V2X ASN.1 C++

Encode/Decode API

User's Guide

Objective Systems, Inc. February 2022

Introduction

The *Objective Systems V2X ASN.1 C++ Encode/Decode API* consists of dynamically-linked libraries (DLL) for encoding and decoding messages as described in SAE J2735 and ETSI standards.

There are currently 4 shared libraries included in the package, supporting different versions of the various V2X standards:

- v2xasn1_j2735
  Supports SAE J2735 201603 - Dedicated Short Range Communications (DSRC) Message Set Dictionary
- v2xasn1_j2735_202007
  Supports SAE J2735 202007 - V2X Communications Message Set Dictionary
- v2xasn1_etsi1supports ETSI specifications as of 2016-11, including:
  - ETSI 302 637-2 v1.3.2 (CAM)
  - ETSI 302 637-3 v1.2.2 (DENM)
  - ETSI 102 894-2 v1.2.1 (ITS-Container)
  - ETSI 103 301 v1.1.1 (SPATEM, MAPEM, IVIM, SREM, and SSEM)
- v2xasn1_etsi supports ETSI specifications as of 2021-03, including:
  - ETSI 302 637-2 v1.4.1 (CAM)
  - ETSI 302 637-3 v1.3.1 (DENM)
  - ETSI 102 894-2 v1.3.1 (ITS-Container)
  - ETSI 103 301 v2.1.1 (SPATEM, MAPEM, IVIM, SREM, and SSEM)

Different versions of the same shared library cannot be used together, due to symbolic name conflicts.

This API has been developed in the C++ programming language. Only basic features of C++ have been used in order to make the API portable to a wide range of platform types. The Objective Systems ASN1C compiler was used to generate the structures and encode/decode functions. Encoders and decoders are included for the Unaligned Packed Encoding Rules (U-PER) binary format as well as JSON and XML Encoding Rules (XER) textual formats. This makes it possible to convert data to and from XML or JSON text to binary format.

Contents of the Package

The following diagram shows the directory tree structure that comprises the API:

```
v2x_api_<version>_<config>
  +- doc
  +- debug
     +- lib
     +- src
  +- release
     +- lib
     +- src
  +- rtsrc
```

```
+- rtjsonsrc
+- rtpersrc
+- rtxsrc
+- rtxersrc
+- rtxmlsrc
+- sample
   +- BasicSafetyMessage
   +- CAMMessage
   += DENMMessage
```

Where <version> would be replaced with a 5-digit version number and <config> by a configuration identifier. The first 3 digits of the version number are the ASN1C version used to generate the API and the last two are a sequential number.

The configuration (<config>) refers to the configuration of the package. Current configurations are limited binary (lb), unlimited binary (ub), and unlimited binary with source code (us).

For example, v2x_dll_v74300_ub would be the initial version generated with the ASN1C v7.4.3 compiler for the unlimited binary configuration (i.e. with license checking removed).

The purpose and contents of the various subdirectories are as follows:

- debug/lib – Contains the debug version of the V2X DLLs and supporting libraries.
- debug/src/dsrc – Contains header files for the debug version of the v2xasn1_j2735 library
- debug/src/j2735_202007 – Contains header files for the debug version of the v2xasn1_j2735_202007 library.
- debug/src/etsi1 – Contains the header files for the debug version of the v2xasn1_etsi1 library.
- debug/src/etsi – Contains the header files for the debug version of the v2xasn1_etsi library.

- release/lib – Contains the release version of the V2X DLLs and supporting libraries.
- release/src/dsrc – Contains header files for the release version of the v2xasn1_j2735 library
- release/src/j2735_202007 – Contains header files for the release version of the v2xasn1_j2735_202007 library.
- release/src/etsi1 – Contains the header files for the release version of the v2xasn1_etsi1 library.
- release/src/etsi – Contains the header files for the release version of the v2xasn1_etsi library.

- rt*src – Contains the header files for the common run-time libraries. In the case of a source kit these directories also contain the source files for the common run-time libraries.

- utilsrc – In source kits only this directory contains code used by the Python wrapper.

- doc – Contains this document.

- sample – Contains sample programs that illustrate how to use the API.

NOTE: There are duplicate symbols between the two J2735 libraries; they cannot be used together, and likewise for the two ETSI libraries. Also, because SAE substantially reorganized the J2735 ASN.1

specification, the generated headers are not the same between the two revisions. The generated types also contain some differences due to changes in header file dependencies (e.g. use of a pointer type where a non-pointer type had been previously used). We provide both libraries to give users time to upgrade their code to account for the header file changes.

Getting Started

The package is delivered as a zipped archive file (.zip) file (Windows) or a tar-gzipped (.tar.gz) file (Linux) that will allow installation to any directory on the target system. The sample programs use relative directory paths, so it is not necessary to create any type of top-level environment variables.

All of the necessary object files have been compiled and installed in the debug\lib and release\lib subdirectories. The code can be tested by executing the sample programs in the sample subdirectory. These sample programs consist of a reader and writer program. The writer program populates a data variable with some data, calls an encode function that writes the encoded byte stream to a file. The reader program reads date from a data file into memory, decodes the data into a C++ object, and then prints the decoded results.

Sample Programs

The following sample programs are included in this package:

- BasicSafetyMessage

Sample programs for Windows can be built by using the Visual C++ *nmake* utility program to execute the provided makefile. The procedure is as follows (note: this assumes package was installed in the c:\ root directory):

1. Open a shell terminal window (or Visual Studio terminal window).

2. Change directory to one of the sample directories above. For example:

   ```
   cd c:\<v2x_root_dir>\sample\BasicSafetyMessage
   ```

3. Execute make or nmake:

   ```
   nmake
   ```

The result should be *writer* and *reader* programs. The writer should be executed first. It will encode a set of test data and write the record out to a *message.dat* file. The reader program can then be executed to read the encoded file and decode the contents.

Note that in order to execute a program that uses the DLL, the operating system must be able to find the file. Two ways this can be done are as follows:

1. Modify the PATH environment variable to include the PATH where the DLL is located. For

example:

```
set PATH=%PATH%;c:\<v2x_root_dir>\debug\lib
```

alternately, the DLL file can be copied into a directory already located in the PATH.

2. Copy the DLL file into same directory in which the executable file is located.

On Linux, the sample program can be built by using the GNU *make* utility program to execute the provided makefile. The procedure is as follows:

1. Open a shell terminal window
2. Change directory to one of the sample directories above. For example:

   ```
   cd <v2x_root_dir>/sample/BasicSafetyMessage
   ```

3. Execute make:

   ```
   make
   ```

The writer and reader programs can be executed as in the Windows case above.

Note that in order to execute a program that uses the shared object on Linux or Mac OS X, the operating system must be able to find the shared object (.so) or dynamic library (.dylib) file. Two ways this can be done are as follows:

1. Create or modify the LD_LIBRARY_PATH (DYLD_LIBRARY_PATH on Mac OS X) environment variable to include the directory where the shared object file is located. For example:

   ```
   export LD_LIBRARY_PATH=$HOME/<v2x_root_dir>/debug/lib
   or

   export DYLD_LIBRARY_PATH=$HOME/<v2x_root_dir>/debug/lib
   ```

2. The shared object/dynamic library file may also be copied into a system directory that is already searched for these files (for example, the /usr/lib directory).

In the case of a limited binary library (which includes the evaluation edition), it may be necessary to assign a second environment variable to allow the license file to be located. The ACLICFILE environment variable should be set to the full pathname to the *osyslic.txt* file that was provided with the product. For example, if you place the license file in the root directory of the installation, the following variable would need to be defined:

```
export ACLICFILE=$HOME/<v2x_root_dir>/osyslic.txt
```

Rebuilding the V2X DLLs (source kits only)

With a source kit you can rebuild the V2X DLLs by following these steps.

For Windows:

1. cd to debug\build to rebuild the debug DLLs or release\build to rebuild the release (optimized) DLLs.

2. Execute `nmake clean` followed by `nmake`.

For non-Windows:

1. cd to debug/build to rebuild the debug shared libraries or release/build to rebuild the release (optimized) shared libraries.

2. Execute `make clean` followed by `make`.

General Procedure to Encode to a J2735 Message

The *MessageFrame* type is the top-level Protocol Data Unit (PDU) type defined for this specification. (In J2735 201603 it is defined in ASN1. module DSRC. In J2735 202007 it is defined in ASN.1 module MessageFrame.) Normally, this is the type one would use to encode a J2735 protocol message, although the API provides that capability to encode any type defined in the specification. For example, a user may want to encoded messages in separate parts and then put them together to form a full message at a later time.

We will assume in this case the user want to encode a complete MessageFrame PDU in one call. The procedure to do this is as follows:

1. Declare a variable of the *ASN1PEREncodeBuffer* class. This is the object that will contain the memory buffer into which the data will be encoded.
2. Declare a variable of the *ASN1T_MessageFrame* class and populate it with data to be encoded.
3. Declare a variable of the *ASN1C_MessageFrame* control class to associate the message buffer object with the data object to be encoded.
4. Invoke the *Encode* method within the control class object to encode the data.

If successful, an encoded J2735 message will have been written to the memory buffer within the encode buffer object. Methods can then be invoked in this object to work with the encoded data. The *getMsgPtr* and *getMsgLen* methods may be invoked to fetch the data. Methods also exist to print the data in a hex dump format or write it to a file.

A more detailed look at each of these steps is as follows. Note that you can refer to the *sample/BasicSafetyMessage/writer.cpp* file within the installation to see actual code associated with these procedures.

Declare a Variable of the ASN1PEREncodeBuffer Class

A variable of this class must first be declared. The basic constructor has a single argument *aligned* which indicates whether aligned or unaligned encoding is to be done. For J2735, this should be set to FALSE to indicate unaligned encoding. Other variants of the constructor allow a static data buffer to be specified or a stream object to writer the encoded data directly to an output stream.

The encode buffer object is used to keep track of all internal variables involved in the encoding process. In a multithreaded application, a separate encode buffer object should be declared per thread to ensure no contention between resources.

The code in the sample program that declares this variable is as follows:

```
/* Create an instance of the compiler generated class.
   This example uses a dynamic message buffer..*/
ASN1PEREncodeBuffer encodeBuffer (FALSE /* aligned */);
```

Populate a Data Variable of the ASN1T_MessageFrame Class

The *MessageFrame* production defined in the J2735 ASN.1 specification is the top-level PDU type for this protocol. In order to encode a message of this type, one must declare a variable of this type and populate it.

The structure contains many nested levels of sub-elements of other complex types. There are different strategies that can be used to deal with these nested types:

1. Declare objects of the sub-elements on the stack and then insert pointer to them in the composite structure. The disadvantage of doing this is the object is only valid within the scope of the current function. This is fine if nothing else is to be done with the object after encoding is complete, but if the user requires that the object persist outside this scope, the second method should be used.

2. Declare a single *MessageFrame* object and then allocate dynamic memory for the sub-elements. If this is done, it is advised to use the built-in *memAlloc* method within the control class to allow the class to manage all memory usage. The memory will then be automatically release when the object is destructed. Otherwise, if standard *new* is used, the user must keep track of all allocated objects.

In the sample program, method 1 was used. This is the full code to populate a test message:

```
ASN1T_BasicSafetyMessage bsm;
bsm.coreData = new_ASN1T_BSMcoreData(messageFramePDU);
bsm.coreData->msgCnt = 0;

const OSOCTET tempID[] = { 0x00, 0x00, 0x00, 0x02 };
memcpy (bsm.coreData->id.data, tempID, 4);

bsm.coreData->secMark = 14800;

bsm.coreData->lat = 0;
bsm.coreData->long_ = 0;
bsm.coreData->elev = 0;
```

```
bsm.coreData->accuracy.semiMajor = 255;
bsm.coreData->accuracy.semiMinor = 255;
bsm.coreData->accuracy.orientation = 65535;

bsm.coreData->transmission = TransmissionState::forwardGears;

bsm.coreData->speed = 716;
bsm.coreData->heading = 6774;
bsm.coreData->angle = -1;

bsm.coreData->accelSet.long_ = 0;
bsm.coreData->accelSet.lat = 0;
bsm.coreData->accelSet.vert = 0;
bsm.coreData->accelSet.yaw = 0;

ASN1C_BrakeAppliedStatus basC (bsm.coreData->brakes.wheelBrakes);
basC.clear();

bsm.coreData->brakes.traction = TractionControlStatus::unavailable;
bsm.coreData->brakes.abs_ = AntiLockBrakeStatus::unavailable;
bsm.coreData->brakes.scs = StabilityControlStatus::unavailable;
bsm.coreData->brakes.brakeBoost = BrakeBoostApplied::unavailable;
bsm.coreData->brakes.auxBrakes = AuxiliaryBrakeStatus::unavailable;

bsm.coreData->size.width = 230;
bsm.coreData->size.length = 600;

ASN1T_MessageFrame messageFrameData;
ASN1C_MessageFrame messageFramePDU (encodeBuffer, messageFrameData);

messageFrameData.messageId = ASN1V_basicSafetyMessage;
messageFrameData.value.t = MessageTypes::T_basicSafetyMessage;
messageFrameData.value.u._MessageTypes_basicSafetyMessage = &bsm;
```

Invoke the Generated Encode Function

In the code shown above, the following declaration links the MessageFrame data object with the encode buffer object:

```
ASN1C_MessageFrame messageFramePDU (encodeBuffer, messageFrameData);
```

The Encode method is then invoked in the following call:

```
if ((stat = messageFramePDU.Encode ()) == 0)
{
   if (trace) {
      printf ("Encoding was successful\n");
      printf ("Hex dump of encoded record:\n");
      encodeBuffer.hexDump ();
      printf ("Binary dump:\n");
      encodeBuffer.binDump ("Data");
   }
   msgptr = encodeBuffer.getMsgPtr ();
   len = encodeBuffer.getMsgLen ();
}
```

If the encode operation is successful, a status value of zero will be returned. The code shown in the good status block illustrates some of the things you can do with the encoded message. You can invoke the *hexDump* method to get a hex dump of the encoded bytes. You can also invoke the *binDump* method to dump a bit trace of all of the encoded fields in the message. This can be very useful for debugging interoperability issues.

The *getMsgPtr* and *getMsgLen* methods can be used to obtain a pointer to the encoded data and the length of the data.

If an error occurs, the *printErrorInfo* method may be invoked to print information on the error. The *getErrorInfo* method can also be used to return the error message as a string for use in GUI applications.

General Procedure to Decode from a J2735 Message

The general procedure to decode from a J2735 message is as follows:

1. Declare a variable of the *ASN1PERDecodeBuffer* class. This object that will contain the information on the binary message that is to be decoded.
2. Declare a variable of the *ASN1T_MessageFrame* class. This is the object into which the message will be decoded.
3. Declare a variable of the *ASN1C_MessageFrame* control class to associate the decoder buffer with the data object to received the decoded data.
4. Invoke the *Decode* method in the control class object.

If successful, the decoded J2735 message content will have been written to the data object declared in step 2.

A more detailed look at each of these steps is as follows. Note that you can refer to the *sample/BasicSafetyMessage/reader.cpp* file within the installation to see actual code associated with these procedures.

Declare a Variable of the ASN1PERDecodeBuffer Class

An object of the *ASN1PERDecodeBuffer* class must first be declared to describe the binary message that is to be decoded. The object can either be instantiated with information on the data to be decoded or without and the data provided later.

The form or the constructor that allows the data to be provided later takes as an argument only a flag that indicates if the message will be in aligned or unaligned PER format. Since J2735 messages are unaligned, this argument should always be set to FALSE. This is code to declare this object in the sample program:

```
ASN1PERDecodeBuffer decodeBuffer (FALSE /* aligned */);
```

In this case, the *readBinaryFile* method will be used to read the message into the buffer.

Declare a Variable of the ASN1T_MessageFrame Class

An object of the *ASN1T_MessageFrame* class must then be declared to receive the results of the decode operation. It is assumed that the message type to be decoded is the MessageFrame message since this is the PDU type defined in the specification, but it is possible to decode a message of any type. However, when working with PER messages, the type of message being decoded must be known in advance, there is no way to infer it from the data alone. Most PER-based message protocols have one or a few top-level types like this which list all of the possible variants in a single type.

Declare a Variable of the ASN1C_MessageFrame Class

An object of the *ASN1C_MessageFrame* class is declared next to tie the decode buffer to the object that is to receive the decoded data. The ASN1C compiler generates two types of classes for each production in an ASN.1 specification:

1. A data class (this has prefix *ASN1T_*) - This is designed to hold data for encoding and decoding, and

2. A control class (this has prefix *ASN1C_*) - This works in conjunction with the data class to allow operations to be performed on the data including encoding and decoding.

The reason for having separate classes is to avoid duplication of control structures in the data classes.

This is the declaration in the sample program:

```
ASN1C_MessageFrame MessageFramePDU (decodeBuffer, data);
```

Invoke the Decode Method

The decode method in the control class would then be invoked to decode the data:

```
stat = MessageFramePDU.Decode ();

if (stat != 0) {
   printf ("decode of PersonnelRecord failed\n");
   decodeBuffer.PrintErrorInfo ();
   return stat;
}
```

If successful, the data will now be available in the data variable where it can be manipulated.

General Procedure to Encode a J2735 Message in XER or JSON

The sample BasicSafefyMessage reader program demonstrates how to encode J2735 message data to XML (XER) or JSON format. For XER the following code is added:

```
// Output decoded data to XER (XML) format

OSRTFileOutputStream xerfos ("message.xml");
ASN1XEREncodeStream xerEncStrm (xerfos);
```

```
stat = MessageFramePDU.EncodeTo (xerEncStrm);
if (stat != 0) {
   printf ("encode MessageFrame to XER failed\n");
   xerEncStrm.printErrorInfo ();
   return stat;
}
```

This creates a file output stream object to write to a file named *message.xml*. An XER encode stream object is then declared using this. The main PDU *EncodeTo* method can then be invoked to write the data contents out in XER format.

Similar code can be used to encode to JSON format. The only difference is that an *OSJSONEncodeStream* stream object would be declared instead of *ASN1XEREncodeStream*.

General Procedure to Decode a J2735 Message in XER or JSON Format

The sample BasicSafefyMessage writer program provides options to input the data to be encoded in XML (XER) or JSON format. This demonstrates how to decode J2735 message data from XML (XER) or JSON format. For XER the following code is added:

```
// XER (XML) file
OSRTFileInputStream xmlfis (infilename);
OSXMLDecodeBuffer xerDecStrm (xmlfis);

stat = xerDecStrm.getStatus();
if (0 == stat) {
   stat = messageFramePDU.DecodeFrom (xerDecStrm);
}
if (0 != stat) {
   printf ("XER decode of MessageFrame failed.\n");
   xerDecStrm.printErrorInfo();
   return stat;
}
```

A file input stream object is first declared with a path to the file containing the XML message to be decoded. An *OSXMLDecodeBuffer* object is then declared that takes this object as an argument. Note the *ASN1XERDecodeStream* class should not be used in this case as this is only maintained for backward compatibility for decoding XER using the older, SAX-based method. A call is then made to the main PDU object *DecodeFrom* method to decode the data to the generated structure where it can subsequently be encoded in binary format.

Decoding from JSON is similar. In this case, an *OSJSONDecodeBuffer* object is used in place of the *OSXMLDecodeBuffer* object. The other logic is the same.

Further Information

This has been a brief overview of the steps required to encode and decode a specific message type from the J2735 ASN.1 specification. Further information on some of the topics referenced in this document are available on the web at the following URL's:

General ASN1C Support Home Page – https://www.obj-sys.com/support/asn1c.php

ASN.1 to C/C++ Type Mappings - https://www.obj-sys.com/docs/acv75/CCppHTML/ch04.html/CCppHTML/ch04.html

Generated C/C++ Source Code - https://www.obj-sys.com/docs/acv75/CCppHTML/ch06.html

ASN.1 PER Encode and Decode Buffer Class Reference - https://www.obj-sys.com/docs/acv75/CCppRunTime/per/annotated.php

ASN1CType Base Class Reference (class from which control classes are derived) - https://www.obj-sys.com/docs/acv75/CCppRunTime/com/classASN1CType.php