0
   ```
   c. Copy the archive to the S32G board using scp.
   ```bash
   $ scp deploy.tar root@<xenbr0_ip>:/tmp
   ```
   After the transfer is done, you will find the deployment archive on the Dom0 root file system in the `/tmp` directory. In case this does not work on Windows systems, please use `pscp` (from putty) instead of scp.
   d. From the S32G Dom0 console, run the FleetWise deployment script:
   ```bash
   $ python3 ~/fleetwise/deploy.py --netif xenbr0 \
   --input-archive /tmp/deploy.tar
   ```
   In case you connected a different port to the internet, please change the `--netif` argument. This script will check if the network interface is connected to the internet, will sync the system date on Dom0 and will deploy the contents of the archive to the AWS IoT FleetWise Edge Agent configuration folder. From this
point, the AWS IoT FleetWise Edge Agent will be started at each boot (before login).
In case one might want to re-start the AWS IoT FleetWise Edge Agent it's enough to run the following command (without specifying the --input-archive parameter):

```
$ python3 ~/fleetwise/deploy.py --netif xenbr0
```

At this point, the Fleetwise Edge Agent is running on the S32G.

## 10.3 Running the use case

After performing the steps described in the previous chapter, return to the EC2 instance console and run the following command:

```
$ cd ~/aws-iot-fleetwise-edge/tools/cloud && 
./demo.sh --vehicle-id fwdemo-s32g --campaign-file campaign-obd-heartbeat.json
```

**Note:** If you changed the vehicle-id parameter in the previous steps, you need to change the parameter for the demo.sh script as well.

This will trigger a collection campaign which will capture incoming traffic sent by the AWS IoT FleetWise Edge Agent. This will deploy a 'heartbeat' campaign in the cloud which will periodically collect OBD data. If the campaign is successfully ran, an HTML web page will be generated. One can visualize the result on his local machine simply by copying the resulting files locally and opening them in a web browser:

```
$ scp -i <PATH_TO_PEM> ubuntu@<EC2_IP_ADDRESS>:~/aws-iot-fleetwise-edge/tools/cloud/*.html
```

## 10.4 Tips and tricks

1. The AWS IoT FleetWise Edge Agent service wrapper allows for easier control of the FleetWise Edge service. Run the following command to cleanup the previous deployment:

```
$ service aws-iot-fwe cleanup
```

This will delete the configuration files from the rootfs. In case you need to re-deploy the configuration, you will need to run the FleetWise deployment script. However, as the configuration file is loaded when the process starts, you will also need to stop the service using the following command:

```
$ service aws-iot-fwe stop
```

2. One can observe the CAN traffic on the S32G by listening to CAN frames on the S32G, if the FleetWise Edge Agent is connected to the cloud:

```
$ candump -i can0
```
In case the setup was done correctly, one shall observe CAN traffic with IDs \textbf{7E0} and \textbf{7E8} when running the command, as in the following log:

\begin{verbatim}
  can0 7E0 [8] 00000010 00001001 ... (request from the Edge Agent)
  can0 7E8 [8] 00000011 01111111 ... (response from CAN simulator)
  can0 7E0 [8] 00000111 00000001 ... (request from the Edge Agent)
  can0 7E8 [8] 00010000 00010001 ... (response from CAN simulator)
...\end{verbatim}

In case no traffic is printed when running the \textit{candump} command, this means that the configuration deployed to the board is invalid or the M7 core is stopped.

3. The logs for CAN simulator and the AWS IoT Fleetwise Edge Agent can be found in the rootfs of Dom0 VM at \texttt{/var/log/cansim.log} and \texttt{/var/log/fleetwise.log} respectively.
11 Ethernet Gateway

The Ethernet gateway currently supports the following use cases:

- Layer 2 (bridge/switch) ETH forwarding
- Layer 3 (router) IP forwarding

Both use cases can be run either in slow-path mode with Cortex-A53 cores handling the forwarding or in fast path mode on SJA1110A switch without any load on A53 cores. For the case of Cortex-M7 forwarding only the L3 option is available since the routing can be done only on IP level in the AUTOSAR COM stack.

GoldVIP provides scripts that can be used to measure performance in slow-path and fast-path mode for both UDP and TCP traffic generated with iperf3 (python for Cortex-M7 use case) from host PC. Slow-path means the traffic is routed by Linux running on A53 cores or AUTOSAR COM stack running on Cortex-M7 cores. Fast-path means the data-path traffic is routed by PFE (Packet Forwarding Engine) or SJA1110 companion switch from the board.

11.1 Prerequisites

- S32G Reference Design Board or GoldBox running GoldVIP images.
- PC with 2 Ethernet ports, running Ubuntu 18.04 (with iperf3, minicom, iproute2, python3, strongSwan) or a built GoldVIP Docker image that already contains all the necessary tools.

11.2 Running the A53 slow-path use cases

1. Connect one host PC ETH port to the board's SJA110A switch Port 2, corresponding to the pfe0 interface in Linux.
2. Connect another PC ETH port to the board's PFE-MAC2 ETH port. An USB-to-ETH adaptor can be used as the second PC ETH card but make sure it supports Gigabit ETH and is plugged into an USB3.0 port.
3. Start GoldVIP Docker container on PC (see Building GoldVIP Docker image chapter)
4. Run on host PC `eth-slow-path-host.sh` script to measure performance for L2/L3 forwarding between pfe0 and pfe2, with UDP or TCP traffic, with various payload sizes, e.g.:

```
$ sudo ./eth-slow-path-host.sh -L 3 -d full -t UDP <eth0> <eth1>
```

The above command is measuring throughput between PC eth0 and eth1 that are connected to the board pfe0 and pfe2, in full duplex mode (-d full), with UDP traffic (-t UDP) with default packet size (ETH MTU). Use -h option to list all available options.

**Note**: run `ip a` command on your host PC to find out the exact names of the interfaces `<eth0>` and `<eth1>` connected to the board.

**Note**: The script is connecting to target console via `/dev/ttyUSB0`. In case tty port is different on your PC, specify it explicitly with `-u` argument, e.g., `-u /dev/ttyUSB1`

11.3 Running the Cortex-M7 slow-path use cases

1. Make sure that SW12 is set to ON-ON position. If not, set it to ON-ON and reboot the board in order to load the custom SJA1110A firmware. If the default firmware is loaded then all traffic going to M7 PFE0 will be forwarded on all SJA’s ports.
2. Follow steps 1-3 from Running the A53 slow-path use cases.
3. Run on host PC `eth-slow-path-m7-host.sh` script to measure the performance of Ethernet packet forwarding between pfe0 and pfe2 but through the AUTOSAR COM stack running on Cortex-M7 core:

```
$ sudo ./eth-slow-path-m7-host.sh -l 10 -d full -t UDP <eth0> <eth1>
```

The above command is measuring throughput between PC eth0 and eth1 that are connected to the board pfe0 and pfe2 for a test length of 10 seconds (\(-l 10\)) in full duplex mode (\(-d full\)), with UDP traffic (\(-t UDP\)) with default packet size (ETH MTU). Use \(-h\) option to list all available options.

**Note:** run `ip a` command on your host PC to find out the exact names of the interfaces `<eth0>` and `<eth1>` connected to the board.

### 11.4 Running the PFE fast-path use cases

1. Follow steps 1-3 from Running the A53 slow-path use cases.
2. Run on host PC `eth-pfe-fast-path-host.sh` script to measure the same performance scenarios as above but this time offloaded in PFE without involving A53 core:

```
$ sudo ./eth-pfe-fast-path-host.sh -L 3 -d full -t UDP <eth0> <eth1>
```

Same notes apply as in previous section.

### 11.5 Running the SJA1110A fast-path use cases

1. Connect both host PC ETH ports to the board's SJA1110A Port 2 and Port 3.
2. Run on host PC `eth-sja-fast-path-host.sh` script to measure the same performance scenarios but this time going through SJA1110A switch Port 2 and Port 3:

```
$ sudo ./eth-sja-fast-path-host.sh -L 3 -d full -t UDP <eth0> <eth1>
```

Same notes apply as in previous sections.

### 11.6 Running the IPsec A53 slow-path use cases

This use case establishes a secured connection through IPsec between the host and the target. IPsec provides security at the level of the IP layer. The payload of IP datagrams is securely encapsulated using Encapsulating Security Payload (ESP), providing integrity, authentication, and encryption of the IP datagrams. strongSwan is used as a keying daemon on both the host and the target to establish security associations (SA) between the two peers. The connection can be established either in transport or tunnel mode, with X.509 authentication.

1. Connect one host PC ETH port to the board's PFE-MAC2.
2. Run on host PC `eth-ipsec-slow-path-host.sh` script to measure the performance for the IPsec-established connection between the host and the board's PFE-MAC2 interface:

```
$ sudo ./eth-ipsec-slow-path-host.sh -l 10 -m transport -d full -t UDP <eth0>
```
The above command is establishing an IPsec connection between PC eth0 and board pfe2 in transport mode (`-m transport`), then it measures the throughput for a test length of 10 seconds (`-l 10`) in full-duplex mode (`-d full`), with UDP traffic (`-t UDP`) with default packet size (ETH MTU). Use `-h` option to list all available options. Same notes apply as in previous section.

11.7 Running the IDPS slow-path use cases

This use case demonstrates the Ethernet IDPS (Intrusion Detection and Prevention System) provided by Argus Cyber Security (https://argus-sec.com/). Argus Ethernet IDPS is a security mechanism designed to provide complementary protection for automotive Ethernet networks, on top of the existing commodity security controls available for Ethernet components.

Argus's IDPS includes Access control capabilities from the Data link layer to the Application layer and supports most of the protocols in those layers (both whitelist and blacklist are supported). In addition, more advanced inspection features are available (stateful inspection, DPI). For information on the full IDPS feature set, please contact Argus.

In this use case, the access control capabilities of the IDPS are demonstrated. The inspection is done on the whole network stack (Ethernet, IP, UDP, SOME/IP) on prerecorded network traffic that contains both valid and invalid SOME/IP packets. Only intrusion detection capability is demonstrated in this use case (no prevention or packet dropping). The prerecorded traffic is injected from PC and sent to the target that runs the Ethernet IDPS.

Please follow these steps in order to run this use case:

1. Connect one host PC ETH port to the board's PFE-MAC2.
2. Run on host PC `eth-idps-slow-path-host.sh` script to send packets from PC ETH port to the board's PFE-MAC2. The IDPS will detect invalid messages and send the log back to the PC. On the host PC, run the following command:

```
$ sudo ./eth-idps-slow-path-host.sh <eth-interface>
```

**Note:** Use `-h` option to see all available arguments.

The output of this use case has two parts:

- **IDPS host log:** Contains information about the number of packets transmitted from the host.
- **IDPS target log:** Contains the output of the IDPS executable:
  - **Valid packets:** The number of packets that matched a whitelist rule of the IDPS and are considered valid.
  - **Invalid packets:** The number of packets that did not match any whitelist rule of the IDPS and are considered invalid.
  - **Relevant packets:** The number of packets that are relevant to this use case (SOME/IP packets), this value should contain the sum of the valid and invalid packets and shall be equal to the number of packets that were transmitted by the host.
  - **Irrelevant packets:** - Packets that are not part of the use case (ARP, general network packets), and not SOME/IP packets. This number varies between different runs of this use case.
11.8 Connecting to a Wi-Fi network

1. Two types of Wi-Fi adapters are supported:
   - PCI-E based adapter using NXP 88W9098 chipset (default);
   - USB based wireless adapters.

2. Insert the adapter you are using in the appropriate port:
   - M2 slot, found between P4 and P5 RJ45 adapters, for PCI-E based adapters;
   - USB slot, found next to the UART consoles for the USB based wireless adapters.

3. In order to use the PCI-E Wireless Adapter make sure to set the SW 17 DIP switch to ON. The position of this switch can be found in the Appendix B.

4. By default, the configuration file at `/etc/wifi_nxp.conf` is designed to work with the NXP 88W9098 wireless adapter. In case you are using a different adapter, modify the interface in the file.

5. Add ssid and passphrase to `/etc/wpa_supplicant.conf`:
   - If your Wi-Fi network uses a password:
     ```
     $ wpa_passphrase SSID PASSPHRASE >> /etc/wpa_supplicant.conf
     ```
   - If you are using a public network:
     ```
     $ echo -e "network={
	ssid="SSID"
	netmask=NONE
}" >> /etc/wpa_supplicant.conf
     ```

6. Restart Wi-Fi service:
   ```
   $ /etc/init.d/wifi_setup restart
   ```
12 CAN Gateway

The CAN gateway is based on EB tresos AutoCore Platform for S32G. It is distributed in binary and source code format. It is loaded from the QSPI Flash, then booted to Cortex-M7 core 0 of S32G platform, by the bootloader.

GoldVIP provides the canperf script to generate CAN traffic from Linux/A53 on FlexCAN and measure CAN forwarding performance between two LLCE-CAN ports connected to two FlexCAN ports.

The following architecture was employed:

![CAN Gateway Diagram]

The CAN gateway currently supports the following use cases:

- **CAN to CAN 1:1 frame routing on Cortex-M7 core (Slow route).** The CAN packets are sent from Linux and received on the M7 through the LLCE filters, then passed through the PDU router instance running on the M7 core.
- **CAN to CAN 1:1 frame routing on LLCE (Fast Route).** The CAN packets are sent from Linux and received on the LLCE. As opposed to the previous use case, the packets are routed directly by the LLCE, without any M7 core intervention.
- **CAN to ETH and CAN 1:1 frame routing on LLCE (CAN to Ethernet Route).** The CAN packets are sent from Linux and received on the LLCE. The packet will then be routed by the LLCE firmware to the output LLCE CAN instance. The packet will then be formatted into AVTP format and sent to the PFE. The packets sent to the PFE firmware are then captured on the AUX0 interface as inbound traffic. When the canperf script detects that the injected packets will also be sent to the AUX0 interface, a network service listener, namely "avtp_listener", will start to capture AVTP packets and log them to a file. In
this case, the canperf script will also report how much data was captured by
the ethernet service listener.

• CAN to ETH routing through M7 core. The CAN packets are sent from Linux
and received on the M7 CAN driver from where they are passed to the
AUTOSAR COM stack which forwards it to the PFE2. The format used for
the ethernet packets is UDP.

• CAN to CAN 1:1 frame routing on Cortex-M7 core with secure onboard
communication implemented in software or hardware. Secure CAN packets
are sent from Linux and received on the M7, then passed through the
PDU router to the Secure Onboard Communication module running on the
M7 core. The Secure Onboard Communication module then verifies the
message authentication code. If the code is valid then the module generates
a new mac code for the authentic payload and sends back a secure can
frame. The Secure Onboard Communication module uses the services of
the hardware accelerated Crypto driver or the software Crypto driver to verify
and generate the message authentication codes.

• CAN to CAN n:m frame routing on the Cortex-M7 core with dynamic multi-
PDU-to-frame mapping. Several different CAN Container frames are sent
from Linux, then passed through the PDU router to the I-PDU Multiplexer
module running on the M7 core. The I-PDU multiplexer first extracts the
smaller I-PDUs contained in each frame and then redistributes them to a
larger number of different frames that are sent back to the Linux core. The I-
PDUs do not have a fixed position inside the container frames but are added
to the containers in a "first in first out" strategy.

12.1 Prerequisites

• S32G Reference Design Board or GoldBox running GoldVIP images

12.2 Running the measurements

1. HW setup:
   Connect FlexCAN0 to LLCE-CAN0 and FlexCAN1 to LLCE-CAN1. To locate the CAN
   connector and pins check the figures from the appendix. The CAN wires should be
directly connected High to High and Low to Low.

2. The canperf.sh or canperf-multi.sh scripts will measure throughput of CAN frames
   routing between the configured CAN ports. The canperf-multi.sh script is only used
for the use case that requires the sending or receiving of multiple can frames. Run the
canperf.sh or canperf-multi.sh (found in the Domain-0 rootfs under the can-gw
directory):
   a. Parameter description:
      -t
      CAN transmit interface -use the values as per CAN-GW
      configuration. For the default CAN-GW configuration provided in
      GoldVIP, use the values as indicated for each flow(e.g. slow path,
      fast path, ...) in below sub-chapters.
      -r
      CAN receive interface -use the values as per CAN-GW
      configuration. For the default CAN-GW configuration provided in
      GoldVIP, use the values as indicated for each flow(e.g. slow path,
      fast path, ...) in below sub-chapters.
-i
id of transmit CAN frame -use the values as per CAN-GW configuration. For the default CAN-GW configuration provided in GoldVIP, use the values as indicated for each flow(e.g. slow path, fast path, ...) in below sub-chapters. For the multi-PDU use case, provide an id for each transmitted container frame.

-o
id of receive CAN frame -use the values as per CAN-GW configuration. For the default CAN-GW configuration provided in GoldVIP, use the values as indicated for each flow(e.g. slow path, fast path, ...) in below sub-chapters. For the multi-PDU use case, provide an id for each received container frame.

-s
CAN frame data size in bytes.

-g
frame gap in milliseconds between two consecutive generated CAN frames, use any integer >=0

-l
the length of the CAN frames generation session in seconds, use any integer > 1

-D
CAN frame data in hex format. For the multi-PDU use case, provide a value for each transmitted container frame.

-m
multi-pdu mode. This flag needs to be set for the multi-pdu use case. This parameter can only be set in the canperf-multi.sh script.

b. For slow route:
- Use the following arguments combinations which match the GoldVIP default configuration for CAN-GW
  
  \`
  \text{-t can0 -r can1 -i 0 -o 4} \\
  \text{-t can1 -r can0 -i 2 -o 3}
  \`

  example:

  ```
  $ ./canperf.sh \text{-t can0 -r can1 -i 0 -o 4} \text{-s 64 -g 0 -l 10}
  ```

  - Optionally, one can use the default script provided in the can-gw directory:

    ```
    can\_slow\_path.sh
    ```

    ```
    $ ./can\_slow\_path.sh
    ```

c. For fast route:
- Use the following arguments combinations which match the GoldVIP default configuration for CAN-GW
  
  \`
  \text{-t can0 -r can1 -i 245 -o 245} \\
  \text{-t can1 -r can0 -i 246 -o 246}
  \`

  example:

  ```
  $ ./canperf.sh \text{-s 64 -g 0 -i 245 -o 245 -t can0 -r can1 -l 10}
  ```
d. For can to ethernet route fast path:
   • Use the following arguments combinations which match the GoldVIP default configuration for CAN-GW
     ```
     -t can0 -r can1 -i 228 -o 228
     -t can1 -r can0 -i 229 -o 229
     example:
     ./canperf.sh -s 64 -g 0 -i 228 -o 228 -t can0 -r can1 -l 10
     ```
   • Optionally, one can use the default script provided in the can-gw directory:
     ```
     $ ./can-to-eth.sh
     ```

e. For secure onboard communication using cryptographic primitives implemented in software or hardware:
   • The GoldVIP default configuration for the CAN-GW is configured to receive a secured frame (with CAN ID 64 on the can0 bus for software crypto and ID 68 for cryptographic primitives accelerated by hardware), verify the message contents, and then send back a secured frame (with CAN ID 66 on can0 for software crypto and ID 70 for cryptographic primitives accelerated by hardware) on the can1 bus.
   • If the message received on can0 is not secured (does not contain a valid MAC code) then the received message is dropped and no reply is sent.
   • The configuration works with the following default settings:
     - Both frames must have a DLC value of 64 (e.g., `-s 64`).
     - The 64-byte frame payload consists of 32-bytes of payload information (the so-called "authentic payload") and a 32-byte Message Authentication Code (MAC).
     - The MAC code is generated over an input consisting of a 2-byte data ID parameter configured for each PDU and the 32-byte authentic payload.
     - The payload over which the MAC code is computed does not contain any freshness value information.
     - The data ID for the Rx pdu is 64 (0x0040) for the software crypto and 68 (0x0044) for the hardware crypto. The data ID for the Tx frame is 66 (0x0042) for the software crypto and 70 (0x0046) for the hardware crypto.
     - The MAC code uses a 32-byte key with a value of:
       ```
       0x4e5850494e5850494e5850494e585049
       4e5850494e5850494e5850494e585049
       ```
     - The same key value needs to be used when generating the MAC code in both the A53 core software and the CAN-GW software (where the key value can be updated in the Nxp_Intgr_M7_Crypto.h file).
     - The algorithm used to generate the MAC key is an HMAC algorithm using the SHA-2-256 hash.
     - A cycle time of at least 2 ms is recommended (e.g., `-g 2`).
• The secured payload (authentic payload + MAC code) can be generated using the `gen_secure_payload.sh` helper script.

• The script expects the following arguments:
  - `i`: the data ID (e.g., `-i 0040`).
  - `k`: the key (e.g., `-k 4e5850494e5850494e5850494e5850494e5850494e585049`).
  - `D`: the authentic payload (e.g., `-D 234D7367207365637572656420757369`).

• The script prints the secure payload (authentic payload + MAC code (generated with SHA-2-256)) or an error message when it receives an invalid input.

• Example of using the `gen_secure_payload.sh` script:

  ```bash
  $./gen_secure_payload.sh -i 0040 -k 4e5850494e5850494e5850494e5850494e5850494e5850494e585049
  -D 234D7367207365637572656420757369
  69E6720484D413205348412D32353623
  ```

• With the default settings the following arguments to canperf should give you a similar number of Tx and Rx frames for both software and hardware crypto:

  ```bash
  -g 2 -s 64 -t can0 -r can1 -i 64 -o 66 -l 10 \
  -D 234D7367207365637572656420757369666720484D\n  D413205348412D323536231a2f6677ad62151b7aaca\n  ae784094607d752433544fd363d75ce942728029bce
  -g 2 -s 64 -t can0 -r can1 -i 68 -o 70 -l 10 \
  -D 234D7367207365637572656420757369666720484D\n  D413205348412D32353623499710c87f594e7f451751\n  91eefc39e86d0f3c87e17773e16f9d8985e1d6a06
  ```

• Optionally, one can use the default script provided in the can-gw directory: `can-secoc-slow-crypto.sh` for secure onboard communication using software crypto and `can-secoc-fast-crypto.sh` for secure onboard communication using cryptographic primitives accelerated by the HSE.

  ```bash
  $ ./can-secoc-slow-crypto.sh
  ```
f. For dynamic multi-PDU-to-frame mapping:
   - The GoldVIP default configuration for the CAN-GW is configured to receive two container frames on the can1 bus (with CAN IDs 95 and 96), unpack the contained PDUs, repackage them into three container frames (with CAN IDs 97, 98 and 99) and send them back on the can0 bus.
   - The following diagram shows the CAN packet flow for this use case:

   - The configuration works with the following default settings:
     - Both input container frames have a DLC of 64.
     - Each 64-byte container frames contain two 12-byte I-PDUs and one 28-byte I-PDU.
     - There are three output container frames.
     - The first output container has a DLC of 48, the second output container has a DLC of 64 and the third output container has a DLC of 16.
     - The first output container contains three 12-byte I-PDUs, the second output container contains two 28-byte I-PDUs and the third container contains the remaining 12-byte I-PDU.
     - Each contained I-PDU is prefixed by a Short Header.
     - The Short Header is made up of a 3-byte ID field used for identifying the I-PDU and a one byte DLC field with the size of the I-PDU payload.
     - The I-PDUs do not have a fixed position within the container but are packaged in a "first come first serve" manner.
A cycle time of at least 1 ms is recommended (e.g., -g 1).

- The container PDUs (container frame payloads) can be generated using the `gen_multi_pdu_payload.sh` helper script.
- The script expects one pair of the following arguments for each contained I-PDU:
  - i: the I-PDU ID (e.g., -i 4).
  - s: the I-PDU DLC (e.g., -s 12).
  - D: the I-PDU payload (e.g., -D 444444444444444444444444).
- The script prints the container frame payload or an error message when it receives an invalid input.
- Example of using the `gen_multi_pdu_payload.sh` script:

  ```sh
  $./gen_multi_pdu_payload.sh -i 1 -s 12 -D 111111111111111111111111
  $./gen_multi_pdu_payload.sh -i 2 -s 12 -D 222222222222222222222222
  $./gen_multi_pdu_payload.sh -i 3 -s 28 -D 333333333333333333333333333333333333333333
  ```

- With the default settings the following arguments to canperf -multi should give you a similar number of Tx and Rx I-PDUs:

  ```sh
  $ ./canperf-multi.sh -m -g 2 -l 10 -s 64 -t can1 -r can0
  $./canperf-multi.sh -m -g 2 -l 10 -s 64 -t can1 -r can0
  ```

- Optionally, one can use the default script provided in the can-gw directory:

  ```sh
  $ ./can-multi-pdu-path.sh
  ```

Note: Please run `./canperf-multi.sh -h` or `./gen_multi_pdu_payload.sh -h` to see all the available options.

3. Running CAN to ethernet slow path:
- Connect one host PC ETH port to the board's PFE-MAC2 ETH port.
- Start GoldVIP Docker container on PC (see Building GoldVIP Docker image chapter)
- Run on host PC `can-to-eth-slow-path-m7-host.sh` script to measure performance for CAN to ethernet routing, with various payload sizes and time gaps between CAN frames e.g.:

  ```sh
  $ sudo ./can-to-eth-slow-path-m7-host.sh -s 64 -g 10 <can> <eth>
  ```
**Note**: Due to the fact that CAN communication has a higher priority than ethernet communication you will see a significant reduction in the number of outbound ethernet packets when the system becomes overloaded by CAN frames (e.g., frames with size 4 and a gap between them of 0 ms).

**Note**: run `ip a` command on your host PC to find out the exact name of the ethernet interface `<eth>` connected to the board.

**Note**: The script is connecting to target console via `/dev/ttyUSB0`. In case tty port is different on your PC, specify it explicitly with `-u` argument, e.g., `-u /dev/ttyUSB1`. Also, no other process should use the port during the test.

### 12.3 Building the M7 Application

The distributed CAN-GW binary is compiled from an EB tresos AutoCore Platform that requires some updates for the tresos plugins to get the same functionality as in the distributed binary image:

1. **Download and install the Elektrobit tresos ACG version mentioned in Software prerequisites.** All the components provided in the installer shall be selected and a new empty directory shall be used as the Install Directory.

2. **Patch the Csm tresos plugin** by first navigating to the `<EB_tresos_install_path>` and then applying the patch found in the `<GoldVIP_install_path>/configuration/can-gw/patches` folder. For example:

   ```bash
cd <EB_tresos_install_path>
git apply --ignore-whitespace \ 
<GoldVIP_install_path>/configuration/can-gw/
patches/0001_fix_csm_in_tresos_folder.patch
```

3. **Download S32 Design Studio v3.4 from your nxp.com account and install it.** The GCC compiler needed for the build process is included in S32 Design Studio.

4. **Update NXP plugins:**
   - Replace the `McalExt_TS_T40D33M1I0R0` plugin found in the `<EB_tresos_install_path>/plugins/` directory with the contents of the `McalExt_TS_T40D33M1I0R0.zip` archive, which can be found in the `<GoldVIP_install_path>/configuration/can-gw/plugins` directory.
   - The archive bundles the PFE and LLCE drivers referenced in Software prerequisites chapter along with the required Real-time Drivers.
   - **Note**: EB tresos needs to be restarted after performing this change, in order to load the newly installed plugins.

5. **Update the build environment:**
   - Adapt `<GoldVIP_install_path>/configuration/can-gw/workspace/goldvip-gateway/util/launch_cfg.bat` to your particular system needs. In particular `TOOLPATH_COMPILER` needs to point to the compiler that you installed at step 2 and `TRESOS_BASE` needs to point to tresos install location from step 1.

6. **Open tresos and import `goldvip-gateway` project located at `<GoldVIP_install_path>/configuration/can-gw/workspace/goldvip-gateway`.**

7. **If you have a valid system model (SystemModel2.tdb file) you can right click the project and hit the generate button.** Otherwise, if the system model is not valid anymore or if you have done any changes to the configuration it is best to use `CodeGenerator` wizard. You can launch this wizard by going in Project->Unattended Wizards tresos menu and select Execute multiple tasks(`CodeGenerator`) entry.
8. You should be ready to build the project. Open a Command Prompt and run the following commands:

```
$ cd <GoldVIP_install_path>/configuration/can-gw/workspace/goldvip-gateway/util
$ launch.bat make -j
```

To create a binary file from elf run the following command in the same Command Prompt:

```
$ C:/NXP/S32DS.3.4/S32DS/build_tools/gcc_v9.2/gcc-9.2-arm32-eabi/arm-none-eabi/bin/objcopy.exe -S -O binary \
../output/bin/CORTEXM_S32G27X_goldvip-gateway.elf \
../output/bin/goldvip-gateway.bin
```
13 Over-the-Air (OTA) Updates

This use case demonstrates the capability of doing Over-the-Air Updates of the GoldVIP components using Airbiquity’s OTAmatic Software and Data Management Platform. More information on the solution can be found on https://airbiquity.com.

Contact Airbiquity for additional information about the OTAmatic Software Management Platform and/or full feature set evaluation (otamatic-nxp@airbiquity.com).

13.1 OTA Software Management

OTAmatic orchestrates and automates secure vehicle software update campaigns from the cloud. The following components are used:

- **OTAmatic Service Delivery Platform (SDP)**, which is a cloud-based service used for the management of vehicle software updates (i.e., customizing updates, creating update campaigns).
- **OTAmatic Client**, which is responsible for the communication with the back-end server (OTAmatic SDP) and supervision of update campaign of all ECUs.
- **Update Agent (UA)**, which represents the application that installs an update on an ECU.

The architecture described below exhibits where each OTA component resides and how they communicate with each other:

The OTAmatic Client is started automatically when the system boots up and it connects to OTAmatic Service Delivery Platform (SDP) to fetch the available updates. The OTAmatic software suite provides secure updates via integration of the Uptane security framework that is used to sign and check the authenticity of the delivered updates.

The OTAmatic Client checks the integrity of an update package and then sends it to the registered Update Agent that should install it. Once the Update Agent receives the update package, it does a full Uptane verification to avoid the possibility of installing updates received from malicious actors, and then proceeds with the installation if the authenticity of the update was validated.

The architecture was shaped by the access policies to the hardware resources that are applicable for the virtual machines (as described in Xen chapter):

- the OTAmatic Client resides in the v2xdomu virtual machine, so the resources that require privileged access are protected from outside interferences.
- the Update Agents reside on the dom0 virtual machine and have unrestricted access to the resources used by the components they handle (i.e., access to QSPI memory for...
the Update Agent that updates real-time images, access to disk partition table for Linux VM Update Agent).

By default, both components run as containerized applications in a local K3s cluster. The OTAmatic Client and the GoldVIP Update Agents communicate over the v2xbr virtual switch through Airbiquity's Update Agent REST Interface. The Client gets the configuration of every registered update agent and based on this it fetches the available updates from the OTAmatic SDP backend over the xenbr0 interface. The OTAmatic Client monitors a list of preconditions that are necessary before starting an update (e.g., ignition is on, vehicle is parked, or battery voltage level is above a certain threshold). The vehicle state is simulated through a set of files (exposed in the rootfs of the v2xdomu VM at /home/root/ota/vehif_state by the application container) that can be used to notify the Client about an update regarding any precondition. Once all the preconditions are met, the OTAmatic Client commands each Update Agent to update the component it handles according to the update campaign that was configured in the OTAmatic SDP.

13.2 Prerequisites

- An update campaign configured in OTAmatic SDP and the initial Uptane configuration files for both the OTAmatic Client and Update Agents. GoldVIP is released with a pre-configured campaign and both the OTAmatic Client and Update Agent are already provisioned with their respective configuration. Please contact Airbiquity for details regarding the creation of other update campaigns.

Note: The pre-configured update campaign has been set with a “Passive” policy to automatically perform downloads and installations of OTA update packages without requiring user approval through an HMI/UI. This is a campaign-specific policy setting that can be configured through OTAmatic based on customer preference. Please contact Airbiquity for more details or information about OTAmatic features and functionality.

13.3 Updates for Real-time images

One of the available Update Agents is the one that handles the updates for the GoldVIP real-time images (i.e., real-time bootloader, GoldVIP Gateway application, SJA telemetry applications). It employs A/B partitioning scheme for updates that target the QSPI-resident images: the new binary image is copied to a different flash region instead of over-writing the currently running image. This way, the situations that may lead to an unusable device after an update can be avoided since it is possible to instantly roll back to the previous image.

This Update Agent recognizes update packages that match the following layout:

```
realtime-update-package.tar.gz/
|--images.json
| `--bin-image1.bin
| `--bin-image2.bin
| `--bin-image3.bin
|--...```

The number of images that can be updated is theoretically unlimited with this format. The update manifest - images.json file - contains information regarding each real-time image that should be updated. The Update Agent uses this data to install each binary image, either by flashing it in QSPI (when the type attribute from the update manifest entry is set to “QSPI”) or by copying it in the dom0 VM file system (when the type attribute is set to...
"rootfs"; applicable for SJA1110 binaries). The expected format of the update manifest is the following:

```json
images: [

  { 
    "filename": "<binary image filename (i.e., bootloader, goldvip-gateway.bin)>",
    "description": "<small description of the updated image>",
    "size": <disk size occupied by the binary image>,
    "type": "QSPI"
    "config": { 
      "address": <address in QSPI where the image will be written>,
      "pivot": <null or the address of the IVT entry that should be altered>
    }
  },

  { 
    "filename": "<binary image filename (i.e., sja1110_uc.bin, sja1110_switch.bin)>",
    "description": "<small description of the updated image>",
    "size": <disk size occupied by the binary image>,
    "type": "rootfs",
    "config": { 
      "path": "<path in the dom0 file system where the binary will be copied to>" 
    }
  }

...]
```

Once the update was installed, the system needs to be restarted for the updates to be applied. One can do this by rebooting the board.

The pre-configured update campaign installs a new real-time bootloader, GoldVIP Gateway application, and SJA firmware. All the installed applications will have the same functionalities as their predecessors, to not affect the experience of using any GoldVIP component. All the new QSPI-resident binaries will be placed at other addresses in NOR flash (i.e., 0x500000 for bootloader, and 0x800000 for GoldVIP Gateway app), and the old application will be kept and can be used again if rollback is requested.

The system is configured to use the new real-time bootloader that will start the newly installed applications. After a reboot, one can observe the flash address of each booted application in the logs printed by the bootloader, similar to the following example:

```plaintext
# Before the realtime OTA update:
Load application id 0 [...] from 0x10XXXX flash address, [...] 
Load application id 1 [...] from 0x300000 flash address, [...] 

# After the update:
Load application id 0 [...] from 0x10XXXX flash address, [...] 
Load application id 1 [...] from 0x800000 flash address, [...] 
```

**Note:** The addresses presented in the previous example are indicative and they may be adjusted in the future.
13.4 Updates for Linux Virtual Machines

Another Update Agent applies updates on the available Linux unprivileged Virtual Machines (i.e., v2xdomu VM). It is capable of installing a new kernel image and its corresponding root file system that can be used to boot the v2xdomu VM, while keeping the old resources in place for roll back purposes. The GoldVIP SD-card image is partitioned during the first boot of the Linux image in such way that A/B system updates for v2xdomu are possible - an additional partition is created to store the new roofs, if the storage medium have enough space left. Once the update package is received, the Agent chooses the inactive slot where the root file system is copied to. It also saves the new kernel in dom0 VM file system and then changes the configuration of the v2xdomu VM, so that the new image can be used after reboot.

**Note:** An SD-card with the recommended size described in Hardware prerequisites is required to run this use case, in order to have enough space for the additional partition. If a smaller SD-card is used, the update for this component will not be installed.

This Update Agent recognizes update packages that match the following layout:

```
linuxvm-update-package.tar.gz/
|-update_manifest.json
|-Image-kernel
|-rootfs/
 | |-boot/
 | |-bin/
 | |-etc/
 | |-dev/
 | |-...
```

The update manifest is used only to specify the type of file system to build in order to use the new roofs. The expected format of the update manifest is the following:

```
{
    "partition_type": "<file system type (i.e., ext4)>"
}
```

The v2xdomu virtual machine must be restarted to apply the update. Unfortunately, the VM can't be rebooted by the Update Agent because that would interfere with the update flow. If the reboot of the board isn't desired, the changes can be applied by running the following commands in dom0 console:

```
# Stop the current v2xdomu VM and then start another one using the config file.
$ xl destroy v2xdomu
$ xl create /etc/xen/auto/V2Xdomu.cfg
# Restart the k3s agent to connect to the new cluster.
$ service k3s-agent restart
```

**Note:** The k3s agent must be restarted in order to connect to the k3s server that gets initialized once the updated VM boots up.

When connecting to the new v2xdomu VM, one can notice the new hostname: v2xdomu-v2.

13.5 Running the OTA updates use-case

The OTAmatic Client and the GoldVIP Update Agents should automatically start when Linux boots up, if the Uptane configuration files are in place. The Client will start the
download and installation of the updates once all the preconditions are met and the application is notified about the system change.

Note: The Uptane configuration files are required for the applications to start successfully.

1. Log into the v2xdomu virtual machine using the command:

   ```
   $ xl console v2xdomu
   ```

2. Start the update of Real-time images and Linux VMs using:

   ```
   $ python3 ~/ota/demo/update_monitor.py
   ```

   Note: The update of Linux VMs will take a while, depending on your ethernet speed and the write speed of the used microSD card. One can pass `--only-realtime-update` as a parameter to the previous command to install the updates only for Real-time images.

This will prepare the environment required to run the use-case (i.e., setup the network interface, updating the system date and time, setting the preconditions to the required values) and display in real-time the progress of download and installation steps. An update starts once all the configured preconditions are met:

- for Real-time images update, `IGNITION` must be set to `ON`. This information is exposed in the v2xdomu rootfs in the file `/home/root/ota/vehif_state/ignition`.
- for Linux VMs update, `IGNITION` must be set to `ON` and `BATTERY_LEVEL` must be greater than or equal to 11.5. This information is exposed in the v2xdomu rootfs in the file `/home/root/ota/vehif_state/battery`.
- in order to start an update, the script writes a value of 1 into the `/home/root/ota/vehif_state/update` in order to notify the OTAmatic Client on the change in vehicle state.

After the preconditions are set, the Client will start to look for any available updates, will download their install packages, and then will communicate with every known Update Agent in order to apply the update.

First, the new real-time images will be downloaded and will be installed by the corresponding Update Agent. After the update was installed, the Client checks again the versions of each installed component and will detect that the system is ready for another update, this time applicable for Linux VMs. The installation of this update starts once the preconditions configured in the OTAmatic backend are met.

Running the update monitor script will generate an output similar to the following image:

```
0:00:03 The network interface was configured.
0:00:15 System date and time are now synchronized.
0:00:20 (Linux VMs, Realtime images) can be updated.
0:00:21 Preconditions were set. Ignition ON.
0:00:28 Update available. Start downloading the update package.
0:00:40 The update package for Realtime images was downloaded.
0:00:40 Starting update installation.
0:00:41 Update successfully installed for Realtime images.
0:00:40 Update available. Start downloading the update package.
0:04:16 The update package for Linux VMs was downloaded.
0:05:29 Starting update installation.
0:05:29 Update successfully installed for Linux VMs.
0:10:57 2 updates installed, done!
```
Note: OTAmatic Client might fail to connect to the OTA backend server if your network uses an HTTP proxy. In such case, the OTAmatic Client shall be configured to use that HTTP proxy by running from the console of the v2xdomu VM:

```
$ python3 ~/ota/demo/add_http_proxy.py <HTTP_PROXY_URL>
$ /home/root/ota/demo/otamatic_controller.sh restart
```

3. Reboot the board to apply the updates.

Each OTA application logs its output in a file on the file system. For debug purposes, one can check these files:

- Check `/var/log/otamatic` on the v2xdomu VM for logs from the OTAmatic Client application. Use `tail -f /var/log/otamatic` to check the logs in real-time.
- Check `/var/log/ota/goldvip_uas` on the dom0 VM for logs from the GoldVIP Update Agents. Use `tail -f /var/log/ota/goldvip_uas` to check the logs in real-time.

The state of the OTAmatic Client and GoldVIP Update Agents can be controlled using:

For GoldVIP Update Agents, run on dom0 VM:

```
$ /home/root/ota/demo/ota_agents_controller.sh {start | restart | stop}
```

For OTAmatic Client, run on v2xdomu VM:

```
$ /home/root/ota/demo/otamatic_controller.sh {start | restart | stop}
```

Note: The OTAmatic Client expects that the registered Update Agents are already running when the Client is started. If the GoldVIP Update Agents are restarted, one can make the Client re-establish the connections to them by either triggering an update in the vehicle state.

```
$ echo "on" > /home/root/ota/vehif_state/ignition
&& echo "1" > /home/root/ota/vehif_state/update
```

or by restarting the OTAmatic service

```
$ /home/root/ota/demo/otamatic_controller.sh restart
```

13.6 Resetting the OTA update demo

After a successful update, the Uptane metadata are replaced according to the new system configuration. In order to run the OTA update use case again, one has to restore the environment to factory defaults:

1. Run `~/ota/demo/reset_update_agents.sh` on dom0 VM to reset the Uptane metadata to the initial version for all the Update Agents. This also reverts any installed update for any component.

   Note: The currently running real-time images and/or v2xdomu VM aren't replaced with the previous ones until a reboot of the board.

2. Run `~/ota/demo/reset_sdk.sh` on v2xdomu VM to reset the OTAmatic Client configuration.

3. Run `~/ota/demo/otamatic_controller.sh restart` on v2xdomu VM to restart the OTAmatic Client.
13.7 OTA applications as orchestrated containers

The OTA services can run in both containerized and non-containerized environments with little to no modification required. When a non-containerized environment is chosen, the OTA applications are running as separate SysVinit services, while in the containerized context they are managed by a local Kubernetes cluster.

By default, both the GoldVIP Update Agents and the OTAmatic Client are packaged as containerized applications, and they are deployed on the local K3s cluster on the S32G device. The Kubernetes cluster is used to manage and supervise these services in order to ensure these applications run continuously.

The OTA applications are automatically deployed to Kubernetes cluster when the system boots up. Because only the v2xdomu node acts as a master in the cluster, the auto-deploying manifests for both applications are found at `/home/root/containers/` on the roots of v2xdomu VM:

- **goldvip-update-agents.yaml** manifest file contains the resources required to run the GoldVIP Update Agents in a containerized environment on Dom0. For demo purposes this manifest file can be found at the same path on Dom0 VM, being used to manage the state of the Update Agents through `/home/root/ota/demo/ota_agents_controller.sh` script.
  The manifest defines a pod that is bound to run on Dom0 VM and uses a container image that packages the GoldVIP Update Agents application. The container image is packed as an OCI image and can be found at `/var/lib/rancher/k3s/agent/images/goldvip-ota-agents-container-image-s32g274ardb2.oci-image.tar` on Dom0 VM. The container creates volume mounts to specific host paths in order to keep the data persistent across reboots.
  **Note:** Any change in the manifest file on one of the virtual machines should be duplicated on the other VM to ensure the proper functioning of the demo scripts.

- **otamatic.yaml** manifest file defines the containerized OTAmatic Client application. This manifest describes a pod that runs only on v2xdomu VM and creates a container based on the OTAmatic Client image found at `/var/lib/rancher/k3s/agent/images/goldvip-ota-client-container-image-s32g274ardb2.oci-image.tar` on v2xdomu VM. In order to keep the data persistent across reboots, this manifest creates volume mounts to specific host paths.
14 Virtualization

Virtualization uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer—processors, memory, storage and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs).

Virtualization via containers (containerization or OS-virtualization) emulates an operating system on top of the host OS, leveraging its features to isolate processes and control their access to hardware resources (e.g., CPU, memory, network).

14.1 Xen

By default, the GoldVIP deliverable includes Xen hypervisor. Xen is a type 1 hypervisor (bare metal) that makes possible running multiple instances of the same operating system seamlessly on the hardware. Xen allows creation of virtual machines from the command line or automatically at startup. Xen virtualizes CPUs, memory, interrupts and timers, providing virtual machines with virtualized resources.

Two types of virtual machines are defined by Xen:

- Privileged (Dom0 or Domain-0): The first machine that runs natively on the hardware and provides access to the hardware for unprivileged domains.
- Unprivileged (DomUs): Virtual machines spawned by Dom0. These machines use the hardware resources allocated from the privileged domain (CPU, Memory, Disk, Network).

In the GoldVIP, two virtual machines are started by default, before the user logs in:

- Domain-0, which has access to all the system resources and creates a network bridge for the unprivileged guest. This bridge, namely xenbr0, is the network interface that forwards packets to the DomU;
- v2xdomu, unprivileged domain, which has access only to a limited number of resources.

14.1.1 Using the DomUs

Xen provides several commands via the xl tool stack which can be used to spawn/restart/shutdown unprivileged domains. Several commands can be used from the Domain-0 command line:

- xl list: lists all the active domains running in the system.
- xl create <domain configuration file>: spawns a DomU. An example can be found in the /etc/xen/auto/V2Xdomu.cfg file.
- xl console <domain name / domain ID>: logs into the console of another unprivileged domain.
- CTRL+J]: Resumes to Dom0 console (can be run only from a DomU)
- xl shutdown: Shuts down a DomU

14.1.2 Networking in DomUs

In order to have network access in the DomUs, a bridge must be created from Domain-0. By default, in the provided example, a bridge (xenbr0) is created at boot time. After the
DomU boots, a virtual interface will be created in Domain-0 and will forward packets to the DomU. After logging into the DomU console, the interface will be visible as eth0.

Another bridge can be created with the following commands, after choosing a physical interface to be shared:

```
$ ifconfig <eth interface> down
$ ip addr flush <eth interface>
$ brctl addbr <bridge name>
$ brctl addif <bridge name> <eth interface>
$ ip link set dev <bridge name> up
$ ip link set dev <eth interface> up
```

You can then assign an IP to the newly created bridge and use it in Domain-0 using `ifconfig <bridge name> ip`.

**Note:** Do not set an IP address for the physical interface.

### 14.1.3 Configuring the V2X domU

The V2X domU configuration is stored in the `/etc/xen/auto` directory and it is started before the user logs in. In order to prevent the machine from auto-starting at boot time, it is necessary to move the configuration file to a different directory (for example `/etc/xen`). After reboot, only Domain-0 will be started.

Several configuration fields are present in the V2X domU configuration file:

- **kernel**: The image that will be used in order to boot the DomU;
- **memory**: Allocated memory for the domU, in MB;
- **name**: DomU name. This name can be used to connect to the DomU using the `xl console` command;
- **vcpus**: Number of virtual CPUs that are to be used for the VM;
- **cpus**: Physical CPUs that are allocated to the VM;
- **disk**: Physical storage device that stores the file system for the VM;
- **extra**: Root device, console setting;
- **vif**: Network bridge that forwards frames from the physical interface, created automatically at Domain-0 boot time.

For more detailed information please consult the Xen official documentation: [https://xenproject.org/](https://xenproject.org/)

### 14.2 Containers

Containers are a lightweight virtualization technique used to package and confine applications with their entire runtime environment. A container is a set of one or more processes that run isolated from the rest of the system, starting from a container image, which bundles the files required to support the processes. The isolation provided by containers leverages several underlying technologies built into the Linux kernel: namespaces, cgroups, chroots.

There are two general usage models for containers:

- **Application containers** – Running a single application or executable program in a container.
- **System containers** – Booting an instance of user space in a container; usually used to run multiple processes at the same time.
14.2.1 OCI container images

A container image is a way to bundle an application to be used as a container. The application and any run-time requirements are included in this package, which is just a directory of files with metadata about how to operate the container. A container image is composed of:

- Independent layers (i.e., directories with files), which through union mounting can be composed together and create the rootsfs available in a container.
- Digests used to uniquely indentify the layers compressed as tarball archives. Through this mechanism, one layer can be referenced by multiple images.
- Container configuration metadata contains information about how the container should be run (i.e., the environment variables, exposed ports, command to be executed).
- The container image manifest, which describes the components that make up a container image, listing the layers referenced by the image, and some other metadatas that uniquely identify an image.

Different tools and methods exist to create such images, such as buildah, umoci, or translating Docker container images to OCI format through skopeo. To convert a Docker container image that is publicly available on DockerHub, one can use skopeo:

```bash
$ # Get the OCI layout for the latest BusyBox image from DockerHub.
$ skopeo copy docker://busybox:latest oci:busybox
$ # Archive the OCI layout to be later imported by container runtimes.
$ tar -cf busybox.tar -C busybox .
```

GoldVIP containerized applications are packaged in Open Container Initiative (OCI) Image format. OCI container images can be used with most of the available container engines, hence the GoldVIP containers can be deployed using Docker, Podman, containerd, and many other implementations.

In order to create OCI images for GoldVIP applications, one can start from the GoldVIP Yocto layer. Check Build and deploy OCI containers chapter for more details.

14.2.2 Container runtimes

GoldVIP includes runC and containerd container runtimes. These can be used to spawn and run containers based on OCI Image format separately from the Kubernetes cluster deployed on GoldVIP. containerd provides the ctr CLI, which can be used to spawn and manage standalone containers and images, outside of K3s:

- `ctr images import <image-tar-archive>`: Import a container image from a tar archive. Can be used to import container images produced in the GoldVIP Yocto build.
- `ctr images ls`: List the known container images.
- `ctr images pull <source>`: Fetch and prepare an image to be later used by containerd engine.
- `ctr run <image-src> <container-id>`: Run a container based on image-src container image.
- `ctr container ls`: List the running containers.
To run a standalone BusyBox container using the latest image published on Docker public registry, the following commands must be executed:

```bash
$ # Ensure that containerd daemon is running.
$ containerd &
$ # Fetch the latest BusyBox image from DockerHub.
$ ctr image pull docker.io/library/busybox:latest
$ # List the available images. The BusyBox image should be now listed.
$ ctr image ls
$ # Run a container based on the fetched image.
$ ctr run --rm -t docker.io/library/busybox:latest demo /bin/sh
```

A shell within the container will be obtained. To exit the container console, press **Ctrl+D**.
15 Container Orchestration

Container orchestration automates the deployment, scheduling, managing and scaling of containerized workloads in dynamic environments. One of the most used container orchestration platform is Kubernetes. K3s is a lightweight production-ready Kubernetes distribution built for Edge computing, and optimized for ARM architecture. K3s is integrated by default in the GoldVIP image and makes possible to deploy, scale and manage highly available workloads on the S32G platform.

GoldVIP configures K3s to optimize the resource consumption and to automatically start some services (e.g., OTAmatic Client, GoldVIP Update Agents) immediately after the platform was booted up. Along the specific GoldVIP container images, the images required for the proper functioning of the cluster (i.e., pause-container) and of the other Kubernetes default components (i.e., coredns) are already present in the rootfs of the VMs. This ensures that the cluster will work when there is no Internet connection available (also known as air-gapped environment). Check the Containers chapter for more information on how the GoldVIP application are packaged as container images.

15.1 The K3s cluster

A Kubernetes cluster is a collection of nodes that work together to run containerized applications and workloads in a scalable, automated, and distributed manner. Using Kubernetes clusters, one may orchestrate and monitor containers across numerous physical, virtual, and cloud nodes, allowing for more flexible and reliable deployments by decoupling the containers from the underlying hardware.

A Kubernetes cluster consists of its control-plane components (deployed on master nodes) and node components (each representing one or more host machines running a container runtime and kubelet service, or the primary node agent; also known as worker nodes). The master node manages the state of the cluster (e.g., which applications to run, and where they should be deployed), by coordinating processes such as maintaining a desired state of the cluster, scheduling and scaling applications and implementing updates on the cluster components.

The worker nodes are the components that run the containerized workloads, performing tasks assigned by the master node. In Kubernetes architecture, the applications are encapsulated as pods, namely a group of one or multiple containers tightly coupled, and share the hardware resources of the worker node where it was deployed. A Pod is the smallest execution unit used in Kubernetes. Other built-in workload resources are available, such as Deployments, DaemonSets, and CronJobs, which build upon the Pod concept. Please check the Kubernetes documentation for more details regarding these concepts.

GoldVIP uses K3s to create a Kubernetes cluster that share the hardware resources from both virtual machines. In K3s architecture, a node that runs the k3s server command will acts both as a master and a worker node, deploying the control plane components and executing different workloads, while a node that runs the k3s agent will act only as a worker node. The following image exhibits the cluster deployed in the context of GoldVIP, the roles of each node, the default containerized applications and the way the nodes communicate in the cluster:
To learn more about the K3s architecture, please visit [https://k3s.io/](https://k3s.io/).

The cluster is composed of the following nodes:

- **v2xdomu**, which starts the `k3s` server process and acts as a master node. This node runs the control plane components and also acts as a worker in the cluster, running the applications bound to v2xdomu VM. The server listens for connections on the V2X interface, using `goldvip` token for authenticating agents that try to join the cluster. In order to optimize the hardware resource utilization, the K3s server starts with most packaged components (such as `coredns`, `metrics-server`, `traefik`) disabled. More advanced applications might need some of those components, so they must be enabled by editing the configuration file and restarting the K3s server process. The configuration used for the K3s server can be found at `/etc/rancher/k3s/config-server.yaml`. One can start the K3s server process with their custom configuration by editing this file, and then run `service k3s-server restart` to apply them.

  **Note**: It is not recommended to use such a weak shared secret for cluster authentication in production environments. A good practice is to use the token generated by the master node, which can be found at `/var/lib/rancher/k3s/server/node-token`. The `goldvip` authentication token is used only for demo purposes, also considering that the connections to the cluster can be established only from the internal network (i.e., V2Xbridge).

- **s32g274ardb2**, which acts as worker node and runs the applications bound to Dom0 VM. The K3s agent process is already set up to connect to the server counterpart, using the configuration available at `/etc/rancher/k3s/config-agent.yaml`. One can edit this configuration file, and then run `service k3s-agent restart`, in order to apply custom settings to the K3s agent process.

The applications can be automatically deployed on the cluster when the S32G board starts-up using auto-deploying manifests. Any file found in `/var/lib/rancher/k3s/server/manifests` on the roots of the v2xdomu VM will be used to deploy the defined components at the startup of the `k3s` server process.

The GoldVIP containerized applications use containers that are not publicly available on any public registry. Those images are present on the VMs' roots at `/var/lib/rancher/k3s/agent/images` and are automatically loaded at the startup of the node components (for both `k3s` server and `k3s` agent commands).
15.2 Orchestrating containers with K3s

The Kubernetes command-line tool, kubectl, provides means to deploy applications, inspect and manage cluster resources, or debugging components of the cluster. The Kubernetes objects are defined in configuration files (usually in YAML format) that are passed to the kubectl command to create or delete resources. Several commands can be used from within the v2xdomu VM to manage the local K3s cluster:

- kubectl create -f <configuration file>: Create the objects defined in a configuration file. Examples can be found in the /home/root/containers directory.
- kubectl get nodes: List the available nodes. Currently, two nodes are present in the local cluster, namely v2xdomu and s32g274ardb2.
- kubectl get pods: List all pods and their status.
- kubectl describe pod <pod name>: Get details of the specified pod, including the list of running containers.
- kubectl exec -it <pod name> -c <container name> -- /bin/sh: Get an interactive console to the specified container. Used for debugging.
- kubectl delete -f <configuration file>: Delete the objects defined in the configuration file.

For more details regarding the kubectl command, please check the official Kubernetes documentation: https://kubernetes.io/docs/reference/kubectl/

The kubectl command can be executed from any node that deploys control-plane components, such as v2xdomu node. By default, the s32g274ardb2 node also gets capabilities for accessing and managing the cluster. A DaemonSet component is automatically deployed on the Dom0 VM to sync the kubeconfig file (found at /etc/rancher/k3s/k3s.yaml) between the master node and the s32g274ardb2 node. Therefore, the kubectl command can be used from the context of Dom0 VM once the cluster is initialized.

15.3 Deploying custom applications in K3s cluster

Using the provisioned Kubernetes cluster, various applications can be deployed on the S32G board. For starters, one can deploy an Nginx web server starting from the example manifest file provided by GoldVIP:

1. Log into the v2xdomu VM using the following command:

   $ xl console v2xdomu

2. Deploy the Nginx web server components:

   $ kubectl apply -f /home/root/containers/nginx.yaml

   This will create a deployment with one pod and a service used to route requests to the containerized Nginx web server. The pod will spawn one container based on the nginx-alpine image that will be automatically pulled from the known registries (usually the Docker public registry). By default, the pod will be deployed only on s32g274ardb2 node.

3. Use kubectl commands to monitor the status of the deployment:

   $ # Check the status of the deployment. The web server was deployed once the 'READY' value is set to '1/1'.

   $ kubectl get pods
   $ kubectl get nodes
   $ kubectl describe pod nginx-pod
4. The web server is now running and can be accessed using:

```
$ wget -qO- http://10.0.100.20:<SVC_EXTERNAL_PORT>
```

Where `<SVC_EXTERNAL_PORT>` is the external port used to expose the deployed application outside of the cluster, and it can be obtained by running $ kubectl describe service ngixn | grep 'NodePort:'. The web server can also be accessed from a browser if the S32G board and the host PC are in the same local network. In the browser of your choice, type http://<V2XDOMU_IP>:<SVC_EXTERNAL_PORT> and the default Nginx HTML page will be displayed.

In order to forward the port used by the service to another host port, run the following command:

```
$ kubectl port-forward deployment/nginx 80: \
--address="0.0.0.0"
```

The command will start redirecting the traffic incoming on the host port 80 to the external port used by the service. Now, the web page is accessible without the need of specifying the service port: http://<V2XDOMU_IP>:<SVC_EXTERNAL_PORT>. The port will be forwarded as long as the process runs. Use CTRL+C keys combination to stop the port forwarding.

5. Updating the deployment can be done by editing the manifest file and re-apply the changes. In order to increase the number of pods deployed, one must change the replicas entry in the manifest file. To achieve this, use a text editor of your choice (i.e., vim) to manually edit the file, or simply run the following command to increase the replicas count to 5:

```
$ sed -i 's|replicas:.*|replicas: 5|g' /home/root/containers/nginx.yaml
```

Optionally, some of the other replicas may be scheduled on the v2xdomu node if the nodeSelector option is replaced with a common label or removed from the manifest file:

```
$ sed -i 's|vmttype:.*|kubernetes.io/os: linux|g' /home/root/containers/nginx.yaml
```

Now the changes can be applied by executing the same command used to initially deploy the application:

```
$ kubectl apply -f /home/root/containers/nginx.yaml
```

The number of entries returned by the last command should match the value specified in replicas field. If the nodeSelector option was changed to match a label set on v2xdomu node, some of the pods will be scheduled on it as shown in the output of the command.
6. Deleting the deployment and its resources can be done by running:

```bash
$ kubectl delete -f /home/root/containers/nginx.yaml
```

For more resources and examples, please check the official Kubernetes documentation: https://kubernetes.io/docs/home/
16 IPCF Shared Memory

Inter-Platform Communication Framework (IPCF) is a subsystem which enables both AUTOSAR and Linux applications, running on multiple homogenous or heterogeneous processing cores, located on the same chip, to communicate over shared RAM. In GoldVIP, IPCF is used to communicate between applications running on the Cortex-A53 cores (inside Domain-0 VM) and the Cortex-M7_0 running an AUTOSAR application.

The IPCF introduces two key concepts:

- Instance: identifies the connection between two cores (i.e. A53 and M7) over shared memory.
- Channel: identifies a distinct communication topic between cores. Each IPCF instance can have multiple channels assigned to it.

The shared memory is exposed in Linux as a character device driver, inserted at boot time by the Kernel. If both the M7_0 and the Linux initialize the shared memory properly, a new driver named `ipcfshm` will appear in the `/dev` directory. This directory is further split, containing one entry per IPCF instance (remote core). In GoldVIP only one remote core is available (`M7_0`). Each IPCF instance is further split into communication channels, displayed as files in the Linux rootfs. Two communication channels are available currently in GoldVIP:

- `echo`: Any message sent through this channel will be echoed back by the Cortex-M7_0.

The communication channels buffer data through a FIFO mechanism. In case the buffers are not read in time by the user (buffer overflow), the oldest message in the queue will be overwritten. Reading and writing data to the IPCF channels can be done using regular read and write operations. (`cat/echo`).

16.1 Running the use case

In order to test the IPCF functionality and data buffering, one can send data to the M7_0 and read the output using read and write operations, from the Domain-0 console:

```
<table>
<thead>
<tr>
<th>command</th>
<th>argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>echo</td>
<td>message#1 &gt; /dev/ipcfshm/M7_0/echo</td>
</tr>
<tr>
<td>echo</td>
<td>message#2 &gt; /dev/ipcfshm/M7_0/echo</td>
</tr>
<tr>
<td>echo</td>
<td>message#3 &gt; /dev/ipcfshm/M7_0/echo</td>
</tr>
<tr>
<td>cat</td>
<td>/dev/ipcfshm/M7_0/echo</td>
</tr>
</tbody>
</table>
```

The sent messages will then be printed by the `cat` command.

16.2 Limitations

The following limitations were put in place in order to limit the memory consumption and the core load generated by intensive read/write operations:

- Maximum message size: 128B. Any message with a longer size will not be sent to and from the remote core.
- Maximum number of buffered messages: 64. If more than 64 messages are received and the user does not process (read) them, the last message received will be lost.
- Maximum number of messages per second: ~1000. If more than 1000 messages per second are sent by the A53 core, the M7_0 will not be able to process them.
17 Machine learning

This use case demonstrates the capability of deploying trained models from a deep learning training environment to an embedded application using the NXP eIQ Auto framework. eIQ Auto is a set of offline tools and a runtime framework that enables the porting, execution, and validation of trained neural networks on these heterogenous embedded systems.

The offline tooling is used to prepare a neural network for heterogenous inference by: partitioning the model, assigning the workloads to compute resources, and optimizing the workloads for the target hardware. The runtime executes the workloads using a platform-agnostic API that abstracts the scheduling, data movement, and synchronization of workloads on different compute engines.

Check the eIQ Auto release mentioned in Software prerequisites for more extensive documentation, details and examples.

GoldVIP demonstrates how to use eIQ Auto to deploy different AI/ML models to the S32G. GoldVIP provides the models, datasets and the runtime applications needed to demonstrate different use-case studies, such as predictive maintenance and Battery Management System. One can use the data provider environment from the GoldVIP Docker container to provide live data points to the Machine Learning inference system that is running on the S32G device. The predictions are then published to the Cloud via the GoldVIP Telemetry application in order to gather, easily visualize, and analyze the behavior of the machine learning model on different inputs.

The following general architecture was employed:

17.1 Prerequisites

- For visualization purposes, the GoldVIP Telemetry application must be first deployed in order to connect the board to the Cloud. Please follow the steps described in the Cloud Edge Gateway chapter to deploy the Telemetry application on your device and connect the S32G board to the Cloud.
- The ML data feeder environment provided within the GoldVIP Docker container. Please follow the steps described in the Docker chapter to deploy a container based on the GoldVIP Docker image.
17.2 Predictive Maintenance use-case

The predictive maintenance use case showcases two machine learning models (one with an Long Short-Term Memory architecture and the other with an Temporal Convolutional Network architecture) running on two inference runtimes (ONNX and Glow) provided by elQ Auto. They predict the remaining unit lifecycle (RUL) of aircraft engines from an input of 50 consecutive cycles. Each cycle contains 25 data points that describe the current state of the engine.

The predictive maintenance application receives the input data over an ethernet connection, which it feeds into the two models, each run on both inference engines. The four outputs (one for each model-runtime pair) consist of the predicted RUL and the model's execution time. These, as well as the real RUL which came with the input data, are sent to an AWS SiteWise Dashboard via the Telemetry application described in Cloud Edge Gateway chapter.

The input dataset contains data from multiple engines. Each engine has multiple (at least 50) consecutive cycles and the RUL after each cycle. The script which sends the data to the predictive maintenance application will send multiple inputs one after another. It will send all available inputs from an engine before moving on to the next one. For example, if engine X has 60 cycles, and an initial RUL of 100, the script shall send cycles 1 through 50 as the first input, then 2 through 51, until it will send the last input for this engine: 11 through 60. Each input will be sent along with the real RUL after the 50 cycles (in this example it will be 50, 49, and finally 40).

17.2.1 Running the demo

Ensure that all the prerequisites listed in the Prerequisites chapter are installed and/or deployed. Use the data feeder environment from the deployed GoldVIP Docker container to start providing input data to the predictive maintenance runtime application, by running the following command:

```
$ python3 ~/ml-data-feeder/pd_data_generator.py \
   --board-ip <dom0_ip> --time-step <seconds> \ 
   --stop-after <count>
```

Where --board-ip is the IP of the dom0 VM, --time-step is the time in seconds between the transmission of subsequent inputs (default is 1), and --stop-after is the count of inputs to send (default is None, in which case the script will send all the available inputs).

The predictive maintenance runtime application will receive the data and then inputs into the available ML models. Once the prediction results are obtained, they are sent to the Cloud via the Telemetry application, where can be visualized within the specific Machine Learning AWS IoT SiteWise Dashboard.

17.2.2 Visualization

The Machine Learning Dashboard contains graphs that contain the real RUL and the predicted RUL for each model architecture (i.e., TCN and LSTM). The dashboard contains a graph showing the execution times for each model-runtime pair that can be used to further analyze the performance of each individual model.

The real RULs will form a straight line, while the predicted RULs will converge on the real value as each engine approaches the end of its lifetime. When the real RUL value jumps to a higher value it means that the input data changed from one engine to the next.
Each model generally gives the same results on both runtimes, what may differ is the execution time.

17.3 Battery Management System use-case

The Battery Management System (BMS) use-case illustrates multiple machine learning models (Keras 2.x h5 and PyTorch models based on Long Short-Term Memory architecture, and a PyTorch model based on the Transformer architecture) running on two different inference runtimes (ONNX and Glow) provided by eIQ Auto. These models are used to predict the State of Charge (SoC) of a Li-ion Battery from an input of 70 consecutive temperature and voltage measurements.

The BMS runtime application receives input data over an ethernet connection, which is fed into the three available models:

- Keras 2.x h5 model based on LSTM architecture, running on both inference runtimes.
- PyTorch model based on LSTM architecture, running on both inference runtimes.
- PyTorch model based on Transformer architecture, on ONNX runtime.

The outputs of each model consist of the predicted SoC and the execution time. These, along with the real SoC received as the input, are then sent to an AWS SiteWise Dashboard via the Telemetry application described in Cloud Edge Gateway chapter.

The input dataset contains data from a set of temperature, voltage, and electrical current measurements done on a 2.9Ah Panasonic 18650PF cell, as part of the Kollmeyer, Phillip (2018), "Panasonic 18650PF Li-ion Battery Data", Mendeley Data, V1, doi: 10.17632/wykht8y7tg.1 study.

17.3.1 Running the demo

Ensure that all the prerequisites listed in the Prerequisites chapter are installed and/or deployed. Use the data feeder environment from the deployed GoldVIP Docker container to start providing input data to the BMS runtime application, by running the following command:

```
$ python3 ~/ml-data-feeder/bms_data_generator.py \
--board-ip <dom0_ip> --time-step <seconds> \
--stop-after <count>
```

Where --board-ip is the IP of the dom0 VM, --time-step is the time in seconds between the transmission of subsequent inputs (default is 1), and --stop-after is the count of inputs to send (default is None, in which case the script will send all the available inputs).

The BMS runtime application will receive the data and then inputs into the available ML models. Once the prediction results are obtained, they are sent to the Cloud via the Telemetry application, where can be visualized within the specific Machine Learning AWS IoT SiteWise Dashboard.

17.3.2 Visualization

The Machine Learning Dashboard contains graphs showing the prediction results and execution times for each Battery Management System-specific ML models. The predicted SoC by each model, along with the real SoC is available in separate graphs for each model format (i.e., PyTorch, Keras 2.x h5).
Each model generally gives the same results on both runtimes, what may differ is the execution time.
18 Offloading cryptographic operations to HSE

The NXP Hardware Security Engine (HSE) is a security subsystem aimed at running relevant security functions for applications with stringent confidentiality and authenticity requirements. In GoldVIP, the HSE subsystem is used by applications running on different cores (Cortex-M7 and Cortex-A53) to offload multiple cryptographic operations and to accelerate cryptographic primitives.

18.1 Offloading cryptographic operations from Linux

The HSE crypto driver provides support for offloading cryptographic operations to HSE’s dedicated coprocessors through the kernel crypto API. In order to use the HSE cryptographic offloading capabilities, the HSE firmware must be installed and the key catalogs must be formatted. The GoldVIP bootloader configures the HSE key catalog in order to be used by both realtime and Linux components.

GoldVIP enables the broadly-used OpenSSL library to make use of the cryptographic offloading capabilities. OpenSSL is a popular software library, a robust and full-featured toolkit for the SSL and TLS protocols, and also a general-purpose cryptography library. Its cryptography library is based on a pure software implementation that needs to use lots of CPU resources which may affect the performance of the entire system. The OpenSSL library residing on the dom0 virtual machine is configured in such a way to use the AF_ALG interface for accessing the Linux kernel crypto APIs exposed by the HSE crypto driver. Therefore, any userspace application residing on dom0 and linking with the OpenSSL library will make use of the capabilities of the NXP Hardware Security Engine and will conserve CPU cycles.

**Note:** The current implementation of the OpenSSL AF_ALG engine provides support for accelerating a small pool of symmetric algorithms. Use $ openssl engine -c -t to check the available off-loadable algorithms.

In order to test the cryptographic offloading capabilities, one can use the OpenSSL command line tool:

```
$ # Get the initial number of interrupts handled on the HSE MU instance.
$ cat /proc/interrupts | grep hse
$ # Issue some commands to make use of an off-loadable cryptographic algorithm.
$ openssl speed -evp aes-128-cbc -elapsed
$ # Check the number of HSE interrupts; the count should have increased.
$ cat /proc/interrupts | grep hse
```

OpenSSL can be re-configured on the fly to fall back to the software implementation, by commenting out the `openssl_conf = goldvip_openssl_def` line from the `/etc/ssl/openssl.cnf` file. One can use the following command to do this automatically:

```
$ sed -i 's/^openssl_conf = goldvip_openssl_def/#&/g' /etc/ssl/openssl.cnf
```

To re-enable the usage of AF_ALG engine by default, use:

```
$ sed -i '/openssl_conf = goldvip_openssl_def/s/^##//g' /etc/ssl/openssl.cnf
```
**Note:** OpenSSL with the AF_ALG engine enabled will fallback to the Kernel Space Cryptographic Implementation if the HSE crypto driver is not probed. This can lead to higher computation times for the off-loadable algorithms through the AF_ALG engine.
19 OpenEmbedded/Yocto project for GoldVIP

GoldVIP project manifest files are used together with GoldVIP Yocto meta layer to build the NXP S32G Vehicle Integration Platform (GoldVIP).

19.1 First Time Setup

To get and build GoldVIP you need to have repo installed and its dependencies. This only needs to be done once.

- Update the package manager ($ sudo apt update)
- Install repo tool dependencies:
  - Ubuntu 18.04 LTS or later
  - python 2.x - 2.6 or newer ($ sudo apt-get install python)
  - git 1.8.3 or newer ($ sudo apt-get install git)
  - curl ($ sudo apt-get install curl)
- Install repo tool:

```
$ mkdir ~/bin
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
PATH=${PATH}:~/bin
```

19.2 Building GoldVIP

The following steps will build a GoldVIP image based on NXP Auto Linux BSP image.

**Note:** A Yocto build needs at least 50 GB of free space and takes a lot of time (a few hours, depending on the system configuration). It is recommended to use a powerful system with many cores and a fast storage media (e.g., SSD). The recommended RAM size is 8 GB.

19.2.1 Download the Yocto project environment

Get the latest GoldVIP manifest files and bring other required repositories:

```
$ mkdir nxp-yocto-goldvip
$ cd nxp-yocto-goldvip
$ repo init -u https://source.codeaurora.org/external/autobsps32/goldvip/gvip-manifests -b develop -m default.xml
$ repo sync
```

**Note:** for a specific GoldVIP release or engineering build, please use the proper branch (-b <branch>) and manifest file (-m <manifest>).

**Note:** to fetch internal development repositories use Bitbucket URL:

```
$ repo init -u ssh://git@bitbucket.sw.nxp.com/gvip/gvip-manifests.git
  -b develop -m default-bitbucket.xml
```

Manifest files description:

- default.xml -> fetch GoldVIP repositories from https://source.codeaurora.org
- default-bitbucket.xml -> fetch internal development repositories from Bitbucket
- alb.xml -> default NXP Auto Linux BSP without GoldVIP extensions
19.2.2 Setup the build environment

- Install all prerequisites before starting the Yocto build (first time only):
  
  ```
  $ ./sources/meta-alb/scripts/host-prepare.sh
  $ sudo apt-get install libssl-dev
  ```

- Create a build directory and setup build environment:
  
  ```
  $ source nxp-setup-alb.sh -D fsl-goldvip -m <machine> -e "meta-aws
  meta-java"
  ```

This release includes support for the following `<machine>` (NXP boards):
  s32g274ardb2, s32g399ardb3.

PFE firmware, Xen hypervisor and bridging utilities are mandatory and configured by
the fsl-goldvip distro.

fsl-goldvip distro includes optional features such as GoldVIP CAN Gateway, GoldVIP
Real-time Bootloader, GoldVIP OTA applications, GoldVIP Machine Learning use case,
GoldVIP containers deployed via K3s, and SJA1110 firmware. Any of these features
can be removed from the image.

- Download GoldVIP binaries from your nxp.com account and append the following line
to the file `build_<machine>/conf/local.conf`:

  ```
  GOLDVIP_BINARIES_DIR = "<path to the local GoldVIP binaries
directory>"
  ```

- Download PFE firmware from your nxp.com account and append the following line to
  the file `build_<machine>/conf/local.conf`:

  ```
  NXP_FIRMWARE_LOCAL_DIR = "<path to the local s32g_pfe_class.fw file>"
  ```

Note: To remove any of the optional GoldVIP features, such as CAN Gateway,
GoldVIP Real-time Bootloader, GoldVIP OTA applications, GoldVIP Machine Learning
use case, or GoldVIP containers support, append

  ```
  DISTRO_FEATURES_remove = "<GOLDVIP_FEATURE>"
  ```

to the file `build_<machine>/conf/local.conf` for each
use case that shouldn't be included. Appending the following lines to the aforementioned
file will remove every optional GoldVIP features:

  ```
  DISTRO_FEATURES_remove = "goldvip-gateway"
  DISTRO_FEATURES_remove = "goldvip-bootloader"
  DISTRO_FEATURES_remove = "goldvip-ota"
  DISTRO_FEATURES_remove = "goldvip-ota"
  DISTRO_FEATURES_remove = "goldvip-containerization"
  ```

Note: To remove SJA1110 firmware, append the following lines to the file

```
build_<machine>/conf/local.conf:
```

  ```
  SJA1110_UC_FW = ""
  SJA1110_SWITCH_FW = ""
  ```

Note: To use internal development GoldVIP repository add the following line in

```
build_<machine>/conf/local.conf:
```

  ```
  GOLDVIP_URL = "git://bitbucket.sw.nxp.com/gvip/gvip.git;protocol=ssh"
  ```

19.2.3 Build the image

```
$ bitbake fsl-image-goldvip
```
Running the above command would be enough to completely build ARM Trusted Firmware, kernel, modules and a rootfs ready to be deployed. Look for a build result in `build_<machine>/tmp/deploy/images/`.

19.2.4 Deploy the image

The file `<image-name>.sdcard` is a disk image with all necessary partitions and contains the bootloader, kernel and rootfs. You can just low-level copy the data on this file to the SD card device using `dd` as on the following command example:

```
$ sudo dd if=fsl-image-goldvip-s32g274ardb2.sdcard of=/dev/<sd-device> bs=1M conv=fsync,notrunc status=progress && sync
```

Ensure that any partitions on the card are properly unmounted before writing the card image, or you may have a corrupted card image in the end. Also ensure to properly "sync" the filesystem before ejecting the card to ensure all data has been written.

**Notes:**

- Builds with `bitbake` accumulate in the deployment directory. You may want to delete older irrelevant images after repeated builds.
- The first build will take a very long time because a lot of one-time housekeeping and building has to happen. You want to have a powerful build machine.
- `SOURCE_THIS` file has to be sourced when going back to build with a new shell.

19.2.5 Build and deploy OCI containers

By default, some GoldVIP applications run as containers built with the Yocto Project. The GoldVIP Yocto layer includes recipes to build these containers and pack them as OCI images in order to be used with any tool that can manage OCI containers (e.g., Docker, Podman, Kubernetes, K3s). To see the available GoldVIP container images and build them individually, run:

```
$ # Get the list of container images provided within GoldVIP layer
$ bitbake-layers show-recipes '*container-image*'  
$ # Build a specific container image
$ bitbake <container-image>
```

The OCI image will be available in `build_<machine>/tmp/deploy/images/` as a tar archive. The tarball can be copied on the target device for creating containers based on that image. Please check Container runtimes and Deploying custom applications in K3s cluster chapters for details on how to create containers starting from custom images.
20 Support

Our S32G GoldVIP community:


For technical support please visit:

- https://www.nxp.com/support

For contacting NXP please visit:

- https://www.nxp.com/about/about-nxp/about-nxp/contact-us:CONTACTUS
21 APPENDIX A: S32G-VNP-RDB2 connectors

The following figure shows the USB, UART, CAN and power connectors.

The following figure shows the Ethernet connectors.
The following figure shows the DIP switches position on the RDB2 board.
# APPENDIX C: Revision History

This table summarizes the changes done to this document since the initial release.

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0.1.0    | 11/2020 | - Cloud Gateway: telemetry use case based on AWS IoT Greengrass and AWS IoT SiteWise  
- Real-time bootloader running on Cortex-M7  
- Ethernet Gateway: slow path and fast path use cases  
- Docker image for host PC used for running Ethernet use cases |
| 0.1.1    | 12/2020 | - CAN Gateway: slow path  
- Improvements in Ethernet Gateway |
| 0.2.0    | 01/2021 | - Created AWS CloudFormation templates for rapid deployment of the Cloud Gateway telemetry use case  
- Ethernet Gateway: added PFE fast path support for all existing use cases |
| 0.3.0    | 02/2021 | - Ethernet Gateway: added Ethernet L3 forwarding slow path scenario with IDPS |
| 0.4.0    | 04/2021 | - Added CAN Gateway slow path with EB tresos AutoCore  
- The Cloud Gateway telemetry application is now deployed using AWS Serverless Application Repository |
| 0.5.0    | 04/2021 | - Adoption of Linux BSP 28.0  
- Adoption of PFE firmware 0.9.3  
- Adoption of RTD RTM 1.0.0  
- Cloud Gateway telemetry includes Cortex-M7 core load  
- Added ability to switch between GMAC and WiFi |
| 0.6.0    | 05/2021 | - GUI adoption  
- Implemented secure boot for bootloader image  
- Added XRDC configuration  
- CAN Gateway: added support for LLCE fast path routing |
| 0.7.0    | 06/2021 | - Added XEN hypervisor  
- Created 2 domains on the A53 cores: Dom0 and v2xdomu  
- CAN Gateway improvements |
| 0.8.0    | 07/2021 | - Separate root file system for each domain managed by the XEN hypervisor  
- Cloud Gateway and AWS IoT Greengrass have been moved to V2Xdomu  
- Changed HTTP port used by AWS IoT Greengrass |
| 0.9.0    | 08/2021 | - Adoption of Linux BSP 29.0 HF1  
- Adoption of PFE firmware 0.9.4  
- Added temperature statistics to the Cloud Gateway telemetry use case  
- GUI: added functionality and interface for Cloud Gateway use cases |
| 0.10.0   | 09/2021 | - Cloud Gateway: added telemetry data from the SJA1110 switch  
- Added general improvements for the GUI |
<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0.11.0   | 10/2021| • Added startup safety checks in Bootloader  
          • CAN Gateway Multicore application on M7 cores  
          • GoldVIP now includes Airbiquity’s OTAmatic software and data management platform for Over-the-Air Updates  
          • Adoption of various dependencies: Linux BSP 30.0, PFE firmware 1.0.0, PFE driver (M7) 0.9.5, LLCE firmware 1.0.2  
          • OTA Updates: added support for updating the real-time images and Linux Virtual Machines  
          • Cloud Gateway: the telemetry application was ported to AWS IoT Greengrass V2 |
| 0.12.0   | 12/2021| • Implemented Ethernet slow path over M7 core  
          • Ported M7 application to EB ACP 8.8.2  
          • Added Wi-fi support for v2xdomu  
          • Improved support for application images flashing  
          • DomU hostname has been changed to be easily identified  
          • OTA Updates: added out of the box support for OTA updates for Linux Virtual Machines and real-time images |
| 1.0.0    | 02/2022| • Adoption of Linux BSP 31.0  
          • Updated EB troses AutoCore to version 8.8.3  
          • Added IDPS over CAN (Cortex-M7 slow path)  
          • Added CAN to Ethernet use case, running on Cortex-M7  
          • Added IPsec slow path on A53 using strongSwan  
          • Added boot time measurement debug prints  
          • User experience improvement. A live dashboard informs the user regarding the download and/or installation progress  
          • Added ability to send custom application data from dom0 to AWS using the Cloud Gateway  
          • Updated AWS IoT Greengrass to version 2.5.3 |
| 1.1.0    | 05/2022| • Adoption of Linux BSP 32.0  
          • Adoption of ATF  
          • Added support for NXP M2 Wi-Fi dongle  
          • IPCF has been integrated, allowing for communication between A53 and M7  
          • Reduced .sdcard image size  
          • Added Secure Onboard Communication for CAN (running on Cortex-M7)  
          • Updated AWS dashboard.  
          • Added the transmission of IDPS statistics to cloud |
| 1.2.0    | 06/2022| • Added support for multi-PDU communication over CAN  
          • Added IDPS over CAN (LLCE fast path)  
          • Cryptographic offloading to HSE through the OpenSSL library  
          • Predictive maintenance using the eIQ Auto toolkit  
          • Container orchestration support using K3s  
          • Integration of AWS IoT FleetWise service  
          • V2X extensibility - facilitate the addition of client devices and AWS IoT SiteWise dashboards to the cloud gateway |
| 1.3.0    | 08/2022| • Battery Management System (BMS) use-case using the eIQ Auto toolkit  
          • Adoption of various new dependencies: Platform Integration 2022.06, HSE Firmware 1.0.5, eIQ Auto toolkit 3.4.0  
          • Telemetry data is also visible in a locally-deployed HTTP Server  
          • Updated AWS IoT Greengrass Core to version v2.6.0 |
## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.0</td>
<td>09/2022</td>
<td>• Removal of the CAN IDPS use cases</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Improvements to the GoldVIP Telemetry application</td>
</tr>
</tbody>
</table>
24 Trademarks

How to Reach Us:
Home Page: nxp.com
Web Support: nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTGX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltVec, C-5, CodeTest, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetics, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. ARM, AMBA, ARM Powered, Artisan, Cortex, Jazelle, Keil, SecurCore, Thumb, TrustZone, and μVision are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. ARM7, ARM9, ARM11, big.LITTLE, CoreLink, CoreSight, DesignStart, Mali, mbed, NEON, POP, Sensinode, Socrates, ULINK and Versatile are trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.
Contents

1 Introduction ......................................................... 2
  1.1 Overview ....................................................... 2
  1.2 High Level Architecture ..................................... 2
2 Hardware and Software Prerequisites .................... 3
  2.1 Hardware prerequisites ...................................... 3
  2.2 Software prerequisites ...................................... 4
3 Host system ........................................................ 5
  3.1 Hardware ....................................................... 5
  3.1.1 PC Ethernet Ports ......................................... 5
  3.2 Host Environment ........................................... 5
  3.2.1 Host system ................................................ 5
  3.2.2 Docker container ........................................ 5
4 GoldVIP Quick Start ............................................. 6
  4.1 Setup cable connections .................................... 6
  4.2 Deploy GoldVIP images .................................... 6
  4.2.1 Deploy GoldVIP SD-card image ....................... 6
  4.2.2 Deploy images to Flash .................................. 6
5 Docker ................................................................. 8
  5.1 Building GoldVIP Docker image ......................... 8
  5.2 Deploy GoldVIP Docker image ............................ 8
  5.3 Running the GoldVIP Graphical User Interface (GUI) in Docker container .......................... 9
6 Boot Flow ............................................................ 10
  6.1 Bootloader ..................................................... 10
7 Cloud Edge Gateway ........................................... 11
  7.1 Introduction .................................................. 11
  7.2 Prerequisites .................................................. 11
  7.3 AWS IAM Permissions ....................................... 12
  7.4 Supported Regions .......................................... 12
  7.5 Deployment of the Telemetry Stack in AWS ........ 12
  7.6 SJA1110 Telemetry Setup ................................... 13
  7.7 Connecting the board to AWS ............................ 13
  7.8 Accessing the AWS IoT SiteWise dashboard .......... 14
  7.9 GoldVIP telemetry dashboards ......................... 15
  7.10 Testing the Telemetry Application ..................... 15
  7.11 Deleting the Telemetry Application ................... 15
  7.12 Configure AWS IoT Greengrass after reboot ........ 15
  7.13 Sending custom data to AWS IoT MQTT client .... 16
8 Telemetry Server .................................................. 16
9 SJA1110 Telemetry Application ............................ 18
  9.1 Data flow ..................................................... 18
10 AWS IoT Fleetwise .............................................. 20
  10.1 Hardware setup ............................................ 20
  10.2 Preparing the use case .................................... 21
  10.3 Running the use case ..................................... 23
  10.4 Tips and tricks .............................................. 23
11 Ethernet Gateway .............................................. 25
  11.1 Prerequisites ................................................ 25
  11.2 Running the A53 slow-path use cases ............... 25
  11.3 Running the Cortex-M7 slow-path use cases ....... 25
  11.4 Running the PFE fast-path use cases ................. 26
  11.5 Running the SJA1110A fast-path use cases ........ 26
11.6 Running the IPsec A53 slow-path use cases .......... 26
11.7 Running the IDPS slow-path use cases ............... 27
11.8 Connecting to a Wi-Fi network ......................... 28
12 CAN Gateway .................................................... 29
  12.1 Prerequisites ................................................. 30
  12.2 Running cluster requirements ......................... 30
  12.3 Building the M7 Application ......................... 36
13 Over-the-Air (OTA) Updates ............................... 38
  13.1 OTA Software Management ............................ 38
  13.2 Prerequisites .............................................. 39
  13.3 Updates for Real-time images ......................... 39
  13.4 Updates for Linux Virtual Machines ................. 41
  13.5 Running the OTA updates use-case .................... 41
  13.6 Resetting the OTA update demo ....................... 43
  13.7 OTA applications as orchestrated containers ...... 44
14 Virtualization .................................................. 45
  14.1 Xen ......................................................... 45
  14.2 Networking in DomUs ..................................... 45
  14.3 Configuring the V2X domU ............................... 46
  14.4 Containers .................................................. 46
  14.5 OCI container images .................................... 47
  14.6 Container runtimes ....................................... 47
15 Container Orchestration ....................................... 49
  15.1 The K3s cluster ............................................ 49
  15.2 Deploying custom applications in K3s cluster .......... 51
16 IPCF Shared Memory ........................................ 54
  16.1 Running the use case .................................... 54
  16.2 Limitations ................................................ 54
17 Machine learning ............................................... 55
  17.1 Prerequisites ................................................ 55
  17.2 Predictive Maintenance use-case ..................... 56
  17.2.1 Running the demo .................................... 56
  17.2.2 Visualization ........................................... 56
  17.3 Battery Management System use-case ............... 57
  17.3.1 Running the demo .................................... 57
  17.3.2 Visualization ........................................... 57
18 Offloading cryptographic operations to HSE .......... 59
  18.1 Offloading cryptographic operations from Linux .......... 59
19 OpenEmbedded/Yocto project for GoldVIP ........... 61
  19.1 First Time Setup ........................................ 61
  19.2 Building GoldVIP .......................................... 61
  19.2.1 Download the Yocto project environment .......... 61
  19.2.2 Setup the build environment ....................... 62
  19.2.3 Build the image ....................................... 62
  19.2.4 Deploy the image ...................................... 63
  19.2.5 Build and deploy OCI containers .................. 63
20 Support ............................................................ 64
21 APPENDIX A: S32G-VNP-RDB2 connectors .......... 65

© NXP B.V. 2022. All rights reserved.
GoldVIP 1.4.0 — 28 September 2022
71 / 72