Commsignia V2X Communication and Security Software Stack and SDK Description

Document version: 23-03
Software version: y20
Publication date March 9, 2023
Document ID: STK-002

Table of Contents

List of Figures

List of Tables

1. Introduction

Vehicle-to-Everything (V2X) is a technology that enables the communication between vehicles as well as with their surrounding traffic infrastructure to relay safety, positioning, and traffic efficiency related information. The technology is based on decentralized communication between onboard units (OBUs) and roadside units (RSUs) that transmit and receive standardized signals over IEEE 802.11p or Cellular-V2X radio channels.

1.1. Introduction to the software stack

The Commsignia V2X software stack is a modular system that handles the communication of standardized V2X messages with security features built in to its core. The stack is platform agnostic and it can run independently on any compatible V2X hardware with various operating systems. It is based on a modular architecture, compliant with the ISO/CEN/ETSI/IEEE/SAE ITS architectural standards. Its main building blocks are interfacing with sensors (for example, the global navigation satellite system (GNSS) module) and transmitters (such as the IEEE 802.11p radio). Network protocols, management, security, and facility services also interface with applications. For more information, see section "Service modules and their architecture" [19].

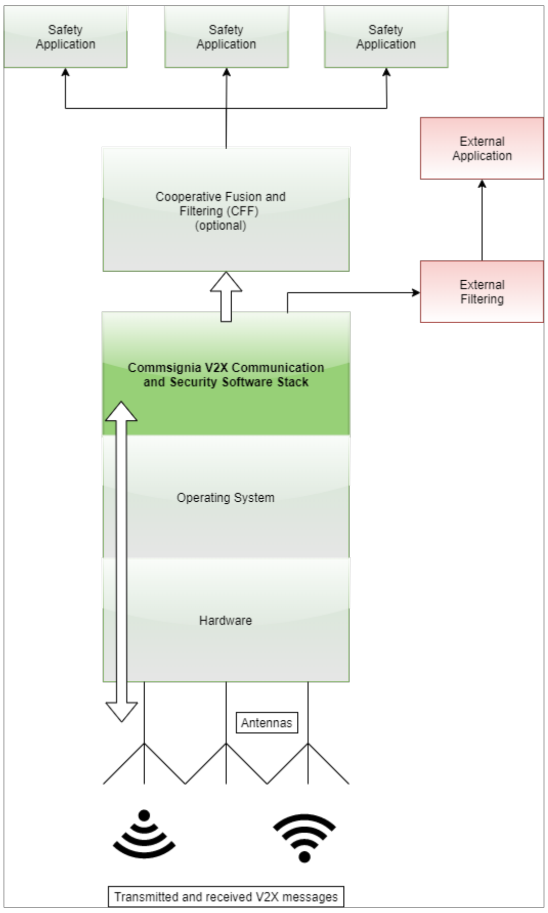

Safety Application
Safety Application
Safety Application
External Application
Cooperative Fusion and Filtering (CFF) (optional)
External Filtering
Commsignia V2X Communication and Security Software Stack
Operating System
Hardware
Antennas
Transmitted and received V2X messages

Figure 1. Relation of the software stack to the rest of the V2X architecture

The main role of the stack is to act as an orchestrator between interfaces linked to the hardware and the operating system, and the upper layers such as the safety applications or the Cooperative Fusion and Filtering (CFF) software. The CFF is another solution offered by Commsignia, which acts as a middleware layer between the software stack and the applications. Customers can also create their own filtering software and external applications.

Security is built in as a core feature; thus, the software stack handles all security certificates as well as the signing and verification of V2X messages. For more information on the implementation of the V2X security, see section "V2X Security in the software stack" [40].

Commsignia also provides a versatile application programming interface (API) to all service modules, enabling convenient development and customization of your own human–machine interface (HMI) and/or safety applications.

1.2. V2X message handling in the stack

The stack is capable of handling all received messages. The messages pass through multiple layers of filtering and routing services (networking, security, and facility layers) as shown in Figure 2. After the facility services verify the correct message content, it can be extracted from the stack for further aggregation and filtering to be used for external applications. The CFF software offered by Commsignia is a solution designed specifically for this purpose; however, the data extracted from the stack can be used by any compatible custom-made filtering software.

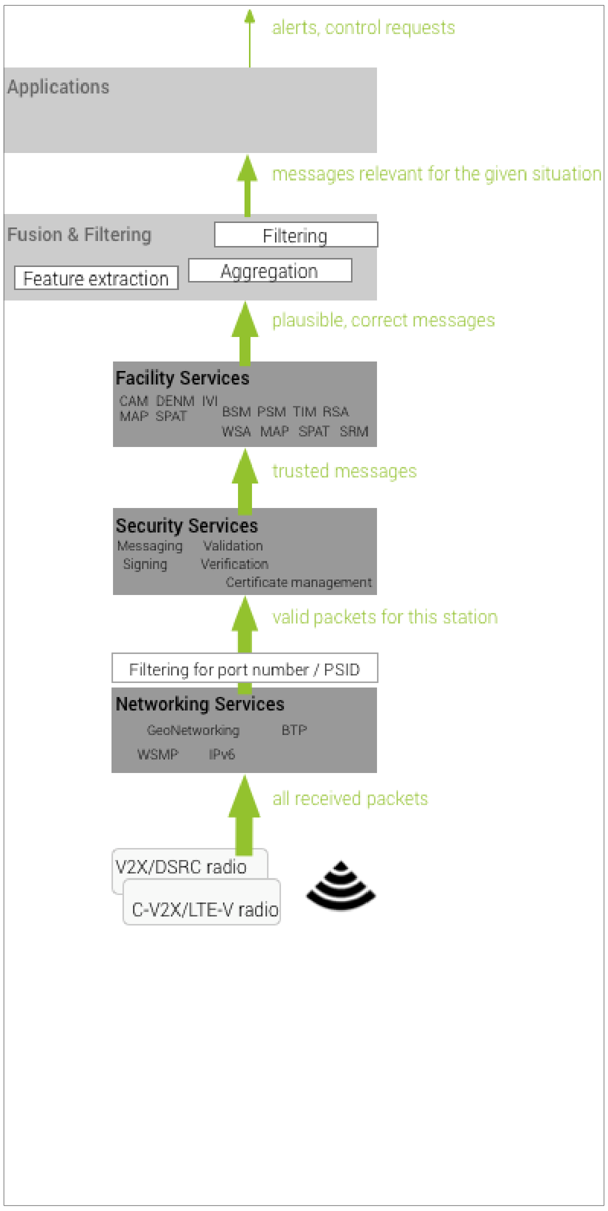


Figure 2. Routing and filtering of the received messages

1.3. Requirements of running the stack

- The host platform must be an IEEE 802.11p radio or C-V2X enabled hardware platform (as specified in [3GPP Rel. 14 PC5]) with one or more CPUs and an operating system to run the software stack.
- The stack itself can be provided by Commsignia as a pre-compiled or pre-flashed binary and delivered on the various supported platforms. More information on the binaries can be found in the Release Note attached to each release.
- The Unplugged-RT software stack also requires a license key for activation. This may be provided separately, part of a software delivery, or already included in the platform configuration (if the software comes bundled with a Commsignia platform).

1.4. Introduction to the software development kit (SDK)

The Commsignia SDK is a library to access low level functionality of the Commsignia V2X stack. It handles the connection to the stack through a proprietary Transmission Control Protocol (TCP) and sends remote procedure call (RPC) commands. Commsignia provides an application programming interface (API) access using this developer kit to certain functions of the software stack.

1.4.1. External API Server

Commsignia provides a client for network communication to the service modules of the software stack.

The External API Server (EAS) is a service module that starts a TCP server on a single, specified port, that uses a proprietary protocol to access the interfaces of the other service modules of the software stack. Commsignia provides a client for the EAS in Python and C languages and that can be integrated into custom application. Using the Remote API enables TCP-based connections between the client of the external application and the service modules of the stack, through the EAS.

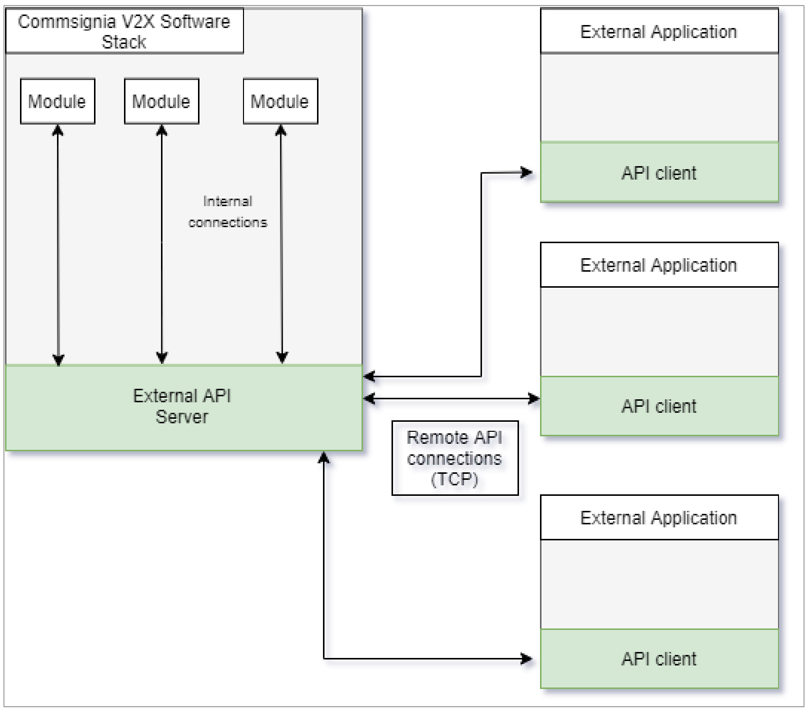


Figure 3. EAS connections within the stack to each available module

All connected clients are considered trusted for connection purposes; thus, the EAS does not filter for potentially sensitive calls or client source addresses. Configuring a firewall for this connection is the responsibility of the integrator.

The binary protocol of the EAS is subject to change in each release; make sure that all connected clients use the same software version as the software stack. Updating the client library does not require a source change (except upon a major version change), only the recompilation of the client for the new library.

2. Getting started with the Commsignia V2X Software Stack

Deploy the software stack on a compatible hardware device with a preset configuration and validate its successful operation.

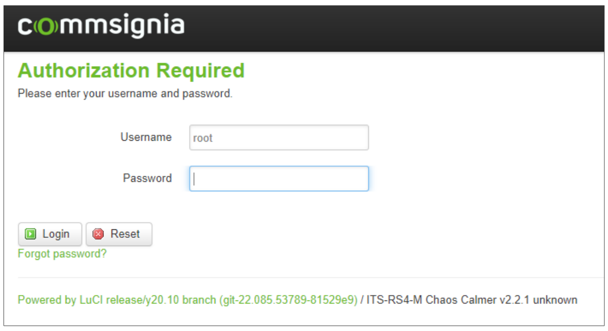
- For validating the examples, at least one compatible V2X device is required.
- To connect to the device, a computer running Ubuntu Linux LTS 18.04 or Windows is required.
- To send out signed messages, a certificate pack with a root certificate authority (CA) certificate is required. For testing purposes, the certificate pack provided by Commsignia can be used.

2.1. Connecting to the device

To configure the features of the Commsignia V2X software stack, a connection needs to be established to at least one V2X device running the stack. The device can be accessed via wireless or wired connection using a graphical user interface (GUI) or a command line interface (CLI).

Please note that the device has two separate IP addresses for wireless and wired connections. Please ensure that your computer is connected to the same wireless or wired network as the RSU. All passwords are case-sensitive.

1. Connect all antennas and accessories to your hardware before powering it up. For more information, refer to the hardware description of your device.
2. Connecting to the device over wireless (Wi-Fi) network:
   If your device is provided by Commsignia, then its SSID is **ITS-DEV-XXXXXXX**, where DEV is OB4 or RS4 if it is an OBU or RSU, respectively, and XXXXXXX is the last seven digits of the serial number of the device, which can be found on the product label. The default Wi-Fi password is **Commsignia**.
   a. In a web browser enter the IP address of the device, which is **192.168.1.54** by default. Use the username **root** and enter the default password, **UK5BJLFZVBPZLIM55Y**, to log in, as shown in Figure 4.



Authorization Required
Please enter your username and password.
Username: root
Password:
Powered by LuCI release/y20.10 branch (git-22.085.53789-81529e9) / ITS-RS4-M Chaos Calmer v2.2.1 unknown

Figure 4. Login screen of the GUI

   b. Alternatively, an SSH connection can be established from the CLI as

```
ssh root@192.168.1.54
```

   and entering the same root password, **UK5BJLFZVBPZLIM55Y**, when prompted.
3. Connecting to the device over wired (Ethernet) connection:
   a. Please ensure that your computer is connected to the same wired network as the device. Your computer needs to use the same subnet as the Commsignia default of the device.

b. In a web browser enter the IP address of the device, which is **192.168.0.54** by default. Use the user name **root** and enter the default password, **UK5BJLFZVBPZLIM55Y**, to log in, as shown in Figure 4.
c. Alternatively, an SSH connection can be established from the CLI as

```
ssh root@192.168.0.54
```

and entering the same root password, **UK5BJLFZVBPZLIM55Y**, when prompted.

4. It is recommended to change the root user password after the first successful login.
   a. To change the password using the GUI, open the **System** → **Administration** menu item and change the administrator password of the router.
   b. To change the password using the GUI, use the command `passwd` and change the password, when prompted.

2.2. Enabling secure message transmission

2.2.1. Enabling security

To enable security on the device proceed as follows:

1. Copy your certificate pack to its designated folder on the device. Commsignia provides its own snake oil certificate pack with a root Certificate Authority (CA) certificate that can be used for testing purposes. Messages sent out with this certificate pack are only recognized as trusted by other devices if they have the same Commsignia root CA certificate. For more information, see section "V2X Security in the software stack" [40].
2. Enable security on the device using the GUI:
   a. Log into the GUI and open the **V2X Core** → **Core stack** menu item.
   b. Expand and check the box next to **Security configuration**.
   c. Check the box next to `Enable security` and set its value to `Yes`.
3. Enable security on the device using the CLI:
   a. Log into the device using SSH.
   b. Open the `/rwdata/etc` directory from a terminal.
   c. Using an editor, open the `its.json` file and add the following item:

```
{
 ...
  "security": {
   "enable": "Yes",
   "checkLoadedCertificates": true
  },
 ...
}
```

If security is enabled and valid signer certificates are not available, then the transmission will be rejected. If security is disabled, then all outgoing messages will be sent with "unsecured" headers. Auto mode means that if certificates are present and can be loaded, then the security module will be turned ON (if certificates are invalid, then messages will not be sent). If they are not present, then the module will be turned OFF.

2.2.2. Enabling pseudonymity

The pseudonymity module enables the pregular changing of MAC addresses, identifiers, and certificates.

To enable pseudonymity the security module needs to be enabled as well.

To enable pseudonymity on the device proceed as follows:

1. Enable pseudonymity the device using the GUI:
   a. Log into the GUI and open the **V2X Core** → **Core stack** menu item.
   b. Expand and check the box next to **Pseudonymity module configuration**.
   c. Check the box next to `Enable pseudonymityE` and set its value to `Yes`.
2. Enable security on the device using the CLI:
   a. Log into the device using SSH.
   b. Open the `/rwdata/etc` directory from a terminal.
   c. Using an editor, open the `its.json` file and add the following item:

```
{
 ...
  "pseudonymity": {
    "enable": "Yes"
  },
 ...
}
```

If pseudonymity is disabled, all outgoing messages will be sent with a fixed station ID and no automatic certificate change will occur.

3. Installing the SDK

The Commsignia SDK is a library to access low level functionality of the Commsignia V2X stack. It handles the connection to the stack through a proprietary TCP protocol and sends remote procedure call (RPC) commands. Commsignia provides an API access using this developer kit to certain functions of the software stack. Code examples are available in both Python and C programming languages.

To use the SDK, the following items are required:

1. A computer running Ubuntu Linux 18.04 or later operating system that is connected to a V2X device running the Commsignia Software Stack.
2. The Commsignia V2X Software Stack installed, licensed and running on a compatible V2X device. For more information, see section "Getting started with the Commsignia V2X Software Stack" [5].
3. The Remote SDK package provided by Commsignia. The SDK package contains the headers, libraries, examples, and documentation required for development for the software stack.

3.1. Installing the Python SDK

To install the Python SDK, the following items are required:

- The Unplugged-RT Remote Python SDK package provided by Commsignia. The SDK package contains libraries, modules, and examples required for development for the software stack.
- To install the Python package, the pip3 package manager is required; the version can be checked using the `pip3 --version` command.
- To run the Python examples, Python 3.7 or later is required; the version can be checked using the `python3 --version` command.

To install the Python SDK, proceed as follows:

1. Open a terminal.
2. Extract the Unplugged-RT Remote Python SDK package using the following command:

```
tar xf Unplugged-RT-<Release-version>-pythonsdk.tar.xz
```

   Here, `<Release-version>` is the applicable release version number of the SDK package provided by Commsignia.
3. Open the directory of the extracted files and use the pip3 package manager to install the Python modules as:

```
cd Unplugged-RT-<Release-version>-pythonsdk
pip3 install pycmssdk-<Release-version>-py3-none-any.whl
```

4. To test the installation, open the directory `./examples` and run a `.py` file, for example, `get_status.py` as:

```
cd examples
python3 get_status.py
```

Please note that in the example files the IP address of the localhost ("127.0.0.1") is used. The IP address of the device can be entered directly in the code after '`host=`' or modify the code such that it reads the argument after the command (e. g., "`host=sys.argv[1]`").

The successful execution of the code can be verified using the Commsignia Capture Protocol C2P interface (see section "Commsignia Capture Protocol (C2P) module" [33]) of the stack to capture sent or received messages or by any other diagnostics tool of your choice.

3.2. Installing the C SDK

To install the C SDK the following items are required:

- The Unplugged-RT Remote C SDK package provided by Commsignia. The SDK package contains the headers, libraries, examples, and documentation required for development for the software stack.
- A developer environment capable to make executable files from `.c` source codes and run them, such as `build-essentials`, `cmake`, etc.

1. Open a terminal.
2. Extract the Unplugged-RT Remote C SDK package using the following command:

   ```
   tar -xf Unplugged-RT-<Release-version>-linuxm_generic_F-
   remote_c_sdk.tar.xz
   ```

   Here, `<Release-version>` is the applicable release version number of the SDK package provided by Commsignia.
3. The SDK needs to be initialized before its first use by issuing the `cmake` command. Open the extracted directory and create a directory (such as `scratch`) and open it:

   ```
   cd Unplugged-RT-<Release-version>-linuxm_generic-remote_c_sdk
   mkdir scratch
   cd scratch
   ```

   Open the directory of the extracted files and open the directory `./examples`. There is a **.c** file and a **Makefile** in each directory. Use the **make** command to compile an executable from the `.c` codes. See the example below for compiling a code sample:
4. Then issue a `cmake` command as:

   ```
   cmake ../
   ```
5. Open the `./examples` directory and use the command `make` to compile executables from the `.c` files:

   ```
   cd examples
   make
   ```

   Use the command `make` for any subsequent compilations.
6. To test the build, access the directory `/examples`, and run an executable file and providing the IP address of the V2X device, for example

   ```
   ./get_version 192.168.1.52
   ```

The successful execution of the code can be verified using the C2P interface (see section "Commsignia Capture Protocol (C2P) module" [33]) of the stack to capture sent or received messages or by any other diagnostics tool of your choice.

4. SDK usage and examples

The Commsignia SDK also provides examples for typical use cases both in Python and C programming languages. In this section the structure and implementation of these applications are detailed using certain examples.

4.1. Python SDK

4.1.1. Connecting to the Commsignia V2X Stack using a basic API command

The following example illustrates a basic API call that enables connection to the Commsignia software stack and get and print the version of the stack.

1. Import the following Python module for the API:

   ```
   from pycmssdk import create_cms_api
   ```

2. Create an API session:

   ```
   with create_cms_api(host="192.168.0.54") as api:
   ```

   Here, the IP of the device is `192.168.0.54`. There can be multiple concurrent API sessions. Creating the session does not provide a connection to the target stack process on its own.
3. Get the current stack status by calling the following API function:

   ```
       response = api.stat_get_device_status()
   ```

4. To print the stack version, use the `print` function:

   ```
       print("The version of the running stack:",
             response.data.device_data.version_info)
   ```

5. Run the code to print out the version of the stack.

4.1.2. Integrating vehicle data to enrich sent V2X messages

To integrate proprietary vehicle data extracted from the CAN interface into the software stack, it needs to be transformed and forwarded to the software stack before it can handle and use it to transmit V2X messages containing vehicle data (for example BSM or CAM).

The CAN bus of a vehicle has to be connected to a hardware capable of extracting data over an interface (for example SocketCAN for Linux).

To integrate vehicle data to the stack, proceed as follows:

1. Extract the vehicle data according to your own CAN scheme.
2. Convert the proprietary CAN data to the appropriate format, usable by the stack. Check the Station Information (STI) module for more information about the acceptable data types.
3. Import the following Python modules:

   ```
   from pycmssdk import (
       StiItem,
       StiNotifData,
   ```

```
    StiSetItems,
    StiType,
    StiTypeList,
    create_cms_api,
)
```

4. Use the `api.sti_set` function to include the collected CAN data in the station information that is processed by the stack to create messages that contain vehicle data. Multiple data points can be set at the same time. However halting the process should be avoided until a certain amount of data is collected; send as many data points over to the stack as many you receive. For further reference see the code example below:

```
api.sti_set(
    StiSetItems(
        items=(
            StiItem(type=StiType.STI_STEERING_WHEEL_ANGLE, value=12),
            StiItem(type=StiType.STI_VEHICLE_LENGTH, value=4000),
            )
        )
    )
```

5. Run the code to integrate the data. A detailed example file `sti.py` is also available.

Sending of the V2X messages (such as CAM, BSM, PVD) containing vehicle information obtained from CAN can be validated by checking the reception of the messages on another V2X device or by using the C2P tool (see section "Commsignia Capture Protocol (C2P) module" [33]).

4.1.3. Sending custom V2X messages

Custom messages can be used for drafting standards or validating unstandardized advanced use cases (such as platooning, telemetric data sending, or other forms of cooperative messages). Sending out custom data can be achieved by wrapping it in a standard V2X message network layer packet (for example GeoNetworking or WSMP). The software stack offers a process to create and send out custom V2X message content by wrapping it with standard network layer elements.

To send out custom V2X messages,—in this example a security signed WSMP message—proceed as follows:

1. Create a custom sender application, integrated with the Commsignia SDK and use it to collect the custom data packet that needs to be sent out. Make sure not to exceed the maximum transmission unit (MTU) of the V2X radio technology used for transmission (for example DSRC).
2. Add the network header to the custom message data, as defined in the applicable V2X standard (for example SAE.... for WSMP or ETSI.... for GeoNetworking).
3. If necessary, enable security validation for the sent custom messages. This requires a valid certificate; see section Security implementation for more details.
4. Import he following Python modules:

```
from pycmssdk import (
    WILDCARD,
    MacAddr,
    RadioTxParams,
    SecDot2TxInfo,
    SecDot2TxSignInfo,
    SignMethod,
    WsmpSendData,
    WsmpTxHdrInfo,
    WsmpTxNotifData,
    create_cms_api,
)
```

5. Define a callback function that handles WSMP transmission:

```
def wsmp_tx_callback(key: int, data: Any, buffer: bytes) -> None:
   //...
```

6. Create a WSMP send header as:

```
    with create_cms_api(host="127.0.0.1") as api:
        api.wsmp_tx_subscribe(WILDCARD, wsmp_tx_callback)
        psid = 130
        send_data = WsmpSendData(
            radio=RadioTxParams(interface_id=1,
                dest_address=MacAddr(0xFF, 0xFF, 0xFF,
                                     0xFF, 0xFF, 0xFF)),
            wsmp_hdr=WsmpTxHdrInfo(psid=psid),
            security=SecDot2TxInfo(sign_info=SecDot2TxSignInfo(
                sign_method=SignMethod.SIGN_METH_SIGN_CERT,
                psid=psid)),
        )
```

7. Call the `api.wsmp_send` function:

```
        api.wsmp_send(send_data, buffer=b"\x01\x02\x03\x04")
        time.sleep(1)
```

8. Run the code to sent out the custom message over the V2X radio. An example code `wsmp_send.py` is also avaliable.

4.1.4. Processing received facility messages

The stack provides an option to handle received custom V2X messages to use them for creating custom applications. To subscribe to a certain type of custom message required for an application, proceed as follows:

1. Import the following Python modules:

```
from pycmssdk import WILDCARD, FacNotifData, create_cms_api
```

2. Define a callback class that handles the received notification:

```
class FacRxCtx:
    def __init__(self):
       //...

    def __call__(self, key: int, data:
                 FacNotifData, buffer: bytes) -> None:
       //...
```

3. Create a subscription through the API as:

```
    with create_cms_api(host="127.0.0.1") as api:
        api.fac_subscribe(WILDCARD, FacRxCtx())
```

4. As the code is event/notification driven, insert a dummy while cycle to prevent the code from terminating:

```
        while True:
            time.sleep(1)
```

5. Run the code to receive raw V2X messages forwarded by the stack. Messages can be validated by printing out their content. An example code `fac_subscribe.py` is also available.

4.1.5. Processing received custom messages

The stack provides an option to handle received custom V2X messages to use them for creating custom applications. To subscribe to a certain type of custom message required for an application,—in this example a WSMP message—proceed as follows:

1. Import the following Python modules:

```
from pycmssdk import WsmpRxNotifData, create_cms_api
```

2. Define a callback class that handles the received notification:

```
def wsmp_rx_callback(key: int, data: Any, buffer: bytes) -> None:
    print("------ Received WSMP Rx notification ------")
    print(f"PSID: {hex(key)}")
    print_wsmp_rx_notif_data(data)
    print("-------------------------------------------")
```

3. Create a subscription throught the API as:

```
    with create_cms_api(host="127.0.0.1") as api:
        FILTERPSID = 0x20
        api.wsmp_rx_subscribe(FILTERPSID, wsmp_rx_callback)
```

4. As the code is event/notification driven, insert a dummy while cycle to prevent the code from terminating:

```
        while True:
            time.sleep(1)
```

5. Run the code to receive raw V2X messages forwarded by the stack. Whenever a custom message is received, the stack forwards it to the pointer that you specified whenever they are received. Messages can be validated by printing out their content. An example code `wsmp_rx_subscribe.py` is also available.

   Please note that unsigned and unvalidated messages are also received and forwarded.

   For secure custom messages, make sure that the `data.security.verify_result` field contains appropriate value.

4.2. C SDK

4.2.1. Connecting to the Commsignia V2X Stack using a basic API command

The following example illustrates a basic API call that enables connection to the Commsignia software stack and get and print the version of the stack.

1. Add the base include file for the API:

```
#include <cms_v2x/api.h>
```

2. Add the include file for the device status function:

```
#include <cms_v2x/stat.h>
```

3. In the initialization phase (at the beginning of `main`) create an API session:

```
cms_session_t session = cms_get_session();
```

There can be multiple concurrent API sessions and the `session` variable is used to differentiate between them. Creating the session does not provide a connection to the target stack process on its own.

4. Connect to the stack process as

```
cms_api_connect_easy(&session, "192.168.0.54");
```

Here , the first parameter is the session, the second is the IP address of the device.
The call may return with an error flag. In this case an error message is logged on the client side and possibly by the stack process as well. The first connection also automatically initializes all global states of the API and starts the background threads.

5. To get the current stack status, call the following API function:

```
cms_stat_get_device_status(&session, NULL, &data);
```

Here, the first parameter is the session, the second is an unused input parameter. This is necessary because all API calls follow the same format: session, input, output are always added even if there are no inputs for the call. This is implemented to ensure backwards compatibility. The third parameter is filled by the device status.
The call may return with an error flag. In this case an error message is logged on the client side and possibly by the stack process as well.

6. To print the stack version, use the `printf` function:

```
printf("The version of the running stack: %s\n", ►
data.device_data.version_info);
```

7. Close the connection and clean up the session:

```
cms_api_disconnect(&session);
```

8. Clean up the rest of the API resources, such as the background threads and global data structures:

```
cms_api_clean();
```

If there are subsequent `cms_api_connect` calls, do not call this function so that the global API structures remain intact and they do not need to be reinitialized.

9. To print the version of the stack compile the code using the `make` command and run the compiled executable.

4.2.2. Integrating vehicle data to enrich sent V2X messages

To integrate proprietary vehicle data extracted from the CAN interface into the software stack, it needs to be transformed and forwarded to the software stack before it can handle and use it to transmit V2X messages containing vehicle data (for example BSM or CAM).

> The CAN bus of a vehicle has to be connected to a hardware capable of extracting data over an interface (for example SocketCAN for Linux).

To integrate vehicle data to the stack, proceed as follows:

1. Extract the vehicle data according to your own CAN scheme.

2. Convert the proprietary CAN data to the appropriate format, usable by the stack. For more information on the acceptable data types, please refer to section "Station Information (STI) module" [22].
3. Use the `cms_sti_set` function to include the collected CAN data in the station information that is processed by the stack to create messages that contain vehicle data. Multiple data points can be set at the same time. However, halting the process should be avoided until a certain amount of data is collected; send as many data points over to the stack as many you receive. For further reference see the code example below:

```
cms_sti_set_items_t set_in = {0};
set_in.items[0].type = CMS_STI_VEHICLE_LENGTH;
set_in.items[0].value = 2500LL;
set_in.length = 1UL;
cms_sti_set(&session, &set_in, NULL);
```

4. Compile using `make` and run it. A detailed example file `sti_set_get.c` and the corresponding executable is also available.

Sending of the V2X messages (such as CAM, BSM, PVD) containing vehicle information obtained from CAN can be validated by checking the reception of the messages on another V2X device or by using the C2P tool (see section "Commsignia Capture Protocol (C2P) module" [33]) .

4.2.3. Processing received facility messages

The stack provides an option to handle received facility V2X messages to use them for creating custom applications. To subscribe to a certain type of facility message required for an application, proceed as follows:

1. Add the following headers:

```
#include <cms_v2x/fac_types.h>
#include <cms_v2x/fac_subscribe.h>
```

2. Use the `cms_fac_subscribe` function to subscribe to a certain type of facility message that needs to be received from the stack (for example MAP). Use the name of the message type as a parameter. Use a pointer to your own code, where the message content from the stack needs to be received:

```
/* Create a context for the subscription callback */
notif_ctx_t ctx = {
        .param = 2U,
        .recv_cnt = 0U
};

/* Subscribe to all messages */
cms_subs_id_t wildcard_subs_id = CMS_SUBS_ID_INVALID;
error = error || cms_fac_subscribe(&session,
                 CMS_FAC_SUBSCRIBE_WILDCARD,
                 l_fac_notif_cb_f,
                 &wildcard_subs_id,
                 &ctx);
if(error)
  { printf("Unable to subscribe to all facility notifications\n");
  // ...
error = error || cms_fac_unsubscribe(&session, wildcard_subs_id);
```

3. As the code is event/notification driven, insert a dummy while cycle to prevent the code from terminating.

4. Compile the code using the `make` command to receive raw V2X messages forwarded by the stack to the specified pointer whenever they are received. Messages can be validated by printing out their content. An example code `fac_subs_notif.c` and a corresponding executable are also available.

4.2.4. How to process received custom messages

The stack provides an option to handle received custom V2X messages to use them for creating custom applications. To subscribe to a certain type of custom message required for an application—in this example a WSMP message—proceed as follows:

1. Add the following header:

```
#include <cms_v2x/wsmp.h>
```

2. Use the `cms_wsmp_rx_subscribe(&session, psid, callback, context, &subscription_id);` function to subscribe to a certain type of custom message that needs to be received from the stack (for example WSMP). Use the name of the message type as a parameter. Use a pointer to your own code, where the message content from the stack needs to be received:

```
static void wsmp_notif_cb(cms_psid_t psid,
                          const cms_wsmp_rx_notif_data_t* notif,
                          cms_buffer_view_t msg,
                          void* ctx)/* Create a context for the ►
subscription callback */




/* Subscribe to a specific PSID */
    static const uint64_t FILTER_PSID = 0x20UL;
    cms_subs_id_t filtered_subs_id = CMS_SUBS_ID_INVALID;
    error = error || cms_wsmp_rx_subscribe(&session,
                                           FILTER_PSID,
                                           &wsmp_notif_cb,
                                           &filtered_ctx,
                                           &filtered_subs_id);

    if(error) {
        printf("Unable to subscribe to WSMP Rx for PSID 0x%llx\n", ►
(unsigned long long)FILTER_PSID);
    //...
    }/* Subscribe to all messages */
    /* Unsubscribe */

    error = cms_wsmp_rx_unsubscribe(&session, filtered_subs_id);
```

3. As the code is event/notification driven, insert a dummy while cycle to prevent the code from terminating.
4. Compile the code using the `make` command to receive raw V2X messages forwarded by the stack to the specified pointer whenever they are received. Messages can be validated by printing out their content. An example code `fac_subs_notif.c` and a corresponding executable are also available.

Please note, that unsigned and unvalidated messages are also received and forwarded.

To secure custom messages, make sure that the `header->security_info.verify_result` field is set to `CMS_SEC_ECDSA_VERIFY_SUCCESS`.

4.2.5. Sending custom V2X messages

Custom messages can be used for drafting standards or validating unstandardized advanced use cases (such as platooning, telemetric data sending, or other forms of cooperative messages). Sending out custom data can be achieved by wrapping it in a standard V2X message network layer packet (for example GeoNetworking or WSMP). The software stack offers a process to create and send out custom V2X message content by wrapping it with standard network layer elements.

To send out custom V2X messages—in this example a security signed WSMP message—proceed as follows:

1. Create your own sender application, integrated with the Commsignia SDK and use it to collect the custom data packet you want to send out. Make sure not to exceed the maximum transmission unit (MTU) of the V2X radio technology that you use for transmission (for example DSRC). Create a custom sender application, integrated with the Commsignia SDK and use it to collect the custom data packet that needs to be sent out. Make sure not to exceed the maximum transmission unit (MTU) of the V2X radio technology used for transmission (for example DSRC).
2. Add the network header to your custom message data, as defined in the V2X standard you want to use (for example SAE.... for WSMP or ETSI.... for GeoNetworking).
3. If necessary, enable security validation for the sent custom messages. This requires a valid certificate; see section Security implementation for more details.
4. Include the following header:

```
#include <cms_v2x/wsmp.h>
```

5. Create a WSMP send header as:

```
int main(int argc, char* argv[])
{
    const char* host = (argc > 1) ? argv[1] : "127.0.0.1";

    /* Create a session */
    cms_session_t session = cms_get_session();

    /* Connect to the host */
    bool error = cms_api_connect_easy(&session, host);

    static const uint8_t BROADCAST_ADDR[CMS_MAC_ADDRESS_LENGTH] = {0xFF, ▶
0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

    /* Create send header */
    cms_wsmp_send_data_t wsmp_send_info = {0};

    wsmp_send_info.radio_info.datarate_kbps = 0;
    memcpy(wsmp_send_info.radio_info.dest_address, BROADCAST_ADDR, ▶
CMS_MAC_ADDRESS_LENGTH);
    wsmp_send_info.radio_info.expiry_time_ms = 0;
    wsmp_send_info.radio_info.interface_id = 1;
    wsmp_send_info.radio_info.sps_index = 0;
    wsmp_send_info.radio_info.tx_power_dbm = 0;
    wsmp_send_info.radio_info.user_prio = 0;

    wsmp_send_info.wsmp_hdr_info.channel_id = true;
    wsmp_send_info.wsmp_hdr_info.datarate = true;
    wsmp_send_info.wsmp_hdr_info.psid = 0x82;
    wsmp_send_info.wsmp_hdr_info.tx_power = true;

    wsmp_send_info.security_info.sign_info.psid = 0x82;
    wsmp_send_info.security_info.sign_info.sign_method ▶
= CMS_SIGN_METH_SIGN_CERT;

    uint8_t msg_payload[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    /* Create a buffer view as a handle to the actual payload buffer */
    cms_buffer_view_t msg = {
        .data = msg_payload,
        .length = sizeof(msg_payload)
```

6. Call the `send` function as:

```



    /* Send the prepared WSMP message */
    error = error || cms_wsmp_send(&session, &wsmp_send_info, msg, NULL);

    /* Close connection and cleanup */
    cms_api_disconnect(&session);
    cms_api_clean();

    return (int)error;
}
```

7. Compile the code using `make` and run it to sent out the custom message over the V2X radio. An example code `wsmp_send.c` and the corresponding executable are also available.

5. Service modules and their architecture

The software architecture of the Commsignia V2X stack consists of various interconnected modules, each providing their own service.

5.1. Architecture overview

The Commsignia V2X software and security stack utilizes a modular architecture to provide various different services related to transmitting and receiving either standard or custom V2X messages. The Security service is an integral part of the stack, its modules are directly connected to each other module. The following image provides an overview about the whole architecture and the rest of this chapter explains the main categories of each service group of the modules.

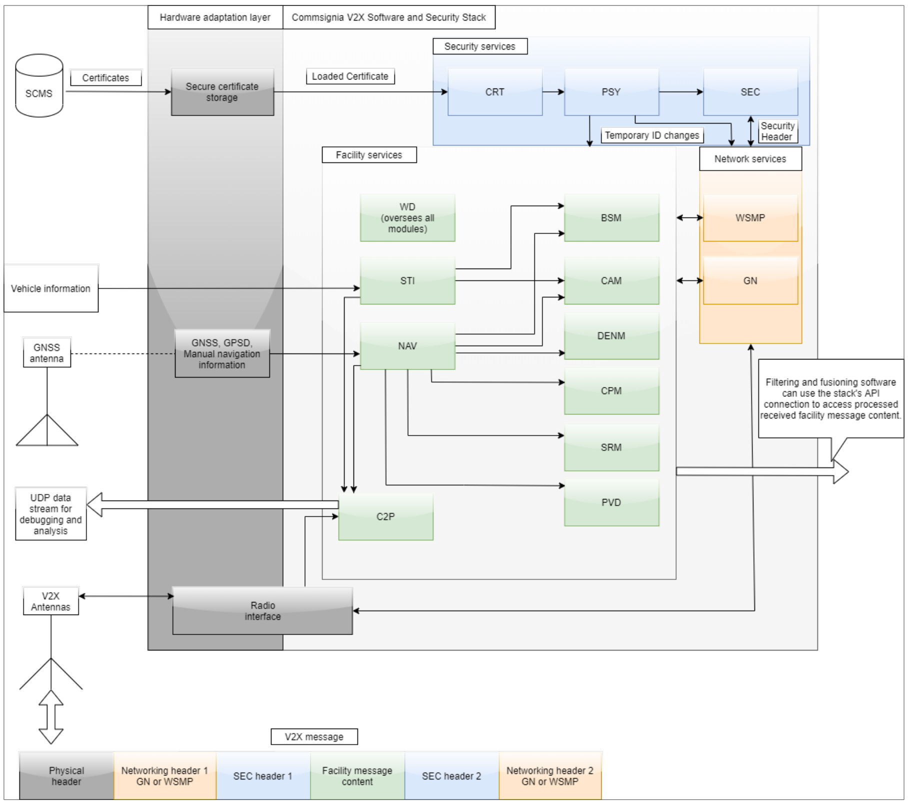

Hardware adaptation layer
Commsignia V2X Software and Security Stack
Security services
SCMS
Certificates
Secure certificate storage
Loaded Certificate
CRT
PSY
SEC
Temporary ID changes
Security Header
Facility services
Network services
WD (oversees all modules)
BSM
WSMP
Vehicle information
STI
CAM
GN
GNSS antenna
GNSS, GPSD, Manual navigation information
NAV
DENM
CPM
Filtering and fusioning software can use the stack's API connection to access processed received facility message content.
SRM
PVD
UDP data stream for debugging and analysis
C2P
V2X Antennas
Radio interface
V2X message
Physical header
Networking header 1 GN or WSMP
SEC header 1
Facility message content
SEC header 2
Networking header 2 GN or WSMP

Figure 5. Architecture of the Commsignia V2X software stack

A Hardware Adaptation Layer (HAL) connects the software stack to the different hardware chips that provide information for the various functions. Hardware connections include:

- Connection to the OS
- GNSS
- Radio
- Private keys stored in the Hardware Security Module (HSM) sign outgoing messages
- Cryptographic accelerator verifies incoming messages

Connection to the Operating System is also handled through an adaptation layer so the software stack can function on various different Operating Systems in a platform-independent manner.

5.2. Network services

Network services provide low-level communication handling for both transmitted and received V2X message packets. The GeoNetworking (GN) module of the software stack will wrap and unwrap the message content based on your regional standard. Incoming messages are recognized by their header and the network module handling it requests the security module to verify it. The security service marks it and returns it to the network module where it is dropped or forwarded to the facility services based on your configuration.

Outgoing messages are wrapped in a standardized header (based on your regional standards) by the network services and the same service requests the security module to sign them, before they are forwarded to the radio for transmission.

This means that not only standard facility messages, but also custom message packets can be sent out wrapped in a standard GN header. For more information about custom messages, see sections "Sending custom V2X messages" [11] and "Processing received custom messages" [13] for Python and "Sending custom V2X messages" [17] and "Processing received custom messages" [13] for C.

5.3. Security services

The modules of the security services of the stack handle signing, verification, encoding, and pseudonymity. The security module communicates directly with the network services to sign outgoing or verify incoming messages. Certification management and pseudonymity are handled separately by their respective modules. For a more detailed explanation, see section "V2X Security in the software stack" [40].

5.4. Facility services

Each facility module acts as an independent service that handles messages based on its configuration. Message contents are assembled and forwarded to the network service automatically, where they are sent over to the security module for signing before transmission.

Incoming facility messages are handled in a unified way: after validation by the security module and reception by the network module. The facility reception service of the stack decodes the received packets and validates their format. The facility modules subscribe to this common service within the stack. For more information, see section "Facility Reception service" [36].

5.5. Security (SEC) module

The security module is a core feature of the software stack that functions as a common layer between Network-level components, the Hardware Security Module (HSM) and Elliptic Curve Digital Signature Algorithm (ECDSA) adaptations and the region-specific security modules.

5.5.1. Security module connections

All loaded region-specific security modules can be used simultaneously.

The main responsibility of the SEC module is to handle requests from the region specific network layer components of the stack (either GeoNetworking or WSMP or DSMP) and sign outgoing messages or validate the incoming secured messages.

The privacy aspect of security is handled by periodically setting new IDs for each module, which is handled by the PSY module (see section "Pseudonymity (PSY) module" [21]).

The certification loading and handling is handled by the CRT service (see section "Certificate loader (CRT) module" [20]).

Certificate secure storage and hardware security are platform specific and are only indirectly related to the dedicated SEC module of the stack.

The security service of the stack is connected to all services described above; however, they act independently and the main responsibility of the SEC module is restricted to signing and verification, as well as handling the encryption related tasks.

5.6. Certificate loader (CRT) module

The CRT module is responsible for loading and forwarding various certificates used for signing. The CRT service of the software stack loads the signer certificates that will be used (CA, intermediate, implicit, and explicit end-entity certificates) from the filesystem. Then the CRT forwards the certificates to the services involved in message signing and verification. Private keys are not loaded or otherwise accessed by the CRT.

The certificates are stored in plain text format. The operation of the CRT module assumes that if the encryption of the stored certificates is required, then it is handled by the operating system (using a filesystem-level encryption such as eCryptfs).

The loader can differentiate between implicitly trusted and untrusted certificates, based on their file path. The directory storing trusted certificates is assumed to be integrity-protected by the operating system. Only the certificates that can be chain-validated up to a trusted certificate are used for signing and verification. The only exception is when the `checkLoadedCertificates` configuration parameter is enabled, which forces all loaded certificates to be trusted by the stack (this is only recommended for testing purposes).

To enable the `checkLoadedCertificates` configuration parameter, proceed as follows:

Using the GUI:

1. Log into the GUI and open the **V2X Core** → **Core stack** menu item.
2. Expand and check the box next to **Security configuration** option, check the box near `Check Loaded Certificates`, and set it to `true`.

Using command line:

1. Log into the device using SSH.
2. Open the `/rwdata/etc` directory from a terminal.
3. Add the following item to the `its.json` file:

```
{
 ...
  "security": {
    "enable": "Yes",
    "checkLoadedCertificates": true
  },
 ...
}
```

5.7. Pseudonymity (PSY) module

The Pseudonymity (PSY) module orchestrates the changes of certificates and temporary IDs for transmission services.

The stack obtains pseudonym certificates from the Security Credential Management System (SCMS) over a secure interface and obtains new batches of pseudonym certificates when necessary and connectivity to the SCMS is available. On every startup, the PSY module of the software stack randomly selects a signed certificate from the available valid pool stored securely in a non-volatile memory as defined in [SAE J2945] and instructs all transmission services to generate a random temporary identifier. The PSY module then periodically orchestrates a change in the used certificate and instructs all service modules to change their temporary IDs as well, based on the distance from the point of last change and the elapsed time. PSY also orchestrates the change of the MAC address of the radio interface and notifies the Security module of the software stack about the certification change.

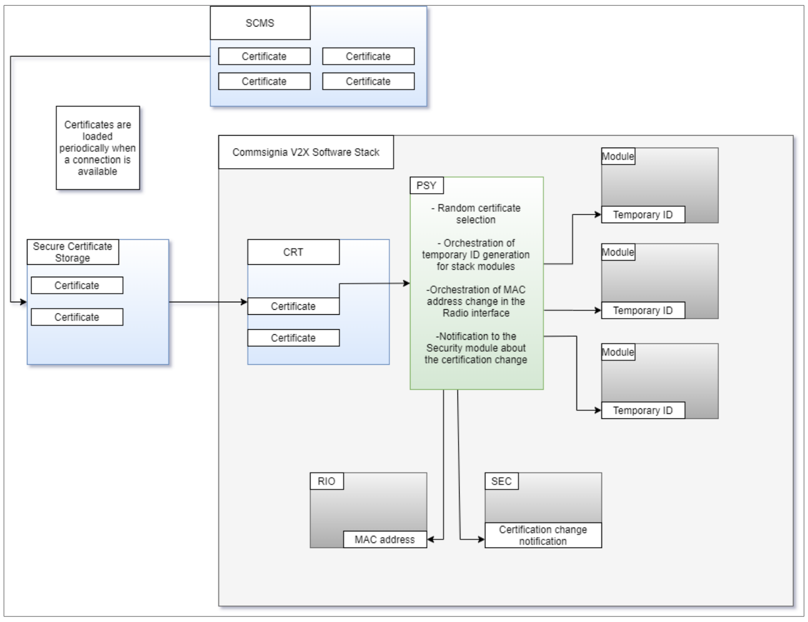

SCMS
Certificate
Certificate
Certificate
Certificate
Certificates are loaded periodically when a connection is available
Secure Certificate Storage
Certificate
Certificate
Commsignia V2X Software Stack
CRT
Certificate
Certificate
PSY
- Random certificate selection
- Orchestration of temporary ID generation for stack modules
-Orchestration of MAC address change in the Radio interface
-Notification to the Security module about the certification change
Module
Temporary ID
Module
Temporary ID
Module
Temporary ID
RIO
MAC address
SEC
Certification change notification

Figure 6. Pseudonymity and Certificate loader modules in relation to the rest of the stack

The same algorithm is used for all transmission services, such as for CAM (see section "Cooperative Awareness Message (CAM) module" [26]), BSM (see section "Basic Safety Message (BSM) module" [24]), and PVD (see section "Probe Vehicle Data (PVD) module" [30]).

5.8. Station Information (STI) module

The Station Information service stores all vehicle data not directly related to navigation, that other modules can use to communicate the vehicle's status.

5.8.1. Information stored in the STI

STI stores two types of data: the physical measurements iof the vehicle in scaled SI units and logical states using enumerations. The physical measurements are used to describe the real-world characteristics of the vehicle (for example the length and height) or measurements that are directly related to physical measurements (such as the height of the bumper on the vehicle). The enumeration based logical states describe certain aspects of the vehicle that are not fixed but can take a pre-defined set of states, for example whether the airbags are turned on, off, or turned on and engaged at the moment. All values of the same unit type (such as speed, distance, mass) must have the same measurement units (for example distance is always set in mm) except in noted cases. CAM, BSM and PVD needs the STI data and data from the NAV module to send out their respective messages. For the full list of the STI information types, see the **sti.h** header file in the attached code reference.

The range check only validates the integer limit within the STI, however the forwarded information may be out of range for some modules that receive STI data (for example BSM). The STI module will not handle any range checking, so in the BSM and CAM messages, the stack sends N/A or OutOfRange if the STI input exceeds their limitations.

5.8.2. Interfacing with the STI

Commsignia provides various API commands that can be used to interface with the STI module, for example the sti_set command enables the setting of certain information points in the STI; and the sti_get lets you read data from the STI service. For the full list of STI related API commands and their usage, see the attached code reference.

A detailed example explaining the integration of CAM data to the STI service of the software stack is also available in sections "Integrating vehicle data to enrich sent V2X messages" [10] (for Python) and "Integrating vehicle data to enrich sent V2X messages" [14] (for C).

5.8.3. Automatically calculated values

The STI module can be configured to calculate the value of certain data types if they are not already set.

Table 1. Automatically calculated values of the STI module

Data type	Description
Acceleration	The STI can be configured to automatically calculate acceleration from speed. The automatically calculated value periodically overwrites any other previously set value, even if it was set by the configuration or over the API.
Yaw rate	The STI can be configured to automatically calculate the yaw rate of the vehicle from the heading information received from the NAV module. The automatically calculated value periodically overwrites any other previously set value, even if it was set by the configuration or over the API.
ABS status	The STI can be configured to automatically calculate from the acceleration of the vehicle if the ABS of the vehicle is enabled (which is also a value that can be automatically calculated). If the deceleration exceeds a configurable threshold, then the STI sets the **ABS enabled** flag to ENGAGED, otherwise it is set to DISENGAGED.

5.8.4. Special values

All data fields (both numeric and enumerated) can have the following special values.

Table 2. Special values of the STI data fields

Value	Description
STI_NA	The value of the data field is unknown.
STI_OOR_MIN	The value of the data field is less than what can be represented or it was out of range (OOR) on the input side. This constant is less than any valid value to aid range checking.
STI_OOR_MAX	The value of the data field is larger than what can be represented or it was out of range (OOR) on the input side. This constant is larger than any valid value to aid range checking.

5.9. Navigation (NAV) module

The stack uses a client-based navigation adaptation service and also supports manual configuration of navigation data to enable custom location-based solutions.

NAV supports multiple sources for navigation data, that the module processes and forwards to other service modules that require location based data (for example BSM or CAM).

Only one source at a time can be configured that NAV uses and this cannot be changed runtime, only with a new configuration and a restart.

The supported navigation sources are:

- Manual: manually specified set of parameters for a navigation fix.
- Real: direct interface with the GPS chip on the device.
- GNSS: global navigation data directly from the antenna. Supported standards are GPS, GLONASS, and Galileo.
- GPSD: A service that collects GPS data from a provider and forwards it as a verified GPS signal. The stack can connect to a GPSD instance as a client to receive this data.

5.10. Basic Safety Message (BSM) module

The BSM module automatically generates BSMs based on data from various other modules of the system and transmits them as signed WSMP packets.

The main use of the BSM is to transmit vehicle data to other participants of traffic. The data sources of BSM are NAV (location information) and STI (for example CAN information from the vehicle). The PSY and SEC modules are responsible for secured message transmission. The basic safety message (BSM) is used in a variety of applications to exchange safety data regarding vehicle state. This message is broadcast frequently to surrounding vehicles with data content as required by safety and other applications as described in the [SAE J2735] standard.

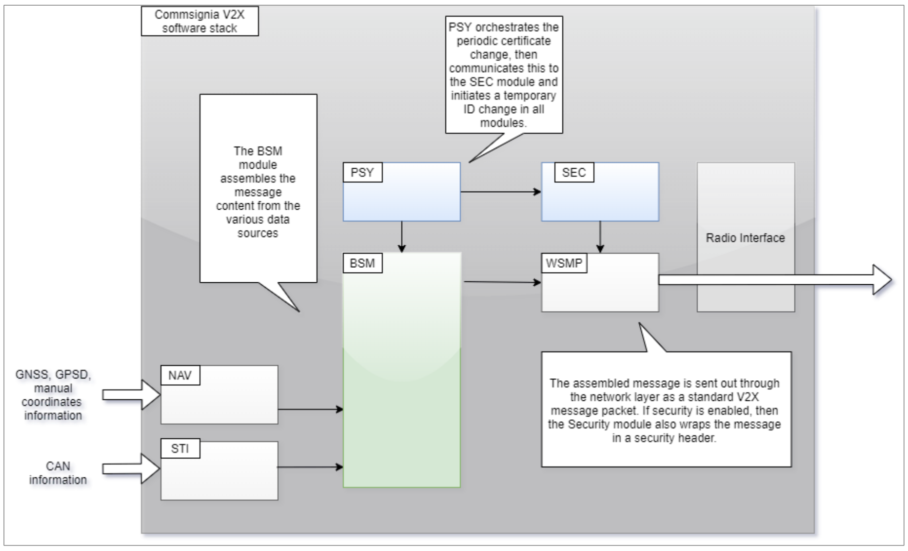

Commsignia V2X software stack
PSY orchestrates the periodic certificate change, then communicates this to the SEC module and initiates a temporary ID change in all modules.
The BSM module assembles the message content from the various data sources
PSY
SEC
BSM
WSMP
Radio Interface
GNSS, GPSD, manual coordinates information
NAV
CAN information
STI
The assembled message is sent out through the network layer as a standard V2X message packet. If security is enabled, then the Security module also wraps the message in a security header.

Figure 7. Interconnectivity of the BSM module and its related input sources within the stack architecture

BSM scheduling is implemented as [SAE J2945] specifies.

5.10.1. BSM core data fields

Table 3shows the supported specific BSM fields in the **BSMcoreData** container, and the name of their data source modules (AC stands for Automatically Calculated). For the full description of each data field, refer to the [SAE J2735] standard.

Table 3. BSM Core data fields and their sources

Data field	Data source
msgCnt	AC
id	STI
secMark	AC
lat	NAV
long	NAV
elev	NAV
accuracy	NAV
transmission	STI
speed	NAV
heading	NAV
angle	STI
accelSet.long	STI
accelSet.lag	STI
accelSet.vert	STI

Data field	Data source
accelSet.yaw	STI
brakes.wheelBrakes	STI (If the value is not N/A or zero, the flag will be 1, else it will be 0.)
brakes.traction	STI
brakes.traction	STI
brakes.abs	STI
brakes.scs	STI
brakes.brakeBoost	STI
brakes.auxBrakes	STI
size.width	STI
size.length	STI

5.10.2. BSM data fields - safety extension

Table 4shows BSM fields in the **VehicleSafetyExtensions** container, and their data sources (AC stands for automatically calculated).

Table 4. BSM safety extension data fields

Data field	Data source
event.eventHazardLights	STI
event.eventStopLineViolation	STI
event.eventABSactivated	STI
event.eventTractionControlLoss	AC
event.eventStabilityControlactivated	AC
event.eventHazardousMaterials	STI
event.eventReserved1	Unsupported
event.eventHardBraking	AC
event.eventLightsChanged	AC
event.eventWipersChanged	AC
event.eventFlatTire	STI
event.eventDisabledVehicle	STI
event.eventAirBagDeployment	STI
pathHistory	PTH
pathPrediction	PTH
lights	STI

5.10.3. Automatically calculated BSM data fields

Table 5 summarizes which BSM data fields can be automatically calculated and the calculation process.

Table 5. Automatically calculated BSM fields

Automatically calculated data field	Calculation
msgCnt	The initial **msgCnt** value is a random number between 0 and 127. Every time a BSM packet is sent the msgCnt increases by 1 until 128, then it is set to 0. When a Pseudonimity (PSY) change occurs, the internal **msgCnt** is set to a random value.
secMark	The **secMark** data member of the BSM packet is always set to the timestamp of the last navigation fix. The timestamp is in "milliseconds in the minute" format.
Brake system event flags: • **ABSActivated** • **TractionControlLoss** • **StabilityControlActiva-ted**	Brake system event flags are calculated from the **abs**, **traction** and **scs** flags of the **BrakeSys-temStatus** data element: • If the status is ON for more than 100 ms, the event flag is set to ON. • If the status is OFF, the event flag is set to OFF.

Automatically calculated data field	Calculation
LightChanged	The **LightChanged** event flag is on if **ExteriorLights** flags changed within 2 s.
WipersChanged	The **WipersChanged** event flag is set to ON if any of the data values in the **WiperSet** container changed within 2 s.
HardBraking	The **HardBraking** event flag is set if the deceleration is greater than 0.4 G.

5.10.4. Interfacing with BSM

Commsignia provides an API command that you can use to set the Special Extension field of the BSM message. Use the **bsm_set_special_ext** function to add Extension Data as an ASN.1 UPER encoded **SpecialVehicleExtensions** object.

5.11. Cooperative Awareness Message (CAM) module

The CAM module automatically generates CAMs based on data from various other modules of the system and transmits them as signed GeoNetworking packets.

The main use of the CAM is to transmit vehicle data to other participants of traffic. The data sources of CAM are NAV (location information) and STI (for example CAN information from the vehicle). The PSY and SEC modules are responsible for secured message transmission. The module collects data from these sources and hands them to the GeonNetworking module to send them as 1609.2 signed GeoNet SHB packets over the radio. The Cooperative Awareness Message (CAM) is used in a variety of applications to exchange safety data regarding vehicle state. This message is broadcast frequently to surrounding vehicles with data content as required by safety and other applications as described in the [ETSI EN 302 637-2] standard.

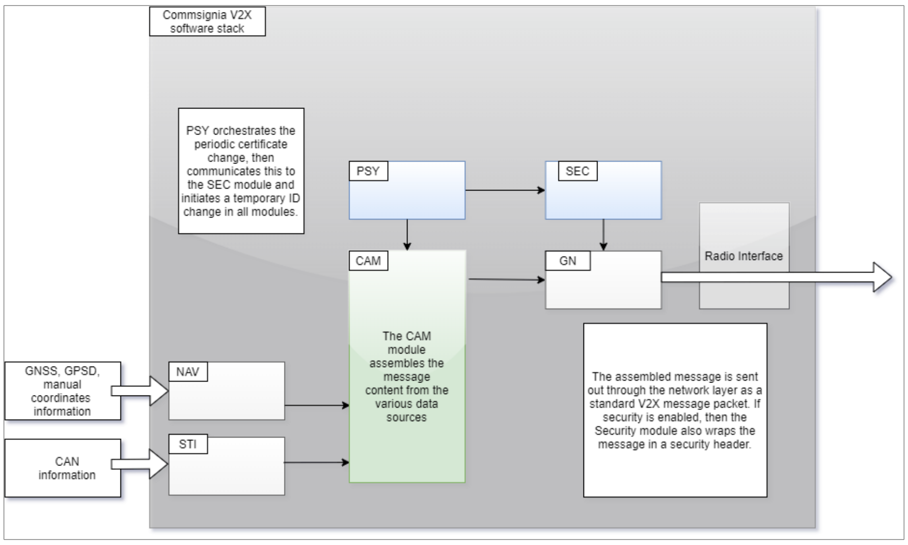

Commsignia V2X software stack
PSY orchestrates the periodic certificate change, then communicates this to the SEC module and initiates a temporary ID change in all modules.
PSY
SEC
CAM
GN
Radio Interface
The CAM module assembles the message content from the various data sources
GNSS, GPSD, manual coordinates information
NAV
STI
CAN information
The assembled message is sent out through the network layer as a standard V2X message packet. If security is enabled, then the Security module also wraps the message in a security header.

Figure 8. CAM structure

Scheduling is performed internally and it is only affected by the current DCC interval in compliance with the [ETSI EN 302 637-2]standard.

5.11.1. CAM supported data fields

Tables 6–9 summarize the supported data fields of the CAM module and the names of the data source modules (AC stands for automatically calculated).

Table 6. CAM header data fields

Data field	Data source
header.protocolVersion	AC
header.messageID	AC

Data field	Data source
header.stationID	AC
cam.generationDeltaTime	AC
cam.camParameters.basicContainer.stationType	STI
cam.camParameters.basicContainer.referencePosition	NAV

Table 7. BasicVehicleContainerHighFrequency data fields

Data field	Data source
heading	NAV
speed	NAV
driveDirection	NAV
vehicleLength	STI
vehicleWidth	STI
longitudinalAcceleration	STI
curvature	PTH - NAV
curvatureCalculationMode	PTH - NAV
yawRate	STI
accelerationControl	STI
lanePosition	STI
steeringWheelAngle	STI
lateralAcceleration	STI
verticalAcceleration	STI
performanceClass	-
cenDsrcTollingZone	-

Table 8. BasicVehicleContainerLowFrequency data fields

Data field	Data source
vehicleRole	STI
exteriorLights	STI
pathHistory	PHPP - NAV

Table 9. SpecialVehicle data fields

Data field	Data source
publicTransportContainer.embarkationStatus	STI
publicTransportContainer.ptActivation	-
specialTransportContainer.specialTransportType	-
specialTransportContainer.lightBarSirenInUse	STI
dangerousGoodsContainer.dangerousGoodsBasic	STI
roadWorksContainerBasic.roadworksSubCauseCode	-
roadWorksContainerBasic.lightBarSirenInUse	STI
roadWorksContainerBasic.closedLanes	-
rescueContainer.lightBarSirenInUse	STI
emergencyContainer.lightBarSirenInUse	STI
emergencyContainer.incidentIndication	-
emergencyContainer.emergencyPriority	-
safetyCarContainer.lightBarSirenInUse	STI
safetyCarContainer.incidentIndication	-
safetyCarContainer.trafficRule	-
safetyCarContainer.speedLimit	-

5.11.2. Interfacing with CAM

The CAM module cannot be interfaced directly. The data elements can be set through their respective modules, except auto-generated and unsupported data.

5.12. Decentralized Environmental Message (DENM) module

The DENM module of the stack handles standard ETSI DENM generation and scheduling.

5.12.1. DENM module overview

Potentially dangerous traffic situations are broadcast in DENM messages in the EU region, as defined by the [ETSI EN 302 637-3] standard. The module operates with raw DENM triggers, updates and terminations. This means that triggering and updating over the API requires the user to send correctly encoded new/updated DENMs in ASN.1 format, apart from the management containers. Therefore, all DENM fields described by the standard are supported by the software stack.

A table of both originated and received messages is stored in the RAM to correctly handle repetition, updates and termination.

The trigger/update/terminate functions behave as described in the standard, including performing negation instead of termination when necessary. Repetition is handled by the module, based on the repetition parameters of the trigger/update function.

5.12.2. Trigger or update requests

These requests trigger DENM sending or an update for an existing DENM. If the **ActionID** parameter is not set then a new **ActionID** is created and parameters is set for this new action. This ID can then be used to update the action or terminate it. After calling this function, the stack starts sending DENM packets with the specified interval for the specified time.

The message payload must be an UPER encoded DENM message. Only the situation, location, and a-la-carte containers can be set. The management container describes the rest of the parameters related to the DENM. If the management container is not empty, it is removed and rewritten. For the full details of the DENM trigger and update request parameters, see the attached code reference.

Table 10. DENM trigger and update request management container parameters

Parameter	Description
ActionID	This is used to identify a DENM event. If it is not set, a new action will be created. If it is already set, an existing action is updated. If set, and it does not correspond to an existing action, the function returns with an error.
Event detection time (Unix timestamp in milliseconds).	If it is set to N/A, it takes the value of the current system time.
Event position (latitude, longitude, altitude, confidence ellipse).	If values are all N/A, this is set to the current location based on navigation data.
Validity duration in seconds, starting from the event detection time.	The default validity is 600 s. The expiration time will be in the future, or the call results in an error.
Repetition duration in milliseconds	If this is set to 0 then the message is sent only once. This value is independent from the validity duration and can be less or equal to it.
Repetition interval in milliseconds	Only used if repetition duration is non-zero.
Relevance distance	Can be N/A; in this case it will not be present in the message.
Relevance direction	Can be N/A; in this case it will not be present in the message.
GeoNetworking Destination area	The destination area of the DENM. It can be one of the following: • Automatic: A circle, where the center is the Event position, and the radius is the relevance distance. Can only be used if relevance distance is set. • Circle • Rectangle • Ellipse

Parameter	Description
GeoNetworking traffic class	The traffic class(es) that the DENM is applicable for based on the definition of the GeoNetworking standard.

5.12.3. Terminate requests

Trigger a DENM termination (negation or cancellation). This function sends a termination DENM request for the currently valid sent or received actions. Most of the fields of the termination are automatically filled based on the already known data from either the **denm_create_or_update** call, or from the received DENM data. (**referenceTime**, **relevance** and target area is set from this data). For the full details of the terminate request parameters, see the attached code reference.

Table 11. DENM terminate parameters

Parameter	Description
ActionID	If it corresponds to a valid sent DENM action, the resulting message will be a cancellation. If it corresponds to a valid received DENM, the resulting message will be a negation. Otherwise this call results in an error.
Event termination time (Unix timestamp in milliseconds)	It corresponds to the **detectionTime** field. If it is set to N/A, it takes the value of the current system time.
Validity time of the termination	Describes the time frame in which the termination request is considered valid.
Repetition duration in milliseconds	If this is set to 0 then the message is sent only once This value is independent from the validity duration and can be less or equal to it.
Repetition interval in milliseconds	Only used if repetition duration is non-zero.

5.13. Collective Perception Message (CPM) module

The stack supports the generation and scheduling of standard CPM transmissions.

The software stack supports ETSI Collective Perception Service message generation and scheduling according to the [ETSI TR 103 562] standard.

Please note that this is a draft standard and its definitions as well as Commsignia's provided implementation are subject to change.

Commsignia provides an API connection to set most of the data fields in the message. In addition to most of the station information data, the following information can be set by the user in raw ASN.1 format:

- Sensor information
- Perceived object information
- Free space addendum

Tables 12–15 describe the data sources of each data field. AC stands for automatically calculated, meaning certain values of data fields are automatically calculated by the CPM module based on other data sources. For the full list of parameters, see the attached code reference.

Table 12. CPM header data fields

Data field	Data source
header.protocolVersion	AC
header.messageID	AC
header.stationID	AC

Data field	Data source
cpm.generationDeltaTime	AC

Table 13. CpmParameters data fields

Data field	Data source
numberOfPerceivedObjects	AC
sensorInformationContainer	API
perceivedObjectContainer	API
freeSpaceAddendumContainer	API
numberOfPerceivedObjects	AC

Table 14. Management Container

Data field	Data source
stationType	STI
perceivedObjectContainerSegmentInfo	Unsupported
referencePosition	NAV

Table 15. StationDataContainer.originatingVehicleContainer data fields

Data field	Data source
heading	NAV
speed	NAV
vehicleOrientationAngle	Unsupported
driveDirection	NAV
longitudinalAcceleration	STI
lateralAcceleration	STI
verticalAcceleration	STI
yawRate	STI
pitchAngle	Unsupported
rollAngle	Unsupported
vehicleLength	STI
vehicleWidth	STI
vehicleHeight	STI
trailerDataContainer	Unsupported

5.14. Probe Vehicle Data (PVD) module

The stack supports the generation and scheduling of PVD message transmission.

This module automatically generates PVD messages based on inputs from various other modules in the system (mainly STI and NAV, but certain values can be automatically calculated as well) and sends them as 1609.2 signed WSMP packets over the radio.

PVD consists of base data identifying the probe vehicle and snapshot data that is collected by the module from other services upon the received request for PVD.

The scheduling, and snapshot scheduling is performed internally, and is only affected by received WSA and PDM packets. Only mandatory snapshot mechanisms are implemented.

5.14.1. Supported data fields

Tables 16 and 17 describe the supported data fields generated into PVD messages. AC stands for automatically calculated, unsupported fields are marked with -.

Table 16. PVD Base Data

Data field	Data source
timeStamp	-
segNum	AC
probeID.name	-
probeID.vin	-
probeID.ownerCode	-
probeID.id	-
probeID.vehicleType	-
probeID.vehicleClass	-
startVector	NAV
vehicleType.keyType	STI
vehicleType.role	STI
vehicleType.iso3883	-
vehicleType.hpmsType	-
vehicleType.responseEquip	-
vehicleType.responderType	-
vehicleType.fuelType	-
vehicleType.vehicleType	-

Table 17. PVD Snapshot Data

Data field	Data source
thePosition	NAV and STI
safetyExt	Calculated like in BSM - LINK
dataSet.lights	STI
dataSet.lightBar	STI
dataSet.wipers	STI
dataSet.brakeStatus.wheelBrakes	STI
dataSet.brakeStatus.traction	STI
dataSet.brakeStatus.abs	STI
dataSet.brakeStatus.scs	STI
dataSet.brakeStatus.brakeBoost	STI
dataSet.brakeStatus.auxBrakes	STI
dataSet.brakePressure	STI
dataSet.roadFriction	STI
dataSet.sunData	STI
dataSet.rainData	STI - The Ohm value in STI converted according to the table defined in [SAE J2735].
dataSet.airTemp	STI
dataSet.airPres	STI
dataSet.steering.angle	STI
dataSet.steering.confidence	STI
dataSet.steering.rate	STI
dataSet.steering.wheel	STI
dataSet.accelSets.accel4way.long	STI
dataSet.accelSets.accel4way.lat	STI
dataSet.accelSets.accel4way.vert	STI
dataSet.accelSets.accel4way.yaw	STI
dataSet.accelSets.vertAccelThres	-
dataSet.accelSets.yawRateCon	STI
dataSet.accelSets.hozAccelCon	STI
dataSet.accelSets.confidenceSet	STI

Data field	Data source
dataSet.object	-
dataSet.fullPos	NAV
dataSet.throttlePos	STI
dataSet.speedHeadC	NAV
dataSet.speedC	NAV
dataSet.vehicleData	-
dataSet.vehicleIdent.name	-
dataSet.vehicleIdent.vin	-
dataSet.vehicleIdent.ownerCode	-
dataSet.vehicleIdent.id	-
dataSet.vehicleIdent.vehicleType	-
dataSet.vehicleIdent.vehicleClass	-
dataSet.j1939data	-
dataSet.weatherReport	STI - Except the friction field - it is already included in **roadFiction**
dataSet.gnssStatus	-

5.14.2. Interfacing with PVD

It is only possible to set data elements through their respective modules but auto-calculated and unsupported data fields cannot be set with API commands.

5.15. Watchdog timer (WDT) module

The WDT module is the global Watchdog time functionality of the stack, which monitors all important threads.

The WDT service reports an error if any of the watched functions of the stack do not report a periodic heartbeat signal at a specified time.

The interface of the watchdog is a periodic report itself, which contains a single status information ("OK" meaning all modules report normally or "Not OK" meaning one or more threads are stuck).

It is recommended to integrate the WDT signal of the stack to a system-wide watchdog service on your platform to monitor this report and if the status is not OK, or the reports stop coming for a specified time (probably meaning that the watchdog thread itself is stuck), the stack must be restarted.

5.16. Statistics (STAT) module

The Statistics (STAT) module is responsible for collecting statistical information from the modules of the software stack.

5.16.1. STAT counter behavior

Each statistics counter starts at zero upon process startup and they each have a range of 0–2^{64}. If a counter value overflows or underflows this range, then that value will wrap around.

API commands can be used to get or set any or all STAT module values of the counter.

5.16.2. List of available statistics counters

The following stack modules have defined counters in the STAT module:

- Radio

- WSMP
- GeoNetworking
- 1609.2
- Facility Rx
- CAM Tx
- DENM Tx
- CPM Tx
- SRM Tx
- BSM Tx
- PVD Tx
- External API
- Security

Use the following command to generate a status log:

```
v2x-status-json-gen
```

The `j` utility can be used to filter the required values, for example the command `v2x-status-json-gen| jq '.statistics.radio.if1'` list the counter values related to radio interface 1 only:

```
{
  "txPacket": 1151711,
  "rxPacket": 7196345,
  "rxUnknownPacket": 0,
  "rxInvalidMacPhyHeaderPacket": 0,
  "rxRssiLastPacket": 0
}
```

5.17. Commsignia Capture Protocol (C2P) module

The Capture interface provides a continuous UDP stream over a wired or wireless connection that sends UDP messages about transmitted and/or received packets.

The C2P data stream is not subscription based, the Capture interface starts upon the startup of the stack providing notifications on system startup if it is enabled in the configuration of the stack. The data can be fikltered to raw radio packets as well as information from the NAV and STI modules of the software stack.

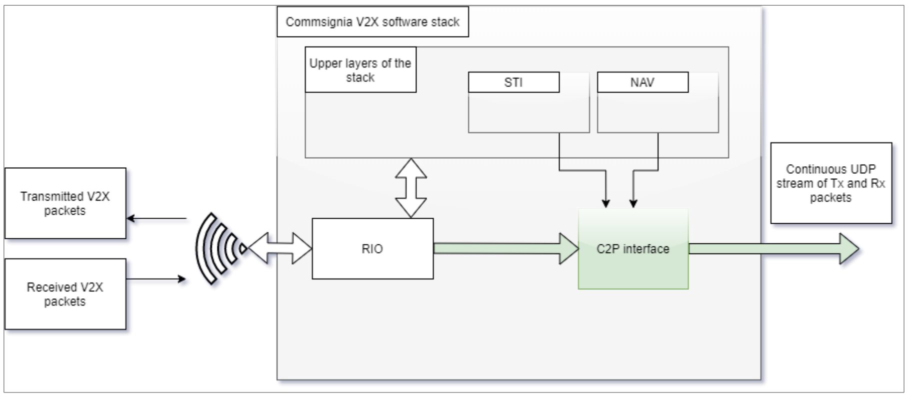

Commsignia V2X software stack
Upper layers of the stack
STI
NAV
Transmitted V2X packets
Received V2X packets
RIO
C2P interface
Continuous UDP stream of Tx and Rx packets

Figure 9. Continuous UDP stream of Tx and Rx data provided by the C2P interface

A filter can be preconfigured for the data needed to be received from C2P. The filter configuration is a system parameter that cannot be changed during runtime and it is not client-specific.

The C2P stream can be filtered to contain the following supported data:

- Receiving radio (IEEE 802.11p or Cellular-V2X) packets as they are received over the radio, right before the network layer starts processing it. C2P transmits all received packets with a correct MAC/PHY frame, even if the packet was dropped by any of the upper layers.
- Transmitting radio (IEEE 802.11p or Cellular-V2X) packets as they are transmitted over the radio, right before the radio adaptations take them over.
- Position (navigational fix) update.
- Updates received on the Station Information (STI) interface.

C2P provides a header for each notification that contains the following information:

- Type of the notification
- Current time and location
- Radio specific parameters (such as Tx power, Channel, RSSI)

5.17.1. Enabling the C2P module on the device

To enable the C2P module on your device proceed as follows:

1. Using a GUI:
   a. Log into the GUI and open the **V2X Core** → **Core stack** menu item.
   b. Check the box next to the option **Commsignia Capture Protocol (C2P)**, and expand it.
   c. To enable the C2P data stream, check the box next to `Enable C2P` and set its value to `true`.
   d. The IP address and the port for the service can be defined at the `Remote server address` and `UDP port number` fields, respectively.
2. Using the CLI:
   a. Log into the device using SSH.
   b. Open the `/rwdata/etc` directory from a terminal.
   c. Using an editor, open the `its.json` file and add the following item:

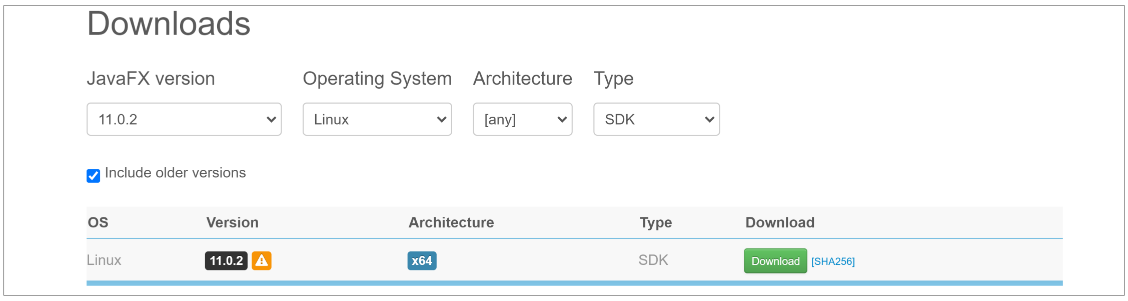
```
{
 ...
  "capture": {
   "enable": true,
   "address": "127.0.0.1",
   "port": 7943
  },
 ...
}
```

5.17.2. Installing the C2P module on your computer

The C2P module requires Java SDK 11 and JavaFX version 11 installed on a Windows or Linux computer.

Installing and configuring the Java environment on a Linux computer

1. Open the Java SE Development Kit 11.0.18 download page, click on the **Linux** tab, and download the file that is appropriate for your Linux distribution. To download JDK files, an Oracle Account is required. After the download has been completed, unzip the downloaded file.
2. Open the JavaFX download page, select the option "Include older versions," and choose JavaFX Version 11.0.2 from the drop down menu. Select the Operating System "Linux" and the Type "SDK," as shown in Figure 10, and download the file. After the download has been completed, unzip the downloaded file
   .


Downloads

JavaFX version: 11.0.2 | Operating System: Linux | Architecture: [any] | Type: SDK

☑ Include older versions

OS	Version	Architecture	Type	Download
Linux	11.0.2 ⚠	x64	SDK	Download [SHA256]

Figure 10. JavaFX download page for Linux

3. To set the Java environment variables open the terminal and use the following commands:

```
export JAVA_HOME="/path/to/java/home"
```

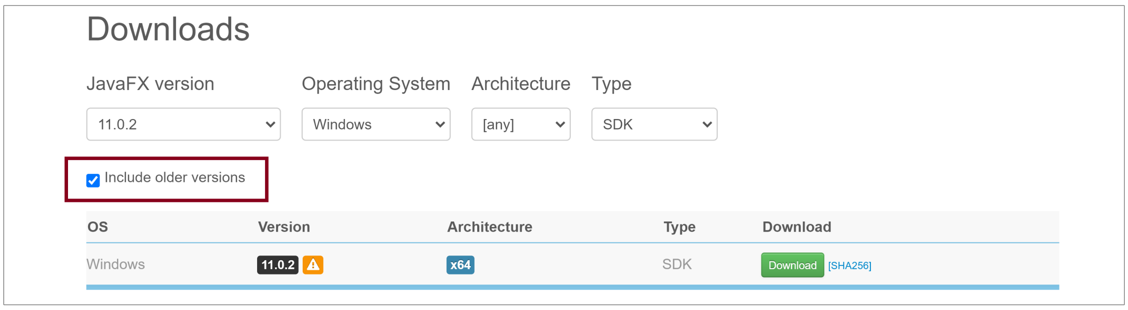
```
export JAVAFX_HOME="/path/to/javafx/home"
```

```
export PATH=$PATH:$JAVA_HOME/bin:$JAVAFX_HOME/bin
```

4. To run the C2P module, start the `./capture-app` executable.

Installing and configuring the Java environment on a Windows computer

1. Open the Java SE Development Kit 11.0.18 download page, click on the **Windows** tab, and download the file "x64 Compressed Archive." To download JDK files, an Oracle Account is required. After the download has been completed, unzip the downloaded file.
2. Open the JavaFX download page, select the option "Include older versions," and choose JavaFX Version 11.0.2 from the drop down menu. Select the Operating System "Windows" and the Type "SDK," as shown in Figure 11, and download the file. After the download has been completed, unzip the downloaded file.


Downloads

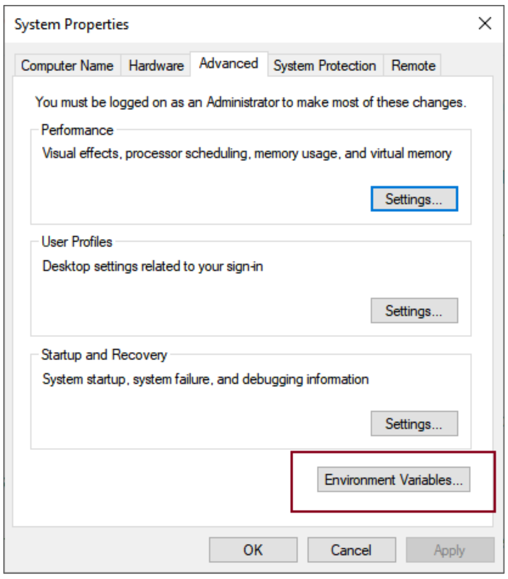
JavaFX version: 11.0.2 | Operating System: Windows | Architecture: [any] | Type: SDK

☑ Include older versions

OS	Version	Architecture	Type	Download
Windows	11.0.2 ⚠	x64	SDK	Download [SHA256]

Figure 11. JavaFX download page for Windows

3. Set the Java environment variables as follows:
   a. Open Control Panel ⟶ System ⟶ Advanced System Settings and click on the "Environment Variables..." button, as shown in Figure 12.


System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your sign-in

Settings...

Startup and Recovery

System startup, system failure, and debugging information

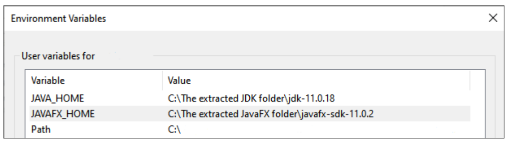
Settings...

Environment Variables...

OK Cancel Apply

Figure 12. Advanced System Settings panel in Windows

b. Set the JAVA_HOME and JAVAFX_HOME environment variables by double clicking on them and specifying the directories of the extracted JDK and JavaFX files, respectively, as shown in Figure 13.

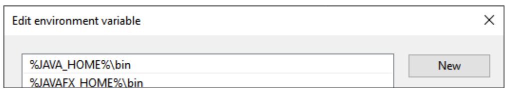

Environment Variables

User variables for

Variable	Value
JAVA_HOME	C:\The extracted JDK folder\jdk-11.0.18
JAVAFX_HOME	C:\The extracted JavaFX folder\javafx-sdk-11.0.2
Path	C:\

Figure 13. Environment Variables panel in Windows

c. Double click on the "Path" user variable and add the lines %JAVA_HOME%\bin and %JAVAFX_HOME%\bin, by clicking on the "New" button, as shown in Figure 14.


Edit environment variable

%JAVA_HOME%\bin

%JAVAFX_HOME%\bin

New

Figure 14. Editing environment variables in Windows

4. Restart your computer for the changes to take effect.
5. To run the C2P module, launch the `capture-app.bat` file.

5.18. Facility Reception service

The stack handles the reception of all facility related messages in a unified way.

The stacks facility reception service decodes the received packets and checks their plausibility. If two Facility messages have the same PSID or AID, the Facility Reception module can still differentiate between them and create the appropriate notification; thus, the subscribing modules do not need to do this separation themselves.

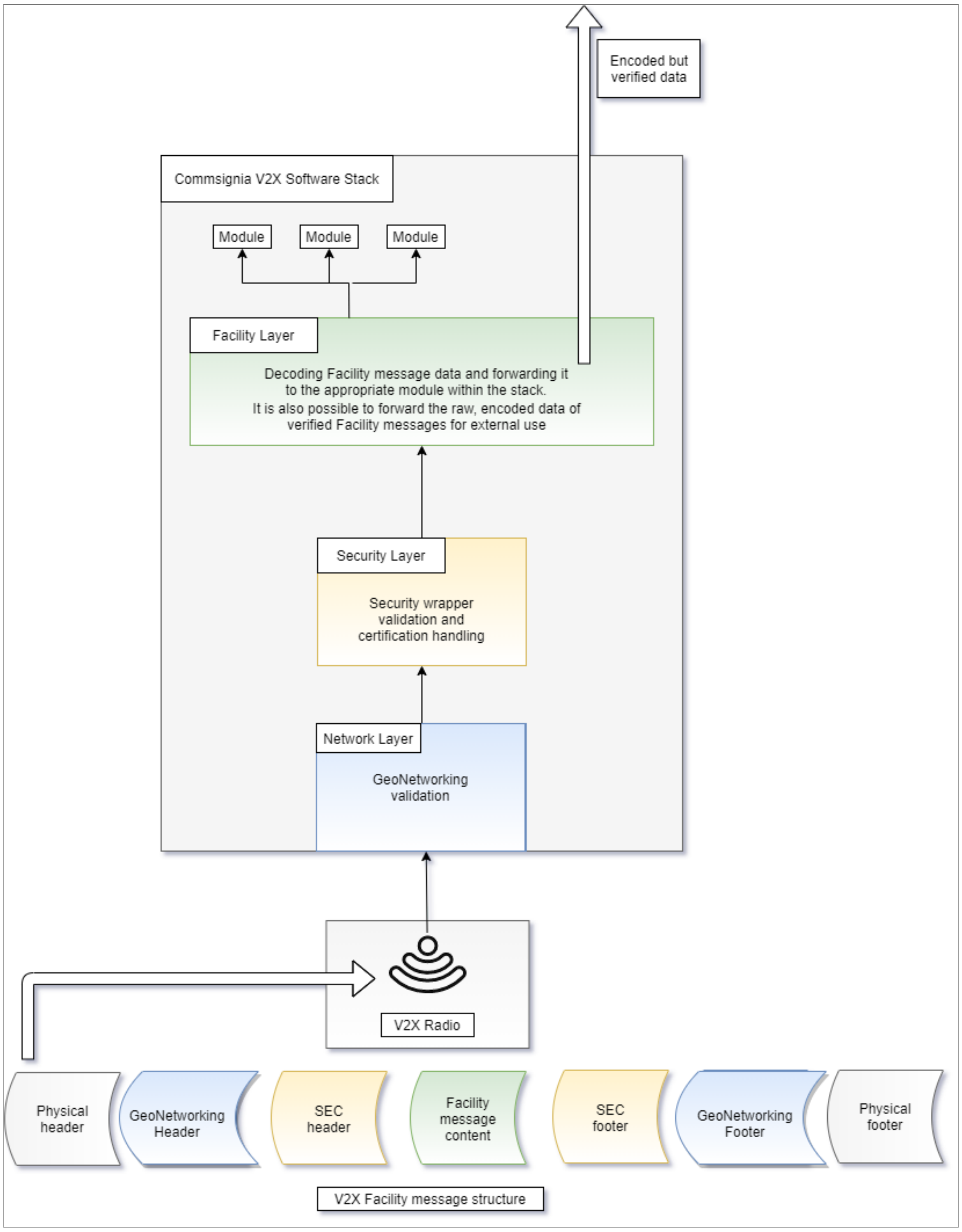

Encoded but verified data
Commsignia V2X Software Stack
Module
Module
Module
Facility Layer
Decoding Facility message data and forwarding it to the appropriate module within the stack. It is also possible to forward the raw, encoded data of verified Facility messages for external use
Security Layer
Security wrapper validation and certification handling
Network Layer
GeoNetworking validation
V2X Radio
Physical header
GeoNetworking Header
SEC header
Facility message content
SEC footer
GeoNetworking Footer
Physical footer
V2X Facility message structure

Figure 15. Facility layer message handling and message structure

The security layer lets packets through even if they failed security and validation checks but it marks them. Depending on your configuration the facility layer can still handle them; however, this is turned off by default.

The stack supports the reception of the following ETSI standard facility messages:

- ETSI Cooperative Awareness Message (CAM)
- ETSI Decentralized Environmental Notification Message (DENM)
- ETSI Infrastructure to Vehicle Information (IVI / IVIM)
- ETSI Road and Lane Topology (RLT, a.k.a. MAP)
- ETSI Traffic Light Manouver (TLM, RLT, a.k.a. SPAT)
- ETSI Traffic Light Control (TLC) Signal Request Extended Message (SREM) and Signal Status Extended Message (SSM)
- ETSI GNSS positioning correction (GPC, a.k.a. RTCM)
- ETSI Collective Perception Message (CPM)

The stack supports the reception of the following SAE standard facility messages:

- SAE DSRC Basic Safety Message (BSM)
- SAE DSRC Traveller Information Message (TIM)
- SAE DSRC Road-Side Alert (RSA)
- SAE DSRC MAP
- SAE DSRC Signal Phase and Timing (SPaT)
- SAE DSRC Signal Request Message (SRM) and Signal Status Message (SSM)
- SAE DSRC Personal Safety Message (PSM)
- SAE DSRC Probe Vehicle Data (PVD) and Probe Data Management (PDM)
- SAE DSRC RTCM

The stack supports the reception of the following YD standard facility messages:

- YD Basic Safety Message (BSM)
- YD MAP
- YD Signal Phase and Timing (SPAT)
- YD Road Side Information (RSI)

5.19. Low-level modules

5.19.1. WAVE Short Message Protocol (WSMP) module

The WAVE Short Message Protocol is a highly efficient network layer messaging protocol that is responsible for transporting standard high priority and time sensitive WAVE Short Messages (WSM).

The WSMP module is a very low level network layer protocol. Upon reception of WAVE Short Massage (WSM) data form the upper layers, WSMP generates a WSMP header to be included in the received unit then transmits the data to the lower layers as specified in [IEEE 1609.3-2016]. The WSMP module communicates with the Security (SEC) module to sign each message.

WSMs can be sent on any channel and the protocol allows applications to directly control physical characteristics. WSMP is also used to send and receive WSAs. A WSM may be accepted by one or more destination devices, depending on the type of MAC-level addressing used (either an individual or a group).

The WSMP module handles the following inputs to generate a standard compliant message that is ready to be sent out as a V2X packet.

Table 18. WSMP module inputs

Input field	Description
Payload	The actual message content. This can be a standard V2X facility message or a custom message.

Input field	Description
Radio information	Transmission related information such as: • Interface ID • Destination MAC address • Data rate • Priority • Tx Power • Expiration Time • SPS channel number
WSMP header	The header defined by the [IEEE 1609.3-2016] standard for all WSMP compliant messages. This contains the PSID as well as any extension fields such as Tx power, Datarate, or Channel information.
Security information	Information about security signing. If singing is enabled, the WSMP module requests the Security service to secure the message before sending it out.

The WSMP module is also responsible for unwrapping the WSMP headers from the received messages and forwarding them towards the upper layers of the stack (for example to the facility services). In this case the WSMP module also requests the Security service of the stack to handle the secure message headers before sending the unwrapped content forward.

5.19.2. GeoNetworking (GN) module

The GeoNetworking protocol is a network layer protocol that provides packet routing in an ad hoc network. It uses the geographical positions for packet transport and it supports communication among individual ITS stations as well as the distribution of packets in geographical areas.

The software stack offers support for ETSI GeoNetworking (Geographical addressing and forwarding for point-to-point and point-to-multipoint communications) compliant with the [ETSI EN 302 636-4-1] and [ETSI TS 102 636-7-1]standards.

The GN module of the stack provides both media-independent and media-dependent functionality combined with the corresponding Basic Transport Protocol (BTP) according to the same standards. Support is provided for both for transmission, reception, and forwarding with the selected GeoNetworking subtypes and forwarding algorithms. The Commsignia V2X software stack supports both IEEE 802.11p and Cellular-V2X radio types.

6. V2X Security in the software stack

Security is an integral part of the Commsignia V2X software stack and this chapter describes the internal logic and the background knowledge related to message signing, secure message verification and the rest of the security aspects.

6.1. Security within the software stack

In a V2X communication environment security is paramount - by definition of V2X technology all messages are sent out over radio, so they can be received and decoded by any participant. Therefore, the sender have to ensure the integration, validity, and security of their messages.

Commsignia provides a security solution, not as a separate layer, but as an integral part of every step of message handling and generation within the software stack, which is compliant with all required V2X security standards and all region-specific technologies can be used simultaneously.

The security module itself is only directly responsible for signing outgoing messages and validating incoming secured messages, as well as handling the technical aspects of wrapping and unwrapping the message content with the standard security headers, upon the request of the region-specific network layer modules (GeoNetworking or WSMP). The other aspects of security are handled by different service modules with their own internal logic and the main security module is only indirectly connected to them.

6.2. Data encryption

Encryption by the security module ensures the confidentiality of the data content so only verified V2X recipients can decipher the message. This is ensured by using symmetric or asymmetric cryptography, using keys known only by trusted participants. The integrity and validity of every message is checked upon receipt.

The following algorithms are approved for use as specified in [IEEE 1609.2-2016]:

- **Signing**: ECDSA over NIST P-256
- **Public key encryption**: ECIES over NIST P-256
- **Hash**: SHA-256
- **Symmetric Encryption**: AES-CCM with 128-bit keys

See the [IEEE 1609.2] standard for normative references to the definitions of the algorithms.

The private keys used for encryption are stored in the Hardware Security Module (HSM) on the device.

6.3. Certificate management

The usage of certificates ensures that all participants of V2X communication can consider each other trusted parties. The certificates contain the public keys used for digital signatures.

A Security Credential Management System (SCMS) service provides certificate packs on a regular basis to sign the messages. Each certificate pack contains several certificates that are randomly selected to ensure privacy and reduce traceability. Certificates are signed with public keys in higher level certificates, forming a chain up to the root certificate(s) of the SCMS provider(s), ensuring the trust and validity of all certificates down to the individual message signing certificates.

You can not generate your own certificates, you must subscribe to a verified provider.

Commsignia provides a certificate pack with a root certificate that you can use for testing purposes; however, this will only result in a trusted status with other Commsignia devices (and devices which have the test root certificate installed as trusted), and will not work in a live enrollment. In production, and most pilot environments, you must obtain the certificates from the verified SCMS provider of the project. The Commsignia V2X software stack is able to use multiple trusted root certificates to support multiple providers simultaneously.

For the details about how the technical aspects of certificate loading are handled, see section “Certificate loader (CRT) module” [20].

6.4. Certificate types

The V2X system uses several types of certificates. SCMS components generate these and in many cases can also revoke them. All certificate lifetimes and renewal periods are defined by the SCMS for each certificate type. All the EE certificates are of implicit type to save storage space and over-the-air bytes. All the SCMS component certificates (the elector, root CA, PCA, and ICA certificates) are of explicit type.

Table 19. Certificate types used by V2X systems

Certificate type	Description
OBE Enrollment	An enrollment certificate is similar to a passport for the OBUs in that it uses the enrollment certificate to request other certificates: pseudonym and identification certificates. It does not have an encryption key.
Pseudonym	Pseudonym certificates are used by an OBU primarily for BSM authentication and misbehavior reporting and do not have encryption keys.
Identification	Identification certificates are used by an OBU primarily for authorization in V2I applications. None of the current V2I applications require encryption by the OBU at the application level; however, there might be a need in the future.
RSE Enrollment	An enrollment certificate is similar a passport for the RSUs in that it uses the enrollment certificate to request application certificates. It does not have an encryption key.
Application	Application certificates are used by an RSU for authentication and encryption; therefore, they might have encryption keys. As there are no privacy constraints for RSUs, an RSU has only one application certificate valid at a time for a given application.
Electors	Elector certificates are not part of the PKI hierarchy of the SCMS, such as verifying a certificate chain in the system does not involve verifying elector certificates. They are used primarily for root CA certificate management, including adding and removing a root CA.
Root CA	The Root CA certificate is the end of trust chain, meaning that verification of any certificate in the system ends at verifying this certificate. The signature on the root CA certificate does not have any cryptographic value as the signature is by the root CA itself, and, therefore, the trust in a root CA certificate is established through out-of-band means. Usually the root CA certificate has a long lifetime, as changing a root CA certificate is a time consuming, and potentially expensive operation. Only a quorum of electors can issue root management messages and add them to a CRL to revoke a root CA certificate
ICA	ICA certificates can be used to only issue certificates to other SCMS components and nothing else. Only the root CA or the ICA can issue, or authorize someone to issue, a CRL to revoke an ICA certificate.
PCA	PCA certificates can be used to only issue certificates to end-entities including OBEs and RSEs. PCA certificates need to have validity periods that are at least as long as the longest validity certificates issued using them. These certificates have an encryption key.

6.5. Privacy through pseudonymity

Privacy is important to negate traceability in a V2X communication environment. The security service of the Commsignia stack achieves this by utilizing a separate pseudonymity service module (PSY) that periodically generates new temporary IDs and MAC addresses for each module within the stack. For the details about the technical aspects, see section “Pseudonymity (PSY) module” [21].

6.6. Enabling secure message sending

To enable the automatic signing of all outgoing standard V2X messages (such as CAM) using a trusted certificate pack proceed as follows:

1. Extract the certificate pack to the certificate location on your device. For example the `/etc/security_eu/` folder.

Make sure that you have the same root certificate that you plan to test on the receiving devices, otherwise the messages will not be recognized as trusted.

2. Enable security, as described in section "Enabling security" [6].
3. Enable the `checkLoadedCertificates` configuration parameter, as described in section "Certificate loader (CRT) module" [20].
4. Enable pseudonymity, as described in section "Enabling pseudonymity" [6] .
5. Enable WSMP configuration using the GUI:
   a. Log into the GUI and open the **V2X Core** → **Core stack** menu item.
   b. Expand and check the box next to **WSMP configuration**.
   c. Check the box next to `enable` and set its value to `true`.
6. Enable WSMP configuration using the CLI:
   a. Log into the device using SSH.
   b. Open the `/rwdata/etc` directory from a terminal.
   c. Using an editor, open the `its.json` file and add the following item:

```
{
 ...,
  "wsmp": {
    "enable": true,
    "expectedPayload": "1609dot2"
  },
 ...
}
```

7. You can verify that the outgoing V2X messages are signed before they are sent by checking the status of the stack. Use the `unplugged-rt-status-gen 127.0.0.1` command to generate an output that contains counters for each service. Look for the `SUS signed` (for US standard) or the `SEU signed` (for EU standard) counter under `Statistics`. If this counter is incremented upon running the status generation again then the outgoing messages are signed.

Check the status of the stack to verify that the outgoing V2X messages are signed before they are sent:

1. Using the GUI:
   a. Log into the GUI and open the **V2X Status** → **Status**.
   b. Expand **statistics** → **security** → **1609.2** and the appropriate region.
2. Using the CLI:
   a. Log into the device using SSH.
   b. Use the command `v2x-status-json-gen | jq ".statistics.security"` to display the counters:

```
{
  "signerCertificate": {
    "eu": 0,
    "us": 0,
    "cn": 0
  },
  "1609.2": {
    "eu": {
      "txSignedPacket": 0,
      "txSigningErrorPacket": 0,
      "rxVerifiedPacket": 0,
      "rxMalformedPacket": 0,
      "rxVerificationFailedPacket": 0,
      "rxUnsecuredPacket": 0
    },
    "us": {
      "txSignedPacket": 0,
      "txSigningErrorPacket": 0,
      "rxVerifiedPacket": 0,
      "rxMalformedPacket": 0,
      "rxVerificationFailedPacket": 0,
      "rxUnsecuredPacket": 0
    },
    "cn": {
      "txSignedPacket": 0,
      "txSigningErrorPacket": 0,
      "rxVerifiedPacket": 0,
      "rxMalformedPacket": 0,
      "rxVerificationFailedPacket": 0,
      "rxUnsecuredPacket": 0
    }
  }
}
```