TRAFFICAUTH®

National Mobility Interchange: Getting Started Guide

Table of Contents

Overview 3

Architecture 3

Integration 4

Network Access 4

Server Endpoints 4

Supported Protocols 4

Authentication 4

Topic Structure and Subscriptions 4

Topic Naming Convention 5

Publishing Usage 5

Subscribing Usage 6

Message Content and Format 7

Message Types 7

Security Model 8

Intended Use Cases 8

Support 8

Versions 9

Appendix 10

Decoding Guidance 10

Decoding ASN.1 (UPER) payloads with asn1c (open source) 10

What asn1c is (and isn't) 10

Step 1 — Get the J2735 ASN.1 specification (source files) 10

Step 2 — Install asn1c (Linux example) 10

Step 3 — Generate a J2735 decoder with OER/UPER support 11

Step 4 — Decode UPER (Unaligned OER) into a MessageFrame 11

Step 5 — Where 1609.2 fits 12

Geohash Guidance 12

Overview

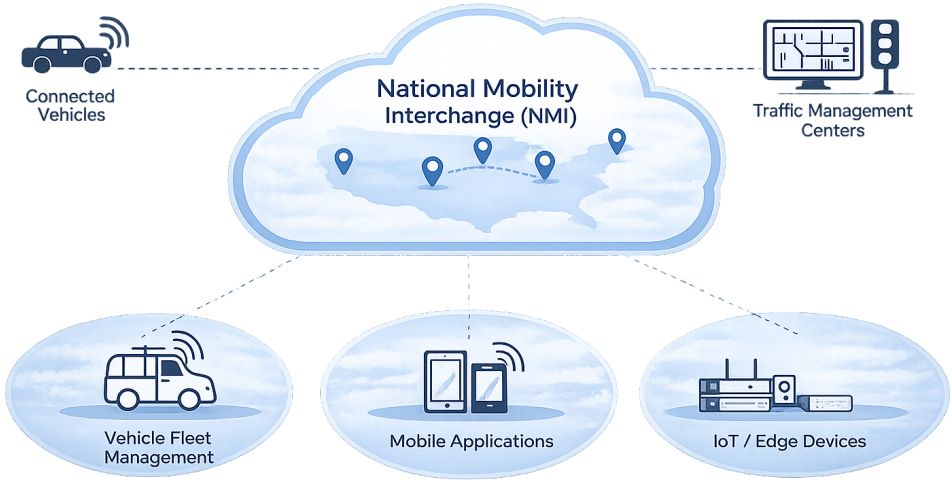
The **National Mobility Interchange (NMI)** is a cloud-hosted V2X data exchange that distributes secure, authenticated, standards-compliant transportation messages over a publish/subscribe model. NMI enables participants to consume real-time V2X messages from public agencies and infrastructure providers using widely adopted industry standards.

This guide covers the information required for an application developer to connect to NMI and begin receiving data.

Architecture

NMI is implemented as a cloud-native, distributed broker designed to support the efficient exchange of time-sensitive V2X data at scale. The platform operates on a publish/subscribe model in which data producers publish messages to structured topics, and consumers subscribe to those topics based on geographic or application-specific relevance.

The system is designed to transport authenticated, standards-conformant messages without modification. NMI expects data consumers to ensure compliance with applicable SAE and IEEE V2X specifications. Clients establish persistent connections to the broker and receive data streams corresponding to their topic subscriptions. The architecture supports high-throughput message delivery with low latency, enabling reliable distribution of real-time transportation data across a diverse set of consumers.



Integration

Network Access

The NMI infrastructure supports multiple protocols and a production and pre-production (test) environment for demonstrations and integration projects.

Server Endpoints

Two server endpoints are supported. One is to be used for initial development and testing and the other is reserved for production use. Each server has a unique URL:

- **Test:** `mqtt.development.v2x.isscms.com`
- **Production:** `mqtt.us.mobilityinterchange.com`

The Test environment is typically used with test or pre-production certificates issued from the SCMS Manager Test Certificate Trust List (CTL), although production certificates will also be accepted in this system. The Production environment must be used with certificates issued from the SCMS Manager Production Certificate Trust List (CTL).

Supported Protocols

Two protocols are currently supported:

- MQTT over TCP on port 1883
- WebSockets on port 80 (non-TLS)

Client systems may publish and subscribe to the NMI servers without authentication. All packets must include a valid IEEE 1609.2 signature from a client system with an active SCMS certificate. Packets without signatures, packets that are malformed, or packets that are published to an incorrect topic may be blocked by the server.

Authentication

- No authentication, tokens, or credentials are required to subscribe.
- Any standard MQTT client library is supported.

Topic Structure and Subscriptions

MQTT topics indicate location and message type. All messages shall use binary payloads encoded according to IEEE 1609.2. The payload shall contain a signed message, which when decoded yields a J2735 message whose type corresponds to the message id indicated in the topic.

Topic Naming Convention

Messages shall be published to topics corresponding to the geographic location at which the event occurs, using a geohash calculated from the event's latitude and longitude coordinates (e.g., via standard base32 geohash encoding at the required precision), with each geohash character mapped to a hierarchical topic level.

The following topic structure encodes location using a hierarchical geohash and appends the message type identifier as a final topic level:

```
/v1/g32/{g1}/{g2}/{g3}/{g4}/{g5}/{g6}/{g7}/{message_id}
```

Where:

- `v1` is a version indicator
- `g32` indicates that what follows is a Geohash in Base 32 notation
- `{g1}...{g7}` are the individual characters of the geohash
- `{message_id}` an integer corresponding to the J2735 Message-Frame messageId

Details:

- A 7-character geohash gives you a location precision of roughly:
  - ±153 meters × ±153 meters (about 500 ft × 500 ft)
- Use one geohash character per topic level.
- For a 7-character geohash like mqqtxyz, the topic becomes:

  ```
  /v1/g32/m/q/q/t/x/y/z
  ```

- message_id (integer) is appended at the final level. For example, BSM messages occurring within the mqqtxyz geohash should use the following topic:

  ```
  /v1/g32/m/q/q/t/x/y/z/20
  ```

  - Where integer message_id for BSM is 20
  - Where message in the topic name matches the message used in the signed payload
- Subscribers can use wildcards to receive broader coverage, for example:
  - To receive all message types in location `mqqtxyz`:
    `/v1/g32/m/q/q/t/x/y/z/#`
  - To receive all BSMs in geohash level 4 location `mqqt`:
    `/v1/g32/m/q/q/t/+/+/+/20`

Publishing Usage

Publishers shall publish all messages to fully qualified topics using the defined geohash and message_id-based topic structure (see Topic Naming Convention). The topic hierarchy shall encode both the geographic scope and the message type.

Requirements:

- Topics shall include the complete geohash hierarchy corresponding to the relevant location of the message.
- Topics shall include the message_id as the final level, consistent with the message type.
- The message_id contained within the message payload shall match the message_id indicated in the topic.
- All messages shall be signed using **valid security credentials**, and the message body shall contain a valid digital signature.
- Client systems shall publish messages to the geohash corresponding to the location relevant to the message content (typically a 7-character geohash).
- Messages that do not conform to these requirements may be rejected or blocked by the service.

Message Distribution:

- Each message shall be published to exactly one topic for point-based events.
- Messages with spatial extent spanning multiple geohash cells shall be published to each topic within the geohash coverage of the message geometry.
- When a message is published to multiple topics, the publisher shall maintain an identical message identifier across all publications to enable downstream deduplication.

Operational Guidance:

- Publishers should transmit messages as a continuous stream where applicable to support real-time data dissemination.
- Topic selection shall align with the geographic relevance and intended consumption of the data to ensure efficient distribution across subscribers.

Subscribing Usage

Subscribers define their subscription patterns based on operational requirements, including geographic scope and message type (Message ID). The topic hierarchy is structured to enable efficient filtering and data consumption using wildcard subscriptions.

Due to spatial coverage, a single message may be published to multiple topics. In such cases, the message retains a consistent identifier across all publications. Subscribers are expected to use this identifier to detect and discard duplicate messages that may be received through overlapping subscriptions or multiple geohash topics.

Requirements:

- Subscribers must be able to:
  - Receive binary MQTT payloads
  - Parse (and optionally validate) IEEE 1609.2 signed messages

- Decode UPER-encoded J2735 payloads using appropriate ASN.1 tooling

Subscriber Guidance Note:

Subscribers should select the narrowest subscription that satisfies their requirements to:

- minimize unnecessary message volume
- reduce processing overhead

Message Content and Format

All NMI payloads are ASN.1 encoded SAE J2735 messages wrapped in an IEEE 1609.2 security envelope as follows:

- **Structure**:
  - IEEE 1609.2 SignedData (OER encoded following the IEEE 1609.2-2022 standard)
    - Containing payload (ASN.1 UPER encoded J2735 Message Frames)
- **Message Set:** SAE J2735
- **Integrity:**
  - Every message includes a **digital signature**
  - Signatures enable validation of message authenticity and integrity

Message Types

The following C-V2X message types are currently supported by the service. Additional message types may be delivered through the system. Clients are expected to be able to interpret the content of any standard J2735 message structure and interpret the message payload.

Message Type
BSM
PSM
RTCM
SPaT
TIM, RSM
WSA
TAM, TUM, TUMack
SDSM
SSM
SRM
MAP

Security Model

The NMI platform operates within a C-V2X over-the-air environment and follows a zero-trust architecture. Accordingly, all data received via MQTT should be treated as untrusted until validated by the consuming system.

All messages are digitally signed at the source using IEEE 1609.2. While the platform enforces publishing rules, the MQTT transport does not guarantee message validity, origin, or correct topic usage.

Client systems are responsible for:

- Verifying message signatures
- Validating data integrity
- Confirming consistency and applicability of the payload

Subscribers are not required to possess certificates to receive data. Signature verification is optional at the protocol level but recommended for production systems requiring trust validation.

Security enforcement, including message signing and SCMS integration, applies primarily to data publishers.

Intended Use Cases

NMI is commonly used for:

- Traffic signal visualization and analytics
- Connected vehicle application development
- Work zone and infrastructure safety applications
- Simulation, testing, and research environments
- Integration with OEM, fleet, or roadside systems

Support

support@TrafficAuth.com

Versions

Version	Date	Notes
1.0	March 19, 2026	Initial release.
1.1	March 25, 2026	Geohash topics
1.2d	April 2, 2026	Geohash naming update, wording
1.2	May 1, 2026	v1.2 released

Appendix

Decoding Guidance

Decoding ASN.1 (UPER) payloads with asn1c (open source)

NMI payloads contain **UPER-encoded SAE J2735 messages** (inside an IEEE 1609.2 secured wrapper). A practical way to decode the J2735 ASN.1 content is to generate a native C decoder using the open-source **asn1c** compiler by Lev Walkin. The project explicitly supports **OER** (including Unaligned OER / UPER) with `oer_decode()` / `oer_encode()` in its runtime API table. GitHub

What asn1c is (and isn't)

- **Is:** An ASN.1 compiler that generates C (C++-compatible) types plus encode/decode functions for multiple encoding rules including OER. GitHub
- **Isn't:** A prebuilt J2735 decoder—you must generate code from the **J2735 ASN.1 module(s)** you plan to support.

Step 1 — Get the J2735 ASN.1 specification (source files)

You'll need the relevant **SAE J2735 ASN.1** module files (version matters). SAE distributes the official ASN.1 "source code" for J2735 via their standards portal (licensed access). SAE Mobilus

Tip: Align the J2735 ASN.1 version you compile against with the J2735 version used by the message producers you're consuming.

Step 2 — Install asn1c (Linux example)

A commonly used install flow is: clone → autoreconf → configure → make → make install. GitHub Wiki

```
# prerequisites (Ubuntu/Debian example)
sudo apt-get update
sudo apt-get install -y build-essential automake autoconf libtool bison flex

# build asn1c
git clone https://github.com/vlm/asn1c.git
cd asn1c
test -f configure || autoreconf -iv
./configure
make
make check
sudo make install

asn1c -v
```

Step 3 — Generate a J2735 decoder with OER/UPER support

Run `asn1c` over your J2735 ASN.1 module file(s). If your J2735 distribution includes multiple interdependent `.asn` files, pass them together (same invocation). GitHub

A typical pattern is:

```
# Example: compile interdependent ASN.1 modules together
asn1c J2735.asn <other_dependencies>.asn
```

Notes:

- `asn1c -h` / `man asn1c` provide option details; the project also provides PDFs (`asn1c-quick.pdf`, `asn1c-usage.pdf`). GitHub+1
- You will include the generated `.c/.h` files + the asn1c runtime support sources in your build.

Step 4 — Decode UPER (Unaligned OER) into a MessageFrame

Once code is generated, you'll typically decode into the **J2735 `MessageFrame`** (or whatever top-level PDU your feed provides). asn1c documents that OER decoding uses `oer_decode()` and supports *basic and Unaligned* OER variants. GitHub

Minimal C skeleton (illustrative):

```
#include <stdio.h>
#include <stdlib.h>
#include <asn_application.h>
#include <asn_internal.h>

// The generated header name depends on your J2735 ASN.1 set and asn1c output.
// Commonly you'll have something like:
#include "MessageFrame.h"

int decode_j2735_UPER(const uint8_t *buf, size_t len) {
    MessageFrame_t *msg = 0;

    // UPER uses the OER decoder in asn1c (Unaligned/basic handled by the OER
support).
    asn_dec_rval_t rval = oer_decode(0, &asn_DEF_MessageFrame, (void **)&msg, buf,
len);

    if (rval.code != RC_OK) {
        fprintf(stderr, "Decode failed: code=%d, consumed=%zu\n", (int)rval.code,
rval.consumed);
        ASN_STRUCT_FREE(asn_DEF_MessageFrame, msg);
        return -1;
    }

    // TODO: inspect msg->messageId and msg->value (content depends on J2735 schema)
    // TODO: route to application logic

    ASN_STRUCT_FREE(asn_DEF_MessageFrame, msg);
    return 0;
```

```
}
```

Step 5 — Where 1609.2 fits

Remember: the MQTT payload you receive from NMI is **not just J2735**—it includes an **IEEE 1609.2 security header with a digital signature** (per the NMI datasheet).

So your pipeline is typically:

1. parse/unwrap **IEEE 1609.2 secured message**
2. validate **digital signature**
3. extract the **UPER J2735 bytes**
4. decode UPER with **asn1c-generated** `oer_decode()`

Geohash Guidance

Implementations may use any geohash library that produces results consistent with the standard geohash algorithm and base32 encoding. Common libraries include ngeohash (JavaScript), geohash2 (Python), and mmcloughlin/geohash (Go).