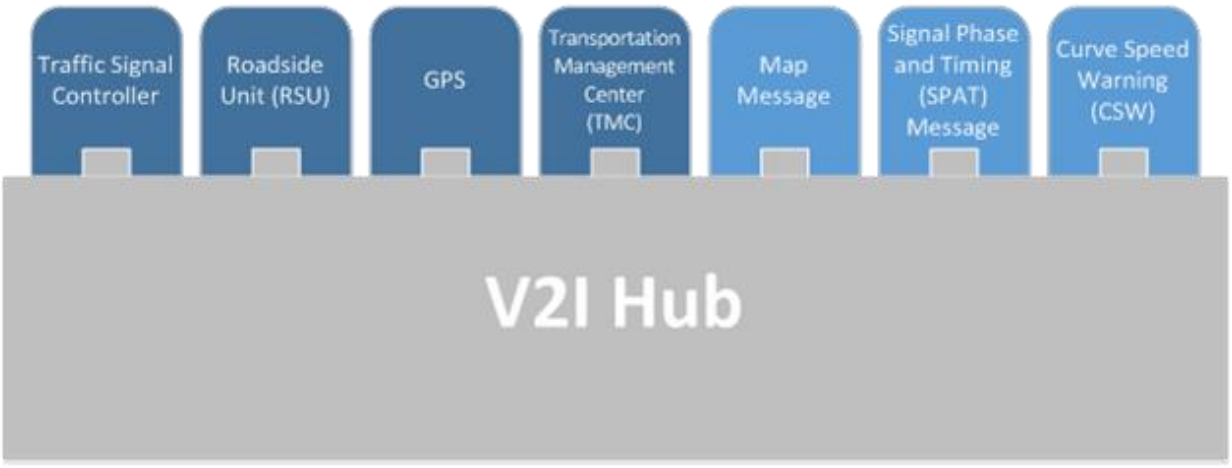
V2I Hub

Plugin Programming Guide

www.its.dot.gov/index.htm

Final Report – July 3, 2018
FHWA-JPO-18-647



U.S. Department of Transportation

Produced by Battelle Memorial Institute under DTFH61-12-D-00040
U.S. Department of Transportation
Office of the Assistant Secretary for Research and Technology
Joint Program Office

Notice

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
FHWA-JPO-647		

4. Title and Subtitle	5. Report Date
V2I Hub Plugin Programming Guide	July 3, 2018
	6. Performing Organization Code: Battelle

7. Author(s)	8. Performing Organization Report No.
Gregory Zink, Greg Baumgardner	

9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
Battelle 505 King Ave Columbus, OH 43201-2693	11. Contract or Grant No.: DTFH61-12-D-00040

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
Federal Highway Administration 1200 New Jersey Avenue, S.E. Washington, DC 20590	Final
	14. Sponsoring Agency Code: FHWA

15. Supplementary Notes

16. Abstract

Connected Vehicle technologies help reduce the number of driving related injuries and fatalities by allowing road users to be aware of potential dangerous situations on the road. There are two main types of Connected Vehicles communications, vehicle-to-infrastructure and vehicle-to-vehicle. Vehicle-to-infrastructure communication takes place between vehicles and deployed roadside communication devices, which capture vehicle generated data while providing information pertaining to safety, mobility, and environmental conditions

This programming guide contains information on how to create a plugin for the V2I Hub. This guide is designed for software developers who will develop plugins for the V2I Hub.

17. Keywords	18. Distribution Statement
Connected vehicle, V2V, V2I, safety, deployment, V2I Hub, IVP, SPaT, V2I Reference Implementation	Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	46	

Form DOT F 1700.7 (8-72) **Reproduction of completed page authorized**

Table of Contents

List of Figures

Executive Summary

The V2I Hub Programming Guide was developed during the V2I Reference Implementation project for developers of plugins for the V2I Hub in supporting Connected Vehicle (CV) technology. This programming guide gives details on how to create a plugin for the V2I Hub based on the Example Plugin created by Battelle. The Example Plugin can be obtained in the U.S. DOT's Open Source Application Development Portal (OSADP) repository, along with the other V2I Hub software that you will need to compile and setup to do your own V2I Hub development.

The V2I Hub consists of software modules written in C and C++ for Linux. The developer wanting to create a plugin for the V2I Hub must be familiar with C, C++ and Linux, and have a have a basic knowledge of object-oriented design. Some other important software development practices to be familiar with are threading, network communications, external library integration, and data serialization just to name a few. Development is done using the following tools: cmake, make, g++ compiler, and optionally eclipse. Developers should have familiarity with those tools to successfully create a V2I Hub plugin.

Additional information on the V2I Hub can be found in the following documents. It is suggested to start with the V2I Hub Guidebook.

- V2I Hub Guidebook
- V2I Hub Plugins
- V2I Hub Deployment Guide
- V2I Hub Software Configuration Guide
- V2I Hub Administration Portal User Guide

Chapter 1. Example Plugin

The Example Plugin showcases the basic functionality needed by a V2I Hub plugin. The plugin has simple Application Programming Interface (API) calls to create, route, and decode messages from the V2I Hub Core. Examples for setting status values, retrieving configuration values and writing logs are also in the example. The Example Plugin is a good foundation to modify when creating your own plugin for the V2I Hub.

Plugin Structure

The Example Plugin structure is as follows:

- Example Plugin
  - src
    - ExamplePlugin.cpp
    - SampleData.cpp
    - SampleData.h
  - CMakeLists.txt
  - manifest.json

The main source code for the plugin resides in the ExamplePlugin.cpp file. This C++ class extends the PluginClient class from the V2I Hub API. The PluginClient has methods that can be overwritten to retrieve settings and messages. The next section will describe the source code needed to do the basic functionality for a plugin, and chapter 2 will walk through the process of creating a plugin based on the Example Plugin. The SampleData files are simply examples on how to include other C++ source files into a V2I Hub Plugin. They are currently not used.

The CMakeLists.txt file contains the needed information for cmake to create the make files needed to compile the project. When creating a new plugin, this file will need to be modified with the plugins information.

The manifest.json file is used by the V2I Hub Core to setup communications between the core and plugin. The file contains a description of the plugin, the executable location, the IP address of the V2I Hub running the core application, port information for communication, messages that are produced by this plugin, and then finally the configuration values for the plugin. The majority of this file will remain untouched, except for the configuration, messageTypes, and description sections. More detail will be given on the modification of those sections in chapter 2.

Sample API Calls

Below are the most common V2I Hub plugin integration and functionality and how to code them in your plugin.

Get Configuration Values

A V2I Hub Plugin may need to contain certain settings that vary from one deployment to the next, or perhaps may require a specific key to be entered. These are examples of V2I Hub configuration values. Configuration values are defined in the manifest file, stored in the V2I Hub configuration database, and then retrieved by the plugin during execution. In the Example Plugin we see two main functions that help with configuration.

void UpdateConfigSettings()

This function updates an internal cache of all the needed Plugin parameters. In the case of Example Plugin, this function is called when the plugin starts and is registered (see State Changes), and when a configuration value is updated.

*void OnConfigChanged(const char *key, const char *value)*

This function is called automatically when a configuration value is changed. The *key* parameter is set to the name of the configuration parameter that has changed and the *value* is set to the new value of that configuration parameter. Although it is possible to check the *key* for the correct value to update in the cached settings of a Plugin, it is often the practice to call UpdateConfigSettings() to re-read all of the values, effectively ignoring the supplied parameters.

In the UpdateConfigSettings() function we can see the API call to retrieve a configuration value in two separate ways:

```
        int instance;
        GetConfigValue("Instance", instance);

        GetConfigValue("Frequency", __frequency_mon.get());
```

In the former case, the Plugin will retrieve its configured value with the key “Instance” and store that value as an integer into the variable *instance*. If the key is found in the configuration database for that plugin, it will set the *instance* variable to the value found in that database. Since all data is stored as strings, there is an implicit conversion (using the default lexical cast from Boost) to an integer type. If the conversion fails, or the key is not found, then the variable is unset. In the code above, this could lead to a programming error because there is no default value for the variable. Also, note that in this example *instance* is defined locally, thus can only be used within the scope of that variable.

In the second example, the “Frequency” parameter is retrieved and stored in a class variable. This could be the simple private variable *_frequency*, which is a thread-safe “atomic” variable of unsigned integer type. However, this example also uses some redirection provided by the V2I Hub Plugin API to add some desired functionality. The PluginDataMonitor.h header adds a type accessible by the macro DATA_MONITOR() that wraps *_frequency* inside a class that knows when that parameter actually changed values. A common use case would be to log a message that the value has changed. In the UpdateConfigSettings excerpt above, the value of *_frequency* is updated when the get() function returns that reference out of the wrapper class. See the Data Monitor section for more on using data monitors.

Subscribing to Messages

To receive message from the V2I Hub, your plugin will need to subscribe to the message types you wish to receive. This is done by using the AddMessageFilter function from the API. In our ExamplePlugin, the constructor of the plugin is registering for three different messages. The pattern for subscribing to messages is below.

void AddMessageFilter(const char *type, const char *subtype)

The simplest form of AddMessageFilter() is to directly pass the V2I Hub message type and subtype. Any message that matches that type and subtype will be forwarded to this plugin, invoking the call back. A special "*" input can be used for subtype to request **all** message subtypes of a given type. The Example Plugin does not directly use this form of subscription.

void AddMessageFilter<MessageType>(this, MessageHandlerFunction)

This templated function formally describes exactly what message is expected to be received and how it is to be handled. Whereas the basic form above only provides a type and subtype and the handling is intended to be inside the call back function, this form describes the exact structure of the message that is expected, which includes the type and subtype, and a specific call back function to use to handle this message type. Each handler function must be structured in the form:

void HandlerFunctionName(MessageType &msg, routeable_message &routeableMsg)

The first parameter *msg* is the converted V2I Hub message of that type defined in the filter request, and the second parameter *routeableMsg* is the broader V2I Hub routing message, which contains the routing information used by V2I Hub, plus a "payload" that is an encoded version of *msg*. The latter parameter is often only used if timing information is needed.

As stated above, there are three messages that Example Plugin is looking to handle, therefore there are three separate handler functions. A full list of supported API messages can be found in Appendix C. When handling a message please refer to the U.S. DOT's OSADP repository for the message structure source.

void HandleMapDataMessage(MapDataMessage &msg, routeable_message &routeableMsg);

void HandleDecodedBsmMessage(DecodedBsmMessage &msg, routeable_message &routeableMsg);

void HandleDataChangeMessage(DataChangeMessage &msg, routeable_message &routeableMsg);

SubscribeToMessage();

The SubscribeToMessages function must be called after the filters are setup, otherwise the plugin will not receive the wanted messages. This function can be invoked more than once, if other filters are added. Currently, the only way to un-subscribe to a message is to restart the plugin.

State Changes

A running plugin will go through a simple set of state transitions with the V2I Hub core server before the plugin is ready to function. The main states are disconnected, connected, registered and error. The default state is disconnected, which transitions to connected when the V2I Hub core server is contacted. After some brief interaction of registration, the state will then become registered. Any failures will resort the plugin to an error state.

Once a plugin is registered, it is safe to retrieve configuration settings from the database. Most plugins will operate as long as the plugin state is not error. In our ExamplePlugin it continues to operate in a loop while we are not in an error state. Once the error state occurs, the plugin exits. The V2I Hub will try to restart the plugin after some configurable timeout.

Whenever a state changes, the plugin will be notified by following call back.

void OnStateChange(IvpPluginState state)

The *state* parameter is an enum value that specifies the new state of the plugin. The typical scenario as seen in the Example Plugin is simply to call the UpdateConfigSettings() function when the state changes to registered. Additional functionality can be handled on disconnect and connect state changes in the OnStateChange call back.

Logging

Logging is done by the V2I Hub core to tmxcore.log using API commands. Logging level can be setup by using the following code:

```
FILELog::ReportingLevel() = FILELog::FromString("DEBUG");
```

This sets logging level to debug. Other levels of logging include: ERROR, WARNING, and INFO. The default is typically INFO, although the configuration parameter “LogLevel” can be set in the manifest to change that.

Once logging level has been set, logging can be done by using the following:

```
PLOG(logINFO) << "My Message";
PLOG(logDEBUG) << "My message: " << intVariable;
```

The levels of logging are logINFO, logDEBUG, logWARNING, and logERROR. There are also more detailed logDEBUG*N* where *N* can be 1-4, but those are mostly reserved for frequently repeated debugging messages that are typically not needed to be seen.

Message Handling

As stated in the Subscribing to Messages section, there are two types of message filters that can be applied. In either case, receiving a message of subscribed type and subtype will automatically invoke the following call back API on the Plugin:

void OnMessageReceived(IvpMessage *msg)

The *msg* parameter is a pointer to the full V2I Hub message structure. This simple C-style struct contains the routing information such as type, subtype, the source plugin and a receive timestamp, along with the payload, which is the actual message, as well as the encoding of the payload. In this handler, however, the developer is responsible for checking the type and retrieving the payload.

void HandleMessageType(MessageType &msg, routeable_message &routeableMsg)

This handler function is associated with the templated AddMessageFilter() function described above in the Subscribing to Messages section. Because this function registers the exact C++ class to use for messages of a given type and subtype, the incoming message payload can automatically be converted to that type. The result is passed into the handler function in the *msg* parameter. The *routeableMsg* parameter is synonymous with the full V2I Hub message pointer passed into the OnMessageReceived() function, but as a fully encapsulated C++ class.

Because the handler in this form has already been converted to a message class, these functions typically can be implemented in few lines of code. In the Example Plugin, the handler functions just use the logging API to write out the message received. The tradeoff for using these handlers, however, is that the time required to convert the incoming message payload to the C++ class may be sub-optimal, particularly if most of the data in the payload is ignored. Therefore, take precaution in choosing how to handle messages that are incoming at a high rate. See the Plugin Threading section for optimized message handling.

Message Decoding

The incoming V2I Hub message payload are generally encoded for performance reasons. Some messages may be just string information, thus encoded as a simple character string, while there may also be encodings of numeric or Boolean data. Each of these can be decoded directly from the payload, and the following API exists in the **routeable_message** class to obtain the payload value as a string: Due to performance issues with the iMX6 architecture, using this method of decoding message is not recommended for high frequency message types, like SPaT or BSM.

std::string get_payload_str()

Although there is no direct function to convert the string values to a Boolean or a number, such an operation is easily done with the aforementioned Boost lexical_cast operation.

The most common payload encoding, however, is a JavaScript Object Notation (JSON) form. The get_payload_str() function in this case, will return the entire JSON as a string. This may be handy for storing or logging and can even be used in conjunction with regular expressions. A direct API is available to convert the JSON information into any V2I Hub message class.

template <typename MessageType> MessageType get_payload()

See the Sample TIM Message Creation section for more information on how a V2I Hub Message must be created.

A special case of the string encoding is what is referred to as “hexstring” encoding. In this case, the string represents a series of bytes written as a string of hex characters. For example, the “bytearray/hexstring” encoding of the string “Hello World!” is “48656c6c6f20576f726c6421”. The most important use of this concept in V2I Hub is to encode the SAE J2735 messages using the available ASN.1 encodings. A “asn.1-ber/hexstring” encoding implies that the message as defined in the J2735 ASN.1 specification is encoded with X.609 Basic Encoding Rules (BER) and stored in the V2I Hub message as a hexstring. The BER encodings are exclusively from the 2015 ASN.1 specification, whereas the 2016 specifications are required to be encoded with the hexstring for Unaligned Packed Encoding Rules (UPER), which is denoted by “asn.1-uper/hexstring”.

There are helper classes in V2I Hub API that assist in decoding a J2735 encoded payload. For example, if the type of the message is “J2735” and the subtype is “BSM”, then the BsmEncodedMessage class can be used to decode the hexstring payload to a BsmMessage class. The decoded message contents can be manipulated through the C-style structures generated from the ASN1C tool, or through its (Extensible Markup Language (XML) representation. See the TmxJ2735Message section in Appendix C for more information on the J2735 message structure.

In order to directly use the OnMessageReceived() function, the programmer must be able to correctly decode the incoming messages. However, with the specific message type handling, the decode is done automatically by the default OnMessageReceived() function in PluginClient. Therefore, simply registering for a BsmMessage would ensure that the call back to the BSM handler function would contain an already decoded BsmMessage.

In the Example Plugin, the three handler functions are set to receive: a MapDataMessage, which is the decoded version of the J2735 MAP message; a DecodedBsmMessage, which is an exploded JSON version of the J2735 BSM; and a DataChangeMessage, which is an internal message for denoting that a monitored value has been changed.

Message Encoding

Encoding a V2I Hub message is effectively the opposite of decoding one. The following API is available in the routeable_message class to assist:

void set_payload(std::string payload)

Encode the payload as a direct string.

void set_payload(bool payload)

Encode the payload as a Boolean type.

void set_payload(int number)

Encode the payload as an integer type.

void set_payload(double number)

Encode the payload as a double type.

void set_payload(tmx_message<JSON> &payload)

Encode the payload as a JSON type. There is also a form to encode as a different format like XML, but all of those payloads are directly converted to a string, therefore have a large performance cost.

void set_payload_bytes(const byte_stream &bytes)

Encode the payload as a hexstring. The encoding by default is set to a “bytearray/hexstring”. Therefore, if the hexstring is ASN.1 encoding, the message encoding value must be changed to reflect the correct encoding.

A V2I Hub J2375 encoded message type understands the correct encodings for passing J2735 messages, version 2016. Therefore, it is always recommended to use this class to handle J2735 messages. There is a simple function available to correctly initialize the message with an encoded payload:

void initialize(MsgType &payload)

The *payload* parameter must be of the correct J2735 message type. For example, a BsmEncodedMessage will only initialize with a BsmMessage. It is also important to note that the most common errors occur inside the encoding of a J2735 message because the *payload* parameter does not contain the correct data and therefore cannot be encoded per the ASN.1 rules. Typical issues include missing values, certain values out-of-bounds, incorrect bit mask settings and invalid array construction. Therefore, it is recommended to always separately test the encoding when generating J2735 messages.

Sample TIM Message Creation

Below is code to create a TIM message with one geographic lane designated by xy offsets from a reference point for a speed restriction zone of 35 MPH. The Curve Speed Warning (CSW) plugin from the U.S. DOT’s OSADP repository contains a more detailed solution for creating a TIM message that contains multiple zones for a speed restriction curve zone. The sample code below is taken from that CSW plugin.

```
//Create a TIM message for a 35 MPH speed restriction
TravelerInformation _tim;

TravelerInformation* tim;

memset(tim, 0, sizeof(TravelerInformation));

//Set Packet ID
tim->packetID = (UniqueMSGID_t *)calloc(1, sizeof(UniqueMSGID_t));
tim->packetID->buf = (uint8_t *)calloc(9, sizeof(uint8_t));
tim->packetID->size = 9 * sizeof(uint8_t);

uint64_t time = GetMsTimeSinceEpoch();

time_t time_sec = time / 1000;
```

```
struct tm *tm;
tm = gmtime(&time_sec);
uint32_t minuteOfYear = (tm->tm_yday * 24 * 60) + (tm->tm_hour * 60) + tm->tm_min;

tim->packetID->buf[0] = 0x00;
tim->packetID->buf[1] = (minuteOfYear & 0xFF0000) >> 16;
tim->packetID->buf[2] = (minuteOfYear & 0x00FF00) >> 8;
tim->packetID->buf[3] = minuteOfYear & 0x0000FF;
tim->packetID->buf[4] = 0x00;
tim->packetID->buf[5] = 0x00;
tim->packetID->buf[6] = 0x00;
tim->packetID->buf[7] = 0x00;
tim->packetID->buf[8] = 0x00;

//Set initial values to NULL
tim->timeStamp=NULL;
tim->packetID=NULL;
tim->urlB=NULL;
tim->regional = NULL;

//Set start time to yesterday

uint64_t time = GetMsTimeSinceEpoch();

time_t time_sec = time / 1000;
struct tm *tm;
tm = gmtime(&time_sec);

int dayOfYear = tm->tm_yday - 1;
if (dayOfYear < 0)
        dayOfYear = 364;
frame->startTime = dayOfYear * 24 * 60;

// Set the duration (minutes) to its max value.
frame->duratonTime = 32000;

frame->startYear = NULL;
frame->url = NULL;
// Priority is 0-7 with 7 the highest priority.
frame->priority = 5;


//Set Region
TiDataFrame *frame = (TiDataFrame*)calloc(1, sizeof(TiDataFrame));

frame->frameType = TravelerInfoType_advisory;

frame->msgId.present = TravelerDataFrame__msgId_PR_furtherInfoID;
std::string tempString = "00";
OCTET_STRING_fromString(&(frame->msgId.choice.furtherInfoID), tempString.c_str());

//Set Geographical Path with XY Nodes
GeographicalPath *geoPath = (GeographicalPath*)malloc(sizeof(GeographicalPath));

geoPath->description = (GeographicalPath::GeographicalPath__description*)
        malloc(sizeof(GeographicalPath::GeographicalPath__description));
geoPath->regional = NULL;

geoPath->name = NULL;
```

```
geoPath->id = NULL;
geoPath->directionality = NULL;
geoPath->closedPath = NULL;
geoPath->direction = NULL;
memset(geoPath, 0, sizeof(geoPath));

geoPath->description->present = GeographicalPath__description_PR_path;

geoPath->description->choice.path.scale = NULL;
geoPath->description->choice.path.offset.present = OffsetSystem__offset_PR_xy;
geoPath->description->choice.path.offset.choice.xy.present = NodeListXY_PR_nodes;

//Add Anchor (Reference Point) at 39.12345, -83.98765
Position3D *anchor = (Position3D*)malloc(sizeof(Position3D));

anchor->regional = NULL;
anchor->elevation = 0 * 10;
anchor->lat = (long)(39.12345 * 10000000.0);
anchor->Long = (long)(-83.98765 * 10000000.0);

geoPath->anchor = anchor;

//Set lane width = 350 cm
LaneWidth_t * laneWidth = (LaneWidth_t *)malloc(sizeof(LaneWidth_t));
long lWidth = 350;
*laneWidth = lWidth;
geoPath->laneWidth = laneWidth;

//Add Direction of Use of Forward
DirectionOfUse_t* directionOfUse = (DirectionOfUse_t *)malloc(sizeof(DirectionOfUse_t)
);
*directionOfUse = DirectionOfUse_forward;
geoPath->directionality = directionOfUse;

//Add two nodes
NodeXY * node = (NodeXY *)malloc(sizeof(NodeXY));
node->attributes = NULL;
node->delta.present = NodeOffsetPointXY_PR_node_XY6;

int16_t xOffset = 0;
int16_t yOffset = 0;
int16_t eOffset = -0;
bool hasElevation = false;

xOffset = 450;
yOffset = -256;

node->delta.choice.node_XY6.x = xOffset;
node->delta.choice.node_XY6.y = yOffset;

ASN_SEQUENCE_ADD(&nodeSet->list, node);

node->attributes = NULL;
node->delta.present = NodeOffsetPointXY_PR_node_XY6;

int16_t xOffset = 0;
int16_t yOffset = 0;
int16_t eOffset = -0;
bool hasElevation = false;
```

```
xOffset = 750;
yOffset = 500;

node->delta.choice.node_XY6.x = xOffset;
node->delta.choice.node_XY6.y = yOffset;

ASN_SEQUENCE_ADD(&nodeSet->list, node);

//add region to frame
asn_set_add(&frame->regions.list, region);
//add frame to frame list
asn_set_add(&tim->dataFrames.list, frame);


TiDataFrame *frame = tim->dataFrames.list.array[0];
frame->content.present = TravelerDataFrame__content_PR_advisory;

//Add ITIS Code for speed restriction
ITIScodesAndText__Member* speed_restriction_member = (ITIScodesAndText__Member*)malloc
(sizeof(ITIScodesAndText__Member));
speed_restriction_member->item.present = ITIScodesAndText__Memberitem_PR_itis;
speed_restriction_member->item.choice.itis = 2564; //speed restriction itis code
ASN_SEQUENCE_ADD(&advisory->list, member);

//Add ITIS Text for 35
std::string text ="35";
int textLength = text.length();

ITIScodesAndText__Member* speed_limit_text_member = (ITIScodesAndText__Member*)malloc(
sizeof(ITIScodesAndText__Member));
speed_limit_text_member->item.present = ITIScodesAndText__Memberitem_PR_text;
speed_limit_text_member->item.choice.text.buf = NULL;
OCTET_STRING_fromString(&(member->item.choice.text), text.c_str());
ASN_SEQUENCE_ADD(&advisory->list, member);

//Add ITIS Code for MPH
ITIScodesAndText__Member* speed_limit_units_member = (ITIScodesAndText__Member*)malloc
(sizeof(ITIScodesAndText__Member));
speed_limit_units_member->item.present = ITIScodesAndText__Memberitem_PR_itis;
speed_limit_units_member->item.choice.itis = 8720; //MPH itis code
ASN_SEQUENCE_ADD(&advisory->list, member);
```

Broadcasting

A V2I Hub routeable_message class represents the encoded message passed through the V2I Hub core server. Therefore, the contents are consistent between the plugin that sends the message and the plugin that receives it. Much of the previous sections discussed how a message is received in the plugin, but a plugin may also be a producer of any messages, whether they are J2735 messages or internal ones. The following API is available in the PluginClient parent class to handle the broadcast of a message through the V2I Hub system:

void BroadcastMessage(const IvpMessage *ivpMsg)

This is the function that directly emits the provided *ivpMsg*. This function is more typically invoked by the other forms below that expect a the routeable_message class reference.

void BroadcastMessage(const tmx::routeable_message &routeableMsg)

This function directly emits the provided *routeableMsg*. This is an efficient form because it just uses the underlying C-style structure contained in the class.

void BroadcastMessage(tmx::routeable_message &routeableMsg)

This function directly emits a *copy* of the provided *routeableMsg*. This is less efficient than the previous form because the underlying C-style structure is first duplicated. The copy, however, is immediately destroyed after it is used.

Since much of the work of a plugin is in building the message to send, most plugins do not really need to worry about the specifics of the V2I Hub routing messages. Therefore, there is a convenience function that can take any V2I Hub message and perform the encoding and the broadcast in one swoop:

template <typename MsgType> void BroadcastMessage(MsgType& message)

This function routes the message through the V2I Hub. Below is code to send the TIM message created above by creating a TimEncodedMessage. Initialize the encoded message by passing a TimMessage which was created by a TravelerInformation. Set the TimEncodedMessage flags to route from the DSRC radio, if the message is to be picked up by the DSRC Message Manger and broadcast out the RSU. Set the DSRC Metadata to its initial state, in this case out channel 178 with Personal Service Identifier (PSID) 0x8003. The values set here are used as defaults, but the PSID and channel will be overwritten by the configuration in the DSRC Message Manager. Create a routable message based on the encoded message and call BroadcastMessage with the routable message to send the message into the V2I Hub.

```
TimMessage timMsg(_tim);

TimEncodedMessage timEncMsg;
timEncMsg.initialize(timMsg);
timEncMsg.set_flags(IvpMsgFlags_RouteDSRC);
timEncMsg.addDsrcMetadata(178, 0x8003);

routeable_message *rMsg = dynamic_cast<routeable_message *>(&timEncMsg);
if (rMsg) BroadcastMessage(*rMsg);
```

Status

Status information can be set by the plugin to be used by monitoring in the admin portal. Status information contains a key and value. The code to set a status value is shown below:

template<typename T> bool SetStatus(const char *key, T value)

The *value* parameter can be either a string, integer, double, or Boolean value. The status values are not persistent between executions of the plugin, but it is still recommended to create default values to use when the plugin initializes. The Example Plugin writes out a count of J2735 maps that have been received when the number changes.

Error Handling

Errors can occur in the plugin, and it is always useful to have information available to help debug a problem that occurs. Aside for logging, any exceptions that occur in the plugin can be handled using the following function:

void HandleException(std::exception &ex)

By default, the exception will be logged, with all helpful debug information, and the plugin will exit. If the problem is recoverable, it is possible to simply log the exception and continue.

Certain V2I Hub API errors can occur, which automatically invoke the following call back:

void OnError(IvpError err)

The *err* parameter specifies what problem occurred. Some problems include errors in the manifest or missing configuration keys. By default, the error is simply logged, but this function can be overridden to provide better error handling.

Chapter 2. Plugin Creation from Example Plugin

Plugin Copy and Modification

The V2I Hub Example Plugin source code includes the base source file ExamplePlugin.cpp, which is the perfect place to start development of a new Plugin. Simply copying the source file to a new name will provide the basic structure of a new V2I Hub Plugin. The following steps detail how to create a new class called MyFirstPlugin from the V2I Hub Example Plugin.

1. Copy the tmx-exampleapps/ExamplePlugin/manifest.json to a new file v2i-hub/MyFirstPlugin/manifest.json.

2. Modify the manifest.json file with the appropriate description, messageTypes and configuration. The name, version and exeLocation will be set automatically at compile time.

3. Copy the tmx-exampleapps/ExamplePlugin/src/ExamplePlugin.cpp to a new file v2i-hub/MyFirstPlugin/src/MyFirstPlugin.cpp.

   Note that this file contains both the class definition and the implementation. This is for convenience since no other V2I Hub software will likely need to use the MyFirstPlugin class. That said, it would be easy to split these into MyFirstPlugin.h with the definition and MyFirstPlugin.cpp for the implementation.

4. In the class source file, change the namespace declaration to MyFirstPlugin.

5. In the class source file, change the class name to MyFirstPlugin, including in the class definition (.h), in the constructor and destructor names, and in the fully qualified declaration of the function implementations. The typical parent class should be PluginClient.

6. Modify the class definition with any new private data or handler functions.

7. Re-write the MyFirstPlugin::UpdateConfigSettings() function to retrieve any configuration parameters added to the manifest.json.

8. Re-write the MyFirstPlugin message handler functions for receiving messages from V2I Hub.

   Note that this also includes changes to the constructor to register the handler function.

9. Re-write the MyFirstPlugin::Main() function to be the main processing loop of the new Plugin.

10. Change the main() function to call run_plugin() with MyFirstPlugin::MyFirstPlugin type in order to launch the new Plugin.

Compiling with OSADP V2I Hub

The V2I compilation processes uses CMake (version 3.5 or higher) to build Makefiles. The steps are hierarchical, meaning first the base V2I Hub platform must be compiled and installed on the target platform prior to the V2I Hub plugins. The V2I Hub plugins expect that the associated platform libraries are present or else CMake fails.

```
$ cd tmx
$ cmake .
$ make
$ sudo make install
```

The CMakeLists.txt file defines the CMake dependencies and operations for a build. In V2I Hub, this file first finds the tmx-plugin.cmake file that was built above. This CMake package file might be located in the compiled area or could be installed on the system. Then, CMake will process any Plugin sub-directories that contain their own CMakeLists.txt file. So, presuming that the MyFirstPlugin directory exists within the V2I Hub source tree, a new CMakeLists.txt can be created directly in the MyFirstPlugin folder:'

```
PROJECT ( MyFirstPlugin VERSION 3.0.0 LANGUAGES CXX )

SET (TMX_PLUGIN_NAME "My Plugin")

BuildTmxPlugin()

TARGET_LINK_LIBRARIES ( ${PROJECT_NAME} tmxutils )
```

The first line in this file declares the project name, which also becomes the executable name, the version of the new Plugin, and that this is a C++ project. The next line sets the name of the Plugin, which will be used in the V2I Hub portal and command lines. The last two lines tell CMake to run the build macro for a V2I Hub Plugin and sets dependencies for any Plugin. Now, the Plugin can be compiled with the following commands:

```
$ cd v2i-hub
$ cmake .
$ make MyFirstPlugin
```

Custom Messages

As seen in Chapter 1, a V2I Hub message is essentially some encoded string that is routed through the core of the system by its specified type and subtype. The most common message types are encoded using a JSON string. As such, it is very easy to create custom messages for V2I Hub as JSON messages. The default JSON-encoded message class is **tmx::message**, so any sub-class of this can be used as the payload to a V2I Hub routing message. The **message** class itself is flexible enough to be used as a payload, but the data contents inside the object can only be added via manipulation of the JSON string. Data can be retrieved through a generic getter function, which requires a default value in case the data does not exist. For example:

```
tmx::message myMsg;
myMsg.set_contents("{\"name\":\"V2I Hub user\",\"age\":30}");
string name = myMsg.get<string>("name", "");               // name is now "V2I Hub use
r"
int age = myMsg.get<int>("age", -1);                       // age is now 30
```

Although using the generic **message** class is sufficient, it sometimes is nicer to have a defined class so that it is clear what data, otherwise known as an *attribute*, is supposed to be available in the message. The V2I Hub API contains the tools needed to construct a custom C++ message class. It is best to create the class as a header-only (e.g. MyMessage.hpp) definition so that it can be used across the system without a separate library. The only downside to this is that if the definition changes, it requires a recompile of every Plugin that uses that message class. A custom message class should extend **tmx::message** for JSON encodings, must have a default no-argument constructor, and should build the attributes through the following macro:

std_attribute(container, type, attributeName, defaultValue, condition)

The *container* is always the this->msg variable for a **tmx::message** type. The *type* is a primitive type for the attribute. The *attributeName* is used to automatically generate getter and setter functions in the form get_*attributeName*() and set_*attributeName()*. The *defaultValue* is used if the attribute is not currently stored in the container. The default is always stored back in the container after the first access. The *condition* is an *if*-style check that is used in the setter function to check for valid values. This is left blank if no validation is needed.

The above custom message can be created in MyMessage.hpp:

```
class MyMessage: public tmx::message {
public:
    MyMessage() { }  // Required no-argument constructor
    MyMessage(std::string name, int age) {
        // An example constructor to prepopulate
        set_name(name);
        set_age(age);
    }
```

```
   std_attribute(this->msg, std::string, name, "", )        // Generates get_name() and
set_name()

   std_attribute(this->msg, int, age, 0, value >= 0)        // Generates get_age() and
set_age() positive value
}
```

Then, the following code can be used:

```
MyMessage myMsg("V2I Hub user", 30);
cout << myMsg << endl;    // Write the JSON
if (myMsg.get_age() < 20) {
   cerr << "Sorry, " << myMsg.get_name() << ", but you are too young for this program.
" << endl;
}
```

There may be a case when C++ class cannot be created for a custom message. For example, commonly there may be some older C-code that has an internal structure. Instead of moving to C++, that structure can be wrapped into a **tmx::message** for transmission and receipt. This can be done with the **tmx::auto_message** class, although it is slightly more work to associate the name of the attributes with their contained value, plus strings may need to be handled separately due to pointers instead of objects. Here is the same example using an auto_message for sending and receiving:

```
// This is the old C-style struct
struct MyMessage {
         char *name;
        int age;
};

// Fill in the message information
MyMessage myMsg;
myMsg.name = strdup("V2I Hub user");       // Will have to free this pointer
myMsg.age = 30;

// Unfortunately, a char * needs to be converted to a C++ string
string tmpStr(myMsg.name);

// Create an auto_message, assigning an attribute name to each value
tmx::auto_message autoMsg;
autoMsg.auto_attribute<MyMessage>(tmpStr, "name");
autoMsg.auto_attribute<MyMessage>(myMsg.age, "age");
cout << autoMsg << endl;   // This will write out the data in JSON

// Create a routeable message for broadcast
tmx::routeable_message v2iMsg;
v2iMsg.set_type("Internal");      // Can be anything.  Just be consistent
v2iMsg.set_subtype("Message1");   // Can be anything.  Just be consistent
v2iMsg.set_payload(autoMsg);      // Payload contents has "json" encoding

this->BroadcastMessage(v2iMsg);
```

To receive this message and convert back to the C-style structure, it is unwrapped from an auto_message.

```
tmx::auto_message recv;
recv.set_contents(ivpMsg.get_payload_str());       // Returns the JSON string

MyMessage recvMyMsg;
// The name attribute is a string, copy to a new char *
recvMyMsg.name = strdup(recv.get<string>("name", "Unknown").c_str());
recv.auto_attribute<MyMessage>(recvMyMsg.age, "age");

cout << "Hello " << recvMyMsg.name << ".  You are " << recvMyMsg.age << endl;

// Do not forget to free the char *
```

Adding Custom Functionality

A V2I Hub Plugin that has any real value would require much additional functionality beyond what is shown in the Example Plugin. Note the following suggestions when adding features to a new Plugin:

1. The Plugin functionality can expand multiple C++ source files and classes, but only the PluginClient implementation class contains the API for obtaining configuration value, broadcasting messages and receiving messages. Therefore, some additional code is commonly required to pass a pointer to the Plugin around for common V2I Hub API requests.

2. Integrating with hardware can be done easily in a Plugin, including using existing C code, but it is always best to separate the hardware interface component with the Plugin component.

3. Multiple threads are running in a Plugin. See Plugin Threading section for more details.

4. The message API provides an excellent base for passing data around, and not just through the V2I Hub Core Server. Consider using a **tmx::message** type to build a basic data adaptor for configuration data as well. The JSON string can be stored in a configuration parameter with attributes as a type-safe way to retrieve the required data. Variable length JSON arrays are supported in any message class through two static functions called to_tree() and from_tree().

5. CMake is capable of finding most build dependencies for a Plugin. See cmake.org.

6. The following chapter will discuss solutions to typical problems and issues needed to create a more functional Plugin.

Chapter 3. Advanced Plugin Capabilities

Frequency Throttle

There a several reasons that certain applications require specific timing of tasks. One example for V2I Hub plugins would be send out a message within a certain time frame, say ten times a second. This can be done several ways, but the V2I Hub provides a convenient class to assist in such timing. The FrequencyThrottle class found in FrequencyThrottle.h provides a very simple interface that remembers how long ago an event occurs. The default measurements are in milliseconds, but that is configurable to other granularities. So, to send a message out ten times a second, a FrequencyThrottle should be created for 100 milliseconds. Each throttle has a key, so the type of the key must be specified. A simple case is to use an integer key.

```
FrequencyThrottle<int> throttle(chrono::milliseconds(100));
```

This is telling the FrequencyThrottle class to count if 100 ms or more has passed since any specific key was last updated. Presuming there is some message available to send, the Monitor() function is used to check every 5 ms or so if it is time to send. The longer the wait before checking again, the smaller the CPU impact, but the more inaccurate the 100 ms sending becomes.

```
while (true) {
        if (throttle.Monitor(1)) {
                this->BroadcastMessage(myMsg);
        }

        this_thread::sleep_for(chrono::milliseconds(5));
}
```

One thing to note is that if *myMsg* is updated in a separate thread, then this code is not thread-safe. Locking is required, and a copy should be made instead. Secondly, the key in this case is hard-coded to 1 because there really is only one key in the throttle. The class is designed, however, to support any number of different keys. So, you can throttle on some numeric ID of an incoming message, for example, and only process one message every so many milliseconds. In that case, the RemoveStaleKeys() function is available to remove any keys that have not been seen in some time. Finally, there is a Touch() operation to update the last access time of that key, if found, so it would not be stale.

Data Monitor

As seen in the Example Plugin, a data member of the class can be monitored for changes by wrapping a PluginDataMonitor container class around it. The actual check for changes is done by the check() function but is **not** automatic.

```
__frequency_mon.check();
```

A nice feature of the check() operation is that it automatically will send a message back to the Plugin if the data changes. Therefore, the Plugin should create a message handler to receive those messages, just like the Example Plugin has done.

```
void ExamplePlugin::HandleDataChangeMessage(DataChangeMessage &msg, routeable_message
&routeableMsg)
{
        PLOG(logINFO) << "Received a data change message: " << msg;

        PLOG(logINFO) << "Data field " << msg.get_untyped(msg.Name, "?") <<
                         " has changed from " << msg.get_untyped(msg.OldValue, "?") <<
                         " to " << msg.get_untyped(msg.NewValue, to_string(_frequency)
);
}
```

The DataChangeMessage is just a JSON string that contains a Name parameter with OldValue and NewValue, all as strings. In the example above, the message is just logged. Therefore, if a Plugin is required to note exactly when its configuration data changes during execution, using a data monitor provides a simple solution.

Plugin Keep Alive

The V2I Hub core server is responsible for starting and stopping the Plugins. By default, the system will track how frequent messages are being sent from the Plugin. So, for example, the MapPlugin is expected to generate a J2735 MAP Message every second, so if after five seconds no message has been seen, the system assumes the Plugin has failed somehow and automatically restarts it.

Since not all plugins generate messages, one is automatically injected as a simple keep alive so that the V2I Hub server does not stop execution of that Plugin. The amount of time expected between keep alive messages is at most eight minutes and 20 seconds (based on 0.5M milliseconds), therefore the default keep alive message will be sent at least every eight minutes, provided that no other message was sent during that time. This capability is implemented using a Frequency Throttle to monitor how often any message was sent for the plugin. The BroadcastMessage() function touches the throttle in order to reset the count, but if roughly 8 minutes pass without calling that function, then a keep alive message is sent. The actual throttle minute interval can be updated by configuration parameter *KeepAliveFrequency*.

Plugin Threading

Every V2I Hub plugin that publically inherits from the PluginClient parent class comes with a built-in threading model. First, there is a main thread, which is kept alive within the Main() function. This main thread is typically used to produce any synchronous messages for the plugin, like TIM or SPaT. Therefore, the thread should be designed to wake up, send a message, and go back to sleep. Every message that is received by that plugin will also spawn a new thread for handling. The implication, therefore, is that receiving messages can occur at the exact same moment as creating a message or receiving others. Some of the most important threading concerns for a plugin, as well as tools to help resolve them, are documented in this section.

Because multiple operations could be happening simultaneously in a V2I Hub Plugin, it is key to ensure proper thread safety. For example, if the goal of the Plugin is to create BSM based on current position of the vehicle, the BSM may be created within threads of incoming messages and broadcast in the Main thread. Since memory is shared between the two or more threads, it is common to use a locking mechanism to ensure thread safe access to the common data. The most commonly used form of locking is a std::mutex, which is sometimes wrapped within a local std::lock_guard to ensure that the lock is always released when finished. Here is an example declaration of a mutex:

```
std::mutex myLock;
```

Then, the critical block of thread safe code would look like:

```
{
std::lock_guard<std::mutex> lock(myLock);
// Thread safe now.
// Do the data operations here
}
```

Even though the above code examples produce a neat, thread safe Plugin, there are some performance considerations. Namely, the time it takes for the critical sections of code to complete ultimately determines how many messages can be processed. For example, if broadcasting the message takes 20 ms, and the location information are already coming in 10 times a second, plus accounting for the time it takes to decode the message, which on some platforms may be much worse than others, then it is only realistically possible to send a single BSM roughly every 150 ms or so, below the desired rate for a BSM of 10 times per second. Therefore, some location messages must be dropped. Using a Frequency Throttle would help slow down the frequency of message handling.

However, not every message type has a predictably known rate. For example, an application that is expected to handle incoming traffic from one intersection plus up to twenty other vehicles can receive as little as 11 messages per second (1 MAP, 10 SPaT) up to a maximum of 211 (Same plus up to 20 BSMs). Depending on the architecture of the application, the V2I Hub can process more messages using appropriate threading. Numbers represented here are used as real-world examples from our testing. In some scenarios there could be more BSMs broadcasting. In this case, spin locking in the various threads is in reality a very bad idea since the receipt of many of the messages may come in much too late to have

any practical safety application. For this reason, V2I Hub comes with a set of helper classes that assist in adding lock-free worker threads to a Plugin for optimized performance of message handling.

LockFreeThread.h

The base class for lock-free threading in V2I Hub is based on the Boost single producer, single consumer lock-free queue solution, which means that as long as only one thread is writing to the queue and only one thread is reading from that same queue, then no locking is necessary. This template class allows both a lock-free input queue and a separate lock-free output queue of any type, so that the actual worker can read some message and write out a different one to be broadcast.

bool push(const InQueueT &item)

The push() operation adds an item to the incoming queue. It returns true if the item was inserted, or false otherwise.

bool pop(OutQueueT &item)

The pop() operation removes an item from the outgoing queue and saves it to the *item* parameter. If no item is on the queue, then the function returns false and the contents of *item* are undefined.

bool push_out(const OutQueueT &item)

Typically used internally to add *item* to the outgoing queue. Keep in mind that only one thread should write to the outgoing queue. Returns true if the item was pushed to the queue and false otherwise.

uint64_t inQueueSize()

Returns the current size of the incoming queue.

uint64_t outQueueSize()

Returns the current size of the outgoing queue.

void doWork(InQueueT &item)

The work function done to process the next incoming *item*. This is a virtual function that must be over-ridden in the implementation class. This function is only called when there is something in the incoming queue.

void idle()

When nothing is in the incoming queue, the idle() operation is called. This gives the thread an opportunity to sleep before trying again. Optimal sleep timing is up to the implementation but should be driven by queue size history, so the thread will be able to wake up in time for another incoming item. Trying to run without an idle operation, however, will lead to high CPU utilization.

ThreadGroup.h

A helper class that maintains a number of lock-free thread implementations and assigns incoming queue items by a 2-byte identifier. The first item for the identifier is assigned based on a strategy of either round-robin (the default), randomized, or by shortest queue. Afterwards, if the identifier is already set to use a specific thread, then that incoming item is assigned to the same thread's queue. This is to facilitate cases where the incoming items must be processed sequentially under the assumption that data from previous incoming messages are still important to any new message handling. For example, to process BSMs from 20 sources, each source can be assigned one of the 5 worker threads in a group, thus not only ensuring each thread only processes 40 messages per second, but also that information from previous BSMs on that source, say path history, can be persisted in the thread local memory since incoming BSMs from that source all are assigned to the same thread.

void set_size(size_t size)

Sets the size of the thread group, initially zero. Once a new thread is added, it is automatically started. Because shrinking the group is not allowed, the size can really only increase and never decrease.

void assign(uint8_t group, uint8_t id, const InQueueT &item)

Assign an incoming item to a thread queue based on the 1-byte group and 1-byte id.

void unassign(uint8_t group, uint8_t id)

Remove a thread assignment for the 1-byte group and 1-byte id. The next assign() operation will pick a new thread.

TmxMessageManager.h

This helper class utilizes a thread group of lock-free threads to set up a network of worker threads to process incoming V2I Hub messages. The class inherits directly from PluginClient, thus implementing classes need to inherit from TmxMessageManager instead. The class is designed to work both as an incoming V2I Hub message decoder and handler using the mechanism described in Message Handling and as a rapid producer of V2I Hub messages. The incoming item may either be a ready-made V2I Hub message pointer, which will simply be passed along to the handler, or a raw set of bytes, which first must be converted into an appropriate V2I Hub message before being passed to the handler. It is important to note that using this class consolidates the processing of messages into a handful of worker threads, including any decoding.

void OnMessageReceived(IvpMessage *msg)

The PluginClient override that takes the incoming message and quickly assigns the message to a worker thread. This is accomplished by invoking one of the IncomingMessage() functions described below. No decoding of the message is attempted. By default, this assigns the message to the next available thread, and does **not** use grouping of sources. This function must be re-implemented in a subclass to provide such functionality.

void IncomingMessage(const uint8_t *bytes, size_t size, const char *encoding, uint8_t groupId, uint8_t uniqId)

Assign a byte array, assuming a specific encoding, to a worker thread based on the 1-byte group and 1-byte unique id. This can be used, for example, to receive bytes from a UDP stream based on the last 2 bytes of the incoming source address. The worker thread will then convert into a V2I Hub message and the handler will be invoked. In this case, however, the handler might just broadcast the message. There is also a similar form of this function for an incoming string message, such as in JSON.

void IncomingMessage(const IvpMessage *msg, uint8_t groupId, uint8_t uniqId)

Assign a V2I Hub message pointer to a worker thread based on the 1-byte group and 1-byte unique id. This is typically done upon receipt of a message from the V2I Hub core server and is used primarily to force the time-consuming handling of messages into separate threads.

void OutgoingMessage(const tmx::routeable_message &msg)

A helpful function that allows the worker threads to pass a message to the outgoing queue.

Figure 1 shows an illustration of the overall processing strategy for a TmxMessageManager implementation, with messages incoming from V2I Hub or raw bytes from other sources entering worker thread queues, which produce messages in their output queues to be sent back into the V2I Hub:

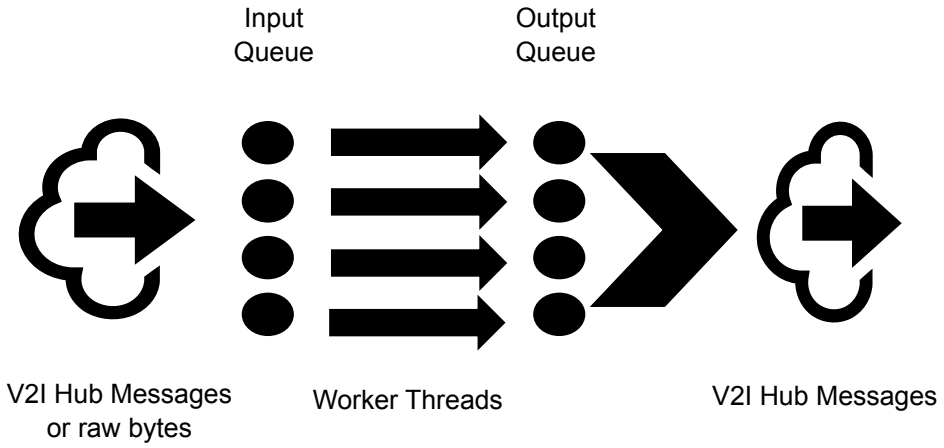


Figure 1. Processing Strategy Diagram

Plugin Testing

As with all complex software systems, one should progress through various stages of testing before deployment. After some level of unit testing is complete on a custom V2I Hub plugin C++ class, the next level of integration testing needed would be to receive actual messages from the V2I Hub core server in order to verify that the plugin can correctly process each type. At minimum, this will require a test environment with a V2I Hub core server process installed and active. Additionally, some ancillary plugins are likely needed for plugin configuration or to help ingest V2I Hub messages. Suggestions include the CommandPlugin in order to use the V2I Hub portal and the MessageReceiverPlugin to receive message bytes from outside the V2I Hub. Note that as long as the plugin manifest.json file has the coreIpAddr entry pointing to the active running core server, there is no need to “install” a plugin on the server. Each plugin will show up as started External to the V2I Hub core server, which simply means that no process control is used.

The MessageReceiverPlugin is a UDP server process that waits for incoming bytes to be written to its port. Those bytes are generally expected to be a J2735 encoded hexstring, that will automatically be converted to the appropriate J2735 V2I Hub message and routed through the server to any plugin that registered for messages of that type. The source of the bytes can be a real DSRC radio, some custom program, or the V2I Hub Vehicle Simulation Tool, which can send BSM and SRM type messages to the MessageReceiverPlugin. Additionally, the MessageReceiver can receive custom messages in JSON form that will be routed as-is. Therefore, if the format of the message is well-known, one can write the JSON string directly to the UDP port using some tool like netcat.

Refer to the V2I Hub Portal Guide for details on configuring the CommandPlugin for UI use. Refer to the V2I Hub Plugin Guide for more details on the MessageReceiverPlugin. Additionally, refer to the V2I Hub Vehicle Simulation Tool User Guide for more details on how to setup the simulation software to send BSM and SRM messages.

Appendix A. Acronyms

API	Application Programming Interface
ASC	Actuated Signal Controller
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
BSM	Basic Safety Message
CSW	Curve Speed Warning
CV	Connected Vehicle
DSRC	Dedicated Short-Range Communications
FCC	Federal Communications Commission
GPS	Global Positioning System
IVP	Integrated V2I Prototype
JSON	JavaScript Object Notation
OBU	On-Board Unit
OSADP	Open Source Application Development Portal
PSID	Personal Service Identifier
RSU	Roadside Unit
SAE	Society of Automotive Engineers
SPaT	Signal, Phase, and Timing
TIM	Traffic Incident Message
TSC	Traffic Signal Controller
U.S. DOT	United States Department of Transportation

UPER	Unaligned Packed Encoding Rules
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
XML	Extensible Markup Language

Appendix B. References

1. U.S. DOT. (2009, September 25). IEEE 1609 - Family of Standards for Wireless Access in Vehicular Environments (WAVE). In *United States Department of Transportation*. Retrieved from https://www.standards.its.dot.gov/factsheets/factsheet/80

2. National Marine Electronics Association. (2008, November). NMEA 0183 Standard. In *National Marine Electronics Association*. Retrieved from https://www.nmea.org/content/nmea_standards/nmea_0183_v_410.asp

3. U.S. DOT. (2009, September 18). NTCIP 1202 - Object Definitions for Actuated Traffic Signal Controller Units. In *United States Department of Transportation*. Retrieved from https://www.standards.its.dot.gov/Factsheets/Factsheet/22

4. U.S. DOT. (2009, September 18). NTCIP 1203 - Object Definitions for Dynamic Message Signs (DMS). In *United States Department of Transportation*. Retrieved from https://www.standards.its.dot.gov/Factsheets/Factsheet/23

5. U.S. DOT. (2016, October 31). DSRC Roadside Unit (RSU) Specifications Document v4.1. In *United States Department of Transportation*. Retrieved from http://www.fdot.gov/traffic/Doc_Library/PDF/USDOT%20RSU%20Specification%204%201_Final_R1.pdf

6. Radio Technical Commission for Maritime Services. (n.d.). RTCM 10402.3 RTCM Recommended Standards for Differential GNSS (Global Navigation Satellite Systems) Service, Version 2.3. In *Radio Technical Commission for Maritime Services*. Retrieved from http://www.rtcm.org/about.html

7. U.S. DOT. (2016, March). SAE J2735 - Dedicated Short-Range Communications (DSRC) Message Set Dictionary. In *United States Department of Transportation*. Retrieved from https://www.standards.its.dot.gov/Factsheets/Factsheet/71

8. U.S. DOT. (2016, March). SAE J2945/1 On-board Minimum Performance Requirements for V2V Safety Communications. In *United States Department of Transportation*. Retrieved from https://www.standards.its.dot.gov/Standard/531

Appendix C. V2I Hub Messages

This appendix details the V2I Hub internal message types, each of which are C++ classes. Whereas V2I Hub provides some basic constructs for the SAE J2735 messages types, the details of what is contained in those message types are only already defined in the specification and are not duplicated here. Additionally, whereas this appendix provides general information about the messages, specific details such as type and defaults are not always included. In order to document the full set of V2I Hub API and messages, a Doxygen configuration file can be created via:

```
$ doxygen tmx/tmx.doxyfile
```

The resulting HTML documentation is generated under the **doxygen/html** sub-directory. To obtain the detailed contents for each class, open that web page in a browser and navigate to the class, which should be under the **tmx::messages** namespace. Expanding the sub-menu in the left nagivation pane will reveal the message attributes for that message. Click each menu item to view the details (as illustrated in Figure 2 below).

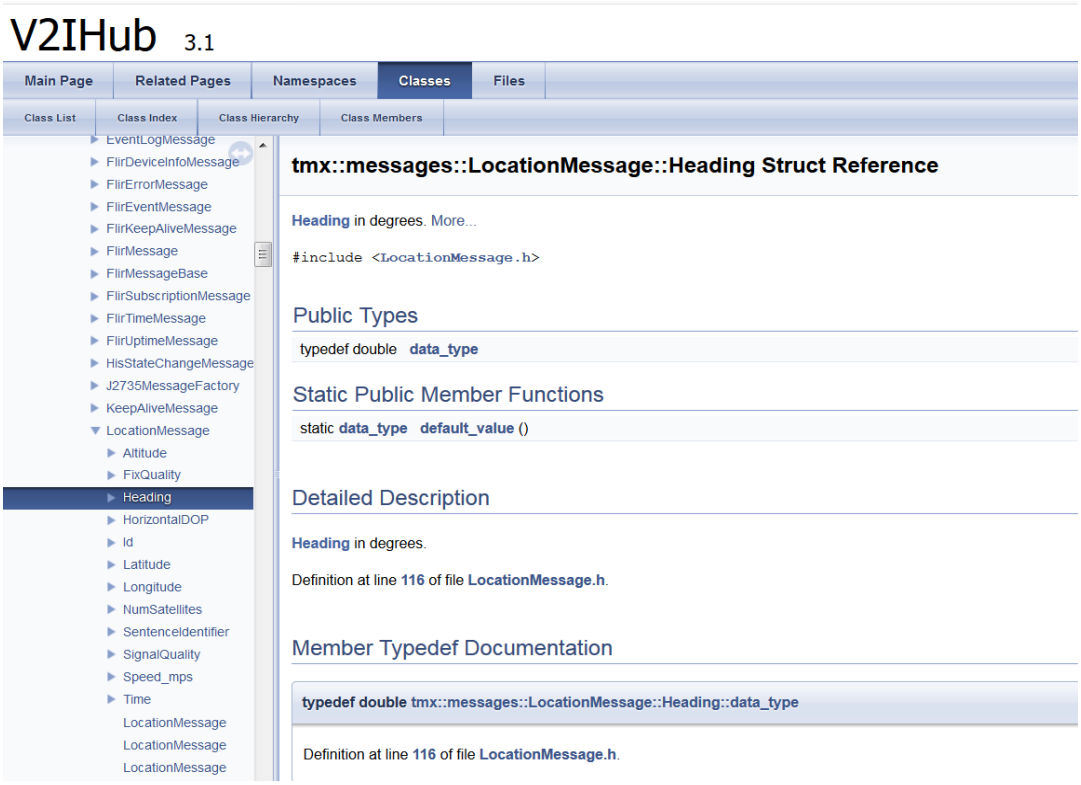


Figure 2. Doxygen-generated HTML Documentation of V2I Hub Messages

ApplicationDataMessage

An internal message containing specific application status information.

- Id – Unique GUID for the application message
- AppId – Unique ID for the application sending the message
- Timestamp – The timestamp of the event
- InteractionId – Null/not present if not currently interacting with things of interest
- IntersectionId – Intersection ID of the current map. Null/not present if not on a map
- DataCode – Code from master list of possible codes
- Data – JSON payload of data fields. Fields vary depending on DataCode

ApplicationMessage

An internal message containing application state transitions and alerts status.

- Id – Unique GUID for the application message
- AppId – Unique ID for the application sending the message
- EventID – Unique message identifier for repeated notifications of this event
- Timestamp – The timestamp of the event
- DisplayDuration – How long to display in milliseconds
- Severity – Info, InformAlert or WarnAlert
- EventCode – Code from the master list of possible events
- InteractionId – Null/not present if not currently interacting with things of interest
- CustomText – Open text field for specific messages related to this event
- DistanceToRefPoint – Distance to the reference point of the map
- AngleToRefPort – Angle to the reference point of the map

DataChangeMessage

The message used in a call back for a data monitor change. See Data Monitor.

- Name – The name of the monitor that has changed.
- OldValue – The value of the monitor that was replaced, as a string
- NewValue – The new value of the monitor, as a string

DecodedBsmMessage

An exploded detailed view of a J2735 BSM message.

- MsgCount – The message count in the range 0 to 127
- TemporaryId – The temporary ID of the sending device. It may change for anonymity
- Latitude – The latitude of the sending device.

- Longitude – The longitude of the sending device.
- Elevation – The geographic position above or below the reference ellipsoid (typically WGS-84)
- Speed_mps – The speed in meters per second
- Heading – The current heading in degrees
- SteeringWheelAngle – The current angle of the steering wheel
- SecondMark – Represents the millisecond within a minute, with a range of 0 – 60999
- IsOutgoing – True if this is an outgoing message being routed to the DSRC radio
- IsLocationValid – True if Latitude and Longitude contain valid values
- IsElevationValid – True if Elevation contains a valid value
- IsSpeedValid – True if speed_mps contains a valid value
- IsHeadingValid – True if Heading contains a valid value
- IsSteeringWheelAngleValid – True if SteeringWheelAngle contains a valid value

EventLogMessage

A message to be written in to the V2I Hub event log.

- Id – An identifier for the message
- Description – A description of the message
- Source – The source plugin of the message
- Timestamp – The time for the message
- LogLevel – The level at which the message is to be logged, i.e. ERROR, WARNING, INFO or DEBUG

HisStateChangeMessage

An internal message indicating a change should have been made on the human interface system (HIS) display.

- Id – Unique GUID for the state change message
- StateChangeTimestamp – Timestamp of the change, in local system time (should be UTC)
- HISType – Display or audible change
- TriggeringAlertType – The application identifier that has triggered the alert
- TriggeringAlertId – The alert identifier that has triggered the change
- HISPreState – What was displayed or played before this change
- HISPostState – What was displayed or played after this change
- Severity – Info, InformAlert or WarnAlert

KeepAliveMessage

Used for Plugin Keep Alive, this message is sent periodically so V2I Hub knows that the Plugin is still available.

- currentTimestamp – The current timestamp

- lastTimestamp – The timestamp of the previous keep alive message

LocationMessage

The current GPS positioning of the V2I Hub device.

- Id – Unique GUID for the location message
- SignalQuality – The quality type of the signal.
- SentenceIdentifier – The sentence identifier for the signal
- Time – UTC of position
- Latitude – Latitude of position
- Longitude – Longitude of position
- FixQuality – The quality of the fix, either 2D or 3D
- NumSatellites – The number of GPS satellites currently available
- HorizontalDOP – Horizontal dilution of precision
- Altitude – Antenna altitude above mean-sea-level
- Speed_mps – Ground speed, in meters per second
- Heading – Heading in degrees

TmxDmsControlMessage

Identifies the action code for the dynamic message sign to use.

- action – The action code to use

TmxJ2735Message

The J2735 message types were directly generated from the ASN.1 specification using the open-source asn1c compiler. This compiler creates C-style structures that follow the rules of the specification. Those structures are not easily wrapped in a V2I Hub message, and may require special handling, thus this templated C++ class is available to use in development. Even though the structure varies from type to type, the generalized class provides some ubiquitous operations that require no knowledge of the actual type, in particular memory management for the underlying structure to avoid memory leaks.

template <typename DataType> class TmxJ2735Message: public tmx::xml_message

The class template parameter *DataType* is expected to be the generated J2735 C-style structures, and the class extends a V2I Hub message type that is XML based. Therefore, these messages can be dumped out in XML form by serializing to an I/O stream, which simply calls to_string().

TmxJ2735Message(DataType *data = 0)

This constructor builds a J2735 message from the given pointer to an already created structure. This form assumes control over the management of the pointer, i.e. it will be freed when the object is destructed. If the pointer is null, which is the default case, the message is considered empty.

TmxJ2735Message(const TmxJ2735Message <DataType> & msg)

A basic copy constructor that directly uses the same underlying J2735 structure pointer, effectively increasing the reference count by one. Therefore, the pointer will only be freed when the last reference to it was destroyed. Therefore, the original object can go out of scope without destroying the underlying structure used in this copy. Because the pointers are shared, changes to data inside will affect all references.

```
TmxJ2735Message<SPAT> *original =
        new TmxJ2735Message<SPAT>(spat);  // Structure is a pointer to spat
…
PLOG(logINFO) << *original << endl;               // Display the contents of the orig
inal SPAT message
TmxJ2735Message<SPAT> copy(*original);    // The copy uses the exact same structure as
original
delete original;                          // The copy still has a reference to the str
ucture
PLOG(logINFO) << copy << endl;            // Should be the exact same SPAT data
// When copy goes out of scope, the spat pointer will be deleted
```

TmxJ2735Message(DataType &msg)

A constructor from an existing J2735 message reference. In this case, it is assumed that memory management is done outside this class, so destroying the object has no impact on the underlying structure.

TmxJ2735Message(const std::shared_ptr<message_type> &other)

A constructor from an existing J2735 message pointer that already has a shared reference count. The current ownership of memory management is maintained, but the reference count is increased by one. If the smart pointer was created outside the V2I Hub API, then cleanup must be done outside the V2I Hub objects, but this could also be a smart pointer obtained from a TmxJ2735Message class, in which case the source object cleanup will be executed upon destruction of all references. As with the copy constructor, the original object can go out of scope without destroying the pointer, but any changes to the structure affects all references.

std::shared_ptr<message_type> get_j2735_data()

Return a pointer to the underlying J2735 message structure. This uses a C++ smart pointer so that the contents can be shared safely without having to pass around copies.

void set_j2735_data(const message_type *data)

As mentioned, the TmxJ2735Message class is copied with shared pointers to the same structure. In fact, the asn1c compiled structures do not have the innate capability to be copied. Thus, deep-copy duplication is only possible via the internal XML container. This function is used to set the contents of the message by serializing the pointer structure to XML and rebuilding an empty structure from XML. The object loses any existing reference to a J2735 structure, so do **not** use this function in replacement of a copy constructor.

```
TmxJ2735Message<SPAT> copy;                // Empty structure
copy.set_j2735_data(spat);                          // New structure filled with conten
ts of spat
// The copy will only clean up its internal structure when it goes out of scope.
// The spat pointer must be deleted manually
```

TmxJ2735EncodedMessage

As mentioned, the J2735 structures are generated from the asn1c compiler. This also provides an API for encoding and decoding the structures, including the BER and UPER encodings for conversion to a byte array and XER encodings for conversion to XML. The XML encodings are used to serialize the TmxJ2735Message from and to a detailed string, but the BER and UPER encodings are used to serialize the message to a hexstring. The class manages the correct encoding and decoding of a TmxJ2735Message so it can be routed through V2I Hub.

template <typename MsgType> class TmxJ2735EncodedMessage: public TmxJ2735EncodedMessageBase

The template parameter *MsgType* is the TmxJ2735Message that is to be encoded and decoded. This class inherits from a simple base class that is just a JSON message type.

static constexpr const char *DefaultCodec = ASN1_CODEC<MsgType>::Encoding;

The default encoding for this V2I Hub message is by default BER for the 2015 specification and UPER for the 2016 specification.

void set_data(const std::vector<uint8_t> &data)

Set the payload contents with the encoded bytes. The encoding type is set to the appropriate encoded hexstring.

MsgType decode_j2735_message()

Return a copy of the TmxJ2735Message payload, decoded from the hexstring. For performance reason, the actual decoding of the bytes is done exactly once (unless the bytes are modified), and the message returned is built as a copy so that the underlying structure is shared amongst all the objects built using this function.

void encode_j2735_message(MsgType &message)

Set the contents of the payload by encoding the *message* to a byte stream.

xml_message get_payload()

Return the payload, which is effectively the same call as decode_j2735_message(), but it also pre-sets the XML container of the message.

int get_msgId()

Writes out the identifier for the J2735 message. These were specified in the 2015 specification but were taken out in the 2016 specification. However, the enumeration was carried into V2I Hub for continuity purposes.

void initialize(MsgType &payload)

A convenience method to initialize this message with an encoded byte stream of the specified payload.

tmx::routeable_message::get_payload<TmxJ2735Message<MsgType>>()

In order for Message Handling to be able to invoke the proper handler function by J2735 type, each TmxJ2735Message must have a template specialization for retrieving the payload as that type. This is typically done by creating a temporary TmxJ2735EncodedMessage type with the supplied encoded byte stream, then invoking the get_payload() on that encoded message type to retrieve the decoded message. The object that is returned can then be used as the input to the handler function.

TmxSignalControllerStatusMessage

Identifies the action to apply to a signal controller

- action – The action code to use. This value is the current active action plan used by the traffic signal controller.

U.S. Department of Transportation
ITS Joint Program Office – HOIT
1200 New Jersey Avenue, SE
Washington, DC 20590

Toll-Free “Help Line” 866-367-7487

www.its.dot.gov

FHWA-JPO-18-647

U.S. Department of Transportation