**STOL ASN1 Project**

1. **Investigation of feasibility and Identification of Best Approach**

Objective: To analyze and investigate the possibility of automatically converting ASN.1 document to ROS structures

The following areas were investigated.

* 1. Understanding the way ASN1C converts ASN.1 programming structures to C files
  2. Understanding the Structures of ROS
  3. Generation of ROS structures
  4. Auto-generate C code for conversion of data from ROS to ASN1 format and ASN1 to ROS format
  5. Auto-generate C code for conversion from ROS to JSON and JSON to ROS
  6. Auto-generate C code for conversion from SAE units to SI units and SI units to SAE units

1.1 **ASN1C**

While analyzing program ASN1C, the locations where the ASN1 program is parsed, compiled into C structures and printed out to .c and .h files have been identified. It was found that ASN1C uses long and unsigned long types for all INTEGER range. This will force any C to ROS parser to generate variables that are no smaller than 4 bytes. Even if the variable takes values 0, and 2, where a byte is enough to represent the whole range of the variable, this will consume 3 extra unused bytes per such variable.

1.2 **ROS**

ROS has structure types that are similar to those in Java - Each file itself acts like an implicit structure type that can be referred to in another ROS file. In the current manual conversion of C structures to ROS, instead of using int32 (long equivalent) and uint32 (unsigned long equivalent) for all integer variables, wherever it is possible to use shorter variables, int16, uint16, int8 and uint8 are being used, and it is a desirable effort, too.

* + 1. **Generation of ROS structures**

There are three ways to generate the necessary ROS structures

* + - 1. C to ROS conversion

This is one option looked into during this investigation as C syntax is thoroughly understood. As a demo, a parser for C-to-C was created. This parser can be directly converted to C-to-ROS parser with some changes. This C-to-ROS parser will take the output C files given by ASN1c program as input. The disadvantage of this approach is the parser will have to remove unnecessary structures, macro definitions and variables that may not have any meaning in a ROS program.

As a proof of concept of this method, initially a program that parses C to C conversion has been generated. While the parser is primitive, it is a work in progress. As the syntax of ROS is being investigated and understood thoroughly, this parser can be readily converted to a C-to-ROS parser.

* + - 1. Generate ROS structure from ASN.1

The syntax of ASN.1 can be directly converted to ROS by creating a parser like asn1c that converts ASN.1 structures to ROS structures.

* + - 1. Modify ASN1C to generate ROS structures together with C structures

Of the 3 options, the best option would be the third option as the first option involves only one tool and so one pass. But from the implementation point of view, it is perhaps the most complicated one. It does involve a lot less work, albeit it requires a very good skillset in programming.

**1.3 Findings**

As mentioned above, one drawback of the C files generated by asn1c is that all integer variables are defined as long or unsigned long. While ASN.1 clearly allows programmer to specify the range a variable can take, ASN1C is clearly ignoring this range while declaring variables. This needs to be rectified to produce less overhead in generated messages that ROS will need to deal with. This will be an added bonus by the end of this project. This step is necessary for either of the two options enumerated above.

As a part of this investigation, the locations where the current asn1c parses and creates core structures of ASN.1 document have been identified. Also, the locations where the C structures are generated have been identified. This knowledge allows a programmer to add hooks to the current asn1c program to generate ROS structures along with the generation of C structures.

**1.3.1 Where to add code to generate ROS structures**

Specifically, asn1c currently generates .c and .h files in asn1c\_save.c file in function **asn1c\_save\_streams**() at line 1098. This is the function where the generated “chunks” are saved into .c and .h files. This is the place where code to generate the ROS .msg files can be added. This is convenient here because .h files are created here. Also, this is the convenient place to generate code for converting

* ROS data to ASN1 data and ASN1 data to ROS data
* ROS data to JSON data and JSON data to ROS data
* SAE units to SI units and SI units to SAE units.

1.3.2 **Where to change asn1c to generate uint16, int16, uint8 and int8 types**

**asn1c\_C.c** aroundline 1380 generates code for typedef for ROS data structures. By default, this line generates “typedef long” for the target variable type. This code needs to be changed so “typedef int16”, “typedefuint16” etc can be generated. Because this code generates the typedefs for .h file, each use of the typed field needs to tracked and the corresponding code adjusted in the affected “.c “ code so “int16”, “uint16”, “int8” and “uint8”.

1. **Strategy for generating code**
   1. **.msg files**

If .h file is being generated, then corresponding .msg file need to be generated. If the .h file contains “typedef uint16 MsgCnt\_t; “ and is named as MsgCnt.h, the corresponding line will be generated as “uint16 msg\_cnt” in MsgCnt.msg. If in another file or location MsgCnt\_t is used as

MsgCnt\_t Count;

in a “.h” file, the ROS equivalent will be

j2735\_v2x\_msgs/MsgCnt count

This approach allows the compiler to generate .msg files with one to one correspondence between .h and .msg files.

* 1. **Auto generation of code for conversion from ROS data to ASN1 data**

The auto generation of code for data conversion from ROS to ASN1 and ASN1 to ROS requires to generate .c and .h files. The .h file will contain the declaration of the functions that do the actual conversion and .c will contain the full definition of the functions doing the conversion.

The programmer/developer will have write code that generate automatically C functions for conversion from ROS data to ASN1 data and ASN1 data to ROS data. For this the developer will have to look at struct streams and expr and generate structures base upon them. The developer should generate code to allocate dynamically memory for each optional variable in the ROS structures that are present there and populate the allocated memory with the corresponding data. The generated structures will have to have fields that specify whether the expression is 1. a function type, 2. FOR loop, 3. WHILE loop, 4. DO loop, 5. IF statements, 6. block statements or 7. plain statements. So, the developer will have to write code to generate code for 1. Functions, 2. FOR loops, 3. WHILE loops, 4. DO loops, 5. IF statements, 6. Block statements and 7. Plain statements. FOR loops can be used to populated structures or elements that are arrays.

Note that the asn1c software also has to generate .h files corresponding to ROS .msg files. Then if the ROS data needs to be converted to ASN1 format, then the C structures generated by ASN1C tool need to be filled using the ROS structures passed to the function in consideration.

Then the data thus converted to C structures from ASN1C can be converted to ASN1 structures using library function upper\_encode().

* 1. **Auto generation of code for conversion from ASN1 data to ROS data**

If the ASN1 data needs to be converted to ROS data, then the first thing is to call function upper\_decode(). So the generated code should call upper\_encode(). Then the C data returned need to be mapped to ROS data. Depending upon the type of data generated, if the type is an array of data, a corresponding array in ROS format needs to be generated using a FOR loop. For each optional element of the structure, the variable will be a pointer on ASN1 side. There should be a bit mapped macro and a variable presence or so on the ROS side that needs to be set appropriately to indicate the corresponding element. The developer needs to take care of the optional data like this.

* 1. **Auto generation of code for conversion from ROS data to JSON**

During the process in 2.2, code can be auto generated for conversion from the ROS data directly to JSON data. This becomes a matter of mapping the ROS format to JSON format. So, code need to be added to the ASN1C files asn1c\_save.c to generate code for generation of .h files for generating JSON structures. These .h files can be used to generate C functions that generate JSON structures on the fly.

* 1. **Auto generation of code for conversion from JSON data to ROS data**

Using the same structures and .h files as in 2.4, functions can be auto generated to map the JSON data as an input to the functions and generate C and ROS data. JSON files contain data that may be optional to ROS, so code should be generated to populate the corresponding flags to indicate the presence of optional data. While generating C data, all optional fields are put into a dynamically allocated memory and the pointer to this memory is assigned to a member of the C structure corresponding to the JSON structure.

* 1. **Auto generation of code for conversion from SAE units in the ROS data to SI data**

Code can be generated per element of the ROS data structure being converted to ASN1 data. This is still not clear which element of the structure needs to be converted and which need not be.

* 1. **Auto generation of code for conversion from SI units in the ROS data to SAE data**

After the ASN1 data is decoded, the resulting data will be in C structures. Code can be generated convert only each of the specific elements of each of the C data structures from SI units to SAE units. This is still not clear which element of the structure needs to be converted and which need not be.

1. **Flowchart for Auto-generation of Code for Conversion between ROS Data and ASN.1 Data**

The flowchart below enumerates steps needed to do be done to generate code for conversion of ROS messages to C messages and encode them into ASN.1 messages. Similarly, code for reverse conversion can be generated.

**Diagram, engineering drawing, schematic

Description automatically generated**

The idea of the flowchart is to ultimately generate a function (code) so the ROS structure is equal to C structure. The C structure is then encoded to generate ASN.1 message. Each function will have a multiple of assignments, like

SensorDataSharingMessageEncode(ROS Data)

{

…

a.b.c.d = e.f.g.h;

…

encoded\_code = upper\_encode();

return encode\_code;

}

where “a” is an structure in ROS and “e” is a structure in C.

1. **First Flowchart - Creation Code for function**

It starts with collections of all message types that constitute the MessageFrame set. It can be obtained from the asn1c. Alternatively, a config file can be created that dictates the set of MessageFrame messages for which the code needs to auto generated for conversion from ROS to ASN.1 format and vice versa. The MessageFrame set can be first generated fully before allowing asn1c to generate any of the .c and .h files, if that is more convenient.

Each MessageFrame message will have its own structure in ROS and C languages. For each structure, there will be a multiple of sub structures and each sub structure may have a multiple sub structures in turn. For each MessageFrame structure (or message), a node is generated, as depicted. Then the node can be printed as a ROS to C function or C to ROS function. The middle flowchart depicts the creation of a node in more detail.

1. **Middle Flowchart - Creation Code For Node**

This flowchart depicts how a node is created for each of the components of the message. Each component may be in turn a structure or a primitive data type(like int). If the next component is a primitive data, code for the accessing of the component within its parent structure is straight-forward. If it is a structure, then the sub-routine depicted in the leftmost flowchart needs to be invoked.

1. **Third Flowchart - Creation of Sub-structure**

This subroutine is very similar to the middle flowchart. It again enumerates all the member of the sub-structure one by one and generates code for each of the member. If the member is a primitive type, the member can be accessed immediately using “.” notation + the member name. If the member is another sub-structure, then it needs to call this same subroutine.

The decode function will be created similarly:

SensorDataSharingMessageDecode(ASN1 Data)

{

decoded\_code = upper\_decode();

…

e.f.g.h = a.b.c.d;

…

Return e;

}