UM12007

S32G Vehicle Integration Platform (GoldVIP) User Manual for S32G3

Rev. 1.13.0 — 20 December 2024 User manual

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 S32G399A.

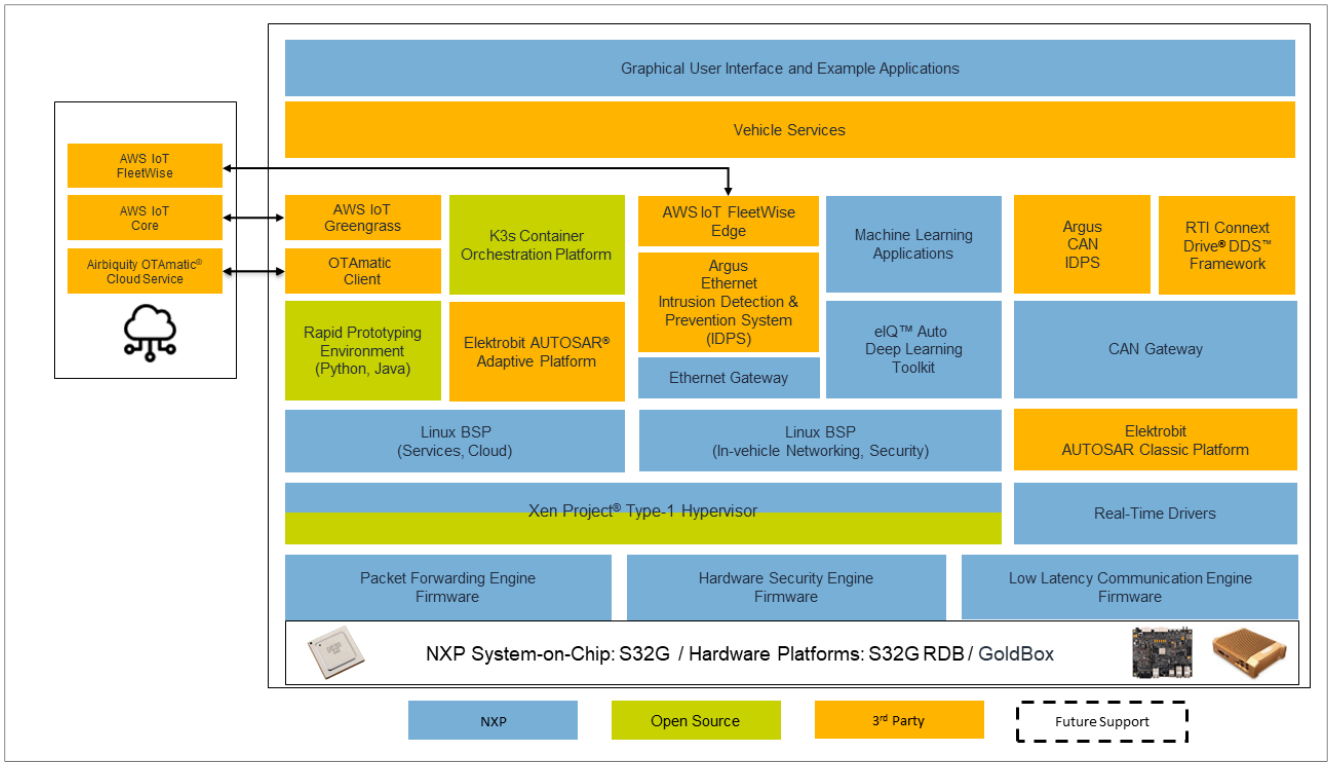
GoldVIP addresses key automotive market trends and use cases:

- 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:

Name	Location
S32G Reference Design Board 3 (RDB3)	https://www.nxp.com/S32G-VNP-RDB3
S32G GoldBox 3	https://www.nxp.com/GoldBox3
Wi-Fi M.2 module based on NXP 88W9098 SoC (optional)	https://www.nxp.com/products/wireless/wi-fi-plus-bluetooth/2-4-5-ghz-dual-band-2x2-wi-fi-6-802-11ax-plus-bluetooth-5-3:88W9098
Wiring harness for CAN	Delivered with the GoldBox
USB to Eth dongle (needed for ETH use cases)	N/A
USB to Wi-Fi dongle (optional for Wi-Fi connectivity)	N/A
SD-card of at least 8GB (16GB or more is recommended)	N/A

2.2 Software prerequisites

GoldVIP integrates the following software products:

Name	Version	Description
Linux BSP	43.0	Linux Board Support Package contains ARM Trusted Firmware and Linux.
Packet Forwarding Engine Firmware	1.10.0	The Packet Forwarding Engine (PFE) is a hardware based accelerator for classification and forwarding the Ethernet traffic.
Packet Forwarding Engine driver (Cortex-M7)*	1.5.0	Packet Forwarding Engine (PFE) Driver for ARM Cortex-M7 processors.
LLCE Firmware and Driver*	1.0.9	Low Latency Communication Engine (LLCE) firmware is intended to offload host applications regarding CAN routing and other time-consuming communication operations.
S32 SDK for SJA1110	1.0.0	The S32 Software Development Kit (S32 SDK) is an extensive suite of hardware abstraction layers, peripheral drivers, RTOS, stacks and middleware designed to simplify and accelerate application development on NXP SJA1110 microcontroller.
S32G Inter Platform Communication Framework	4.10.0	Inter-Platform Communication Framework (IPCF) is a subsystem which enables applications, running on multiple homogenous or heterogeneous processing cores to communicate over various transport interfaces (i.e., shared memory).
S32G Real-Time Drivers	5.0.0	Real-Time Drivers (RTD) is a set of AUTOSAR compliant device drivers responsible for hardware abstraction and enablement for the NXP targets.
S32G Real-Time security Crypto Driver	5.0.0 QLP01	The security Crypto Driver enables the offload of cryptographic operations to the HSE subsystem.
S32G Safety Software Framework	2.0.2 QLP01	S32 Safety Software Framework (SAF) is a software product containing software components for establishing the safety foundation for customer’s safety applications compliant with ISO 26262 functional safety.
Hardware Security Engine Firmware	0.2.51.0	The Hardware Security Engine (HSE) is a security subsystem, which aims at running relevant security functions for applications having stringent confidentiality and/or authenticity requirement.
AWS IoT Greengrass	2.12.6	AWS IoT Greengrass is an open-source edge runtime and cloud service for building, deploying, and managing device software.
AWS IoT FleetWise	1.1.1	AWS IoT FleetWise is a managed service that you can use to collect, transform, and transfer vehicle data to the cloud in near real time.
Elektrobit ACP for S32G3	8.8.7	EB tresos AutoCore lets you configure, validate, and generate the workspaces that are based on EB stack and Autosar OS.
RTI Connext Drive Evaluation	2.0.0	An automotive-grade communication framework built on the Data Distribution Service.
GoldVIP SJA1110 customization	1.13.0	SJA1110 firmware customization environment.

Name	Version	Description
GoldVIP LLCE customization	1.13.0	LLCE firmware customization environment.

Software is available here.

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

The following tools are required to rebuild or modify the GoldVIP applications:

Name	Version	Description
EB tresos Studio	27.1.0	EB tresos Studio lets you configure, validate, and generate your ECU basic software (BSW).
S32 Design Studio IDE	3.5	The NXP supported S32DS, an IDE that can be used to configure and build some of the GoldVIP applications.
S32 Design Studio 3.5 Development Package	3.5.3_D2306	S32 Design Studio 3.5 Development Package with support for S32G devices.
NXP GCC for ARM toolchain	9.2	This GCC (both 32 and 64-bit version) is bundled with S32 Design Studio.
GNU make	4.3	GNU make utility. Version 4.2.1 or higher is required. Download from https://www.gnu.org/software/make/.
Green Hills for ARM toolchain	2020.1.4	The third-party Green Hills supported toolchain. Required if some applications are built with the Green Hills toolchain. Download from GHS.
Python	3.8	High-level, object-oriented, interpreted programming language. Required to generate Bootloader dynamic boot configuration. Download from https://www.python.org/downloads.

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 20.04 or later.

4 Shared Resource Strategy

The NXP S32GXX is an SoC family representing a device that supports a heterogeneous compute architecture consisting of Cortex-M7 and Cortex-A53 cores. Although these cores are from different ARM core architectures (i.e., ARMv7-A and ARMv8-A respectively), both have access to the same on-chip peripherals and resources. Therefore, a strategy of device resource allocation is needed to ensure that all the processors coexist and share peripheral access coherently.

GoldVIP demonstrates a recommended strategy for device resource allocation and sharing. The next paragraphs summarize the general principles of working with the following resources:

- Security module
- Networking
- Memory

4.1 Security

The S32G399A device include a security engine (HSE) that has an important role during boot and also in run-time. In case of secure boot, the HSE core is booted first, so that it can authenticate and decrypt boot images for other cores consequently. In run-time, HSE provides security algorithms to individual applications. HSE has four different Messaging Units (MUs) that enables one CPU core and the HSE within the SoC to communicate and coordinate by passing messages (e.g., data, status and control) through the MU interface.

The default allocation of the available HSE MUs employed in GoldVIP is exhibited in the following table:

Domain	Description
Real-time domain (Cortex-M7 cores)	Access to HSE_MU2 and HSE_MU3 ports.
Linux domain (Cortex-A53 cores)	Access to HSE_MU0 and HSE_MU1 ports.

4.2 Memory mapping

The following table describes how the QSPI NOR Flash and SRAM memories are partitioned in GoldVIP:

Module	Start address	End address	Description
QSPI NOR Flash	0x00000000	0x000000ff	Image Vector Table (IVT) Image
	0x00000100	0x000001ff	Device Configuration Data (DCD) Information
	0x00000200	0x000003ff	QuadSPI Parameters
	0x00000400	0x000c3fff	HSE Firmware images
	0x000c4000	0x000cefff	HSE System Image
	0x000cf000	0x001effff	GoldVIP Real-time Bootloader image (Slot A)
	0x001f0000	0x001fffff	GoldVIP Bootloader dynamic configuration
	0x00200000	0x003fffff	ARM Trusted Firmware binary image
	0x00400000	0x007fffff	GoldVIP Real-time Gateway application image (Slot A)
	0x00800000	0x008fffff	GoldVIP Real-time Bootloader image (Slot B)
	0x00b00000	0x00efffff	GoldVIP Real-time Gateway application image (Slot B)
SRAM	0x34000000	0x343fffff	GoldVIP Real-time Gateway application address space, used by the Cortex-M7_0 and Cortex-M7_1 cores

Module	Start address	End address	Description
	0x34400000	0x344fffff	PFE reserved memory (BMU and RT), used by PFE
	0x34500000	0x3451ffff	Shared memory reserved for the Shared Resource Manager mailboxes, used by the Cortex-M7 and Cortex-A53 cores (used when the SRM functionality is enabled in the images)
	0x34520000	0x345fffff	Shared memory used for Inter-Platform Communication, used by the Cortex-M7 and Cortex-A53 cores
	0x34600000	0x346fffff	ARM Trusted Firmware address space, used by the Cortex-A53 cores
	0x35300000	0x353fffff	GoldVIP Real-time Bootloader application address space, used by Cortex-M7 core

4.3 Networking

The S32G3 SoC contains two ethernet controllers:

- **GMAC**
- **PFE** - an Ethernet accelerator with three EMACs (PFE_EMAC_0, PFE_EMAC_1, PFE_EMAC_2)

These controllers can be programmed to use MII/RMII/RGMII and SGMII interface modes supporting 10/100/1000/2500 Mbit/s speeds.

The S32G399A RDB3 board integrates the aforementioned ethernet controller, including the NXP Automotive TSN/AVB SJA1110A SGMII switch. The available sockets are:

- 1x2500Base-T with AQR113 PHY connected to PFE_EMAC_1
- 2x1000Base-T with AR8035 PHYs connected to the SJA1110A switch
- 1000Base-T with AR8035 PHY connected to the RGMII PFE_EMAC_2
- 1000Base-T with KSZ9031 PHY connected to GMAC
- 100Base-TX directly connected to the SJA1110 switch
- 6x100Base-T1 directly connected to the SJA1110 switch

The following port mapping is used:

SoC port	Interface	Board port	Linux interface
PFE_EMAC_0	SGMII(2.5 Gbps) to switch SJA1110A	P2A and P2B	pfe0sl
PFE_EMAC_1	N/A	N/A	pfe1sl
PFE_EMAC_2	RGMII	P3B	pfe2sl
GMAC	RGMII	P3A	eth0

4.3.1 GMAC

The GMAC port is used exclusively by the Linux (Cortex-A53) Domain. In the default GoldVIP image, the GMAC port is shared between the Dom0 and v2xdomu VMs through a bridge, and the corresponding interfaces are `xenbr0`, and `eth0` respectively.

4.3.2 PFE

GoldVIP makes use of the PFE Master-Slave feature to share the PFE provided connectivity using dedicated host interfaces. In the default GoldVIP image, the Real-time domain acts as the master of the PFE subsystem, while the Linux domain integrates a PFE Slave instance as part of the dom0 VM.

The following Host Interface(HIF) allocation is used:

HIF	Owner	Description
HIF_NOCPY	N/A	N/A
HIF_0	Real-time / Cortex-M7 cluster	Used as the master channel
HIF1	LLCE cores	Used for CAN2Eth scenarios
HIF2	N/A	N/A
HIF3	Linux / Cortex-A53 cluster	Used by the slave instance

5 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.

5.1 Setup cable connections

To locate the connectors and ports from the board check the figures from Appendix A or check the S32G-VNP-RDB3 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.

5.2 Deploy GoldVIP images

Download GoldVIP binary images from here.

5.2.1 Deploy GoldVIP SD-card image

1. Write the GoldVIP .sdcard image (`fsl-image-goldvip-s32g399ardb3.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 the fsl-image-goldvip-s32g399ardb3.sdcard on SD-card plugged into the Linux host PC, e.g.:

```
$ sudo dd if=fsl-image-goldvip-s32g399ardb3.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 RDB3 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.

5.2.2 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 dynamic boot configuration:

```
run write_bootconfig
```

3. Update Arm Trusted Firmware image:

```
run write_atf
```

4. 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 J189 to 1-2 shorted and 3-4 shorted position in order to have the SJA telemetry application loaded to the board by Linux.

6 Docker

6.1 Building GoldVIP Docker image

In order to run some of the GoldVIP use cases, the GoldVIP Docker container image is required. The container image recipe is provided in the layout under the `docker` directory. The Docker image can be obtained by following the next steps:

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 a Linux command line terminal and run:

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

   Note: The fetch for some packages might fail if your network uses an HTTP proxy. In such cases, 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:1.13.0`. 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 container image, please see the official documentation: https://docs.docker.com.

6.2 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:

```
$ sudo docker run -it --rm --name goldvip       \
     --network=host --privileged                \
     -v $HOME/.Xauthority:/home/vip/.Xauthority \
     -v /lib/modules:/lib/modules:ro            \
     -e DISPLAY=$DISPLAY                        \
     -e XAUTHORITY=/home/vip/.Xauthority        \
     docker-goldvip:1.13.0
```

   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. Access the board's Linux console, from the Docker container:

```
$ sudo minicom -D /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
```

3. Login with `root` account (no password required).

Now you can start running the use cases detailed in each gateway chapter.

6.3 Running the GoldVIP Graphical User Interface (GUI) in Docker container

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

1. Start the nodejs service to run GUI application:

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

2. Start a browser of your choosing on your Linux host 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.

Notes:

- The board must have an ethernet connection in the same network as the host machine.
- Please go to the browser settings to enable *Pop-ups and redirects*. For example, one can follow this guide to enable the pop-ups for Google Chrome browser. The steps might be different depending on the used browser.
- If the AWS credentials are exported as environment variables (`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`) in the terminal of the GoldVIP Docker container, there is no need to fill in the corresponding textboxes for the GoldVIP Cloud Telemetry use case.

7 Boot Flow

7.1 Bootloader

The GoldVIP makes use of a System-Level 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.

Note: The Secure Boot flow enabled in GoldVIP covers only the authentication of the bootloader image, Boot Table (optional, if used), and the images loaded by the bootloader component. Therefore, by default, the TF-A BL3x images are not authenticated before being loaded. To enable the TF-A BL3X images authentications, one has to enable the secure boot feature when building the desired GoldVIP distribution with the Yocto Project. Please see Setup the build environment chapter for more details on enabling System-Level Secure Boot.

The Bootloader tresos workspace and source code can be found in the GoldVIP installer in the `<GoldVIP_install_path>/realtime/s32g399ardb3/bootloader` directory. More details on the Bootloader functionality can be found in the `Bootloader User Manual` available in the `<GoldVIP_install_path>/documentation` folder.

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:

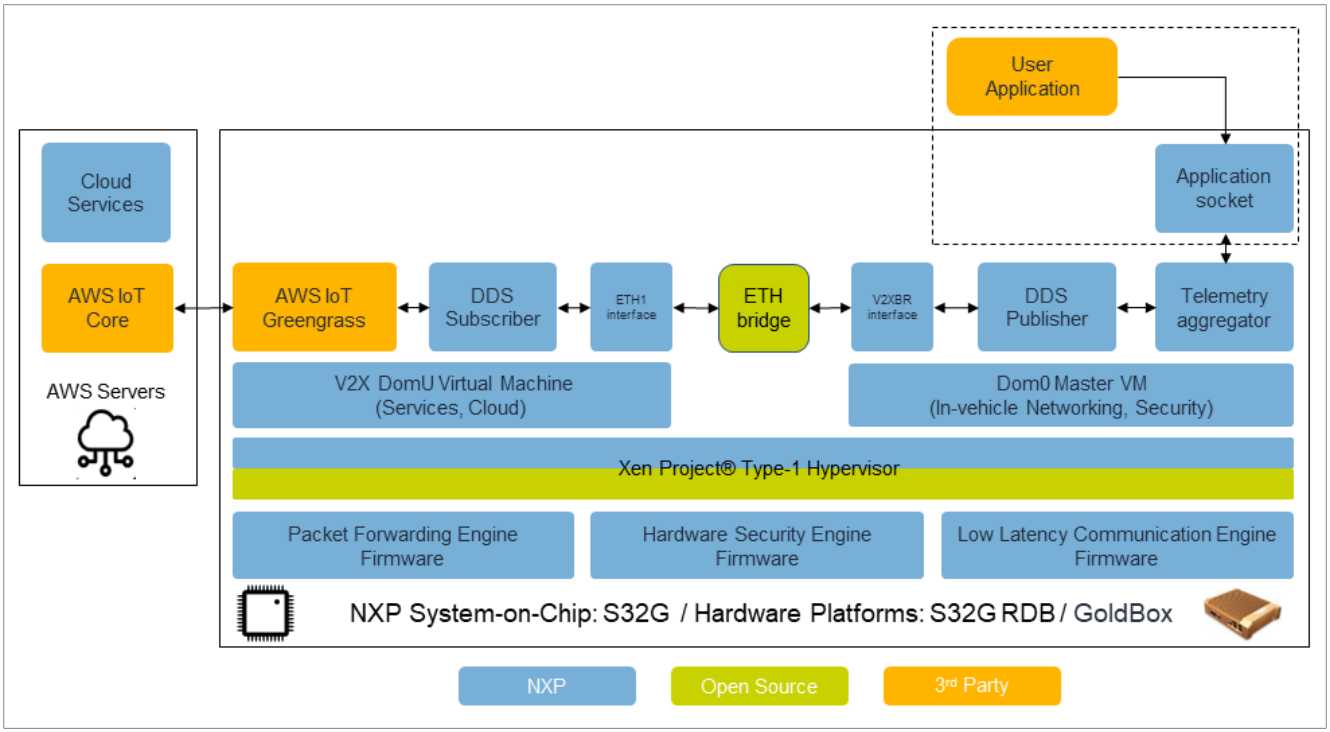
1. Load the Bootloader EB tresos project into workspace, and for all the missing plugins choose *Remove all faulty Modules*;
2. Disable the *Safety Bootloader Enable* field in the Bootloader plugin configuration;
3. Go to the Bootloader workspace and delete the *output* folder;
4. Generate and compile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

8 Cloud Edge Gateway

8.1 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 Section 17.1 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 Section 10 publisher-subscriber 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. 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.

Note: When using the `fsl-goldvip-no-hv` Yocto distro, all the resources mentioned in this chapter which are either in Domain-0 or v2xdomu will be in the same standalone Linux machine.

8.2 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.

8.3 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": "*"
        }
    ]
}
```

Please check the AWS IAM User Guide for steps on how to set this policy.

Note: The examples in this document are intended only for development environments. All devices in your production fleet must have credentials with privileges that authorize only intended actions on specific resources. The specific permission policies can vary for your use case. Identify the permission policies that best meet your bussiness and security requirements. For more information refer to Example policies and Security best practices.

8.4 Supported Regions

The Telemetry Stack is limited to the AWS regions that support AWS IoT SiteWise. The list of supported AWS regions can be consulted here

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

8.5 Deployment of the Telemetry Stack in AWS

1. Go to the AWS Serverless Application Repository (SAR) console: https://console.aws.amazon.com/serverlessrepo/.
2. Go to `Available applications` tab, then select `Public applications` and search for `nxp-goldvip-telemetry`.

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 name of the application in the previous steps, 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:

- An AWS IoT Greengrass V2 telemetry component, this is a python function that runs on v2xdomu and sends data to AWS IoT Core. The provisioning script described in chapter Section 8.8 creates an AWS IoT Greengrass V2 continuous deployment which will run the telemetry component on your board.
- An 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.

8.6 SJA1110 Telemetry Setup

Steps needed to enable SJA1110 telemetry data:

1. Connect the SJA1110 to the internet using the P4 ethernet port on the board (check Section 27). 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 J189 is set to 1-2 shorted and 3-4 shorted. If not, set it to 1-2 shorted, 3-4 shorted and reboot the board.
4. Run the provisioning script (described in chapter Section 8.8) 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 Section 13.5 cannot be used any longer.
- Setting J189 to 1-2 open and 3-4 shorted 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 Section 8.13.

The Section 11 chapter contains more details about the SJA1110 application.

8.7 OP-TEE RPMB secure storage for AWS Certificates

In the SJA1110 Telemetry Setup certificates for the client device are downloaded from the AWS cloud and stored for subsequent provisionings. These certificates can be stored securely using OP-TEEs Replay-Protected Memory Block (RPMB) secure storage. To use RPMB run the SJA1110 setup as described in Section 8.6, but also add the `--use-rpmb` option to the provisioning script.

The Section 18 chapter contains more details about OP-TEE.

Note: The showcased RPMB secure storage functionality is for demo purposes only, it should not be used in production. The certificates stored in RPMB secure storage are availabile in plain text in the Linux normal world prior to their storage, and when they are transmitted to the client devices.

8.8 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:
3. From the v2xdomu console, set the AWS credentials as environment variables:

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

Please check the AWS documentation for additional information on how to retrieve the AWS credentials: https://docs.aws.amazon.com/singlesignon/latest/userguide/howtogetcredentials.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
```

- `<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.
- `<region-name>` is the region in which you have deployed the Telemetry Stack.
- `--setup-devices` is used to start the SJA1110 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.

Notes:

- AWS IoT Greengrass uses ports `8883` and `8443` by default. 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 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.
- 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.
- The network configuration and time are not persistent between reboots. Please check Section 8.13 for further information.

8.9 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 data.

You will now see the live telemetry data from your board.

8.10 GoldVIP Telemetry dashboards

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

1. **SoC Dashboard**: contains widgets dedicated to SoC specific statistics: core loads of the virtual CPUs running in Domain-0, core loads of all Cortex-M7 MCUs, PFE traffic (both RX and TX), SoC temperature sensors values, Domain-0 memory load, and Health Monitoring Voltages.
2. **IDPS Dashboard**: contains dedicated widgets for IDPS anomalies detected. Currently, only anomalies detected by the CAN IDPS running on Cortex-M7 are available. In case an anomaly is detected by the IDPS engine, data is published by the Cortex-M7 application via IPCF. Statistics are then collected and calculated to be sent to the cloud counterpart of the application.
3. **SJA Dashboard 1-2**: contains widgets containing packet statistics per each of the SJA ports available on the S32G.
4. **Machine Learning Dashboard**: contains widgets dedicated to the predictive maintenance application, it shows the prediction results and execution durations.
   Note: The Machine Learning Dashboard will not receive any data because the Machine Learning use-cases are disabled in this release.
5. **GreenVIP Dashboard**: contains widgets with CPU loads reported by GreenVIP.
6. **XS32K3X4EVB-T172 Dashboard**: contains widgets with data from an S32K3 board running the AWS Greengrass Demo.

8.11 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 12.5% on the core load shall be observed in the `Core Loads Dashboard`, per each of the started processes.
   - Kill all the CPU-intensive processes:

     ```
     $ killall dd
     ```

8.12 Deleting the Telemetry Application

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

Note: Upon stack deletion, a new deployment will be created in order to remove any previously deployed components on the device. This deployment will be applied if the device is online during the 3 minutes time frame after the stack deletion is issued. The stack will be deleted regardless of whether the deployment is deployed successfully on the device (i.e., deployment status is "COMPLETED") or not.

8.13 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
```

Note: When using the `fsl-goldvip-no-hv` Yocto distro, skip this step.

- 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, and 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
```

8.14 Sending custom data to AWS IoT MQTT client

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

To send data from your custom application, connect with a socket to `localhost:51001` on Domain-0 virtual machine and send a message with the following format:

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

Where `payload` is a dictionary, or a list of dictionaries. `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.

To see your data in the cloud you must first subscribe to the corresponding topic in the AWS IoT Core console:

1. In the AWS IoT Core console select `Test` from the navigation pane.
2. Choose `MQTT test client`.
3. Choose `Subscribe to a topic`.
4. Enter the topic `<TelemetryTopic>/example/topic/suffix` (you can use the # wildcard to see all topics or see a subset of topics, e.g., `<TelemetryTopic>/#`)
5. Choose `Subscribe`. You should see messages being displayed as they are received.

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

```
$ echo '{
    "app_data": {"payload key": "payload value"},
    "mqtt_topic_suffix": "example/topic/suffix"
  }' | nc -c -v 127.0.0.1 51001
```

9 Telemetry Server

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`, `temperature`, `monitored voltages`, and `IDPS Anomalies` (if the IDPS use case is started). It is an alternative for the AWS IoT SiteWise dashboards created by the GoldVIP Telemetry application.

The telemetry server starts automatically when the v2xdomu VM boots up. In order to access the local telemetry dashboard, the board 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.

10 Data Distribution Service

10.1 General concepts

The Data Distribution Service™ (DDS) is an open standard that defines a middleware protocol and API for data-centric communication using the publisher-subscriber pattern of communication, and it addresses real-time and embedded systems. The DDS publish-subscribe model aims to eliminate the complexity of data communication inside a distributed network of either homogenous or heterogenous nodes. It defines concepts such as nodes (publisher, subscriber or even both roles in the same node) and topics (an object that describes the data).

DDS handles the dynamic discovery protocol which means that the application does not have to know the location of the other endpoints because it can be handled automatically by DDS at runtime.

The DDS Quality of Service (QoS) defines a rich set of parameters which allows the developer to fine tune the performance of the application. Examples of QoS parameters are: reliability, history depth, durability, deadline, resource limits, etc.

For more information on the DDS standard, please visit: https://www.rti.com/products/dds-standard

10.2 RTI Connext Drive

RTI Connext Drive® is the data-centric connectivity framework that includes a suite of safety-certified components and a production-proven architecture that delivers the performance required for software defined vehicles (SDVs), while enabling the co-development and collaboration across multipleplatforms. Based on the DDS standard, Connext Drive enables native development on a DDS base and/or through direct connectivity within ROS 2 and AUTOSAR Classic and Adaptive environments. In addition, Connext Drive supports telematics applications through a UDP-based Real-Time WAN Transport that enables low latency and high throughput communications. It provides flexibility, scalability, compatibility and upgradability while also fulfilling key technical and safety requirements.

For a specific hands-on example of how RTI Connext works, please see Next Generation E/E. Additional information regarding the RTI Connext Drive product is located here. To request the full feature set evaluation or for specific questions, please contact RTI (ConnextDriveNXP@rti.com).

Note: When using the `fsl-goldvip-no-hv` Yocto distro, both the publisher and subscriber reside on the same machine and they communicate through shared memory.

10.3 RTI Connector for Python

RTI Connector provides a quick and easy way to write applications that publish and subscribe to the RTI Connext Drive databus in Python and other languages. In GoldVIP, RTI Connector for Python is used to exchange telemetry information between Domain-0 and v2xdomu using v2xbr interface. The publisher that resides on Domain-0 gathers the data from various sensors (e.g., temperature), and statistics (e.g., cores load), packs it and then sends it to the two subscribers, both of which are running on v2xdomu:

1. The first one runs inside Telemetry Server
2. The second one runs inside the Telemetry Component from Cloud Edge Gateway

RTI Connector is free for non-commercial use and can be installed on supported platforms using the `pip` Python package installer. For more information, click here.

10.4 RTI Connext Micro

RTI Connext Micro is a flavour of Connext DDS which address resource-constrained (memory and CPU) nodes in a DDS Bus. In GoldVIP demo, DDS Micro runs in M7 domain where the application that is created on top of it acts as a gateway between AUTOSAR Com module and DDS. The demo provided collects CAN signals that

are related to a virtual car lights subsystem, it repacks them in DDS topics that are related to a specific exterior light module (Headlamp, Taillight, Stoplamp, LicensePlateLight and HazardLights) and publishes the data on the DDS bus. The data for each light subsystem is either a unsigned value which designates the intensity from a scale of 0 (light turned off) to 100 (light at full intensity) or a boolean value for the on-off status of the light.

In order to run the demo follow this steps:

1. Enable the M7-A53 PFE bridge, follow the steps from Section 13.9 (for Linux aux0 interface IP address use the one listed in the instructions).
2. Connect FlexCAN0 to LLCE-CAN0.
3. In order to have some data that isn't zeroed all the time, a CAN dump file is provided. Replay the file using the following command: `$ canplayer -l i -I ~/dds/rti_dds_lights.pcap &`. The `-l i` option will play the file in a infinite loop. In order to kill the canplayer process you can the following command `$ pkill -9 canplayer`.
4. Run the python script file, `$ python3 ~/dds/rti_dds_lights.py`, which creates a DDS participant that subscribes to all the light topics and prints the received data. In order to stop the process you have to issue the `SIGINT` signal using `CTRL+C` command.

11 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 running 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 can be 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, AWS IoT Thing name, the MQTT topic, v2xdomu virtual machine IP address, the certificates of the SJA1110 IoT 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.

11.1 Data flow

The SJA1110 telemetry data is sent to the cloud in a separate JSON payload than the other cloud telemetry.

The data is published on the `s32g/sja/switch/<stack-name>` MQTT topic, and has the following payload structure:

```
{
    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 RDB3), 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 data will appear in three AWS IoT Sitewise Dashboards named `SJA1110 Dashboard 1/2/3`.

11.2 Building the SJA1110 Telemetry application

GoldVIP makes use of custom SJA1110 firmware binaries that enables the functionality needed for sending telemetry data to the cloud. In order to get the same SJA1110 functionality as in the distributed binary images:

1. Download and install the GoldVIP SJA1110 customization and SJA1110 SDK version mentioned in Software prerequisites.
2. Install the GCC for ARM toolchain referenced in the Software prerequisites chapter.
3. Open a Command Prompt and run the following commands to build the SJA1110 firmware:

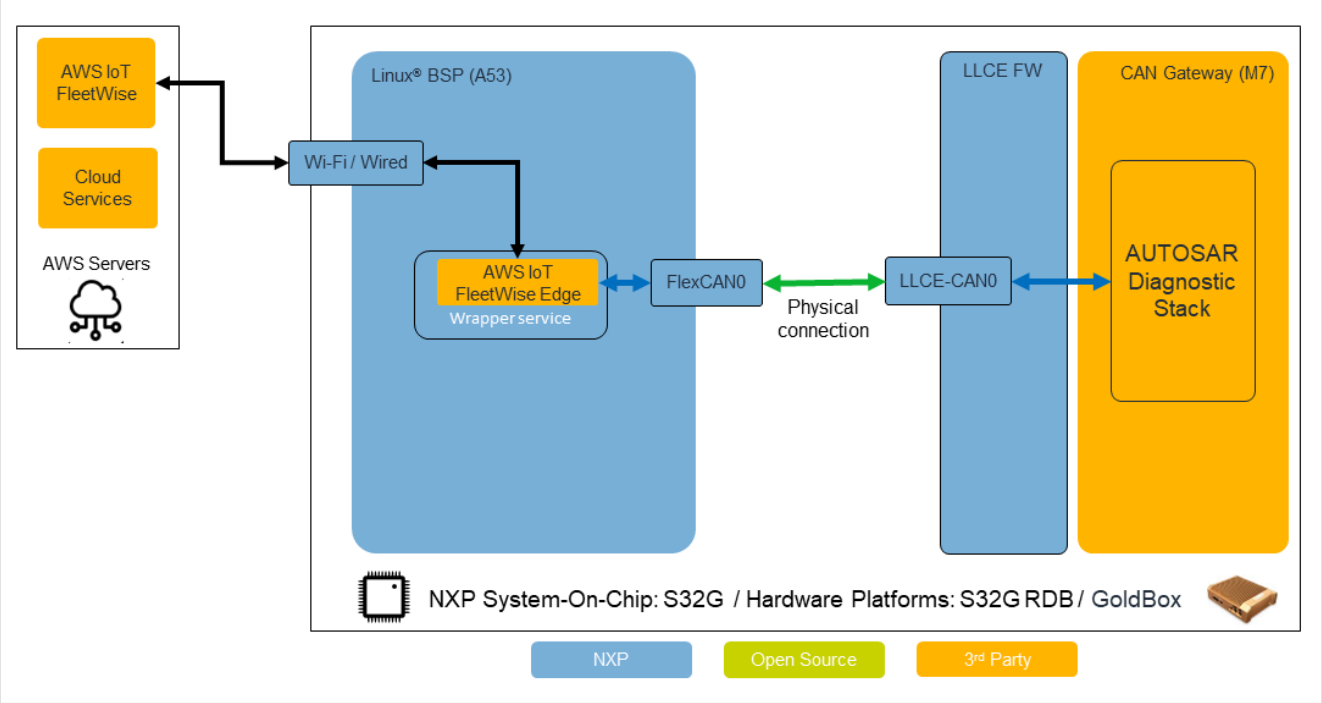
```
$ make build \
CWD=<SJA1110CSTM_install_path>/sja_telemetry/Build/ \
DEPENDENCY_PATH=<SJA1110_SDK_install_path> \
TOOLPATH_COMPILER=C:/NXP/S32DS.3.5/S32DS/build_tools/gcc_v9.2/gcc-9.2-arm32-eabi/arm-none-eabi/bin/
```

   Note: The GoldVIP SJA1110 Telemetry application can also be build using the S32 Design Studio v3.5 by importing the project from `<GoldVIP_SJACSTM_install_path>/sja_telemetry` and then using the *Build Project* option from the IDE interface.
4. After building, the firmware `sja1110_uc.bin` and the static configuration `sja1110_switch.bin` will be available in `<GoldVIP_SJA1110CSTM_install_path>/sja_telemetry/Build/`.
   Copy these files to the board at `/lib/firmware` on the dom0 VM in order to load the SJA1110 FW on the switch at the next boot.
   Note: J189 must be set to shorted-shorted in order to load the custom SJA firmware.

12 AWS IoT FleetWise

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/v1.1.1

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:

- The *GoldVIP Real-Time application*, which runs a AUTOSAR Diagnostic stack to answer on CAN0 interface to OBD2 requests sent by *AWS FleetWise Edge Agent*. The application responds to OBD2 Services `$01`, `$03` and `$09`.
- *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 received by the diagnostic stack running on *GoldVIP Real-Time application*. The application will reply with precomputed data for the following Parameter IDs (PID): Engine Speed, Vehicle Speed, Fuel Tank Level, Ambient Air Temperature, Engine Coolant Temperature and Throttle Position. The GoldVIP application also sets Diagnostic Trouble Code (DTC) number `0x0123` which will be seen as set when *AWS IoT FleetWise Edge Agent* queries for which DTCs are set using OBD2 Service `$03` (show stored DTCs).
- *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.

Note: When using the `fsl-goldvip-no-hv` Yocto distro, all the references to the `xenbr0` network interface in the instructions shall be replaced with `eth0`.

12.1 Hardware setup

In order to connect the CAN gateway to the AWS IoT FleetWise Edge Agent, perform the following steps:

1. Connect the FlexCAN0 to LLCE-CAN0.
2. Connect the S32G board to the internet via a wired or Wi-Fi connection of your choice. By default, the following steps assume that a connection is active on the GMAC port (P3A).
3. Deploy the GoldVIP Real-Time images, and set the board to boot from QSPI. The OBD requests generated by the AWS IoT FleetWise Edge Agent will be answered by the GoldVIP Real-Time application.

12.2 Preparing the use case

Several steps need to be performed before running the use case.

1. **Prepare the workspace:**
   a. Open the AWS CloudShell: **Launch CloudShell**.
   b. Obtain the AWS IoT FleetWise Edge Agent software from GitHub:

```
$ git clone --branch v1.1.1 \
https://github.com/aws/aws-iot-fleetwise-edge.git \
~/aws-iot-fleetwise-edge
```

2. **Provision AWS IoT Credentials:**
   a. Create an empty directory where the FWE configuration data will be temporarily stored:

```
$ mkdir ~/goldvip-fwe-deploy
```

   b. Create an AWS IoT Thing, and provision the credentials for it, using the tools provided in the FWE software repository:

```
$ ~/aws-iot-fleetwise-edge/tools/provision.sh \
--region us-east-1 \
--vehicle-name fwdemo-s32g \
--private-key-outfile ~/goldvip-fwe-deploy/private-key.key \
--endpoint-url-outfile ~/goldvip-fwe-deploy/endpoint.txt \
--vehicle-name-outfile ~/goldvip-fwe-deploy/vehicle-name.txt \
--certificate-pem-outfile ~/goldvip-fwe-deploy/certificate.pem
```

   c. Create the configuration files that will be used by the AWS IoT FleetWise Edge Agent:

```
$ ~/aws-iot-fleetwise-edge/tools/configure-fwe.sh \
--input-config-file ~/aws-iot-fleetwise-edge/configuration/static-\
config.json \
--vehicle-name "$(cat ~/goldvip-fwe-deploy/vehicle-name.txt)" \
--endpoint-url "$(cat ~/goldvip-fwe-deploy/endpoint.txt)" \
--can-bus0 can0 \
--output-config-file ~/goldvip-fwe-deploy/goldvip-config.json
```

   d. The generated certificates and configuration files must be deployed on the S32G board. For this purpose, create an archive with the contents of the folder to facilitate the deployment:

```
$ cd ~/goldvip-fwe-deploy && tar -cf deploy.tar *
```

3. **Deploy the configuration on the S32G board:**
   a. Download this archive on your **local machine**: in the AWS CloudShell console click on the `Actions` dropdown button, select the `Download file` action, then fill in the filename of the previously created archive in the displayed text box, and choose `Download`. If the default paths were used, that would be `~/goldvip-fwe-deploy/deploy.tar`.
   b. Copy the tarball to the S32G board, in the root file system of the Dom0 VM. One can use the `scp` (Secure Copy) utility to accomplish this. Open a shell on your **local machine** and run the following:

```
$ scp deploy.tar root@<xenbr0_ip>:/tmp
```

After the transfer is done, the deployment archive can be found on the Dom0 root file system in the `/tmp` directory.
Note: The board must be connected to internet via the GMAC Port (P3A). To find the IP address of the `xenbr0` network interface, run the following in the console of the Dom0 VM:

```
$ ip address show xenbr0
```

Note: On Windows systems, one can use the `PSCP` (PuTTY Secure Copy) client, if the `scp` utility is not available.

c. On the Dom0 VM, run the deployment script that helps with the set up of the AWS IoT FleetWise Edge Agent:

```
$ python3 ~/fleetwise/deploy.py --netif xenbr0 \
--input-archive /tmp/deploy.tar
```

This script will check if the network interface is connected to the internet, will syncronize the sytem date and time using `ntp`, and then it 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 automatically at each boot.
If the `--input-archive` parameter is ommited, this script simply restarts the AWS IoT FleetWise Edge Agent:

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

12.3 Use the AWS IoT FleetWise demo

The following steps will register your AWS account for AWS IoT FleetWise, create a demonstration vehicle model, register the virtual vehicle created in the previous steps and run a campaign to collect data from it. From the AWS ClouShell console, run the following:

1. Install the dependencies for the demo script:

```
$ pip install --force-reinstall -v numpy==1.26.4 && \
sudo -H ~/aws-iot-fleetwise-edge/tools/cloud/install-deps.sh
```

2. Run the demo script:

```
$ cd  ~/aws-iot-fleetwise-edge/tools/cloud && \
./demo.sh --vehicle-name fwdemo-s32g \
--region us-east-1 \
--campaign-file campaign-obd-heartbeat.json
```

Note: If you changed the `vehicle-name` parameter in the previous steps, the option passed to the `demo.sh` script must be updated accordingly.
This script will create the required resources for this demo in the cloud, and will trigger a collection campaign that will capture the OBD responses from the GoldVIP Real-Time Application. This will deploy a *heartbeat* campaign in the cloud which will periodically collect OBD data.

3. An HTML file will be generated after a successful run of the script. One can visualize the results on his local machine by copying the generated HTML file locally and opening it in a web browser. In the AWS CloudShell window, copy the path to the HTML file, then click on the `Actions` dropdown menu, and choose the `Download file` action. Paste the path in the text box, then click on `Download`. The HTML file can be opened in your preferred browser.

12.4 Clean up the resources

Most of the resources created by the `provision.sh` and `demo.sh` scripts can be deleted using the following command in the AWS CloudShell console:

```
$ cd ~/aws-iot-fleetwise-edge/tools/cloud && \
./clean-up.sh
$ ~/aws-iot-fleetwise-edge/tools/provision.sh \
--vehicle-name fwdemo-s32g \
--region us-east-1 \
--only-clean-up
```

12.5 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 redeploy the configuration, you will need to run the AWS IoT 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, if the AWS IoT FleetWise Edge Agent is running and connected to the cloud:

   ```
   $ candump can0
   ```

   In case the setup was done correctly, one shall observe CAN traffic with IDs `>7DF>`, `7E0`, and `7E8` when running the command, as in the following log:

   ```
   can0  7DF   [8]  02 01 00 00 00 00 00 00 (request from the Edge Agent)
   can0  7E8   [8]  06 41 00 1F FB 80 03 00 (response from CAN gateway)
   can0  7DF   [8]  07 01 00 20 40 60 80 A0 (request from the Edge Agent)
   can0  7E8   [8]  10 10 41 00 1F FB 80 03 (response from CAN gateway)
   can0  7E0   [8]  30 00 00 00 00 00 00 00 (request from the Edge Agent)
   can0  7E8   [8]  21 20 80 02 80 01 40 4C (response from CAN gateway)
   can0  7E8   [8]  22 00 00 70 00 00 00 00 (response from CAN gateway)
   ...
   ```

   In case no traffic is printed when running the `candump` command, this means that the configuration deployed to the board is invalid or the Cortex-M7 core is stopped.

3. The logs for the AWS IoT FleetWise Edge Agent can be found in the rootfs of Dom0 VM at `/var/log/fleetwise.log`.

13 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 (and a python script 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.

Note: The custom SJA1110 switch firmware delivered in the GoldVIP image must be used to exercise any of the use cases listed below.

13.1 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.
- J189 set to shorted-shorted in order to load the custom SJA1110 switch firmware during the boot.

13.2 Running the A53 slow-path use cases

1. Connect one host PC ETH port to the board's SJA1110A 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 adapter 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 the `eth-slow-path-host.sh` script to measure performance for L3 forwarding between `pfe0` and `pfe2`, with UDP or TCP traffic, and 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.

Notes:

- Run `ip a` command on your host PC to find out the exact names of the interfaces `<eth0>` and `<eth1>` connected to the board.
- The script is connecting to target console via `/dev/ttyUSB0`. In case the tty port is different on your PC, specify it explicitly with `-u` argument (e.g., `-u /dev/ttyUSB1`).
- The Layer 2 ETH forwarding scenario makes use of VLANs. VLAN support is needed on the host PC.

13.3 Running the Cortex-M7 slow-path use cases

1. Make sure that J189 is set to 1-2 shorted and 3-4 shorted. If not, set it to 1-2 shorted, 3-4 shorted 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 the `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`) and default packet size (ETH MTU). Use `-h` option to list all available options.
Same notes apply as in previous section.

13.4 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 the `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.

13.5 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 the `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.

13.6 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 the `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`) and default packet size (ETH MTU). Use `-h` option to list all available options.
Same notes apply as in previous section.

13.7 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 the `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.

13.8 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 apropriate 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 SW17 DIP switch to ON-ON-OFF-OFF. The position of this switch can be found in 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 the SSID and passphrase of your Wi-Fi network to the `/etc/wpa_supplicant.conf` file:
   - 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={\n\tssid="<SSID>"\n\tkey_mgmt=NONE\n}" \
    >> /etc/wpa_supplicant.conf
```

6. Restart Wi-Fi service:

```
$ /etc/init.d/wifi_service restart
```

13.9 Ethernet bridge between M7 and A53

The M7 and A53 can use their respective ethernet stacks to communicate through PFE using their AUX interface. In order to create the L2 bridge you have to run the following LibFCI commands:

```
$ libfci_cli bd-update --vlan=1 --uh=FORWARD --um=FLOOD \
  --mh=FORWARD --mm=FLOOD
$ libfci_cli bd-insif --vlan=1 -i hif0 --tag=OFF
$ libfci_cli bd-insif --vlan=1 -i hif3 --tag=OFF
$ libfci_cli phyif-update -i hif0 --mode=VLAN_BRIDGE
$ libfci_cli phyif-update -i hif3 --mode=VLAN_BRIDGE
```

In order to test the network connection with the M7 application you have to assign to the `aux0` interface an IP address from the same subnet used by the M7 application and ping the M7 domain on `169.254.11.12/24`. For example:

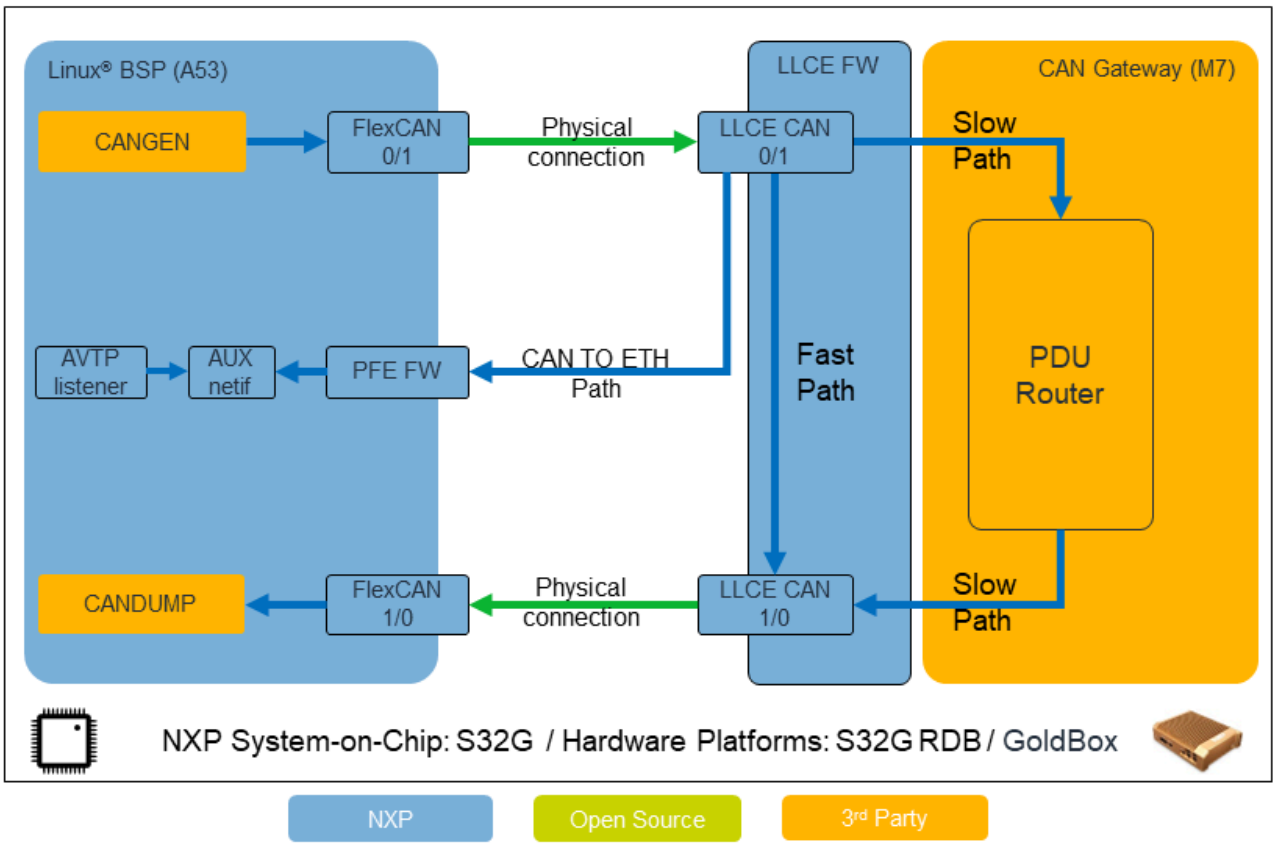
```
$ ip address add 169.254.11.100/24 dev aux0s1
$ ping -I aux0s1 169.254.11.12
```

14 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:



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.
- IDPS filtering using CAN frames inspection provided by Argus CAN IDPS (Intrusion Detection and Prevention System) engine. The CAN IDPS module is provided by Argus Cyber Security (https://argus-sec.com/) for demonstration purposes. Argus CAN IDPS looks for cyber attacks in the CAN network by monitoring frames for deviations in expected behavior and characteristics. Analysis is performed by an advanced rule-based heuristic detection and prevention engine.
  The engine is based on a Ruleset specifically generated for each vehicle model (i.e., in consideration of its architecture, messaging database, communication traffic models and other elements unique to the vehicle line). This Ruleset reflects the vehicle's traffic in normal operating circumstances and based on this Ruleset, Argus CAN IDS determines when a frame is anomalous, valid or in need of further investigation. There are two instances of a demo of the engine running in GoldVIP: one running on M7 core and one running inside the LLCE.
  Every CAN frame is inspected by Argus CAN IDPS based on the configuration of the Ruleset. For the full version of Argus CAN IDPS, the Ruleset may include features such as ensuring that the DLC (Data Length Code) matches the OEM's definitions, that all signals are in their allowed ranges, that frames' timing is as expected and that diagnostic frames are meeting the requirements of the ISO standards (see further details below under Available CAN IDPS features chapter).
  Any frame meeting the rules in the Ruleset is considered “accepted” and is forwarded to be further handled by the gateway and routed to its destination. In the case a frame is detected to include an anomaly, it is considered “rejected”, dropped and an anomaly report shall be sent for further diagnostics and logging.
  There could be cases where an anomaly is identified, yet the specific anomalous frame cannot be pinpointed. In these cases, the frame by which the anomaly was detected shall also be considered “accepted”, but an anomaly report shall be sent for further diagnostics and logging. This can happen, for example, when a Fixed-Periodic frame is received twice in a very short time interval. In this case, the legitimate frame cannot be distinguished from the injected frame, yet it is clearly an anomaly in the CAN traffic.

Note: A standard CAN frame with CAN ID `0x0` is sent on the LLCE CAN 14 bus when the `COMM_FULL_COMMUNICATION` communication mode is entered.

14.1 Available CAN IDPS features

The features detailed below are available in the full Argus CAN IDPS. Each of the available features are enabled via Ruleset configuration. GoldVIP integrates a demo that includes the Signal Range Enforcement and Minimum Transmission Intervals features from the following list.

- **ID Enforcement**: Reports and drops a frame as anomalous when the CAN ID is not expected on the inspected bus.
- **DLC Enforcement**: Reports and drops a frame as anomalous when its DLC (Data Length Code) does not match the value defined by the customer.
- **Signal Range Enforcement**: Reports and drops a frame as anomalous when at least one signal value is out of its allowed range.
- **Unallocated Bits**: Reports and drops a frame as anomalous when bits not allocated to any signal have an unexpected value.
- **Various frame timing inspection features**: The IDPS has several features to check the timing of frames on the bus. When frames appear on the bus at an unexpected timing, the frame shall be reported either as “rejected” and shall be dropped, or as “accepted” but with an anomaly report. This depends on the measure of certainty that the IDPS can determine the specific anomalous frame.
  An example - Minimum Transmission Intervals. In most cases, Event and Event-Periodic messages have a minimum interval that should be kept between transmissions of the message. This IDPS feature checks that this minimum is kept, and can be configured to report on the first violation, or only after several violations.
- **Bus Load detection**: Reports when a bus or several buses are loaded beyond a predefined threshold that suggest that the bus might be flooded. This feature only detects the load.
- **Diagnostic frames inspection features**: The IDPS offers several features for inspecting diagnostic frames of the UDS protocol (Unified Diagnostics Services), including:
  - Inspection of service identifiers (reports and drops “rejected” frames)
  - Detection of Brute force usage of diagnostics
  - Detection of diagnostic services scanning
  - Detection and prevention (drops “rejected” frames) of diagnostic services forbidden during driving.

14.2 Prerequisites

- S32G Reference Design Board or GoldBox running GoldVIP images

14.3 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 Appendix A. 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 multiplexed 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

```
-t can0 -r can1 -i 0 -o 4
-t can1 -r can0 -i 2 -o 3
example:
$ ./canperf.sh -t can0 -r can1 -i 0 -o 4 -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

```
-t can0 -r can1 -i 245 -o 245
-t can1 -r can0 -i 246 -o 246
example:
$ ./canperf.sh -s 64 -g 0 -i 245 -o 245 -t can0 -r can1 -l 10
```

- Optionally, one can use the default script provided in the can-gw directory: `can-fast-path.sh`

```
$ ./can-fast-path.sh
```

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`

```
$ ./can-to-eth.sh
```

e. For M7 IDPS:
- CAN-GW M7 IDPS library ruleset is configured to act on CAN ID 257 on can0 bus. If all preconditions are met then the frame will be routed with CAN ID 256 on can1 bus otherwise the CAN frame is considered malicious and dropped. In case an anomaly is detected, it is also reported via IPCF in order to be published in cloud.
- The preconditions are as follows:
  - The frame shall have a value between 0x00 and 0x20 in the first byte of the payload (e.g., `-D 2000000000000000`).
  - The frame shall have a cycle time of 1000 ms (e.g., `-g 1000`).
- With all the preconditions from above the following arguments to canperf should give you the same count of Tx and Rx frames:

```
-t can0 -r can1 -i 257 -o 256 -s 8 -g 1000 -D 2000000000000000
-t can0 -r can1 -i 257 -o 256 -s 8 -g 1000 -D 3200000000000000
example:
$ ./canperf.sh -t can0 -r can1 -i 257 -o 256 -s 8 -g 1000 -D 2000000000000000 -l
 10
```

- With all the preconditions from above the following arguments to canperf should result in the frames being rejected:

```
-t can0 -r can1 -i 257 -o 256 -s 8 -g 1000 -D 4500000000000000
-t can0 -r can1 -i 257 -o 256 -s 8 -g 100 -D 2000000000000000
example:
$ ./canperf.sh -t can0 -r can1 -i 257 -o 256 -s 8 -g 100 -D 2000000000000000 -l
 10
```

- Optionally, one can use the default script provided in the can-gw directory: `can-aidps-slow-path.sh`

```
$ ./can-aidps-slow-path.sh
```

f. For LLCE IDS:
- CAN-GW LLCE IDS library ruleset is configured to act on CAN ID 289 on can0 bus. If all preconditions are met then the frame will be routed with the same CAN ID on can1 bus otherwise the CAN frame is considered malicious and dropped. In case an anomaly is detected, it is also reported via IPCF in order to be published in cloud.
- The preconditions are as follows:
  - The frame shall have a value between 0x00 and 0x20 in the first byte of the payload (e.g., `-D 2000000000000000`).
  - The frame shall have have a cycle time of 1000 ms (e.g., `-g 1000`).

- With all the preconditions from above the following arguments to canperf should give you no reported anomalies:

```
-t can0 -r can1 -i 289 -o 289 -s 8 -g 1000 -D 2000000000000000
-t can0 -r can1 -i 289 -o 289 -s 8 -g 1000 -D 0000000000000000
example:
$ ./canperf.sh -t can0 -r can1 -i 289 -o 289 -s 8 -g 1000 -D 2000000000000000 -l
 10
```

- With all the preconditions from above the following arguments to canperf should result in the frames being marked as anomolous:

```
-t can0 -r can1 -i 289 -o 289 -s 8 -g 100 -D 2000000000000000
-t can0 -r can1 -i 289 -o 289 -s 8 -g 1000 -D 4500000000000000
example:
$ ./canperf.sh -t can0 -r can1 -i 289 -o 289 -s 8 -g 100 -D 2000000000000000 -l
 10
```

- Optionally, one can use the default script provided in the can-gw directory: `can-aidps-fast-path.sh`

```
$ ./can-aidps-fast-path.sh
```

g. 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 4e5850494e5850494e5850494e585049 4e5850494e5850494e5850494e585049`).
  - D : the authentic payload (e.g., `-D 234D736720736563757265642075736 6E6720484D4143205348412D32353623`).
- 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:

```
$./gen_secure_payload.sh -i 0040 -k 4e5850494e585049\
4e5850494e5850494e5850494e5850494e5850494e585049\
 -D 234D736720736563757265642075736\
696E6720484D4143205348412D32353623
```

- 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:

```
-g 2 -s 64 -t can0 -r can1 -i 64 -o 66 -l 10 \
-D 234D736720736563757265642075736696E6720484\
D4143205348412D323536231a2f6677adf62151b7aac\
ae784094607d752433544fd363d75ce942728029bce
-g 2 -s 64 -t can0 -r can1 -i 68 -o 70 -l 10 \
-D 234D736720736563757265642075736696E6720484\
D4143205348412D32353623499710c87f594e7f451751\
91eefc39e986d0f3c87e17773e16f9d8985e1d6a06
example for software crypto:
$ ./canperf.sh -g 2 -s 64 -t can0 \
-r can1 -i 64 -o 66 -l 10 \
-D 234D736720736563757265642075757\
3696E6720484D4143205348412D323536\
231a2f6677adf62151b7aacae78409460\
7d752433544fd363d75ce942728029bce
example for hardware crypto:
$ ./canperf.sh -g 2 -s 64 -t can0 \
-r can1 -i 68 -o 70 -l 10 \
-D 234D736720736563757265642075573\
696E6720484D4143205348412D3235362\
3499710c87f594e7f45175191eefc39e9\
86d0f3c87e17773e16f9d8985e1d6a06
```

- With the default settings the following arguments to canperf should make the CAN-GW reject all the frames as unsecured and not send any response back:

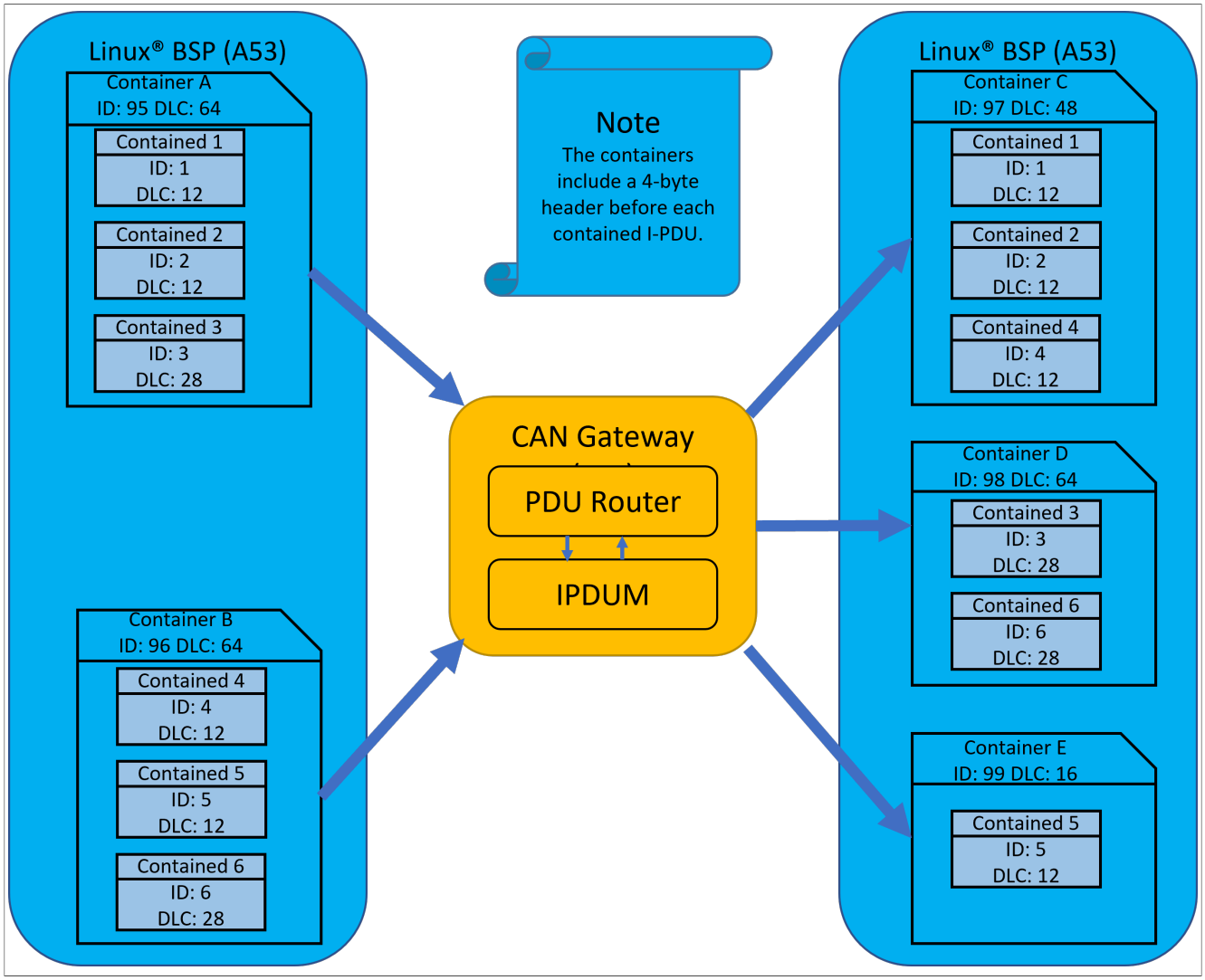
```
-g 2 -s 64 -t can0 -r can1 -i 64 -o 66 -l 10 -D 00
-g 2 -s 64 -t can0 -r can1 -i 64 -o 66 -l 10
example:
$ ./canperf.sh -g 2 -s 64 -t can0 -r can1 -i 64 -o 66 -l 10
```

- 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.

```
$ ./can-secoc-slow-crypto.sh
```

```
$ ./can-secoc-fast-crypto.sh
```

h. 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 44444444444444444444444`).
- 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:

```
$./gen_multi_pdu_payload.sh -i 1 -s 12 -D \
111111111111111111111111 -i 2 -s 12 -D \
222222222222222222222222 -i 3 -s 28 -D \
333333333333333333333333333333333333333333333333333333333333
```

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

```
-m -g 2 -l 10 -s 64 -t can1 -r can0 \
-i 95 -D 0100000C11111111111111111111111110200000C\
22222222222222222222222220300001C33333333333333333\
33333333333333333333333333333333333333333 \
-i 96 -D 0400000C444444444444444444444440500000C\
55555555555555555555555550600001C66666666666666666\
66666666666666666666666666666666666666666 \
-o 97 -o 98 -o 99
example :
$ ./canperf-multi.sh -m -g 2 -l 10 -s 64 -t can1 -r can0 \
-i 95 -D 0100000C11111111111111111111111110200000C\
22222222222222222222222220300001C33333333333333333\
33333333333333333333333333333333333333333 \
-i 96 -D 0400000C444444444444444444444440500000C\
55555555555555555555555550600001C66666666666666666\
66666666666666666666666666666666666666666 \
```

- Optionally, one can use the default script provided in the can-gw directory: `can-multi-pdu-path.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 the `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.:

```
$ sudo ./can-to-eth-slow-path-m7-host.sh -s 64 -g 10 <can> <eth>
```

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

- 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).
- The script is connecting to target console via `/dev/ttyUSB0`. In case the 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 serial port during the test.

14.4 Building the M7 Application

The Gateway application can be compiled using:

- NXP GCC 9.2.0 20190812 (Build 1649 Revision gaf57174)
- Green Hills Multi 7.1.6d / Compiler 2020.1.4

The distributed CAN-GW binary is compiled using the aforementioned GCC toolchain, starting from an adapted EB tresos AutoCore Platform. In order to get the same functionality as in the distributed binary image, use the following steps:

1. Download and install the EB tresos ACP software product mentioned in Software prerequisites. All the components provided in the installer shall be selected and a dedicated folder (e.g., `C:\EB\ACP-S32G3`) shall be used as the *Install Directory*.
2. Patch the Csm and Crypto tresos plugins by first navigating to the `<EB_ACP_install_path>` and then applying the patches found in the `<GoldVIP_install_path>/realtime/s32g399ardb3/can-gw/patches` folder:

```
$ cd <EB_ACP_install_path>
$ git apply --ignore-whitespace \
  <GoldVIP_install_path>/realtime/s32g399ardb3/can-gw/patches/*.patch
```

3. The GoldVIP Real-time Gateway application makes use of different NXP plugins than the ones packaged in the default EB tresos AutoCore deliverable. For this reason, the NXP plugins must be updated in the EB tresos AutoCore installation folder, by replacing the old ones: navigate to the `<EB_ACP_install_path>/plugins/`, delete the existing `McalExt_TS_T40DxM1I0R0`, and copy the contents of the `<GoldVIP_install_path>/realtime/s32g399ardb3/can-gw/eclipse/plugins` directory to update the environment with the latest plugins. One can also create a link to the NXP plugins if copying them is not desirable, by creating a file under the `<EB_ACP_install_path>/links` directory with the following contents:

```
path=<GoldVIP_install_path>/realtime/s32g399ardb3/can-gw/
```

   The NXP PFE, LLCE, IPCF and Real-Time Drivers referenced in Software prerequisites chapter are bundled in the preconfigured plugins package.
   Note: Any opened instance of the EB tresos Studio needs to be restarted after performing this change, in order to load the newly installed plugins.
4. Install a supported toolchain (e.g., GCC, already provided with S32 Design Studio).
5. The build environment must be configured. Edit the `<GoldVIP_install_path>/realtime/s32g399ardb3/can-gw/workspace/goldvip-gateway/util/launch_cfg.bat` file, by replacing the default paths with the actual paths on your host system:
   - The toolchain used:

```
SET TOOLCHAIN=(gnu|multi)
```

   - The path to one of the supported toolchain compilers:

```
:: Using the GCC compiler:
SET TOOLPATH_COMPILER=C:/NXP/S32DS_3.5/S32DS/build_tools/gcc_v9.2
:: Using the GHS compiler:
```

```
SET TOOLPATH_COMPILER=C:/ghs/comp_202014
```

- The path to Elektrobit Tresos Studio installation folder:

```
SET TRESOS_BASE=C:/EB/tresos
```

- The path to HSE firmware:

```
SET HSE_FIRMWARE_DIR=">path to the local directory
                       that contains the HSE FW deliverables<"
```

- The path to PFE firmware:

```
SET PFE_FIRMWARE_DIR=">path to the local directory
                       that contains the PFE FW deliverables<"
```

- The path to LLCE firmware (optional):

```
SET LLCE_FIRMWARE_DIR=">path to the local directory
                       that contains the LLCE FW deliverables<"
```

6. Open EB tresos Studio and import the *goldvip-gateway* project located at `<GoldVIP_install_path>/realtime/s32g399ardb3/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 Project` 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` EB tresos Studio 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>/realtime/s32g399ardb3/can-gw/workspace/goldvip-gateway/
util
$ launch.bat make -j
```

Note: The RTI DDS-related functionalities are not available when the GHS compiler is used. The gateway application must be reconfigured to build successfully using the GHS compiler, by removing the `TcpIpSocketOwner_DDS` entry from the `TcpIpSocketOwnerConfig` list under the `TcpIp` module.

14.5 Building the custom LLCE Firmware

The custom LLCE Firmware can be compiled using:

- NXP GCC 9.2.0 20190812 (Build 1649 Revision gaf57174)

The distributed Gateway application makes use of a custom LLCE firmware tailored for providing the functionalities demonstrated by some use cases. In order to get the same LLCE functionality as in the distributed binary image:

1. Download and install the GoldVIP LLCE customization version mentioned in Software prerequisites.
2. Install a supported toolchain (e.g., GCC, already provided with S32 Design Studio)
3. Adapt `<GoldVIP_LLCECSTM_install_path>/llce_fw/config.mak` to your particular system needs (i.e., `GCC_location`, `DEV_ID`, `RELEASE_ID`). Open a Command Prompt and run the following commands to build the LLCE firmware:

```
$ cd <GoldVIP_LLCECSTM_install_path>/llce_fw
$ make bin TOOLCHAIN=gcc TARGET_DEVICE=S32G3
```

4. After building, the binaries and the C source files will be available in `<GoldVIP_LLCECSTM_install_path>/fdk/llce_bin/gcc/enablement`.
   Copy these files to the gateway workspace:

```
$ cp -a <LLCECSTM_install_path>/llce_bin/gcc/enablement/. \
```

```
<GoldVIP_install_path>/realtime/s32g399ardb3/can-gw/workspace/source/integration/
firmware/llce
```

5. Build the M7 gateway application in order to load the new LLCE FW. Check the Building the M7 Application chapter for more detailed steps.

15 Health Monitoring

The Health Monitoring component running primarily on Cortex-M7 acquires and computes several voltages that are used by the SoC. It showcases and uses several components that are present in an automotive Classic Platform AUTOSAR application:

- The Input/Output AUTOSAR stack: it integrates and configures the OCU and ADC MCAL modules and a demo example of the AUTOSAR IoHwAb module. This demo provides a Service Software Component description file generated based on the configuration in order to provide access to other AUTOSAR Software Components (SWC) to the underlining IOs through the RTE port mapping mechanism described by AUTOSAR.
- The actual Health Monitoring SWC which uses the beforementioned IoHwAb Service SWC to acquire three voltages that are present on the board: 1.1V, 1.2V and, 1.8V. This showcases how a SWC can be developed and integrated in the GoldVIP environment. After the acquisition and processing of the voltages they are sent to A53 domain through IPCF for further processing.

The telemetry collector running on A53 gathers the data and sends it to the cloud and the Telemetry Server. Please check the linked chapters for details on how to visualize the data.

Note: there are no hardware connections to be made between the monitored voltages and the analog channels of the SoC as they are already hardwired.

16 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.

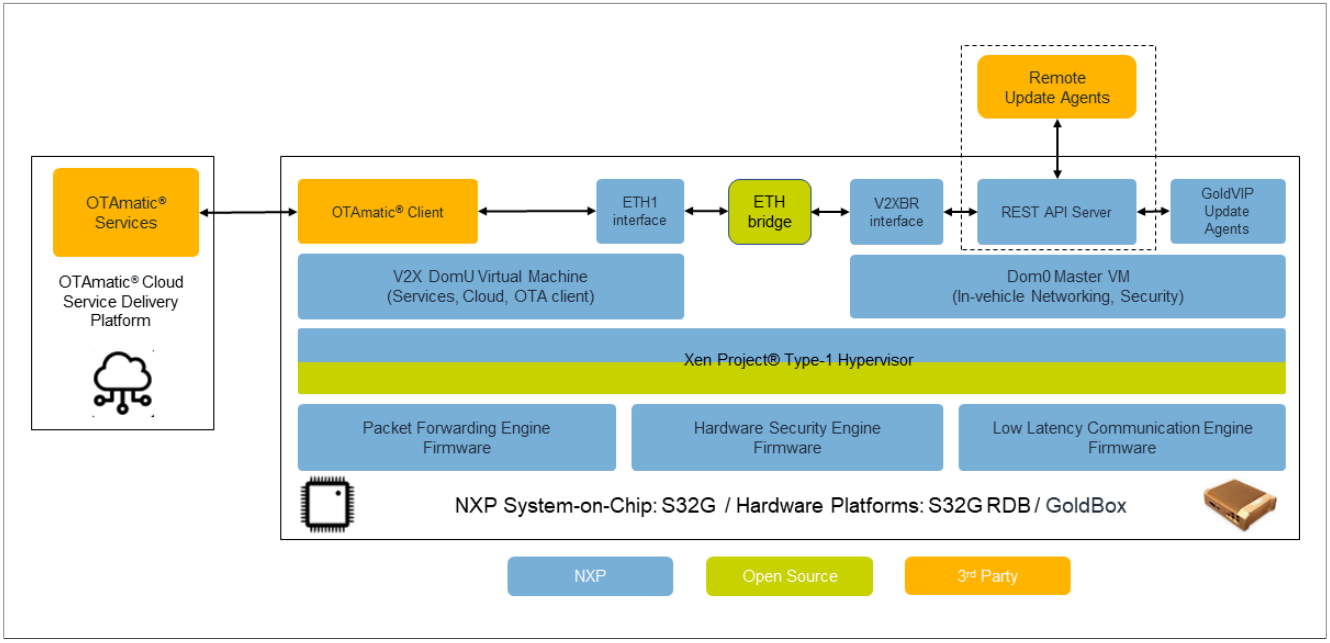
Contact Airbiquity for additional information about the OTAmatic Software Management Platform and/or full feature set evaluation (otamatic-nxp@airbiquity.com).

16.1 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. Based on a precondition file that is used when setting up the campaign, the Client monitors if vehicle conditions are adequate to proceed with the installation of the updates. The OTAmatic Client Integrator Interface component indicates if the conditions to proceed are met, not met, or if the OTA campaign needs to be terminated. Two files are monitored to interact with the OTAmatic Client, located in the rootfs of the `v2xdomu` VM at `/data/aqconfig/`:

- `ota_check` - used to indicate if conditions are adequate to proceed with the OTA campaign. It contains either `ota_no_go` (conditions are not met), `ota_go` (conditions are met), or `<number>` (indicates that conditions are not met, and sets a polling period of this many seconds).
- `request` - used to send requests to the Client, and is removed after being read. It may contain either `updates` (request to check for updates), `proceed` (request to continue with the OTA update), or `status` (request the current OTA status).

A simple text file is used as the preconditions file, with two words ending in a comma. First word will determine the behavior during the download, while the second word will determine the behavior for the installation step. Values expected are:

- `go` - this value marks the preconditions as met.
- `wait` - this value marks the preconditions as not met. Client will wait until a `proceed` request is issued to continue with the download or installation.

Once the preconditions are met, the OTA Client commands each Update Agent to update the component it handles according to the update campaign that was configured in the OTAmatic SDP.

Note: When using the `fsl-goldvip-no-hv` distro, the OTAmatic Client and the Update Agents reside on the same machine, they communicate using a set of dummy interfaces: `dummy1`, and `dummy2`.

16.2 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

16.3 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:

```
images: [
{
  "filename": "<binary image filename (i.e., boot-loader)>",
  "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>,
    "boot_strategy": "BOOTROM"
    "pivt": <null or the address of the IVT entry that should be altered>,
    "clear_bootcfg": <boolean value describing whether to clear the Boot configuration
 word>
  }
},
{
  "filename": "<binary image filename (i.e., 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>,
    "boot_strategy": "BOOTLOADER",
    "application_id": "ID of the application for which the image is updated",
    "image_id": "ID of the image fragment that is updated",
    "load_address": "address where this image fragment should be copied to before boot",
    "entrypoint": "reset handler address of this application"
  }
},
{
  "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 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., *0xb00000* for GoldVIP Gateway application), and the old application will be kept and can be used again if rollback is requested. The system is configured to use 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:

```
# Before the realtime OTA update:
bootloader 0x2 Application 0 [...] from QSPI, address 0x201240.
bootloader 0x2 Application 1 [...] from QSPI, address 0x400000.

# After the update:
bootloader 0x2 Application 0 [...] from QSPI, address 0x201240.
bootloader 0x2 Application 1 [...] from QSPI, address 0xb00000.
```

Note: The addresses presented in the previous example are indicative and they may be adjusted in the future.

16.4 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 rootfs, 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 rootfs. 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*.

16.5 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.

Note: When using the `fsl-goldvip-no-hv` distro, all the resources mentioned below for Dom0 and v2xdomu are on the same machine, at the same paths as they are specified below.

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

```
$ xl console v2xdomu
```

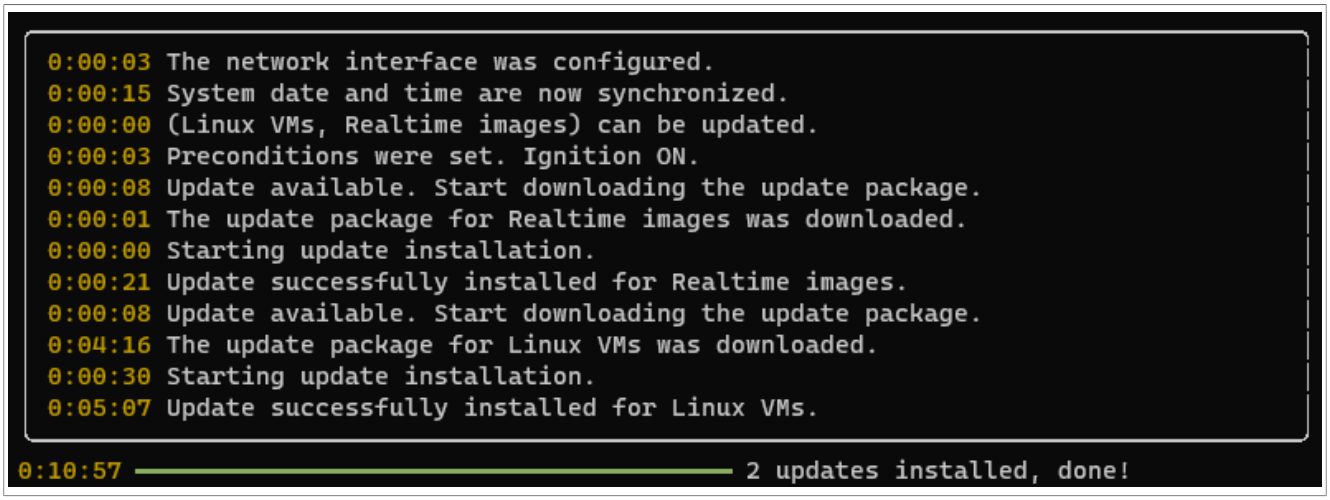
2. Start the update of Real-time images and Linux VMs using:

```
$ 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 OTA conditions to the required values) and display in real-time the progress of download and installation steps. An update starts once all the configured conditions are met and a request is sent to the OTA Client:

- for Real-time images update, the preconditions file contains `go,go,`, therefore the update is installed as soon as the `updates` request is sent to Client (using the `/data/aqconfig/request` file exposed in the `v2xdomu` rootfs).
- for Linux VMs update, the preconditions file contain `go,wait,`, therefore the update is downloaded as soon as possible, but it is not installed until a `proceed` request is sent to the Client (i.e., `echo "proceed" > /data/aqconfig/request`).

After the `updates` request is sent, 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:



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
```

Note: When using the `fsl-goldvip-no-hv` distro, the Linux VMs update is not available.

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 restarting the OTAmatic Client service:

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

16.6 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 `$ /home/root/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 `$ /home/root/ota/demo/reset_sdk.sh` on v2xdomu VM to reset the OTAmatic Client configuration.
3. Run `$ /home/root/ota/demo/otamatic_controller.sh restart` on v2xdomu VM to restart the OTAmatic Client.

16.7 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 that runs on the S32G device. The Kubernetes cluster is used to manage and supervise these services in order to ensure these applications run continously.

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 rootfs 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-s32g399ardb3.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 describe 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-s32g399ardb3.oci-image.tar` on v2xdomu VM. In order to keep the data persistent across reboots, this manifest creates volume mounts to specific host paths.

16.8 Providing updates to remote devices

GoldVIP can be used to provide updates to remote devices connected to the gateway. Each device connected to the S32G must run an OTA Agent that can install updates and is able to register itself to the GoldVIP Update Agents for receiving the OTA updates. GoldVIP Update Agents act as routing points between OTAmatic Client and the GoldVIP Remote Update Agent, forwarding the messages received from the OTA Client to the corresponding device connected to the gateway, and vice-versa. The forwarder component does not keep a static configuration of the connected nodes, so each remote Update Agent must register itself to the GoldVIP Update Agents prior to receiving updates.

In order to register a remote Update Agent to the forwarder component, the remote agent must send its configuration data (i.e., network address, port, update agent ID) by triggering a POST request on `http://<gateway_address:port/generic/register`. Once the request is made and a response is received from the forwarder component, the remote Update Agent can receive updates. The following format is used for the register request:

```
{
```

```
  "elementId": "<specific ID of the remote device/Update Agent>",
  "elementAddr": "<network address used by the remote device>",
  "elementPort": "<port used by the OTA agent to listen for update requests>",
  "elementUrl": "<specific URI used to serve the requests>"
}
```

The GoldVIP Update Agents will start forwarding any request received from OTAmatic Client that targets the remote OTA agent to `http://<elementAddr>:<elementPort>/<elementUrl>/<query_uri>?<query_params>`.

GoldVIP includes an example of a remote Update Agent that can run in the context of Dom0 virtual machine, separately from the GoldVIP Update Agents. Instead of connecting to OTAmatic Client, this agent communicates with the forwarder component of the GoldVIP Update Agents to receive updates. By default, this OTA agent runs as a containerized application that is started once the system boots up, and a specific campaign for it is already preconfigured. The update is received and installed after the other updates (i.e. Real-time images and Linux virtual machine updates) are installed. After the update package is installed, a text file will be placed at `/home/root/ota/remote_ua` path in the rootfs of Dom0 VM. The `/home/root/ota/demo/reset_update_agents.sh` script can be used to reset the demo.

17 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).

17.1 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.

17.1.1 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+]`: Resumes to Dom0 console (can be run only from a DomU).

- `xl shutdown <domain ID>`: Shuts down a DomU.

17.1.2 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>
$ brcrl 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.

17.1.3 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/

17.2 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.

17.2.1 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 rootfs 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.

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.

17.2.2 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:

```
$ # 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`.

18 OP-TEE

The OP-TEE software component provides Linux users with a Trusted Execution Environment (TEE) so that critical applications working with sensitive data can leverage the hardware isolation of Arm TrustZone. Such applications are called Trusted Applications (TAs), and they run at Secure EL0.

The OP-TEE project is split into several modules:

- The `optee_client` module, residing entirely in the user space of Linux, facilitates the communication between normal world applications (running at Non-Secure EL0 level) and secure world through the TEE Client API and the tee-supplicant daemon. When a normal world application invokes a TA via an API call, behind the scenes the normal world - the OP-TEE driver in the Linux Kernel - executes a Secure Monitor Call (SMC) instruction. The Secure Monitor (SM) traps the exception and will switch execution to `optee_os` in the secure world. The process of switching to the normal world from the secure world uses the same mechanisms.
- The `optee_os` module, which runs at Secure EL1, ensures that TAs have all the necessary resources to be executed safely and securely. It has its minimal kernel, drivers and libraries, and implements the TEE Internal Core API that can leverage the TrustZone (TZ) hardware isolation. While `optee_client` implementation is platform-independent, the `optee_os` module is configured according to the platform specifications. Trusted memory zones, UART communication and GIC initialization are configured to ensure proper usage. TF-A is responsible for loading and executing the optee_os binaries. Moreover, as mentioned above, OP-TEE also relies on the Secure Monitor implemented in the TF-A.

Besides the `optee_os` and `optee_client` components, which are mandatory for creating a functional Trusted Execution Environment, the Linux BSP comes with another two modules offered by the OP-TEE project: `optee_examples` and `optee_test`. The contents of both these modules reside in the user space of Linux:

- The `optee_example` module, which contains a number of example applications, all prefixed with `optee_example_`. For example, to run the `optee_example_hello_world` one must enter the following command:

  ```
  # optee_example_hello_world
  ```

  The source code for these applications can be found on the official GitHub page: OP-TEE Sample Applications. The documentation can be found in the official OP-TEE documentation: OP-TEE Example applications.
- The `optee_test` module, which contains several thousand tests checking that the OP-TEE integration has been done successfully. Run the entire test suite with the following command:

  ```
  # xtest
  ```

  More information about running xtest can be found in the official OP-TEE documentation: optee_test.

Note: By default, OP-TEE is configured to have 2 virtual guests. This can be changed via the `CFG_VIRT_GUEST_COUNT` parameter.

18.1 OP-TEE RPMB secure storage

OP-TEE implements a Replay-Protected Memory Block (RPMB) secure storage. RPMB is a partition on the eMMC where the host can read/write blocks of data, but cannot delete them. Every block is encrypted by a device specific key. Both the v2xdomu and dom0 host a Trusted Application called `optee_certificate_secure_storage` which can be used to write and read data from the RPMB. By default, the RPMB partition on the eMMC is emulated, it is not mandatory to boot from an eMMC to use this feature. This can be changed during compile time via the `RPMB_EMU` parameter.

The Cloud-GW client device provisioning can use the RPMB secure storage to store the certificates used by the client devices to connect to the AWS cloud. To use this feature, refer to the OP-TEE RPMB secure storage for AWS Certificates chapter.

More information about RPMB can be found in the official OP-TEE documentation:RPMB Secure Storage.

Note: By default, `tee-supplicant` emulates a RPMB partition and, for safety considerations, this emulation is used. Disabling the emulation translates to a commitment of permanently programming the RPMB Key into the eMMC partition, assuming the corresponding consequences that come with it.

19 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.

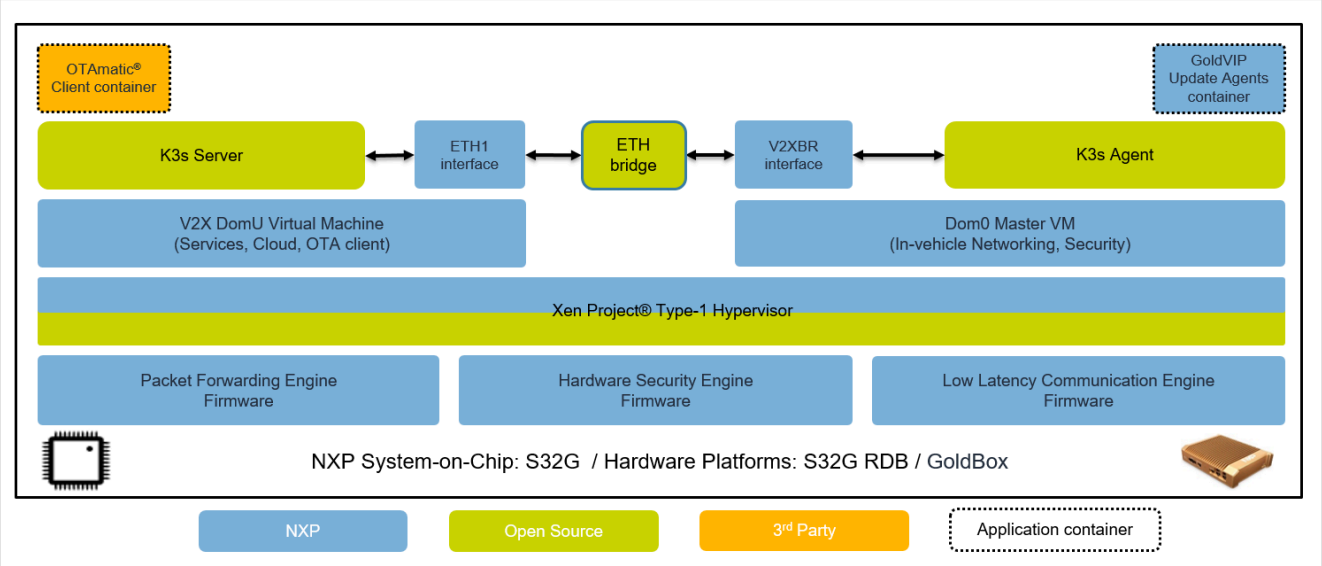
19.1 The K3s cluster

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 *Deployment*s, *DaemonSet*s, and *CronJob*s, 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/.

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 autheticating agents that try to join the cluster. In order to optimize the hardware resouce 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).
- `s32g399ardb3`, 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 setting to the K3s agent process.

Note: When using the `fsl-goldvip-no-hv` Yocto distro, the nodes are running on the same machine.

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 rootfs 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' rootfs 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).

19.2 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 `s32g399ardb3`.
- `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 <ctr 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`, 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 `s32g399ardb3` 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 `s32g399ardb3` node. Therefore, the `kubectl` command can be used from the context of Dom0 VM once the cluster is initialized.

19.3 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 `s32g399ardb3` 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 deployments -o wide
$ # List the deployed pods. One should notice some pods with names formatted
 as `nginx-*-*`.
$ kubectl get pods -o wide
$ # List all the services. One can notice the `nginx` service that was
 deployed previously.
```

```
$ kubectl get services -o wide
```

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 nginx | 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 accesible without the need of specifying the service port: http://<V2XDOMU_IP>. 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:

```
$ # Replace the 'vmtype' label with a common value for both nodes.
$ sed -i 's|vmtype:.*|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:

```
$ # Apply the changes done in the manifest file.
$ kubectl apply -f /home/root/containers/nginx.yaml
$ # List the pods created for the nginx application.
$ kubectl get pods -l app=nginx -o wide
```

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:

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

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

20 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. Four communication channels are available currently in GoldVIP:

- *echo*: Any message sent through this channel will be echoed back by the Cortex-M7_0.
- *idps_statistics*: This channel exposes specific IDPS statistics, collected by the Cortex-M7-0. Writing to this channel produces no results. This channel is used by the cloud telemetry collector in order to send IDPS data to the cloud.
- *benchmark*: Writing to this channel triggers a benchmarking run on Cortex-M7-1, if it is not already running. Reading from this channel retrieves the results from finished runs of the benchmark.
- *health_mon*: This channel exposes the voltages acquired and computed by Cortex-M7-0. Writing to this channel produces no results. The data is used by the cloud telemetry collector in order to send the monitored voltages to the cloud.

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*).

20.1 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:

```
$ echo message#1 > /dev/ipcfshm/M7_0/echo
$ echo message#2 > /dev/ipcfshm/M7_0/echo
$ echo message#3 > /dev/ipcfshm/M7_0/echo
$ cat /dev/ipcfshm/M7_0/echo
```

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

20.2 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: 128 bytes. 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.

21 Performance

21.1 Benchmark on Cortex-A53

The performance of the ARM Cortex-A53 cores on the S32G platform can be measured using the CoreMark tool from EEMBC. The CoreMark benchmarking tool, an industry-standard, produces a single-number score allowing users to make quick comparisons between processors.

GoldVIP includes the CoreMark tool in the root file system of the Dom0 VM. For convenience, for Cortex-A53 a script that runs benchmarks with two different sets of parameters is available for use on Dom0:

```
$ /home/root/benchmark/benchmark-a53.sh
```

The first set of parameters is used to compute the CoreMark score, and the second one is used to validate the operations.

CoreMark's executable takes several parameters as follows:

- 1st, 2nd, 3rd - Seed values used for initialization of data.
- 4th - Number of iterations (0 for auto : default value).
- 5th, 6th - Reserved for internal use.
- 7th - For malloc users only, overrides the size of the input data buffer.

Consult the CoreMark repository (https://github.com/eembc/coremark) for more details about the run parameters and the output format.

Note: The overhead of virtualization on Cortex-A53 (detailed in the Virtualization chapter), along with a set of concurrently-running application may have an impact on the results of the CoreMark benchmark score.

Note: When using the `fsl-goldvip-no-hv` Yocto distro, the resources located on Dom0 are on the Linux machine.

21.2 Benchmark on Cortex-M7

GoldVIP includes an adaptation of the CoreMark benchmarking tool in the CAN Gateway application as a Complex Device Driver. The benchmarking software runs on Cortex-M7-1. There is an IPCF channel (detailed in the IPCF Shared Memory chapter) used to trigger the benchmarking run and to retrieve the results. To trigger a run and get the results use the following script from Dom0 VM:

```
$ /home/root/benchmark/benchmark-m7.py
```

the benchmarking software will run with seeds 0x00,0x00,0x66 and with 4000 iterations. The script will output the score, the run time in seconds and in core ticks.

Note: The overhead of the Autosar OS on Cortex-M7 (mentioned in the CAN Gateway chapter), impacts the results of the benchmark score.

22 HSE Security Support

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.

22.1 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/#&/' \
    /etc/ssl/openssl.cnf
```

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

```
$ sed -i '/openssl_conf = goldvip_openssl_def/s/^#//' \
    /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.

Note: When using the `fsl-goldvip-no-hv` Yocto distro, the OpenSSL library residing on the standalone Linux machine is configured to use the HSE crypto driver.

22.2 PKCS#11 support

PKCS#11 represents a cryptographic token interface standard, used to create and manipulate cryptographic data like public-private key pairs. In the context of GoldVIP, PKCS#11 is used as a low-level interface that performs cryptographic operations using the NXP HSE subsystem, without the need for the application to directly interface the device through its driver. The PKCS#11 support is based on the PKCS#11 Cryptographic Token Interface Base Specification Version 3.0 specification.

An user-space module (`libpkcs-hse.so`) is used to enable the communication with HSE for the user-space applications that make use of the PKCS#11 interface. This module is available by default in the filesystem of the dom0 virtual machine (`/usr/lib/libpkcs-hse.so`) and can be used to access the PKCS#11 APIs. GoldVIP exposes the PKCS#11 functionality in the v2xdomu virtual machine as well, by using the `p11-kit proxy` functionality - therefore, one can store keys in the HSE subsystem, and then use them from the context of the v2xdomu VM using the `/usr/lib/p11-kit-proxy.so` module.

In order to test the PKCS#11 interface functionality, one can use the `pkcs11-tool` to load RSA keys (public/pair), EC keys (public) and AES keys. To make use of the HSE PKCS#11 interface using the `pkcs11-tool`, from both dom0 and v2xdomu virtual machines, one must provide the path to the HSE PKCS#11 library (i.e., passing the `--module /usr/lib/libpkcs-hse.so` option), respectively to the p11-kit proxy module on v2xdomu (i.e., `--module /usr/lib/p11-kit-proxy.so`). For example, to load and then remove a public RSA key from the console of the dom0 virtual machine, one can run the following commands:

```
$ # Generate a RSA key pair using OpenSSL and extract the public key
$ openssl genrsa -out rsa_keypair.pem 2048
$ openssl rsa -in rsa_keypair.pem -outform DER \
    -pubout -out rsa_keypub.der
$ # Load the public key in the HSE NVM key catalog
$ pkcs11-tool --module /usr/lib/libpkcs-hse.so \
    --token-label "NXP-HSE-Token" --type pubkey \
    --write-object rsa_keypub.der --id 000701 \
    --label "HSE-RSAPUB-KEY"
$ # Check that the key was imported and then remove it
$ pkcs11-tool --module /usr/lib/libpkcs-hse.so \
    --token-label "NXP-HSE-Token" --list-objects
$ pkcs11-tool --module /usr/lib/libpkcs-hse.so \
    --token-label "NXP-HSE-Token" --type pubkey \
    --delete-object --label "HSE-RSAPUB-KEY"
```

The `--id` option corresponds to the key's slot (`00`), group (`07`) and catalog (`01`), in hexadecimal, as described in the from the HSE Key Catalog chapter.

22.3 HSE Key Catalog

The GoldVIP bootloader configures the HSE key catalog in order to be used by both realtime and Linux components. The key catalog configuration is exhibited in the following table:

Catalog ID	Group ID	Key type	Number of key slots
0x01 (NVM key catalog)	0x00	AES, maximum 256 bits	10
	0x01	AES, maximum 128 bits	5
	0x02	HMAC, maximum 512 bits	5
	0x03	ECC Pair, maximum 256 bits	2
	0x04	ECC Public, maximum 256 bits	2

Catalog ID	Group ID	Key type	Number of key slots
	0x05	ECC Certificate, maximum 256 bits	1
	0x06	RSA Pair, maximum 2048 bits	2
	0x07	RSA Publick, maximum 2048 bits	2
	0x08	RSA Certificate, maximum 2048 bits	1
0x02 (RAM key catalog)	0x00	RSA Public, maximum 2048 bits	1
	0x01	AES, maximum 256 bits	12
	0x02	HMA, maximum 512 bitsC	6
	0x03	ECC Public, maximum 256 bits	1

23 Adaptive AUTOSAR

The AUTOSAR Adaptive Platform (AP) is the standardized platform for microprocessor-based ECUs supporting use cases like highly automated and autonomous driving as well as high speed on-board and off-board communication.

GoldVIP makes use of the EB corbos AdaptiveCore Adaptive AUTOSAR solution, deploying it on the v2xdomu VM.

This solution together with the EB corbos Studio allows the user to create, configure and deploy Adaptive Applications.

The AdaptiveCore solution is added to the v2xdomu image through the `eb-ara` Yocto layer.

Note: When using the `fsl-goldvip-no-hv` Yocto distro, the AdaptiveCore solution is added to the standalone Linux machine in the same way.

23.1 Preparing the Adaptive AUTOSAR source files

- Download the EB corbos AdaptiveCore deliverable referenced in the Software prerequisites chapter.
- Open the `setup_ADG_sources.sh` script found inside the GoldVIP binaries archive under `<GoldVIP_install_path>/adaptive` and customize the `CORBOS_ARCHIVE` and `CORBOS_FOLDER` variables if needed.
- Run the `setup_ADG_sources.sh` script which will extract the ara components and EB corbos Studio from the provided ADG archive and place them in `CORBOS_FOLDER`:

```
$ cd <GoldVIP_install_path>/adaptive
$ chmod a+x setup_ADG_sources.sh
$ ./setup_ADG_sources.sh
```

The AdaptiveCore solution integrated into GoldVIP only contains the Adaptive Platform and all the building blocks required for deploying and running Adaptive Applications.

In order to create an Adaptive Application using EB corbos Studio and deploy it to the GoldVIP you must first set up the EB corbos toolchain.

23.2 Setting up the EB corbos toolchain

- Run the script that sets up the `ara-cli` multitool :

```
$ bash ~/nxp-goldvip/adg/ara-corbos-AdaptiveCore-deliveries/ara_BuildEnv/files/
starterkit/installers/adg-standard-installer.sh
$ PATH=~/.local/bin:$PATH
```

  The installer also creates a fixed empty directory structure.
- The `setup_ADG_toolchain.sh` script automatically populates the toolchain directory structure. Update the `ADG_ROOT`, `ADG_VERSION`, `ADAPTIVECORE_DELIVERIES` and `CORBOS_FOLDER` paths before running the script if custom install locations are desired:

```
$ cd <GoldVIP_install_path>/adaptive
$ chmod a+x setup_ADG_toolchain.sh
$ ./setup_ADG_toolchain.sh
```

- Build the GoldVIP specific SDK for v2xdomu by following the Build the software development kit chapter.
- Run the SDK installer script and install the software to the `<ADG_ROOT>/adaptivecore/sdk/custom/custom/` directory (default: `~/ara/eb/adaptivecore/sdk/custom/custom/`):

```
$ cd ~/nxp-yocto-goldvip/build_s32g399ardb3/tmp/deploy/sdk/
$ ./fsl-auto-glibc-x86_64-cortexa53-crypto-toolchain-43.0.sh
```

- Include the specific CMake modules from the ara build environment to the installed SDK:

```
$ cp -R ~/nxp-goldvip/adg/ara-corbos-AdaptiveCore-deliveries/ara_BuildEnv/config/cmake/
* \
~/ara/eb/adaptivecore/sdk/custom/custom/sysroots/x86_64-fslbsp-linux/usr/share/
cmake-3.22/Modules
```

- Register the installed SDK with ara-cli:

```
$ ara-cli SdkMgr \
--env-file ~/ara/eb/adaptivecore/sdk/custom/custom/environment-setup-cortexa53-crypto-
fsl-linux \
--platform custom \
--toolchain-file ~/ara/eb/adaptivecore/sdk/custom/custom/sysroots/x86_64-fslbsp-linux/
usr/share/cmake/OEToolchainConfig.cmake \
--version 0.0.1
```

- Activate the EB corbos license by following the instructions from: https://www.elektrobit.com/support/licensing/.
- Register and install EB corbos Studio with `ara-cli`:

```
$ export WORKSPACE=~/ara/eb/adaptivecore/source/2.14.0
$ cp -R ~/nxp-goldvip/adg/ara-corbos-AdaptiveCore-deliveries/ara_BuildEnv/tools/
EB_corbos_Studio \
~/.local/lib/python<python_version>/site-packages/ara/tools
$ corbos-studio-launcher --single-user-license
```

After running this commands, an instance of EB corbos Studio will be installed in `~/ara/eb/adaptivecore/source/2.14.0`.

23.3 Creating, Building and Deploying an Adaptive Application

The build process is described in more detail in the "EB_corbos_AdaptiveCore_user_guide_Linux.pdf" manual found in the EB corbos deliverable. The following guide is a slightly modified version of the build process provided in the manual and involves some extra manual steps since the GoldVIP does not use the EB corbos Linux operating system and does not use systemd.

- Create a new application using ara-cli:

```
$ mkdir ~/workspace/
$ cd ~/workspace/
$ ara-cli Application --create-project \
--app ~/workspace/demo \
--target-os custom \
--target-platform custom
```

- Generate the CMake files needed by the project build system:

```
$ ara-cli Application --generate-cmake \
--app ~/workspace/demo \
--target-os custom \
--target-platform custom
```

- Start EB corbos Studio using the command line:

```
$ cd ~/ara/eb/adaptivecore/source/2.14.0/EB_corbos_Studio-2023_R04_03_00-Linux_x86_64
$ ./EB_Corbos_Studio.bash
```

- Import the demo project into the workspace.
- Configure the `model/AP_R19-03/em_ecu_configuration.ecuconfig` so that:
  - in GeneralSettings, the `Os` parameter is set to `linux_standalone`
  - in GeneralSettings, the `Architecture` parameter is set to `arm64`
  - the Uids and Gids found in the `Group` and `Identity` configuration containers are set to 0 (root).

- Run the em pluget (`AraEmModelGenerator`) to generate the execution manifest by right clicking on the ExecutionManager → Autosar Adaptive → Run pluget → select AraEmModelGenerator → press Finish.
- Run the dlt pluget (`AraDltModelGenerator`) to generate the dlt configuration file by right clicking on the ExecutionManager → Autosar Adaptive → Run pluget → select AraDltModelGenerator → press Finish.
- Build the application:

```
$ ara-cli Application \
--app ~/workspace/demo \
--target-os custom \
--target-platform custom
```

- Before deploying the application, configure the `project_config.json` file, remove the container section and update the deploy-files section so that all the required files (execution manifest, executable, shell script, sandbox JSON file and shared libraries) are copied to the correct places in the file system. We provide a `project_config.json` example file (under `<GoldVIP_install_path>/adaptive`) based on the default settings. The value of the `target-host` field must be updated so that it contains the IP address of the v2xdomu VM. Deploy the application using the `TargetOperation` command:

```
$ ara-cli TargetOperation --app ~/workspace/adg/demo
```

Note: *Automatic deployment works only if the target is connected to a network that is reachable by the host !*

- Another option is doing the deployment manually by copying the files to the correct locations inside the target filesystem:
  - execution manifest (`demo_amd.json`) to `/etc/adaptive/ara_EM/state-manager`
  - shell script (`demo.sh`), sandbox JSON file(`demo.json`), and dlt JSON file (`ara_DLT_am.json`) to `/etc/adaptive/demo`
  - executable (`demo`) to `/usr/bin`
  - shared libraries (`libexample.so`) to `/usr/lib`.
- Make the application binary and the shell script executable:

```
$ ara-cli TargetOperation --app ~/workspace/adg/demo \
  --remote-command='chmod a+x /etc/adaptive/demo/demo.sh'
$ ara-cli TargetOperation --app ~/workspace/adg/demo \
  --remote-command='chmod a+x /usr/bin/demo'
```

- Reset the target.
- Open the v2xdomu console on the board and check if the demo service is installed:

```
$ emtool allProc
```

- Enable the demo application:

```
$ emtool set demo On
```

- The demo application is now writing "Hello World" to the log file:

```
$ cat /tmp/adg.log | grep Hello
```

24 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).

24.1 First Time Setup

All the steps described below have been run and validated on Ubuntu 20.04 LTS (native or through a virtual machine). It is then recommended to install Ubuntu 20.04 LTS before going through the following sections.

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:
  - 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 https://storage.googleapis.com/git-repo-downloads/repo-2.32 \
 > ~/bin/repo
$ chmod a+x ~/bin/repo
$ PATH=${PATH}:~/bin
```

24.2 Building GoldVIP

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

Note: A Yocto build needs at least 250 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 or more.

24.2.1 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 -b develop -m default.xml \
  -u https://github.com/nxp-auto-goldvip/gvip-manifests
$ repo sync
```

Notes:

- For a specific GoldVIP release or engineering build, please use the proper branch (`-b <branch>`) and manifest file (`-m <manifest>`).
- To fetch internal development repositories use Bitbucket URL:

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

Manifest files description:

- default.xml -> fetch GoldVIP repositories from https://github.com/nxp-auto-goldvip
- default-bitbucket.xml -> fetch internal development repositories from Bitbucket

- alb.xml -> default NXP Auto Linux BSP without GoldVIP extensions

24.2.2 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-hv \
 -m s32g399ardb3 \
 -e "meta-aws meta-java meta-vip"
```

This release includes support for:

– Machines: `s32g399ardb3`

– Images: `fsl-image-goldvip`

– NXP boards: `s32g399ardb3`

– Two distributions: `fsl-goldvip-hv` (default) and `fsl-goldvip-no-hv`:

  – `fsl-goldvip-no-hv` distro includes optional features such as GoldVIP Real-time Gateway, GoldVIP Real-time Bootloader, GoldVIP OTA applications, GoldVIP containers deployed via K3s, cryptographic support through HSE, RTI DDS demo applications, SJA1110 firmware, and Adaptive AUTOSAR. Any of these features can be removed from the image.

  – `fsl-goldvip-hv` includes everything from above but also adds the Xen hypervisor.

PFE Slave driver is mandatory and configured by `GoldVIP-specific distributions`.
TF-A BL3X image authentication is optional and is not enabled by default. To enable the complete system-level secure boot flow (that includes the TF-A BL3X images authentication), simply add the follwing line in your `build_s32g399ardb3/conf/local.conf file`:

```
DISTRO_FEATURES:append = " secboot"
```

The desired distro can be configured in `build_s32g399ardb3/conf/local.conf` by modyfing the `DISTRO` parameter.

- In order to use the Real-Time Innovations Connext DDS components, one has to expressly accept the terms of RTI Automotive Software Evaluation License Agreement and append the the following line to the `build_s32g399ardb3/conf/local.conf` file:

```
LICENSE_FLAGS_ACCEPTED:append = " commercial_rti-connext-dds"
```

Note: This step can be skipped if the RTI Connext DDS components are removed from the image.

- Install the Adaptive Autosar depedencies and add layers to workspace:
  1. Download the EB corbos AdaptiveCore package from nxp.com and follow the instructions found in Preparing the Adaptive AUTOSAR source files to install and prepare the EB AdaptiveCore source files and recipes.
  2. Add the `meta-eb-ara` layer to the Yocto workspace:

```
$ bitbake-layers add-layer \
~/ara-corbos-AdaptiveCore-deliveries/ara_Integration/impl/yocto-basic/meta-eb-ara
```

  3. To enable the build of the EB corbos AdaptiveCore components, one has to expressly accept the terms of EB License Terms and Conditions and append the the following line to the `build_s32g399ardb3/conf/local.conf` file:

```
LICENSE_FLAGS_ACCEPTED:append = " commercial_elektrobit"
```

Notes:

– These steps can be skipped if the EB AdaptiveCore components are removed from the image.

  - If the EB AdaptiveCore dependencies were not installed in the default path, then the arguments provided to the `bitbake-layers add-layer` command shall be updated accordingly.
- Download GoldVIP binaries from your nxp.com account and append the following line to the file `build_s32g399ardb3/conf/local.conf`:

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

- Download the HSE firmware release referenced in the Software prerequisites chapter and append the following line to the `build_s32g399ardb3/conf/local.conf` file:

```
NXP_FIRMWARE_LOCAL_DIR = "<path to the local directory that contains the HSE FW
 deliverables>"
```

Notes:

- To remove any of the optional GoldVIP features, append `DISTRO_FEATURES:remove = "<GOLDVIP_FEATURE>"` to the `build_s32g399ardb3/conf/local.conf` file 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-benchmark"
DISTRO_FEATURES:remove = "goldvip-bootloader"
DISTRO_FEATURES:remove = "goldvip-cloud"
DISTRO_FEATURES:remove = "goldvip-containerization"
DISTRO_FEATURES:remove = "goldvip-crypto"
DISTRO_FEATURES:remove = "goldvip-dds"
DISTRO_FEATURES:remove = "goldvip-gateway"
DISTRO_FEATURES:remove = "goldvip-ml"
DISTRO_FEATURES:remove = "goldvip-ota"
DISTRO_FEATURES:remove = "goldvip-adaptive-autosar"
```

- To remove SJA1110 firmware, append the following lines to the file `build_s32g399ardb3/conf/local.conf`:

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

- To use internal development GoldVIP repository add the following line in `build_s32g399ardb3/conf/local.conf`:

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

24.2.3 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_s32g399ardb3/tmp/deploy/images/`.

24.2.4 Deploy the image

The file `fsl-image-goldvip-s32g399ardb3.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-s32g399ardb3.sdcard status=progress \
bs=1M conv=fsync,notrunc of=/dev/<sd-device> && 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 house keeping 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.

24.2.5 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_s32g399ardb3/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.

24.2.6 Build the software development kit

Bitbake offers the capability of creating an SDK installer script for each recipe. This script can then be used to install an SDK that facilitates the development of application software for the GoldVIP target.

For example, in order to create an SDK for the DomU filesystem of the GoldVIP target, run:

```
$ bitbake fsl-image-goldvip-domu -c do_populate_sdk
```

The result is a large script found in `build_s32g399ardb3/tmp/deploy/sdk/`, which can be used to install the SDK.

25 Support

Our S32G GoldVIP community:

- https://community.nxp.com/t5/S32G/bd-p/S32G

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

26 Note about the source code in the document

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

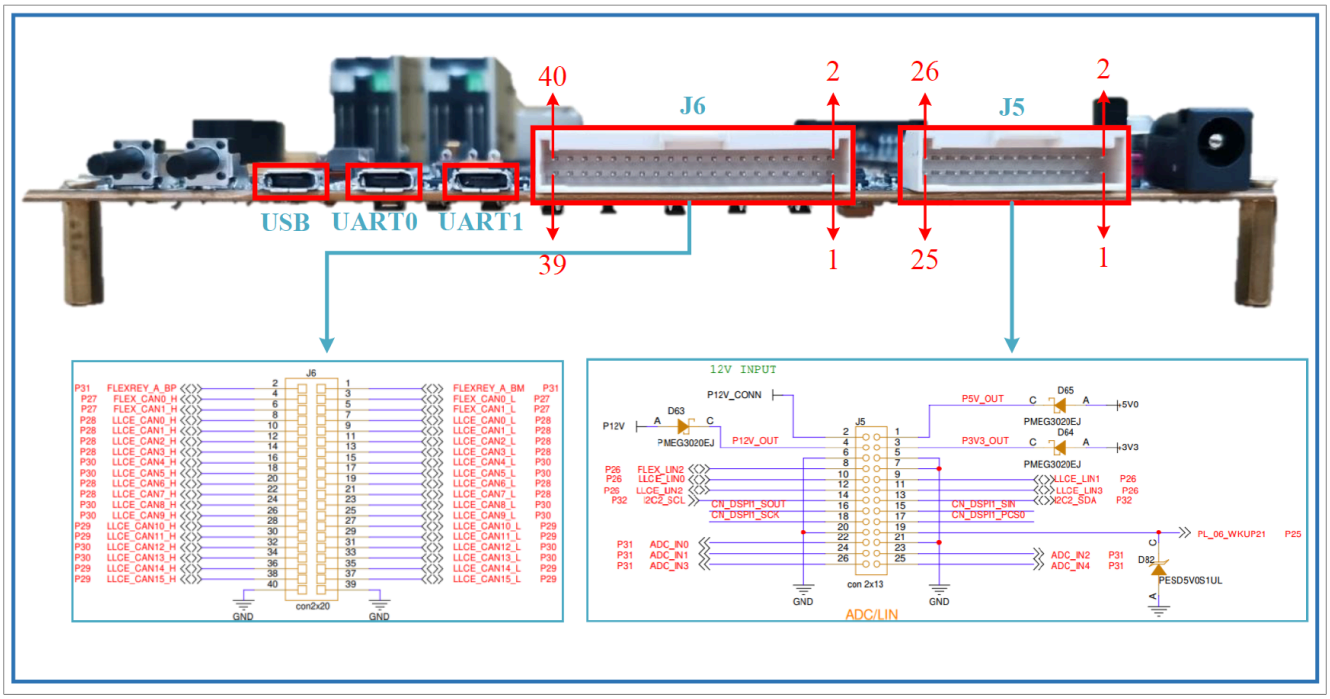
Copyright 2023 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

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

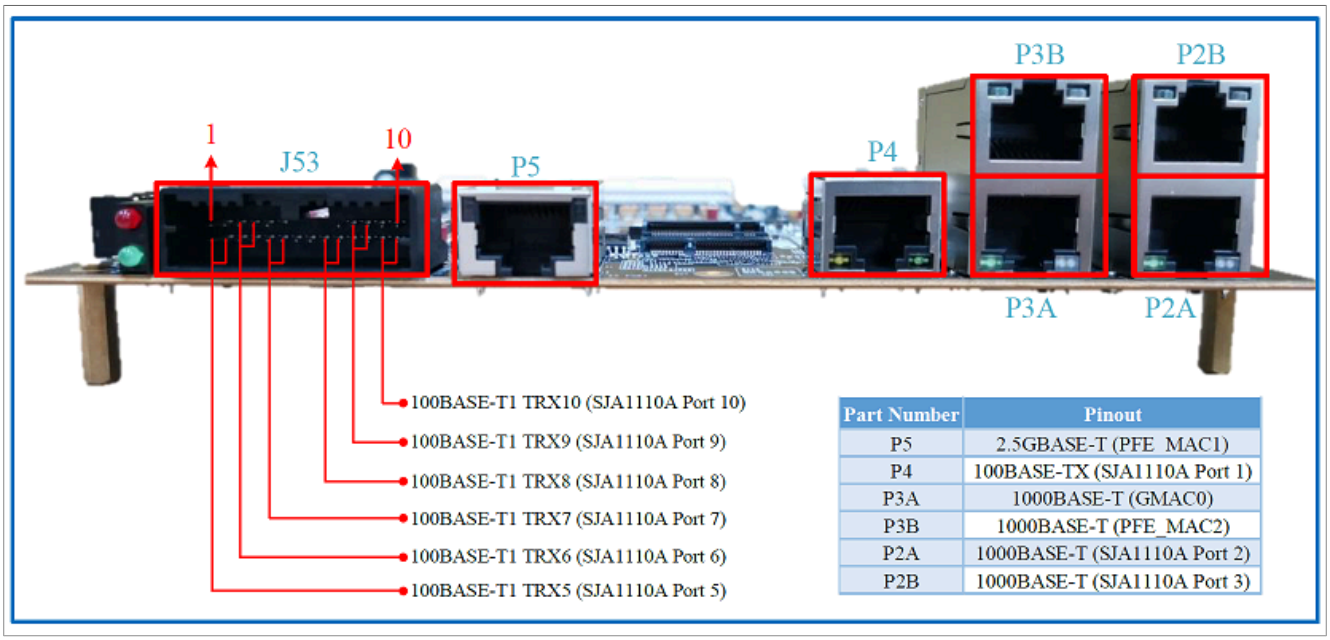
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE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

27 APPENDIX A: S32G-VNP-RDB3 connectors

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



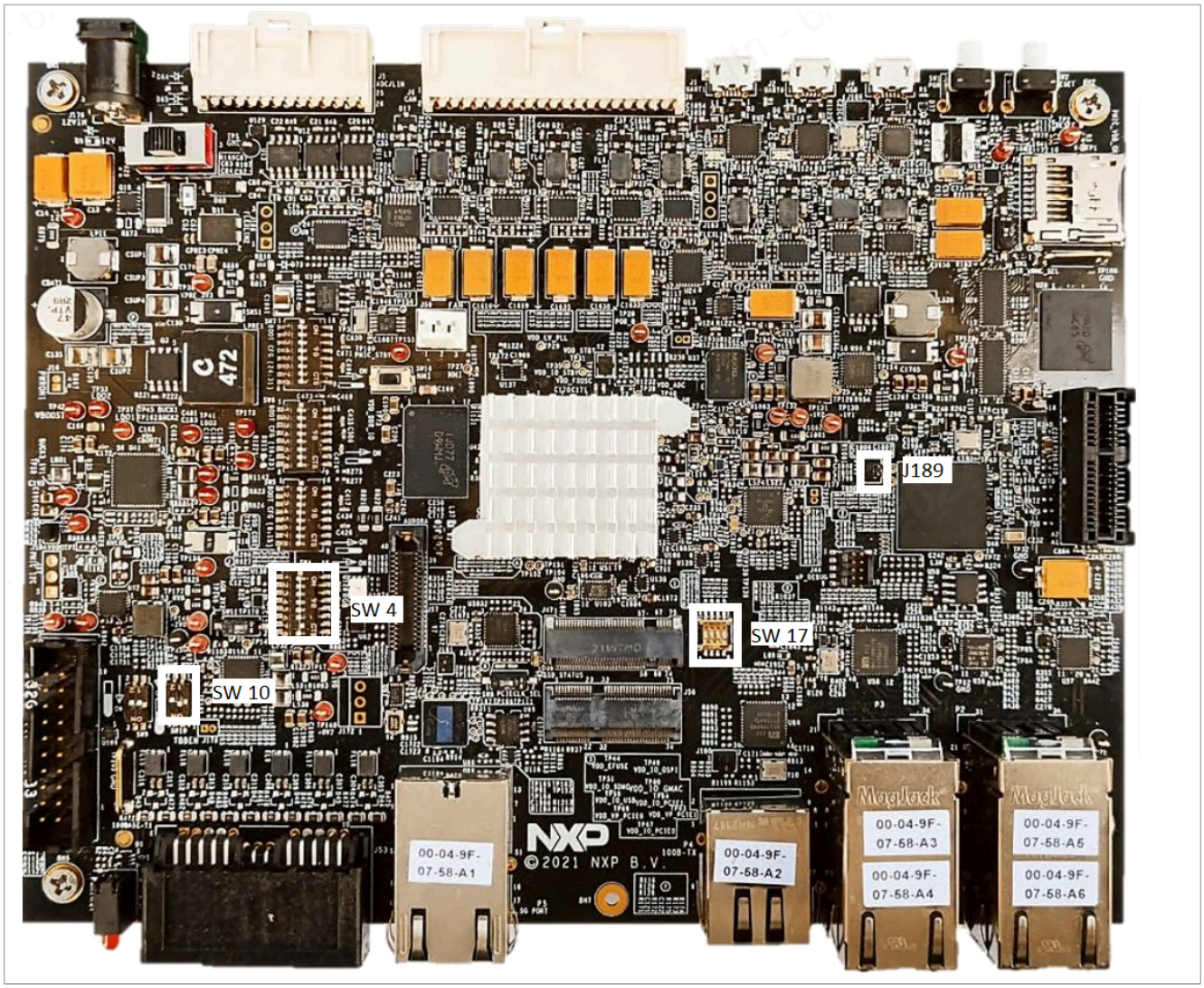
The following figure shows the Ethernet connectors.



Part Number	Pinout
P5	2.5GBASE-T (PFE_MAC1)
P4	100BASE-TX (SJA1110A Port 1)
P3A	1000BASE-T (GMAC0)
P3B	1000BASE-T (PFE_MAC2)
P2A	1000BASE-T (SJA1110A Port 2)
P2B	1000BASE-T (SJA1110A Port 3)

28 APPENDIX B: S32G-VNP-RDB3 DIP switches

The following figure shows the DIP switches position on the RDB3 board.



29 Revision History

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

Revision	Date	Description
1.13.0	12/2024	• Adoption of various new dependencies: Linux BSP 43.0, HSE Firmware 0.2.51.0, Automotive RTD 5.0.0, Automotive Crypto Drivers 5.0.0 QLP01, LLCE 1.0.9, PFE MCAL Driver 1.5.0, PFE Firmware 1.10.0, SAF 2.0.2 QLP01 • Consolidate the image loading functionality using the HSE SMR service on both S32G2 and S32G3 platforms • Implement the secure-boot feature at system level • Remove the eIQ Auto use cases
1.12.0	06/2024	• Adoption of various new dependencies: Linux BSP 41.0, LLCE 1.0.8, IPCF 4.10.0 • Support for dynamic boot configuration in the Bootloader application. • Support for storing the AWS IoT Client Devices certificates in OP-TEE's RPMB Secure Storage. • Add new Linux distribution flavor without Xen hypervisor.
1.11.0	01/2024	• Adoption of various new dependencies: Linux BSP 39.0, PFE FW 1.8.0, PFE MCAL Driver 1.3.0 • OP-TEE component is enabled in the default configuration
1.10.0	11/2023	• Adoption of various new dependencies: Linux BSP 38, PFE FW 1.7.1, PFE MCAL Driver 1.2.0, RTD 4.0.2 HF02, SAF 2.0.1, HSE 0.2.22.0, LLCE FW 1.0.7, IPCF 4.9.0, eIQ Auto 3.6.0, AWS IoT FleetWise 1.0.8. • AUTOSAR Adaptive Platform support via the integration of the EB corbos Adaptive Core solution. • Example integration of the System-Level Bootloader component.
1.9.0	09/2023	• Integration of the RTI Connext Drive DDS Framework in the Real-time Gateway Application • Platform Health statistics available in the Cloud • Adoption of various new dependencies: Linux BSP 37, PFE FW 1.7.0, PFE MCAL Driver 1.1.0, RTD 4.0.1, SAF 2.0.0 QLP1
1.8.0	07/2023	• GHS compiler support for Gateway application • Cortex-M7 as PFE master instance • Support for RDB3 Rev F
1.7.0	05/2023	• Adoption of various new dependencies: Linux BSP 35, PFE FW 1.5.0, RTD 4.0.0, HSE 0.2.16.1, Platform Software Integration 2023.02 • Integration of benchmark code based on EEMBC CoreMark for both M7 and A53 domains • Ethernet communication between M7 and A53 domains through the PFE bridge
1.6.0	02/2023	• Adoption of various new dependencies: EB tresos AutoCore 8.8.7, eIQA 3.5.0, AWS IoT Greengrass 2.9.1, Linux BSP 34, PFE FW 1.4.0, PFE Mcal Driver 1.0.0, LLCE FW 1.0.5, RTD 3.0.4, SAF 1.9.0 • Aggregation of telemetry data through DDS, using RTI Connector for Python • Startup Safety checks enabled in Bootloader • Support for ETH routing on the M7 core • Support for CAN2ETH use cases
0.9.0	11/2022	• GoldVIP support for S32G3 platform

Legal information

Definitions

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

Disclaimers

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

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

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

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

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

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

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

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

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

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

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

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

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

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

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

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

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

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

Trademarks

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

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

Contents