E-AGLE Trento
*Racing Team*

# Telemetry Code Generator

Eugenio Berretta
eugenio.berretta@studenti.unitn.it

Academic year 2019/2020

## Contents


1 Concept 4
- 1.1 Introduction to the problem . . . 4
- 1.2 Project purpose . . . 4

2 Decision 5
- 2.1 The telemetry data structure . . . 5
  - 2.1.1 What is it . . . 5
  - 2.1.2 The structure of the structue . . . 5
  - 2.1.3 An example . . . 5
- 2.2 The Problem . . . 7
  - 2.2.1 Introduction . . . 7
  - 2.2.2 Low-level languages vs Hight-level languages . . . 7
- 2.3 The solution . . . 7
  - 2.3.1 The json format . . . 7
  - 2.3.2 The templates . . . 8
  - 2.3.3 The Typescript language . . . 9
  - 2.3.4 A separate project . . . 9
  - 2.3.5 The documentation . . . 9

3 Implementation 10
- 3.1 General . . . 10
  - 3.1.1 Dependencies . . . 10
  - 3.1.2 Package.json scripts . . . 10
  - 3.1.3 main and bin . . . 11
- 3.2 The code . . . 11
  - 3.2.1 package.json . . . 11
  - 3.2.2 tsconfig.json . . . 11
  - 3.2.3 docs . . . 11
  - 3.2.4 source . . . 11
- 3.3 Transpiling . . . 12
- 3.4 More information about the code . . . 12

4 User usage 13
- 4.1 Local module . . . 13
- 4.2 Global module . . . 14
- 4.3 api . . . 15
  - 4.3.1 signature . . . 15
  - 4.3.2 parameters . . . 15
  - 4.3.3 options parameters . . . 15
- 4.4 special comments . . . 15

5 Future 16
5.1 Storic . . . 16
5.2 Future . . . 16
5.2.1 Config interpretation . . . 16
5.2.2 Message parsing . . . 16

6 Conclusion 17

# 1 Concept

The telemetry of eagletrt is a program written in C, hosted in a raspberry plugged directly to the canbus of the car. It listens to the can and gps messages, parses them in a quite complex structure and every a certain amount of milliseconds saves the structure in a local mongodb and forwards it via mqtt, before discarding it and reapeating the cycle.

## 1.1 Introduction to the problem

The telemetry needed to be written in a low-level language, because performance were essential for the purpose of the team. C is one of the best languages if you want to develope efficient programs, but adds a lot of complexity comparing with higher programming languages and usually entails an harder job. Nonetheless, writing code in C is very funny because it makes you understand what really happens in your machine when the program is executed, in a far wider extend of the other languages. The most annoying problem with C for this project was that it is statically typed. It has to be like that, because it is one of the reasons it is so performant, aber sometimes it means also writing a huge amount of repetitive code and repetitive is never funny.

## 1.2 Project purpose

The purpose of this project, that was initially inside the telemetry and then it was moved, is to reduce to a very significant extend the C code that has to be written when some changes happen in the telemetry, saving a huge amount of time.

# 2 Decision

## 2.1 The telemetry data structure

### 2.1.1 What is it

The telemetry reads the messages that arrive from the canbus and from the gps rover serial port. Every a certain amount of milliseconds it saves in a local mongodb the data and forwards it via mqtt. In order to do this, it needs to parse and accumulate all the messages that arrive in a variable, modeled in base of what we call the "telemetry data structure". In the C code it is a struct that groups the messages. But it reflects also the documents saved as BSON in mongodb and it is consequently inspired by a json object.

### 2.1.2 The structure of the structue

The telemetry data structure follows these rules:

- An instance of the structure, such as a variable of the struct data_t or a saved document in mongodb, represents a block of messages that were accumulated in a certain amount of milliseconds before being saved and discarded by the volatile memory.
- It has an integer "id" property. It is an incremental dentifier of the block of messages.
- It has a long integer "timestamp" property. It is the timestamp taken just before saving the messages of bloks.
- It has a string "session" property. It is the identifier of the session when the structure block was saved.
- Every single message is represented by a "message object", an object containing the properties "timestamp", that is the timestamp taken when the messages arrived in the telemetry, and "value", that is the value of the message and can be a primitive type as well as an object type.
- A part from the "id", the "timestamp" and the "session" properties, the telemetry data structure is composed by message objects that are always inside an array. This is because, usually, during the life of the block of messages, several messages of the same type will be received and accumulated. Hence, every array of messages contains all the messages of a certain type that arrived in a certain amount of milliseconds.
- The array of messages can be logically nested and grouped in other objects.

### 2.1.3 An example

This is an example of document saved in mongodb:

```
{
    "id": 23,
    "timestamp": 10483862400000,
    "sessionName": "2020_04_23__12_00_00__pilot_race",
    "throttle": [
        {
```

```
            "timestamp": 10483862400001,
            "value": 0
        },
        {
            "timestamp": 10483862400002,
            "value": 5
        },
        {
            "timestamp": 10483862400003,
            "value": 6
        }
    ],
    "brake": [
         {
            "timestamp": 10483862400001,
            "value": 0
        },
        {
            "timestamp": 10483862400004,
            "value": 100
        }
    ],
    "bms_hv": {
        "temperature": [
            {
                "timestamp": 10483862400000,
                "value": {
                    "max": 28,
                    "min": 22,
                    "average": 25
                }
            }
        ],
        "voltage": [
            {
                "timestamp": 10483862400000,
                "value": {
                    "max": 312,
                    "min": 200,
                    "total": 250
                }
            },
            {
                "timestamp": 10483862400007,
                "value": {
                    "max": 312,
                    "min": 200,
```

```
                    "total": 250
                }
            }
        ]
    }
}
```

## 2.2 The Problem

### 2.2.1 Introduction

The problem with the structure is that C code is a statically typed language. This means that it is not suited to handle json object like the structure. A structure like the one in the example above, but much longer in the reality, should be used in C code. The result is a structure definition of 350 lines of code, an instance allocation of 50 lines of code, a bson conversion of 400 lines of code and an instance deallocation of 30 lines of code. Taking in account that the structure changed frequently, because the new sensors or can messages were added, mantaining and keeping updated that code was worse than the hell. If only a message changed, we would have to change four different parts of the code and some of them were between 400 lines of code that seem always the same.

### 2.2.2 Low-level languages vs Hight-level languages

Here is the problem: in a statically type like C is very difficult to iterate over the keys of an object, expecially without loosing performance. Almost always you need to manually enumerate every single key. In contrast, dynamically programming languages such as javascript can iterate easily, with a few lines of code, over huge objects with plenty of keys. But of course we could not use Javascript for the telemetry.

## 2.3 The solution

Not only were these parts of codes very long, but also repetitive. This was annoying, but being repetitive came us in favour. What is repetitive is easy to standardize and hence automatize. We managed to write in a few days Javascript code that produced C code for us. All those critical parts that were used to make us loose plenty of time were now written by a Javascript program.

### 2.3.1 The json format

There is a json file, called "structure.json", that represents the structure. Every time the data structure changes, we only need to change that json file and Javascript does all the rest of the job for us.

The json format is almost identical to the data saved in mongodb. The differences are that:

- For all the primitive values, instead of the value a string with the primitive type is put in place
- For every array of messages, only a single object is added, it represents the types of the message, and as second element there is an integer representing the maximum number of messages of that type that can be put in a block of messages

This is an example of structure.json:

```
    {
    "id": "int",
    "timestamp": "long",
    "sessionName": "char*",
    "throttle": [
        {
            "timestamp": "long",
            "value": "double"
        }, 200
    ],
    "brake": [
        {
            "timestamp": "long",
            "value": "double"
        }, 200
    ],
    "bms_hv": {
        "temperature": [
            {
                "timestamp": "long",
                "value": {
                    "max": "double",
                    "min": "double",
                    "average": "double"
                }
            }, 500
        ],
        "voltage": [
            {
                "timestamp": "long",
                "value": {
                    "max": "double",
                    "min": "double",
                    "total": "double"
                }
            }, 500
        ]
    }
}
```

### 2.3.2 The templates

After coding the functions that wrote the C code, we needed a powerful way to inject it in the C code. Making an entire c file would have made the project less clear and understandable. Copying and pasting it manually would have been annoying. As a solution, we decided that every c or header file that would have contained auto-generated code, should have had a .template.c or .template.h extension. Inside, instead of the long and tedious parts of code, had a special comment specifying

which part of code should have been there, such as "// GENERATED_BSON_CODE". The javascript program would have then automatically found all those .template files and produced an identical copy, without the .template extension and with the generated code instead of the special comments.

### 2.3.3 The Typescript language

Initially written in Javascript, we decided to rewrite it in Typescript because the code was quite complicated and adding types to Javascript would have made everything more maintainible.

### 2.3.4 A separate project

The code generator was initially included in the telemetry. After observing that it was very rearly changed, we decided to separate it in this project and to make it as an npm command line module. This was done because the code generator was written in a different language from the telemetry, was enough articulated to be separated and was quite safe and there would have not be the need of emergency changes during tests.

### 2.3.5 The documentation

Document that code would have been a long task. We decided to use typedoc (https://typedoc.org/) to generate the documentation. It is a sort of javadoc, but for Typescript. It reads the doc comments in the code and generates an html documentation, that is now available online. This was also useful, because writing the doc comments is capted by the editors that provide hints.

# 3 Implementation

The code of the generator can be found in this github repo: https://github.com/eagletrt/eagletrt-code-generator

## 3.1 General

The project is written in Typescript and is an npm package, usable both as local and global module.

### 3.1.1 Dependencies

The dependencies are:

- chalk (https://www.npmjs.com/package/chalk) for a coloured output
- dree (https://www.npmjs.com/package/dree) to manage the directory trees
- yargs (https://www.npmjs.com/package/yargs) for the command line part

The dev-dependencies are:

- typescript (https://www.typescriptlang.org/) to transpile the Typescript code into Javascript
- eslint (https://eslint.org/) to lint the code, so that it is clear and follows a standard
- @typescript-eslint/eslint-plugin and @typescript-eslint/parser to use eslint with Typescript
- @types/node and @types/yargs to provide the types definitions for the dependencies
- commitizen (https://www.npmjs.com/package/commitizen) to have a standardized commit on git
- typedoc (https://typedoc.org/) to generate the documentation from comments in the code
- now (https://www.npmjs.com/package/now) to host costlessly the generated documentation site

### 3.1.2 Package.json scripts

The package.json has a few scripts to help develope the module:

- transpile: it transpiles the Typescript in Javascript
- lint: it lints the code
- lint:fix: it lints and fixes the code
- docs: it uses typedoc to generate the documentation site and now to upload it. It uses other "docs:" helper scripts.
- commit: it uses commitizen to commit on git

### 3.1.3 main and bin

In the package.json, the specified main file is the file that will be imported when the module will be imported by another project. The bin is the one that will be executed when the module will be used as a command line program.

## 3.2 The code

### 3.2.1 package.json

In the root folder there is the package.json, that is the manifest of the npm module. It declares some helper npm scripts and both the dependencies and the dev-dependencies.

### 3.2.2 tsconfig.json

The source/tsconfig.json file is the json file that configures the Typescript transiplation of the code

### 3.2.3 docs

The /docs folder contains the assets for the documentation, the generated-by-typedoc documentation site and a text file with the directory tree of the project generated with dree.

### 3.2.4 source

The /source folder contains all the Typescript code. It is divided in two folders /lib for the actual library and /bin, that imports the code inside /lib and yargs to make the module command-line usable.

**lib** The lib folder is structured like this:

- index.ts: It is the root file and the one exported by the npm module that will be published
- /types: It contains the main typescript types, interfaces and general classes
- /generators: It contains all the code reguarding the declarations of the c code generators and their associated special comment.
  - /bson is a folder containing the generator that creates the c code to transform the telemety data structure to bson
  - /structure is a folder containing the generators that reguards the allocation, deallocation and declaration of the c struct. It contains also a structureGenerator.ts file that provides a more-defined generator class that all the structure generators extend
  - index.ts is the flie imported by the root index.ts. It uses dree to iterate through all its siblings folders and files. Only the files ending with .generator.js are considered. All these imported classes are then exported in an array. The extension is js because this task is executed at runtime, when the generators in the siblings folders are already transpiled in js.
- /utils: It contains various ts files that will be used by the index.ts file.

- options.ts exports a function that given the user-provided options object returns it with the default values put where none were provided
- logger.ts exports a class whose purpose is providing a beautiful log
- getCodes.ts exports a function that, given the path of the structure.json file and the array of generators provided by /generators/index.ts, makes the generators creating the code and returns them.
- transpile.ts exports a function that, using dree, scans the given src folder to search the template files, replaces the special comments with the right generated code and creates their analog file without the .template extension.
- parseTemplate.ts exports helper functions for transpile.ts

**bin** The bin folder contains only an index.ts. It imports the lib from /lib/index.ts and the external module yargs, in order to make the module usable also as a command line program.

## 3.3 Transpiling

The Typescript is transpiled in Javascript so that nodejs can understand it. The tsconfig.json contains the configurations of the transpilation and the /source folder is transpiled in a /dist folder that also contains /bin and /lib folders.

## 3.4 More information about the code

To have further information about the code, it is possible to follow two documentation sites made with typedoc:

- user: https://eagletrt-code-generator.euberdeveloper.now.sh/. It contains only the information about the exported functions of the library.
- dev: https://eagletrt-code-generator-dev.euberdeveloper.now.sh/. It contains information about all the files and function of the library.

# 4 User usage

## 4.1 Local module

To use this section as a local module:

1. Install the module locally npm install –save eagletrt-code-generator
2. Given a directory structure such as

```
code/
    structure/
        structure.template.h
        structure.c
    utils/
        utils.h
        utils.template.c
    main.template.c
```

And a script such as

```
const generator = require('eagletrt-code-generator');

const src = './code';
const structure = './code/structure.json';
const options = {
    extensions: ['c', 'h', 'cpp', 'hpp'],
    log: true
};

generator.generate(src, structure, options);
```

The result will be:

```
code/
    structure/
        structure.template.h
        structure.h
        structure.c
    utils/
        utils.h
        utils.template.c
        utils.c
    main.template.c
    main.c
```

Where all the special commments in the .template.c or .template.h files will be replaced with the code generated in base of the structure.json file and put in files with the same name without the .template extension.

## 4.2 Global module

To use this section as a global module:

1. Install the module globally npm install -g eagletrt-code-generator
2. Given a directory structure such as

```
code/
    structure/
        structure.template.h
        structure.c
    utils/
        utils.h
        utils.template.c
    main.template.c
```

And a the command:

```
eagle generate --src code --structure .code/structure.json --extensions c
```

The result will be:

```
code/
    structure/
        structure.template.h
        structure.h
        structure.c
    utils/
        utils.h
        utils.template.c
        utils.c
    main.template.c
    main.c
```

Where all the special commments in the .template.c or .template.h files will be replaced with the code generated in base of the structure.json file and put in files with the same name without the .template extension.

3. To show the help execute:

```
eagle generate --help
```

## 4.3 api

The module exports only the function "generate".

### 4.3.1 signature

```
generate(src, structure, options)
```

### 4.3.2 parameters

- src: Optional. The folder where the template files will be fetched from. The default is the current folder.
- structure: Optional. The path to the json file containing the structure, used by generators to dynamically generate code. The default is structure.json.
- options: Optional. The options object specifying things such as logging, indentation and filters on the files

### 4.3.3 options parameters

- exclude: Default value: /node_modules/. A RegExp or an array of RegExp whose matching paths will be ignored.
- extensions: Default value: undefined. An array of strings representing the extensions that will be considered. By default all extensions will be considered.
- log: Default value: true. If the log will be shown on the terminal.
- indent: Default value: true. If the generated code will be indented the same as the comment it will substitute.

## 4.4 special comments

The special comments usable in the C code are:

- GENERATE_BSON: Generates the code of the function that given the structure variable, creates the bson object
- GENERATE_STRUCTURE_TYPE: Generates the c struct representing the structure
- GENERATE_STRUCTURE_ALLOCATOR: Generates the code of the function that allocates the structure
- GENERATE_STRUCTURE_DEALLOCATOR: Generates the code of the function that deallocates the structure

For examples and more inforamtion about the special comments, read the README of the repo (https://github.com/eagletrt/eagletrt-code-generator/blob/master/README.md)

# 5 Future

## 5.1 Storic

The module was initially a javascript program that ran on nodejs, was located in the same repo of the telemetry and was written in three days. After that it was debugged by using it and improved, until it was eventually moved in an own repository. The last modifies reguarded the traduction traduction to Typescript and the complete reorganization of the code. This is because the telemetry C code needed this module and without it would take lots of time to write manually the code that depends on the data structure. Having only I worked on this project, I had to organize the code and make it clearer so that any other developer could subsequently maintain it quite easily. A .travis.yml was also added so that the linting and documentation site update was done automatically by Travis.ci (https://travis-ci.org).

## 5.2 Future

The module is now stable, but it will be probably extended to automatize other parts of the c code.

### 5.2.1 Config interpretation

The telemetry uses a config.json file to general configuration, such as mqtt and mongodb uri and other stuff. The json config file is quite articulated is interpreted with JSMN (https://zserge.com/jsmn/) which is very cool but requires lots of code. In the future, this task could be automatized with this code generator.

### 5.2.2 Message parsing

Another task, more important, that can and probably will be authomatized in the future is the can message parsing. Currently over 85% of the time used to maintain and improve the telemetry is occupied by the part that parses the can messages and stores the values in the data structure. This is because that part of code is, again, repetitive, long and changes frequently. With the new car fenice the messages ids will be generated automatically and given another json file (messages.json) together with the current structure.json file properly modified, we think that it will be possible to authomatize also the task of parsing the can messages. This was actually (only conceptually because fenice was at the time still to be made) already done and the example can be find following this branch of this repo https://github.com/euberdeveloper/telemetria-c/tree/newTask.

# 6 Conclusion

The telemetry code generator was an example of the power of automatization. We had to write tons of repetitive lines of C code and we could not avoid it because we needed performances. By generating automatilcally the critical parts of the C code we can now do what before occupied the 80% of our time in a matter of seconds, modifying only a single json file.