Brno University of Technology

Faculty of Information Technology

Wireless and Mobile Networks 2024/2025

Project documentation

Pavel Kratochvíl
xkrato61

Radio Data System (RDS) Encoder and Decoder

23.11.2024

1 Introduction

The aim of this project was to implement a radio data system (RDS) encoder and decoder. Their capabilities cover processing Group 0A and 2A message types. The project could be divided into several parts, the implementation of encoder, decoder, testing and other files.

The entire assignment implementation follows the European Standard EN50067 – Specification of the radio data system (RDS) for VHF/FM sound broadcasting in the frequency range from 87.5 to 108.0 MHz.

1.1 Encoder

The encoder encodes provided input (program identification, program type, traffic information etc.) into a specific group message structure.

1.2 Decoder

The decoder takes in the RDS encoded message as an argument and decodes it back into the original fields. During decoding, Cyclic Redundancy Codes (CRC) are used to ensure validity of each 26bit chunk of the message. Additionally, the order of 104 bit groups as well as the order of 26 bit blocks inside a group can be arbitrarily changed and the decoder can still properly decode the message.

2 Implementation

As per the assignment, the final implementation is written in C++, particularly the C++14 version of the ISO/IEC 14882 standard.

The final implementation consists of files: `rds_encoder.cpp`, `rds_decoder.cpp`, `common.cpp` and their respective header files.

For convenience and easier extensibility of both encoder and decoder, a `CommonGroup` class stores functions data which is common for both 0A and 2A group. The classes `Group0A` and `Group2A` inherit from this class and provide more specialized parsing functions, sorting functions and information printing functions.

2.1 Encoder

The encoder program starts with argument parsing and checking the validity of each argument. All arguments must be present. For numerical arguments, the range is checked. For boolean arguments, the values are explicitly checked to be either '0' or '1'. Lastly, for string arguments, the strings are verified to contain only alphanumeric characters or spaces and their length is checked as well. If any of these criteria are not met by the input, the user is informed of the faulty supplied arguments by a helpful message and code `1` is returned.

2.2 Decoder

The decoder program starts with argument parsing and checking of the only `-t` input flag its value, which is a binary string and may only contain '0' and '1' characters. The encoded message length must be divisible by 104 without remainder. The binary string is then sliced into 26 bit chunks for further processing.

After the initial checks, the blocks are sorted in each group in the input by `sort_blocks` function based on block offset values. The CRC takes place here as this is the fist step in the processing and each 26 bit chunk must be verified. The sorting is necessary as the order of four blocks in each group can be arbitrary. After sorting, the group is identified from the first received group's 2nd block.

The order of groups in the input message is corrected by `sort_0A_data` or `sort_2A_data`, which reorder the groups and identify empty (missing) ones based on the decoder segment numbers in each group. After the sorting, the decoding is done using masking and shifting of individual bits in each block. If any inconsistencies are detected across blocks, for example a inconsistent PI value, the program exits with code 1.

Additionally, if any groups are missing in the input, the Program Service value will be incomplete and the missing characters and filled in by underscores (e.g., "PS: __dioXYZ" when the first group containing the first 2 characters is missing).

2.3 Common Files

`common.cpp` and its header file `common.hpp` provide shared constants and functionality such as printing 26 bit long blocks and CRC calculation.

2.3.1 CRC

The CRC calculation has done in both encoder and decoder. While in the former it is the last stage before the outputing binary string, in the latter it has to be one of the first stages of processing to ensure data validity before any further decoding. Only manual bitwise operations are used on template class bitset to calculate the CRC.

We are calculating the CRC value for the upper 16 bit part of the 26 bit block (`val` variable), therefore `and_check` variable is initialized to `1«25`. The result of a bitwise AND of the `val` and the `and_check` variable determines, whether we should XOR the `val` with the standard CRC-10 polynomial used in RDS $x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + 1$. If the AND operation result was `1`, we XOR the `val` with the CRC polynomial bitset `0b10110111001` and shift the polynomial one bit to right. Otherwise, if the result of the and operation was `0`, we just shift the polynomial to the right. In total, we repeat this operation 16 times, so it can be efficiently done in a for loop and the result is stored in the `val` variable as the last 10 bit. This resulting value is then OR-ed with the original one and returned.

3 Testing

For validating outputs both encoder and decoder a testing script `tester.py` was implemented. It contains over 50 test cases in total. This includes, missing flags, missing flags' values, valid and invalid values for each flag, such as underflows, overflows and invalid length of string values for `rt` and `ps` flags. Furthermore, decoder test also verifies the functionality when the order of groups is changes, the order of blocks inside a group is altered and lastly, several tests verify that code `2` is returned when decoder receives corrupted input binary string value. The output of the script is an enumeration of the tests and their results.

```
~> python3 tester.py
------ ENCODER 0A ------
Decoder test # 0  -  basic valid 0A - PASS
Decoder test # 1  -  valid pty - PASS
Decoder test # 2  -  flag --help should print and return 0 - PASS
Decoder test # 3  -  invalid group - PASS
Decoder test # 4  -  pi underflow - PASS
------ ENCODER 2A ------
Decoder test # 0  -  basic valid 2A - PASS
Decoder test # 1  -  basic valid 2A long rt - PASS
Decoder test # 2  -  basic valid 2A empty rt - PASS
Decoder test # 3  -  invalid ab - PASS
Decoder test # 4  -  invalid ab - PASS
------ DECODER 0A ------
Decoder test # 0  -  flag --help should print and return 0 - PASS
Decoder test # 1  -  basic valid 0A - PASS
Decoder test # 2  -  0A swap groups - PASS
Decoder test # 3  -  0A swap groups - PASS
Decoder test # 4  -  0A swap groups - PASS
Decoder test # 5  -  0A CRC corrupt - PASS
------ DECODER 2A ------
Decoder test # 0  -  basic valid 2A - PASS
Decoder test # 1  -  2A missing value - PASS
Decoder test # 2  -  2A missing value - PASS
Decoder test # 3  -  2A swap groups - PASS
Decoder test # 4  -  2A CRC corrupt - PASS
Decoder test # 5  -  2A CRC corrupt - PASS
Decoder test # 6  -  2A CRC corrupt - PASS
Decoder test # 7  -  2A swap missing more groups - PASS
```

Figure 1: A shortened output of `tester.py` script with all test cases passing.

```
~> python3 tester.py
------ ENCODER 0A ------
Decoder test # 0  -  basic valid 0A - PASS
Decoder test # 1  -  valid pty - FAIL
Unexpected stdout.
command:  ./rds_encoder -g 0A -pi 4660 -pty 30 -tp 1 -ms 0 -ta 1 -af 104.5,98.0 -ps "RadioXYZ"
Expected:
00010010001101000...
Actual:
00010010001101001...
```

Figure 2: An example output of a failing test case. The script provides the command used for the test case, and the expected and actual `stdout` of the program. The expected and actual outputs were shortened to fit.