Master's Thesis

# Conflating Best-Source-Selector for telemetry data streams with inter-stream error correction of sounding rockets


| | |
|---|---|
| Author: | Moritz Aicher, München |
| Supervisor: | Dr. Anita Enmark |
| Supervisor: | Prof. Dr.-Ing. Sergio Montenegro |
| Advisors: | Markus Wittkamp, MSc. |
| | Jochen Barf, MSc. |
| Field of study: | Space Science and Technology |
| Date: | 15.11.2019 |


Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

LULEÅ
UNIVERSITY
OF TECHNOLOGY

MORABA
MOBILE RAKETENBASIS

DLR

**Abstract**

During the flight of a sounding rocket often multiple ground stations are used to receive telemetry data. The overlapping time in which the telemetry stations can receive data from the rocket is rather high (up to 100%), compared to other setups like air and space applications, where it is only used to switch from one ground station to another. This, and the knowledge about the data format of the transmission protocol allow new approaches for Best-Source-Selection and error correction.

This thesis provides a working software implementation which enables the enhancement of data streams received by multiple ground stations from the same transmitter by conflation of the streams and performing forward error correction. For the forward error correction the existing protocol had to be analyzed to get the used error correction algorithm and with it the statistical information on wrong detection possibilities. The usage of a higher OSI-Level together with the long overlapping times of data streams allows to build a software with advanced capabilities compared to simple Best Source Selectors. It provides the possibility to replace defect frames in the output by validated ones of other inputs and furthermore eliminate even wrongly detected errors.

# Declaration

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources in the list of references.

Würzburg, November 15, 2019, _____

(Moritz Aicher)

# Contents

**1 Introduction** **2**
1.1 Outline . . . 3
1.2 DLR - German Aerospace Center . . . 3
1.3 MORABA - Mobile Rocket Base . . . 3
1.4 State of the Art . . . 5
1.5 Motivation . . . 6

**2 Background** **7**
2.1 Theoretical Background . . . 7
2.1.1 OSI-Model . . . 7
2.1.2 Noise . . . 7
2.1.3 Bit Error Rate (BER) . . . 8
2.1.4 Hamming Distance . . . 8
2.1.5 Block Code . . . 9
2.1.6 Hamming Code . . . 9
2.1.7 Forward Error Correction (FEC) . . . 9
2.2 Practical Background . . . 10
2.2.1 Sounding Rockets . . . 10
2.2.2 Flight Phases . . . 11
2.2.3 Rocket Range . . . 12
2.2.4 Experiment Data . . . 14
2.2.5 Communication Channel . . . 14
2.2.6 Data Transmission . . . 14
2.2.7 TFrame Structure . . . 16
2.2.8 TMFrame Structure . . . 16
2.3 Best Source Selection Strategies . . . 17
2.3.1 Frame Synchronization Pattern . . . 18
2.3.2 Data Quality Encapsulation . . . 19
2.3.3 Digital Pattern Mode . . . 20
2.3.4 Data Quality Metric Mode . . . 21
2.3.5 Cyclic Redundancy Check Mode . . . 21
2.3.6 GDP Weighted Majority Vote with Data Quality Encapsulation . . . 21

**3 Results** **23**
3.1 Error Correction . . . 23
3.1.1 Reverse Development of FEC Code . . . 23
3.1.2 (12,8) Code . . . 25
3.2 Software Architecture . . . 28
3.2.1 Data Word . . . 29
3.2.2 Object Pooling . . . 32
3.2.3 Input Handler . . . 33
3.2.4 Frame . . . 36
3.2.5 TFrame . . . 38
3.2.6 TMFrame . . . 40

3.2.7 DataStream . . . . . 41
3.2.8 DataQualityManager . . . . . 42
3.2.9 StreamMerger . . . . . 48
3.2.10 Double Exponential Smoothing . . . . . 57
3.2.11 Statistics . . . . . 59
3.2.12 TimeManager . . . . . 60
3.3 Verification . . . . . 62

**4 Conclusion** **64**

**A Matlab Code** **67**
A.1 Hamming Distance Calculation . . . . . 67
A.2 Possibility of Undetectable Errors in Code Word . . . . . 68

**B DataGenerator Framework** **69**
B.1 DataWord . . . . . 69
B.2 Frames . . . . . 71
B.3 Stream . . . . . 74

# List of Figures

1 Worldwide sounding rocket launches per year with an apogee above 80 km of the past 25 years, numbers generated from entries in sounding rocket database of [McDowell(2019)] . . . 2
2 Organizational chart of Space Operations and Astronaut Training and MORABA . . . 4
3 OSI-Model layer representation. Stream merging takes place in highlighted layers. . . . 7
4 Bit Error Rate for PCM-FM signals according to [IN-SNEC(2009), Typical PCM-FM Bit Error Rate Efficiency] . . . 9
5 Sounding rocket MAPHEUS 5 with two motor stages (S30 and S31) and several payload modules, by Mobile Rocket Base (MORABA) . . . 11
6 Schematic representation of a sounding rocket flight with two motor stages. Telemetry stations are located close to the launchpad. Rocket schematic by MORABA . . . 13
7 Typical ground station setup for an aircraft test range, taken from [Nicolo(2018)] . . . 13
8 Schematic representation of the continuous transmission from experiment and housekeeping data to users with two ground stations . . . 15
9 Simple Best Source Selector switch with three inputs and one output . . . 18
10 Time shift between different streams. Taken from [Nicolo(2018), p. 49] and modified . . . 19
11 Bit-by-Bit majority vote between different streams using data from Quality Encapsulation. Taken from [Nicolo(2018), p. 54] and modified . . . 20
12 Correction circles for a Hamming Distance of between a tuple of code words $X_i$ and $X_j$, taken from [Tran-Gia(2015)] and modified . . . 26
13 Schematic representation of the data flow inside the software. Red: data flow, blue: DataWord object flow, green: DataWord with filled objects flow, yellow: metadata flow, gray: time data flow . . . 28
14 Three streams with a maximum ID count of 5. Green frames are matching, red frames do not match. The second stream is shifted by one cycle count compared to stream one. The third stream has frame loss. . . . 46
15 Matching frames of each stream are added to a list for comparison and majority vote . . . 52

# List of Tables

1 Transfer frame (TFrame) structure . . . 16
2 Telemetry frame (TMFrame) structure . . . 17
3 Count of code words with the weight A . . . 27

# 1 Introduction

For a long time Best Source Selectors have been in use on test ranges all over the world. The development of new technologies in this field is inert and was not in the main focus for test engineers for a long time. However, over the past years with new and faster processors and Field Programmable Gate Array (FPGA) different approaches became feasible while real-time data analysis became progressively important to the operators, especially with recent development of autonomous systems. This brought the development of source selection back into the focus.
One possible challenging use case for best source selection are sounding rockets, which present a special application on test ranges and space centers. With a very characteristic trajectory and application domain they fill a niche in scientific research platforms and have special demands compared to aircraft, drones and other aerospace systems especially when compared to launch systems performing orbit injection. Most flights differ very much in the mission objectives based on the changing scientific background, reaching from micro gravity experiments to technology demonstrations. With a flight monitored by several redundant ground stations this results in the need of a source selection that does not only switch between incoming telemetry streams but is also able to extract the best possible data to provide a clean outcome while being flexible to adapt to new demands at the same time.
When looking at past 25 years of recorded sounding rocket launches across the world that reached an apogee of at least 80 km height, it can be seen that the market did not change radically during that time (See Figure 1). Around 84 launches can be expected per year, mostly form the same launch providers. This makes clear that the market for products dedicated to sounding rockets, like a specialized Best Source Selector, is small and therefore a lot of products are tailor-made. The software framework developed in this thesis is designed precisely for such a custom purpose.

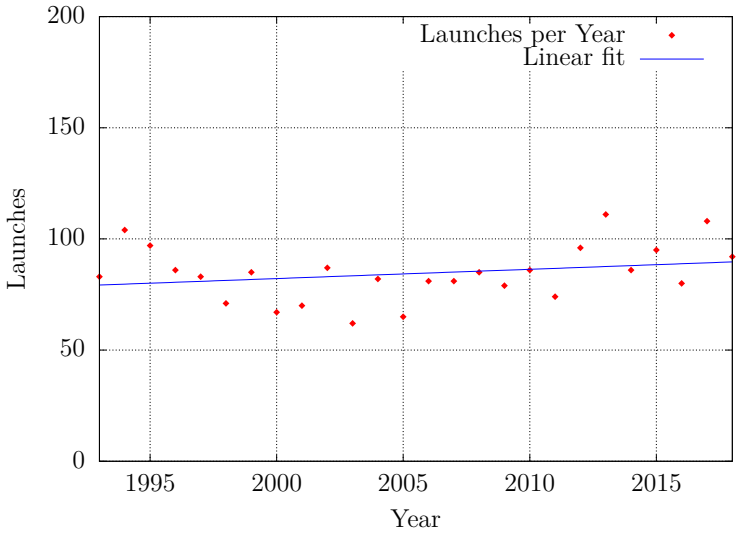


Figure 1: Worldwide sounding rocket launches per year with an apogee above 80 km of the past 25 years, numbers generated from entries in sounding rocket database of [McDowell(2019)]

## 1.1 Outline

This thesis provides a working implementation of a data stream merger to be used as a technology demonstration to determine its further capability of using it to replace state-of-the-art Best-Source-Selectors. This merger is capable of parallel decoding of incoming data streams of different sources as well as of detection and correction of errors package-wise in soft real-time. Errors will be corrected by correlating frames in different streams and using different approaches to find the best suitable outcome. The output result is a data stream of the same format with the best possible set of data extracted from the input provided by all telemetry stations. It will give an overview of the error correcting algorithms and the detection probabilities that are being used to conduct the forward-error-correction. Furthermore, it will be shown how the software has been implemented and which issues had to be addressed in order to provide a stable software solution.

## 1.2 DLR - German Aerospace Center

The German Aerospace Center (German Aerospace Center (DLR)) in which environment this thesis was written is the German national center for research and development of aerospace applications but also in areas of energy, transportation and robotics. Its 8700 employees are distributed over 26 locations across Germany and four locations outside the country. With a research and operation budget of over 1 billion euro and a dedicated budget of over 1.5 billion euro it provides a powerful institution for cutting edge research. (See [DLR(2019)]) Besides research DLR functions as the German Space Agency providing operational support for space missions. The operational part is located at the Institute for Space Operations and Astronaut Training at the site in Oberpfaffenhofen.

The Institute for Space Operations and Astronaut Training is responsible for space operations and for research in related subjects. It consists of Mission Operations which is part of German Space Operation Center (GSOC), Mission Technology, Ground Stations and Communications (GSOC), Spaceflight Technology (GSOC), Microgravity User Support Center (MUSC), Astronaut Training, Controlling & Acquisition and the Mobile Rocket Base (MORABA) as shown in Figure 2. For more information see reference [RB(2019)].

## 1.3 MORABA - Mobile Rocket Base

MORABA is doing research in the field of sounding rockets as well as carrying out unmanned parabolic flights and experimenting under micro gravity conditions at high altitudes. Applications are related to atmospheric, astronomy, geophysics, material sciences or hyper sonic research (see [MOR(2019)]). The mobility of this unit enables scientists to perform these tests all over the world, from arctic climate above the polar circle up to hot humid climate at the equator. The MORABA itself consists of five groups. This thesis is written with the data handling group which provides hard- and software know-how on collection of experiment data, observes housekeeping data during test and launch campaigns and delivers data to the scientists and engineers during and after the flight. The Telemetry, Tracking & Command group is responsible for operating the telemetry ground stations and the transmitters on the rocket as well as for the transmissions itself.

On request radar services can be provided to the customer. Control and Sensorics provide attitude and rate control to achieve certain trajectories and rotation rates. Launch Services provide everything necessary to launch a rocket system and Mechanical Flight Systems are concerned with structural development and production of rocket parts.

Telemetry, Tracking & Command is closely connected to the Data Handling unit where which this thesis was conducted. A typical mission can be roughly divided into six parts. It starts with the experiments design which has to be done by the scientists themselves. However, they will be provided guidance in how to build and fabricate their constructions to be space harden and secure. Secondly, the experiments need to proof their feasibility during design reviews. Afterwords the hard- and software components will be produced. Before being accepted and mounted on the rockets they have to be verified for functionality under stress conditions by environmental tests including vibration and vacuum tests. If all reviews are successful the experiment can take place on a suitable launch side for example at Esrange Space Center in Sweden or Andøya Space Center in Norway. A typical sounding rocket for micro gravitation missions is launched in a parabola flight, separates the motor and performs a de-spin to reach micro gravity without external influences, reentering through the atmosphere and landing the payload with a parachute. Then it is picked up by a helicopter and brought back to the range where experimenters investigate their results. A more detailed explanation on the flight phases can be found in section 2.2.2. During the flight experiment data can be sent down together with the housekeeping of the rocket through the onboard data handling system, a radio link, a ground station system and a data proxy to the operators. While the data passes trough all this stations errors might occur, especially in the radio link.

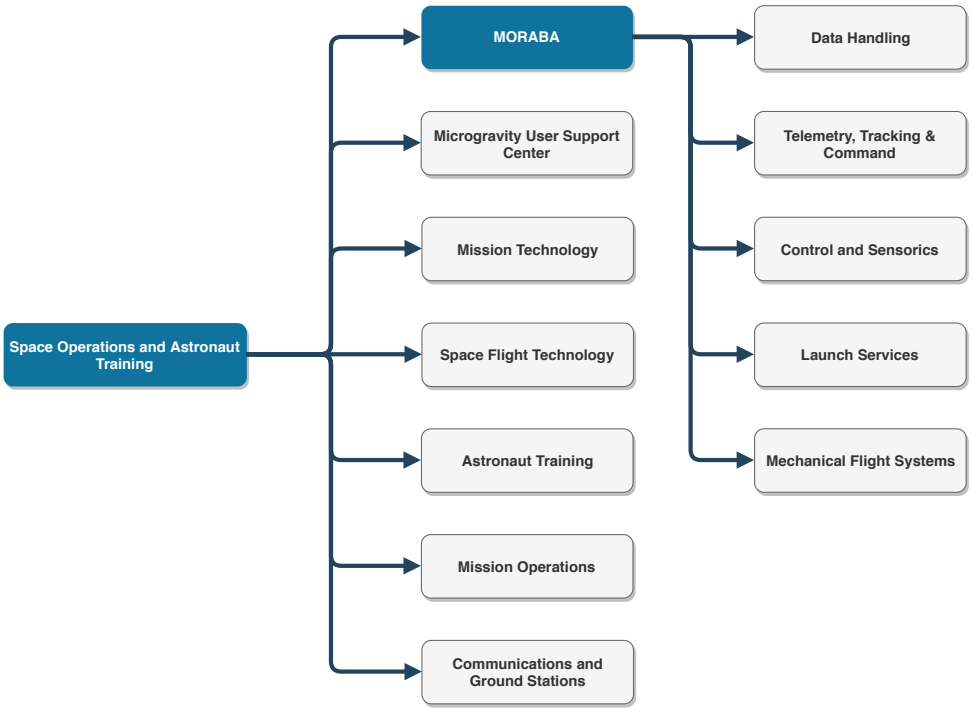


Figure 2: Organizational chart of Space Operations and Astronaut Training and MORABA

## 1.4 State of the Art

This chapter shall provide an overview of the current state-of-the-art technology used on rocket and military test ranges for data path selection and fusion. Since especially military test ranges have a very conservative information policy for obvious reasons, only few papers are available describing the technology that is used currently. More details can be gained from data sheets and leaflets of manufacturers like GDP Space Systems for telemetry equipment. For this reason the information has to be looked at with some precaution regarding the timeliness and actual performance.

Data transmission on test ranges is always an important but difficult topic. Test parameters, subjects and configuration change constantly and rapidly from mission to mission and sometimes even during the operation itself. The objects (e.g. rockets, aircraft) to be examined might not survive the procedure and extracting data afterwords can be difficult or impossible. So it is particularly important to receive, enhance and store as much data as possible in real-time. Not only for the later use by the scientists but also by the experiment operators who for example utilize the data for adjusting test parameters, controlling outputs. Also, this data is used to maintain range safety. On top of these requirements the objects to be tested move fast, in case of super sonic rockets up to ten times the speed of sound. Antennas have to track the sender continuously until the signal gets too weak to be used and other antennas have to take over or have to compensate data loss, since all this has to be done without interrupting the flow of information.

The development of best source selectors reaches back into the mid of last century. Simple techniques were used to switch between data streams of pre-prioritized antennas. A schematic illustration of this can be seen in Figure 7.

A simple Best Source Selector (BSS) switches between several inputs. The inputs are prioritized by the operators beforehand. If a signal gets too weak, the BSS switches to the next stream. This comes with several disadvantages which are further explained in Chapter 2.3.

Based on the encryption of data streams a pattern recognition with state machines was introduced, comparing pattern in all streams to do the matching. This was the birth hour of Correlating Best Source Selector (CBSS). The appearance of CBSS was necessary since single frames could not be determined in an encrypted stream. In a further developed advanced approach data received could be encapsulated with additional data generated in the telemetry ground stations and then sent to the BSS at a rocket base. This method is called Data Quality Encapsulation (DQE) and the additional information can be used as a basis for decision-making in the switching process without using pre-prioritized streams. Both methods can be used to enhance each other as can be seen in Figure 11. A bit flagged with poor quality is compared to the other streams and the majority of the answers is chosen. A third method is to use a Cyclic Redundancy Check (CRC) or Forward Error Correction (FEC) to detect errors in transmitted data (see [Nicolo(2018)]). The advantages and how to overcome the down sights are discussed in Chapter 2.3.

## 1.5 Motivation

After being founded in 1966, MORABA experienced technical development over many decades. This means that there is also lots of legacy involved that cannot be altered easily because of system dependencies. This reaches from analog technology, logic implemented in hardware to newer software components relying on aged formats and connectors. However, there is also a constant force in lifting the technology to the next level to keep up with the newest research and developments. Naturally, one of the fields heavily under construction is the software. As mentioned before, many components which used to be implemented in hardware can now be replaced by software making processes more flexibly. Not only does modern computing hardware become constantly cheaper, the reliability of recent operating systems gets better with time. Based on this above described a feasibility test for stream selection was in discussion, where the software implementation could allow a maximum of flexibility while being cost-efficient at the same time.
With further investigation into the topic new possibilities arose. Currently, the ground station only consists of one or two ground stations with one or more receivers depending on the mission, but more would be possible. This means that the data streams could not only be switched between the best or strongest signals but also a fusion of streams could take place to reach a better overall stream quality. And on top of that, even error correction could be done in real-time, not only in the post-processing. With sites at different sections of the ground path it is possible to increase the reception time further, however, in this case it is imported to switch between the stations without data loss.

All these points sum up to the motivation to conduct a concept design and a test-wise implementation of such a software. Further, in this thesis background research, architecture, implementation and results will be discussed.

# 2 Background

With this section background information is given to lay a foundation and a more detailed understanding on how the implementation took place and what circumstances had to be considered.

## 2.1 Theoretical Background

The theoretical background gives an overview of the most relevant technical terms and techniques as well as basic numbers used in the implementation.

### 2.1.1 OSI-Model

The Open Systems Interconnection (OSI) reference model provides a basis for system interconnections to exchange information. It reaches from the physical layer up to the user interface and divides the communication into different layers with different objectives. By this schematic any kind of data transmission can be fit to the OSI-layer model. For a more detailed information see the reference [ISO(1994)]. It to get mutual understanding of different transmission methods as well as recognition implementations.

Figure 3 shows the bottom-up view of the layer system. Transmissions handled by the Stream Merger build in this thesis operate in highlighted layers two and three. For this work it is assumed that there are no insights in the layers above, like the application or session layer.

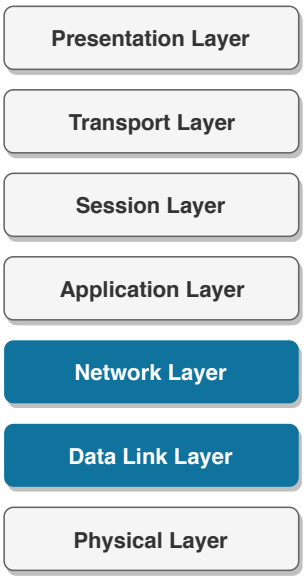


Figure 3: OSI-Model layer representation. Stream merging takes place in highlighted layers.

### 2.1.2 Noise

Noise $N$ in signal processing stands for an unwanted disturbance value on a measured signal and is a mostly random fluctuation on top of the signal. On the receiver side it can

lead to a false interpretation e.g. due to quantization of the analog signal. Noise cannot be avoided completely but as one can see in Section 2.1.3 a more powerful transmission reduces the chance of a bit failure by rising the signal power higher than the noise power. An additional method to avoid bit errors on the physical layer is to correct damaged bits by performing error correction on higher OSI levels if their occurrence is below a certain boundary.

### 2.1.3 Bit Error Rate (BER)

The Bit Error Rate (BER) of a data transmission is not only dependent on the energy per bit ($E_b$) to noise spectral density ($N_0$) with the inversed Gaussian error function $erfc$. The error is also dependent on the modulation used for the transmission. The energy per bit is calculated by dividing the carrier power by the bit rate $[E_b] = \frac{J}{\frac{1}{s}} = Js$, while the noise unit is also $[N_o] = \frac{J}{Hz} = Js$ it means that $\frac{E_b}{N_0}$ is dimensionless (See [Breed(2003)]). It has to be distinguished between the transmission BER and the overall BER of the process. The here considered one is the raw error rate without correction. Error correction on a higher OSI-level can improve the performance further. The inversed Gaussian error function $erfc$ is described by the integral:

$$erfc(x) = 1 - \frac{2}{\sqrt{\pi}} * \int_x^\infty e^{-\xi^2} d\xi \tag{1}$$

Since there is a dependency between the $\frac{E_b}{N_0}$ and the BER it can be calculated as follows:

$$BER = \frac{1}{2} * erfc\left(\sqrt{\left(1 - \frac{\sin(2\pi h)}{2\pi h}\right) * \frac{E_b}{2N_0}}\right) \tag{2}$$

This equation is taken from the handbook of the Telemetry Receiver [IN-SNEC(2009)]. The $h$ variable describes the modulation index, which is set to $h = 0.7$.
The modulation used for the down-link is Pulse Code Modulation (PCM)/Frequency Modulation (FM) as defined in the Inter Range Instrumentation Group (IRIG) standard (IRIG106). Telemetry Tracking & Command group of MORABA (1.3) tries to maintain a $\frac{E_b}{N_0}$ above 13.5 dB which leads to a maximum BER of around $1 * 10^{-7}$ per bit as can be read from Figure 4. This means that one out of $1 * 10^7$ bit can be considered to be damaged at the receiver side. For further calculations this value is used assuming the worst case for operations.

### 2.1.4 Hamming Distance

The Hamming distance ($d$) between two data words $(u, v)$ with the exact same length is equal to the number of symbols in which these words are differing. The Hamming distance is described as $d(u, v)$. If $d(u, v) = 0$ the information content of the data words is the same. (See [Butterfield et al.(2016)Butterfield, Ngondi, and Kerr])

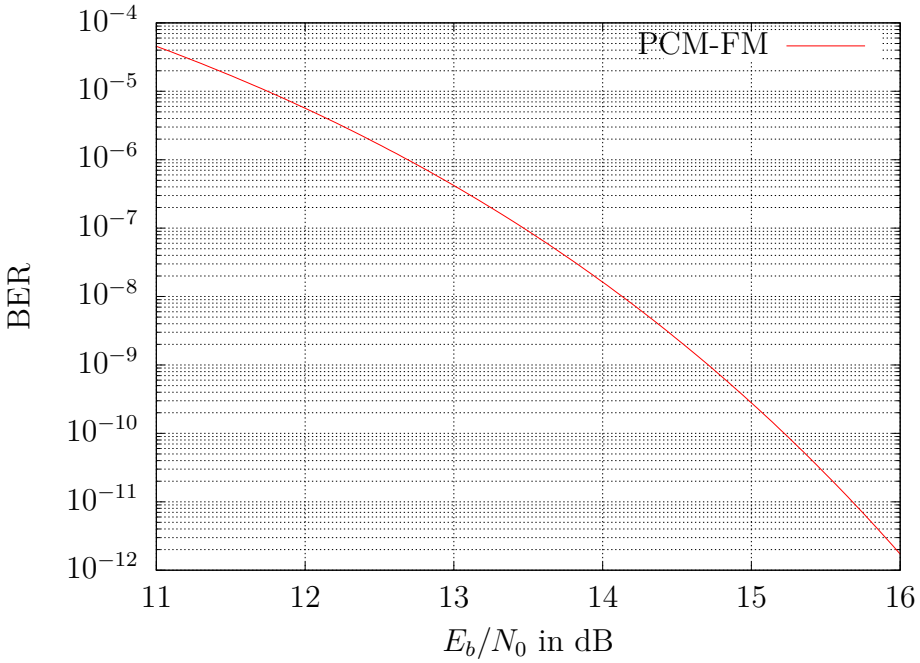


Figure 4: Bit Error Rate for PCM-FM signals according to [IN-SNEC(2009), Typical PCM-FM Bit Error Rate Efficiency]

### 2.1.5 Block Code

Block Codes are codes used for error-correction and detection. They are characterized by a fixed length of symbols which can be decoded uniquely without knowledge of other blocks. A block is referred to as a code word with a length $n$ and an information content of $k$ symbols. Some additional symbols are added, so that $n > k$, creating a redundancy in the information content. This can be used to detect and correct errors. A block code is described as $(n, k)$-code which indicates the overhead but also the maximum of wrong symbols that can be detected and corrected. (See [Butterfield et al.(2016)Butterfield, Ngondi, and Kerr])

### 2.1.6 Hamming Code

The family of Hamming Codes are a set of error correcting, linear and perfect block codes (2.1.5) fulfilling the following characteristics:

$$n = 2^m - 1, k = n - m \tag{3}$$

Where m is the number of additional added symbols used as control and correction redundancy.

### 2.1.7 Forward Error Correction (FEC)

Protocols relying on two-way-communication channels a checksum for detecting errors is sufficient, e.g. TCP. A client can use it to test its received data for errors and in case an

error occurred a re-transmission of the data will be initiated until a correct package has been verified at the receiving side.

Unfortunately the communication channels used on sounding rockets often rely on a single-way communication channel. Which means that it just consists of a down-link and a package that has been altered during the transmission process will not be resent by the sender. For this case a Forward Error Correction technique is used.

## 2.2 Practical Background

In this chapter on practical background the technical environment is described in which the development took place. Implementation fundamentals as well as basic knowledge will be provided to the reader.

### 2.2.1 Sounding Rockets

Sounding Rockets are a special group within the rocket family. For their scientific purpose they are launched following a parabolic trajectory without performing an orbit injection and descending back to earth. During the flight they can reach trough all layers of the Earth's atmosphere up till the boarder of the Exosphere 700 km above the ground. These high altitudes cannot be reached balloons. The scientific purpose is mostly but not exclusively civil. The flight through the different layers of the atmosphere enables scientists for example to measure gas configurations and particle densities like plasma which provides important data on climate and global warming. But not only chemical experiments can be performed, the parabolic flight allows a short duration of micro gravity of at least $10^{-4}\frac{m}{s^2}$ down to $10^{-7}\frac{m}{s^2}$ ranging from some seconds up to 12 minutes depending on the height of the apogee (see [Pallone et al.(2018)Pallone, Pontani, Teofilatto, and Minotti]). This can be used to execute biological or physical experiments. Also, the high altitude with space conditions enables scientists to conduct technological and material verification. Since the parabola has a high altitude but the ground distance is rather short, the rockets payload can be picked up and returned to the rocket range again for investigation and data extraction.

This huge variety of capabilities makes sounding rockets a unique tool for science and research. On top of this, a sounding rocket mission is inexpensive compared to a satellites mission or a flight to the International Space Station (ISS). Disused military rocket motors like M112 Hawk rocket motor can be depleted for these civil purposes (see [Ast(2019)]).

A sounding rocket usually consists of the following parts. It starts with a rocket motor where fins are attached for a stabilized flight by introducing a spin to the whole rocket of around 15 Hz. The motor can be, but not always is solidly fueled, which means that it is ignited once and then burns until the fuel is exhausted. A thrust profile has to be build in while manufacturing the motor and it cannot be changed during the flight like it is possible with liquid fuels where e.g. a valve can be opened or closed to provide more or less fuel to a burning chamber. However, solid fuel is more easily to be handled since it does not need maintenance like cooling, pressure and has only very few mechanical parts. The rocket is build modular with different module rings of multiple heights for different purposes. The amount of modules varies with the mission and only the standard modules are listed here. On top of the motor there is a so called Recovery Module attached. The

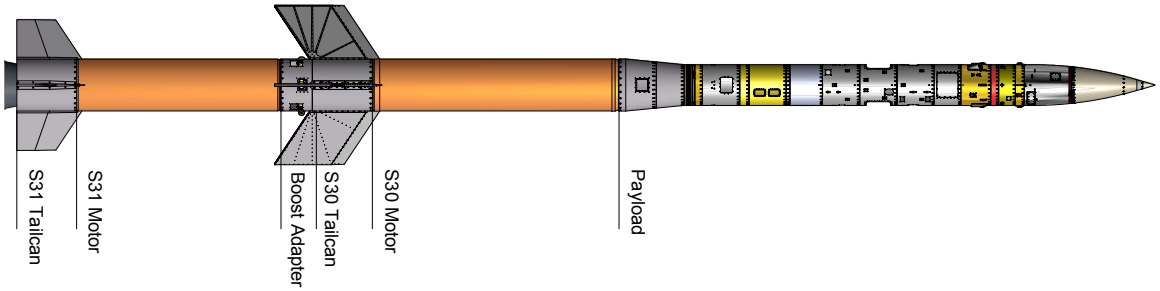


Figure 5: Sounding rocket MAPHEUS 5 with two motor stages (S30 and S31) and several payload modules, by MORABA

attachment is locked by a ring which can be removed by firing small pyrotechnic initiators to separate both parts after the engine burned out. (See Section 2.2.2)

As an example, the possible configuration of a MAPHEUS rocket is described. Other configurations are possible, depending on the used system and mission requirements. In this case the Recovery Module contains the parachute and the pressure sensor to land the payload safely on ground. Attached to the recovery module there is also a mechanism for mechanical de-spinning of the payload after separation. Depending on the configuration it is followed by a rate control system to maintain a precise rotation rate or to keep the rate close to zero. On top of the Recovery Module there is the Service Module containing all the electrical equipment necessary for the flight like batteries, on-board computers, timers, sensors, radio transceivers etc. It also provides connectors for the experiments which are in separate modules and contains the connector for the umbilical power and data lines while attached to the launcher or during testing. As mentioned every experiment is placed in a separate module, it is provided a connector for power and telemetry data. The scientists are more or less free in what they are doing as long as it meets the electromechanical requirements. Also, the choice of the data format for the down-link is up to them the scientists, since the data is repacked inside an internal protocol format (see Sections 2.2.4 and 2.2.6 for further information).

The experiment modules are stacked an placed on top of the Service Module. At the tip of the rocket there is the nose cone which as well can house experiments. Is also has an antenna for the GPS receiver in its tip and can as well be removed by a ring connector. But its main purpose is of course the aerodynamic shape to reduce drag during launch. A completely assembled sounding rocket with a dual stage S30 and S31 motor can be seen in Figure 5. To reach higher altitudes or different flight profiles a second stage can be attached. The bandwidth can change with the flight phases as experiments are started and stopped facing different criteria variations for example micro gravity or pressure.

### 2.2.2 Flight Phases

At the beginning of a mission launch the rocket is placed at the launchpad, usually attached to a rail. Just before the countdown reaches zero the engines are ignited. The rockets starts moving along the rail still attached to the umbilical which consists of power and data wires. Just by the lifting force the wires are plugged out of the service module.

Before that the power was switched to the internal batteries but the data transmission over wire gets interrupted. This interruption is defined as "Lift-off" and this exact moment will be used as a fix time reference later the process. However, no data gets lost, the ground stations were already receiving data and transmitting it to the operators.

Since the engines used are former military rocket motors the acceleration during the first flight phase is enormous and reaches up to 21 times the gravitational force (for Improved Orion motors). The flight is stabilized by spinning the rocket up to around 11 Hz. After this short but rapid burning phase the motor is burned out and decoupled from the payload modules. Because of the velocity, the height above ground is still increasing. Now the "de-spinning" takes place where two weights on wires are released to bring the rotation down to roughly 0 Hz. It is possible to reduce the spin further by using an active rate control system. With the de-spinning the zero gravity phase starts. No more forces are acting on the payload system. Often the experiment phase of the scientists starts at this moment. For some experiments live data is essential during the short micro gravity phase in case a rocket cannot be recovered afterwords or since parameters have to be changed using an up-link. Also, verification of is important to give feedback to the construction engineers. This state will be maintained during descending until it starts to interact with increasing density of the atmosphere again. At a certain point the rocket loses the telemetry connection since it descents behind the horizon and the telemetry is switched off by a timer after touchdown. Just before touchdown the landing position will be sent via a dedicated Iridium Satellite up-link to the recovery team. This transmission is not recorded by a ground station. Depending on the landing zone the payload is collected by a helicopter or ship and brought back to the rocket range for further investigation by the scientists. The rocket motor might not be recovered immediately.

### 2.2.3 Rocket Range

Throughout this thesis the term "rocket range" or "test range" sometimes is used. A "rocket range" is a restricted access area where, as the names intends, rockets are launched. It does not only contain the launcher, but everything that is necessary to support and maintain scientific or technology experiments. This reaches from a range safety department which concerns things from handling toxic to explosive material as well as safety during launch to the actual launch staff and scientists.

Rockets launched within the area do not necessarily land within that area since the distances covered by the vehicle can be immense. Ranges can usually be found in little inhabited areas like "Esrange Space Center" in Sweden above the polar circle or the well known military test range "White Sands Missile Range" in the desert of New Mexico, USA. Usually receiving antennas are located along the flight path of an aircraft observed, like displayed in the schematics by GDP Space Systems Figure 7 while the the Best Source selection takes place on the range (see [Nicolo(2018)]). For the special family of sounding rockets which have a parabolically shaped trajectory this does not apply. The ground range is rather short compared to the height which means the antenna systems are close to each other. Therefore, the rocket is in the field of view of several antennas at the same time. Actually, the overlapping time can be considered above 95%. This configuration is the basis for the conflating best source selection.

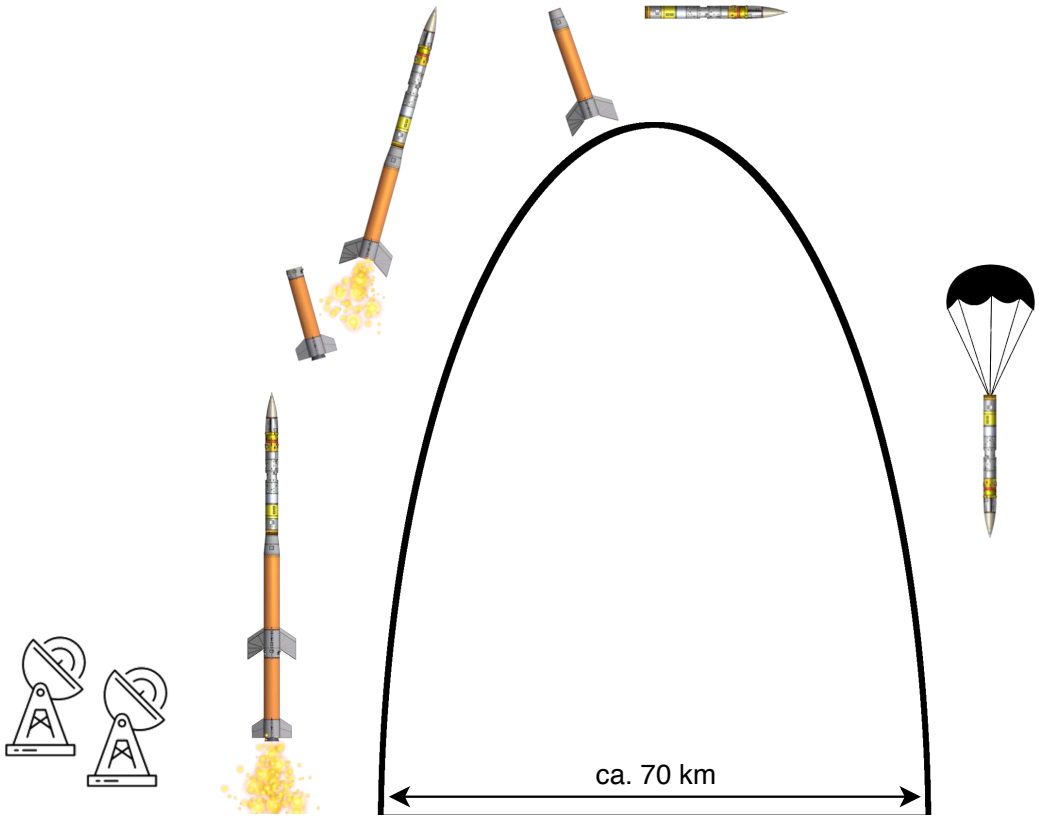


Figure 6: Schematic representation of a sounding rocket flight with two motor stages. Telemetry stations are located close to the launchpad. Rocket schematic by MORABA

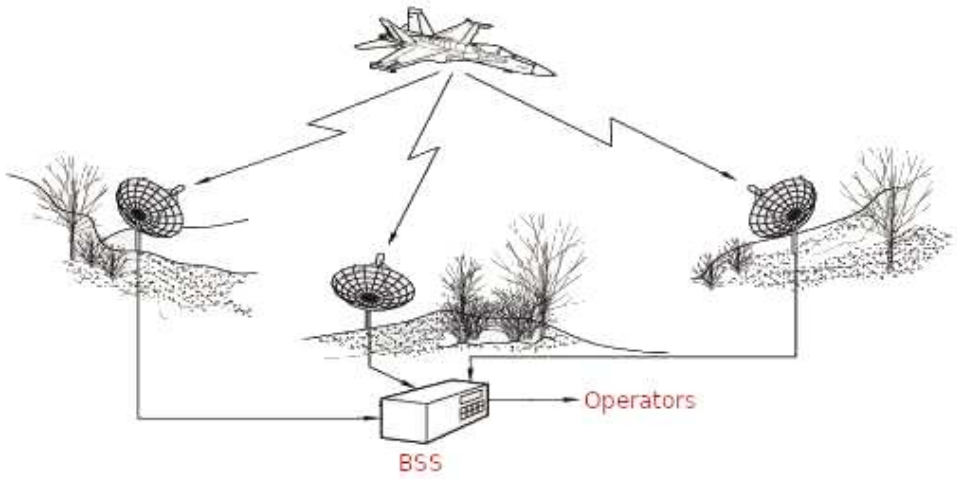


Figure 7: Typical ground station setup for an aircraft test range, taken from [Nicolo(2018)]

### 2.2.4 Experiment Data

As stated in Section 2.2.1 the rocket's payload with the experiments will be recovered after the flight, however, often it is necessary for the scientists to receive live data during the flight, even so most of the time only a down-link is provided instead of a bi-directional connection and the experiment phases performed are autonomously triggered by pre-defined signals like Lift-Off. Normally the majority of the data is stored onboard and only the most important parameters are sent to the ground station, this has to be done since a safe pickup of the payload can never be guaranteed. The rockets Service Module can be seen as a black box for the experiments. A RS-422 or Ethernet connector is provided where a fully transparent asynchronous down link can be established (see [Schuettauf et al.(2018)Schuettauf, Kirchhartz, et al.]). The same interface is provided at the science center where the scientists operate their experiments. This means that the whole transmission process is transparent to the scientists and the only limitation is to stay within a pre-defined transmission rate. The protocol for performing the transmission is up to the customers, providing maximum of flexibility.

### 2.2.5 Communication Channel

During the whole phase from lift-off until the experiments are shut off data is transmitted from the rocket to the ground stations. Figure 8 shows the logical data path. In the upper part, one can see the path inside the rocket. Data from the experiments and housekeeping like rates, acceleration, GPS-position, currents, temperatures etc. are collected inside the "Data Handling"-subsystem which is placed inside the "Service Module" where it is chopped and packed into IRIG-106 frames, see Section 2.2.7. The packages are then modulated with PCM/FM on a carrier signal and transmitted down to one or more ground stations. In this case there are two station which receive the signal and demodulate it. Inside a so-called "Proxy" the frames receive a second header with additional information like size, version, quality etc. This will be explained in detail in Section 2.2.8. From these Proxy the frames are sent via an Ethernet User Datagram Protocol (UDP) to the Stream Merger described in this thesis. Streams of several ground stations are merged in this process into one single stream. A "Stream Extractor" removes all frame headers on OSI-layer 2 and 3 and unpacks the data. It is then send to the scientists and operators.

### 2.2.6 Data Transmission

The housekeeping and experiment data is transferred via different paths to the scientists. In a first step incoming data is atomized into fix block sized data junks inside the Onboard Data Handling module. These junks are then repacked using a protocol close to the IRIG-106 standard for digital on-board recording. This standard is still widely used in telemetry of aeronautical telemetry applications of the Range Commanders Council (RCC) member ranges. (See [Baggerman(2019)])

Before lift-off, while the rocket is still attached to the launcher, data is transmitted via an umbilical, which is a bundle of different wires plugged into the service module. During the lift-off the umbilical is being pulled out of the rocket just by the force of the moving rocket. At that point the stream of data packages has to be received by a telemetry ground station via S-band. The signal gets received amplified and in the end

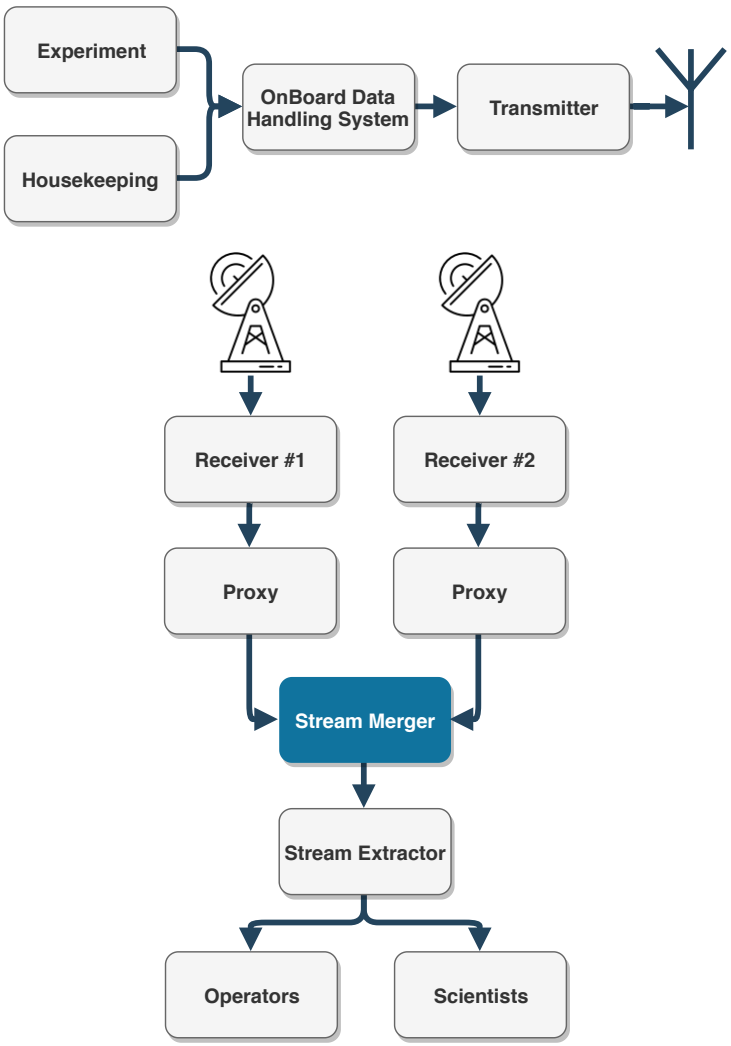


Figure 8: Schematic representation of the continuous transmission from experiment and housekeeping data to users with two ground stations

synchronized by a Cortex receiver. The Cortex receiver is a broadband radio telemetry receiver produced by Zodiac Aerospace used within MORABA telemetry stations. Then it is transmitted using Ethernet to a proxy service, which repacks the frames send by the Cortex in a suitable format which can be understood by the operators' software. Sometimes the Iridium satellite constellation is used for long range transmission. The sender is a satellite transceiver unit from where the data is routed through a network of 75 satellites (see [Tan et al.(2019)Tan, Qin, Cong, and Zhao]) in low earth orbit and down to a ground station. From there it is also sent through an Ethernet connection.

### 2.2.7 TFrame Structure

The transfer frame (TFrame) generated inside the Service Module has a very simple structure. It orients itself on the IRIG-106 frame format but implements a reduced version with less metadata inside the header, so that the overhead for the transmission gets reduced to a minimum. The header begins with a 3-byte sync word (0xFAF320) to easily be found in the stream. It is followed by 12 Bit words, called DataWord in this thesis. The first 8 bits are data with a 4 bit redundancy for FEC. Further information on the FEC will be given in Section 3.1.2 Theoretically an infinity number of DataWords can be inside a frame, however, the length is limited by the length counter in the TMFrame, see Section 2.2.8. The length is pre-defined before a flight, so that all frames have the same number of DataWords. However, it can vary from mission to mission.

Special attention in the implementation is required by the first DataWord. Even so the format is the same for all DataWords, the first word belongs to the header carrying a counter value which overflows every 256 counts. This is the only counter within the whole format. Since the format is part of legacy and will remain the standard for a longer period of time all soft- and hardware implementations require this frame format so the counter cannot be increased to improve the matching of frames inside the buffer as described in Chapter 3. For example the telemetry department does slant range measurements on the data stream which require the counter to stay in this shape. For this purpose even empty frames, only consisting of the header, the count and empty DataWords can exist, if there is no data to be transmitted at that moment. Since one DataWord consists of 12 bit an uneven count of DataWords plus one DataWord carrying the count have to be send to get an even byte number.

| | Sync | Count | DataWord #1 | DataWord #2 | DataWord #n |
|---|---|---|---|---|---|
| Size (Bit) | 24 | 8+4FEC | 8+4FEC | 8+4FEC | 8+4FEC |

Table 1: Transfer frame (TFrame) structure

### 2.2.8 TMFrame Structure

The telemetry frame is generated by the proxy and exists only in the ground segment packed in Ethernet and UDP frames. Its purpose is to provide additional information along with the TFrame and with this it allows a maximum of flexibility on the user side. One TMFrame can carry exactly one TFrame. The frame header is modular and can be extended with various information. On wired connections inside the ground segment the

overhead does not matter much compared to the radio transmission between rocket and telemetry stations. As the TFrame, the TMFrame starts with a synchronization word but consisting of 4 Bytes instead of 3. It is followed by a length value which describes the whole size of the frame in bytes. As described in Section 2.2.7, the count of DataWords is limited in length by this value. This means that a total size of 65536 bytes is possible. After the size a version number follows, which can be used to differentiate different versions of headers with different extra information. To allow a downward compatibility an offset field is given. It describes the offset from the first byte of the TMFrame-header to the first byte of the TFrame header. This means that if there is additional information the system is not aware of, it can just ignore this fields and jump directly to the frame body. The last header field in the standard header contains a double value (64 bits) to describe a quality index which can be calculated inside the Cortex receiver. For example, it can be based on values like bit synchronization errors, frame loss etc. After the header, the data field follows. It consists of a whole TFrame as in Table 2. No additional error correction is implemented in the frame's header however this is not necessary since the frames are transported over an Ethernet protocol like UDP (with CRC).

| | Sync | Length | Version | Offset | Quality | Body |
|---|---|---|---|---|---|---|
| Size (Bit) | 32 | 16 | 8 | 8 | 64 (double) | <TFrame> |

Table 2: Telemetry frame (TMFrame) structure

## 2.3 Best Source Selection Strategies

As described in Section 1.4, it is difficult do determine what are the details of the most recent approaches for best source selection in industrial and military applications. What all BSS have in common is that they have to monitor all available input data streams continuously to be able to adjust the output in a way so that the output has the best possible quality while preventing interruptions. A simple selector monitors the bit error rate and switches from one data stream to another if the bit error rate reaches a critical predefined threshold (see Figure 9). The switch just redirects the input to the output channel, all data from other channels is ignored. A rapid switching is possible. This implementation leads to a low technical effort and does not limit the maximum throughput of the system since the measurement can easily be done in real-time during the bit synchronization which is normally done by the receiver internally, even if there is no source selection taking place. This kind of BSS has no limit on the input channels since other channels than the selected one do not have to be processed.

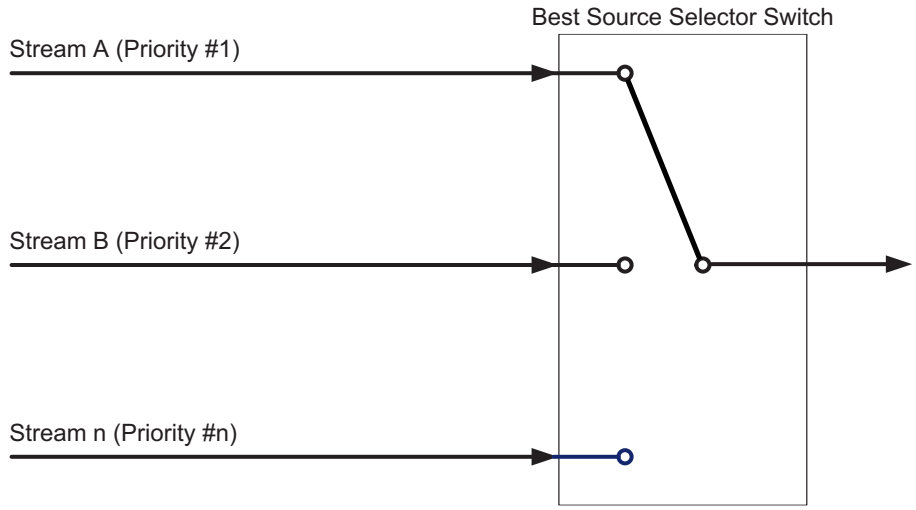


Figure 9: Simple Best Source Selector switch with three inputs and one output

Yet, the above described method has a lot of disadvantages. The incoming data streams are most certainly asynchronous due to different travel times of the signal to the antenna and then to the selector. If the output is switched to another input source there is also a jump in time in the future or past depending on the position of the current data stream in time. If the time jump is into the future, data will be lost since the output stream has lost frames or has a gap. Therefore, all the frames in between the points in time will just be omitted. The time jump is accompanied by a phase jump which requires an electronic system to find the lock on the stream again as can be seen in Figure 10. A lock means that the system's internal clock is synchronized to the arrival interval of new frames. This makes it easy to determine the beginning and the ending of a frame. Burst errors can lead to a rapid switching between input channels since the overall quality of the streams might be good, but the stream only has a short period of defective frames due to a lot of errors arriving in a short time interval. Also, the streams have to be prioritized e.g. based on the sequence of overflights over the antennas. This is especially important if the experiment is done over a large area like experiments with aircraft. In case of sounding rockets where the rocket is received by all ground stations roughly over the whole duration of the experiment a different approach has to be chosen.

However, there has been some development since the use of simple Best Source Selectors. To determine the functions of more recent applications it is useful to have a look on publications like [Nicolo(2018)] of manufacturers of BSS components. Several methods to improve the selection are mentioned in the Section 1.4 "State of the Art". There will be a more detailed discussion of these options, and they can contribute to a custom approach for the data stream improvement for sounding rockets.

### 2.3.1 Frame Synchronization Pattern

Apart from this simple switching done in the 80' and 90', with more powerful FPGA sophisticated ideas could be realized. The biggest disadvantage of the simple approach is

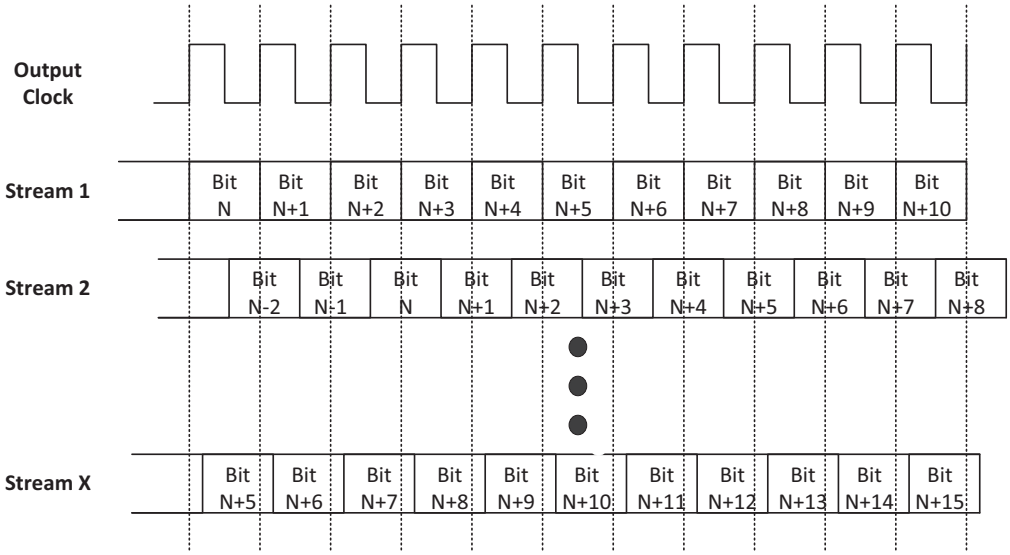


Figure 10: Time shift between different streams. Taken from [Nicolo(2018), p. 49] and modified

the data loss during switching. Buffers were used to synchronize the incoming frames in phase and time. Each stream was handled simultaneously and the synchronization offset could also be used as a quality index. With this a dynamic lossless switching between the streams became possible while maintaining an unlimited bandwidth. However, the correlation was only done based on the input timing by the frame synchronization pattern not on the data itself. The improvement of this idea is to use state machines to match patterns from one stream against the other streams and once a pattern has been detected the process is repeated after some time to verify that the streams are still synchronous. This was done in the more recent years from 2000 to 2010. It also allows matching of encrypted streams with random data. In fact random data is even better for this method since repeating patterns can load to a false positive in the correlation process. (See [Nicolo(2018)])

### 2.3.2 Data Quality Encapsulation

With advanced hardware it was possible to gather information not only inside the Best Source Selector but also directly at the remote located ground stations. The received data is repacked in a second frame carrying this information to the CBSS. This encapsulation of additional quality around the data itself is called Data Quality Encapsulation (DQE). This helps to make use of additional and more reliable information for decision-making for each stream but has the downside of a lager overhead potentially limiting the throughput of the transmission. In the worst case every bit has an additional bit describing whether

the signal was strong or weak during the reception, which leads to an overhead of 100%. But if this is acceptable it can be used to do a bit-wise majority vote on the data. In the majority vote a single matching bit of each stream is compared and the majority is accepted as the true value of this particular bit. This might sound very inefficient but leads to an output that can potentially be better than each single input stream. This process can be seen in Figure 11. Poor bits have a lower value than bits with a high signal quality. The majority is then calculated based on this information and placed on the output stream. Of course this requires powerful logic processing capabilities.

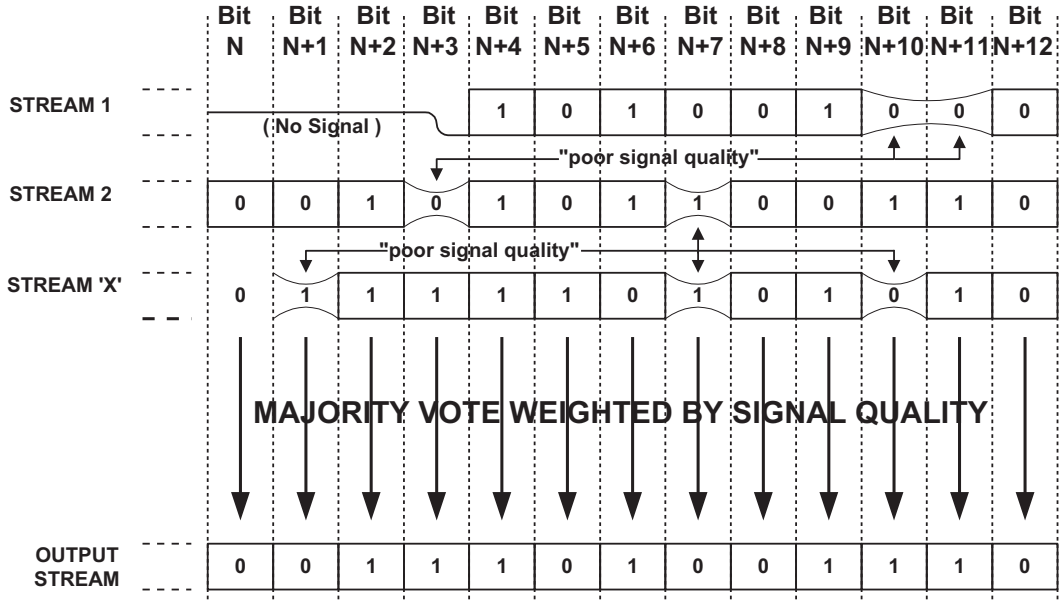


Figure 11: Bit-by-Bit majority vote between different streams using data from Quality Encapsulation. Taken from [Nicolo(2018), p. 54] and modified

### 2.3.3 Digital Pattern Mode

The digital pattern mode is a mode similar to the Data Quality Encapsulation. At first frame synchronization is used to correlate the sources. With the frame synchronization pattern a quality index can be derived by investigating the synchronization process. An advantage is that existing receivers provide $E_b/N_0$ (see Section 2.1.5) and Best Source Selectors can be used. No powerful hardware is necessary to calculate that information since it is already available in the correlation process. However, it needs to process quite a lot of data until trends or degrading infect the quality index. This mode can also be used together with majority vote. Nevertheless, in this case one needs pay attention to the fact that there are three sources available from the beginning or that the switch-over

to majority vote is fast in case a new stream appears to prevent data loss in between. (See [Nicolo(2018)])

### 2.3.4 Data Quality Metric Mode

Data Quality Matrix (DQM) is a new mode, developed by the Range Commanders Council in the US in the year 2017. Bit Error Probability (BEP) is used in the Data Quality Encapsulation but not bit-wise. Instead, a block of 4096 bits is encapsulated with a header of 48 bit containing 16 bit frame sync pattern and 16 bit for the Data Quality Metric additional with some other information. The DQM is translated from a table to the Bit Error Probability. This reduces the overhead in the stream and allows to use fully encrypted streams where no information about the content is available and the data seems to be random. The downside is a lack of information on the quality of smaller data junks. (See [Nicolo(2018)])

### 2.3.5 Cyclic Redundancy Check Mode

A mode working with the data itself instead of information based on external measurements like the $E_b/N_0$ is the CRC-Mode. CRC stands for Cyclic Redundancy Check and uses additional information added to a block of data. By adding additional structured bits generated from the content of the data block to the header or footer of a transfer frame it is possible to calculate whether the data in the data block is correct or if bit flips occurred during transmission. The code words with the redundant information are calculated using a generator polynomial. On the receiver side the same polynomial is used to calculate the code word. Now both code words can be compared. If they match, the data is most likely to be correct. "Most likely" means that the likelihood is dependent on the polynomial used and cannot be ensured to 100%. Based on this error detection a quality index can be calculated. The best source selection can take place based on this index. Since the data block has to be a static size to work the polynomial, it might be necessary to fill it up with zeros. This leads to an unnecessary transmission of data which can be seen as an overhead to the transmission together with the additional CRC-bits. This approach could lead to a difficulty in correlation since frames filled with zeros might look the same if no random data or variable data like frame counters are sent within the header. Also, buffering is needed since only whole frames can be analyzed. The buffer therefore depends on the data block size and the correlation interval. However, this approach also works with encrypted data. (See [Nicolo(2018)])

### 2.3.6 GDP Weighted Majority Vote with Data Quality Encapsulation

In the paper [Nicolo(2018)] submitted by GDP, this mode is said to be the most powerful available mode. It consists of three subsets. The quality is determined based on the analog signal during the bit synchronization and quality information is provided for every bit received. The second approach is using Data Quality Encapsulation based on the analog signals quality located at the ground station. By using DQE the quality information is sent inline with the data. The third does the same by transmitting the data directly out of the receiver over e.g. an Ethernet connection. The signal quality is monitored over a 256bit for every stream as a basis to do a weighted majority vote in case there are at least

three streams available. For two streams a bit-by-bit decision can be done as shown in Figure 11. It can be switched lossless between those modes for a dynamic stream management. With this setup the streams can correct each other so that the output is better than single inputs by a factor of up to 3. By the bit-by-bit DQE the required bandwidth is doubled between the receiver and the best source selector (See [Nicolo(2018)]).

In this approach of developing a software based on BSS it will be shown that it is possible to fuse all these functions together to provide the best possible output of the system and therefore doing a Best Data Selection. The data can even be enhanced by doing error correction in the process. It is not switched as in a BSS or CBSS so no frames can be lost during the process. The streams are monitored over time as well as a frame wise analysis of the data is done. This is made possible by the fact, that there is an insight in the protocols used on higher OSI levels. The downside is that the protocols have to be known by the system and, therefore, the system cannot be used when the protocol is unknown. Moreover, the software bound to the special use in the environment of MORABA. To improve the data further than with error correction it is necessary that several data streams are received in parallel at a time like it is the case for typical sounding rocket applications. The framework is build in a way, so that additional header information of the transfer frame can easily be added and even a replacement of IRIG-106 with a different protocol like the ones used by Consultative Committee for Space Data Systems (CCSDS) would be possible with small effort.

# 3 Results

In this chapter an overview of the implementation process will be given. It starts with a detailed analysis of the error detection and correction capabilities of the undocumented FEC implemented in the DataWords of the TFrame described in Section 3.1.2. Then an overview of the data flow inside the software will be given followed by a detailed insight in the modular structure of the system. Each processing step will be discussed in detail along with which problems occurred and how they were solved.

## 3.1 Error Correction

At first here is an overview of the process of obtaining the error detection and correction code for the further use in the correction process. Further criteria for the convolution of previous corrupted frames are discussed.

### 3.1.1 Reverse Development of FEC Code

Since there exists description nigher of the Forward Error Correction used in the transmission and nor of its mathematical capabilities, it had to be figured out by reversing the process to get to the mathematical model. Only the error syndromes for doing the correction and detection are available but no further information on the statistics. This information could be obtained from the data sheet of the CMX909 Package Data Modem of CML Microcircuits [CML(2008)].
Given is the following parity check matrix $H$ from [CML(2008), 5.5.2 FEC]:

$$H' = \left[\begin{array}{cccccccc|cccc} 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \tag{4}$$

From this matrix everything else has to be derived by calculations and educated guesses. The first striking thing to notice is that the matrix ends with an identity matrix of $4 \times 4$. This leads to the unverified assumption that a systematic code is used. Systematic codes are separated in data bits and parity bits for a better handling by e.g. FPGA. So the usage of such a code would make sense. With the formulas given in [Pless(1998), 1.2] it is now tested if our assumption can be translated into a model. Error syndrome take the following form:

$$H = (-A^{\mathsf{T}}|I_{n-k}) \tag{5}$$

And for binary codes $-A = A$:

$$H = (A^{\mathsf{T}}|I_{n-k}) \tag{6}$$

By looking at the Matrix 4 with the identity matrix at its end and the knowledge that a set of data is 8-bit per code word, it can further be assumed the following for the Equation 6. Where A is a matrix with $k \times (n-k)$ and $I$ as the $k \times k$ identity matrix. It can be derived from that $n - k = 12 - 8 = 4$, this means that the parameters would be $n = 12$,

$k = 8$ and therefore $I_k \times k$ is $I_{8\times 8}$. And following for $-A^\mathsf{T}$:

$$-A^\mathsf{T} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{7}$$

And the transformed version of the matrix:

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{8}$$

The standard form of the generator matrix $G$ can be described as in [Ling(2004), Definition 4.5.3 p. 52]:

$$G' = (I_k|A) \tag{9}$$

This leads to a possible generator matrix $G$:

$$G' = \left[\begin{array}{cccccccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right] \tag{10}$$

With a possible generator matrix it can now be tested whether the previous assumptions were right. A code word $c$ can be calculated by multiplication of the generator matrix $G$ with 8 bits of data [Ling(2004), p. 57]:

$$c = G \times d = d^\mathsf{T} \times G^\mathsf{T} \tag{11}$$

In the manual an example can be found for testing a code word on validity. This 12-bit code word can now be used to test the derived generator matrix.

$$d = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{12}$$

Which supposes to result in a code word of:

$$c = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{13}$$

This can now be used to verify the calculated generator matrix and therefore the code structure with the parameters $k$ and $n$:

$$c' = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \times \left[\begin{array}{cccccccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right] \tag{14}$$

$$c' = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{15}$$

With $c = c'$ follows that the previously made assumptions on the matrix structure in Equation 4 were right. Following is the code used for Forward Error Correction a (12,8)-Code where $n = 12$ and $k = 8$. Furthermore, a generator matrix was obtained which can now be used to generate valid input data for the software stream merger. With this knowledge the mathematical capabilities can be derived.

### 3.1.2 (12,8) Code

A (12,8) code produces a 12-bit output for a 8-bit input. With Equation 3 it can be seen that the (12,8)-code does not fulfill the requirements to be a simple Hamming code:

$$n = 2^m - 1, k = n - m \tag{16}$$

Calculating m from the second part of the requirements:

$$m = n - k = 12 - 8 = 4 \tag{17}$$

For setting $m = 4$, the result should be $n = 12$.

$$n = 2^m - 1 = 2^4 - 1 = 7 \neq 12 \tag{18}$$

This proofs that this is not a simple Hamming code.
The block code could also be an extended or shortened Hamming Code. An extended code has a minimum Hamming Distance of 4 with an additional parity bit. By calculating the distance between every word using a Matlab script (see Appendix A.1) it could be proven, that there can be found tuple of words for every word with a minimum Hamming distance of 3. This way we can rule out the usage of an extended version of Hamming Code, since the minimum distance would have bin 4 in this case. So this code has to be a shortened (15,11)-code, since there are 4 parity bits the number of symbols which were reduced by 3 from 11 to 8. This reduces the overall length of the code word to 12. It was most likely done to fit the criteria of easy handling in hardware implementations as discussed above.

Figure 12 shows two code words $X_i$ and $X_j$ of the same code space $X$. These words have a Hamming distance of 3 to each other which means that there are two unused words

lying in between. The difference both words is 3 bits.
If a code word would be received and be placed on position 1 on the line it is more likely that the original word is $X_i$ than $X_j$ since the distance to $X_i$ is 1 and to $X_j$ is 2. It is assumed that the received word has a single bit error and therefore will be corrected to $X_i$. However, if exactly two bit errors occur, it will be placed on position 2. As a result it would be corrected to $X_j$ which is wrong, but we still can see that a defect word has been transmitted. This is displayed by the so called correction circles. Every code word within the red circle would be corrected to the red code word, every word within the blue circle to the blue word.

This means that the code is capable of detecting two bit errors and correcting single bit errors. Three bit errors in $X_i$ would be detected as a valid code word $X_j$.

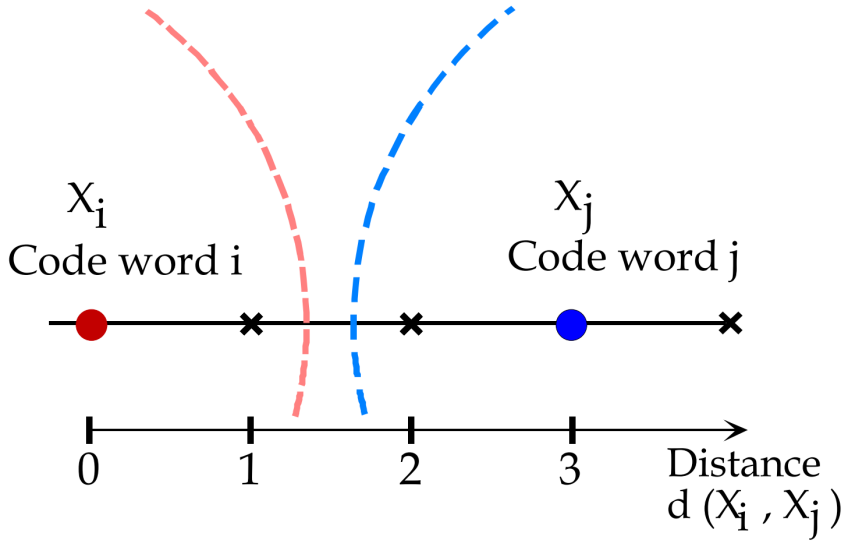


Figure 12: Correction circles for a Hamming Distance of between a tuple of code words $X_i$ and $X_j$, taken from [Tran-Gia(2015)] and modified

From this the possibilities of such errors can be calculated and later be used to compare the reliability of words that were received over different communication channels.
From Section 2.1.3, it can be derived, that BER is $1 * 10^{-7}$ in the worst case, which implies that on average every $1 * 10^{7}$ bit is damaged. And the possibility of an error free transmission with a code word of 12—bit length is:

$$P_{e=1} = (10^{-7})^0 * (1 - 10^{-7})^{12} = 0.9999988 = 99.99988\% \tag{19}$$

The possibility of one and two transmission errors is:

$$P_{e=1} = (10^{-7})^1 * (1 - 10^{-7})^{11} = 9,99 * 10^{-8} = 9,99 * 10^{-10}\% \tag{20}$$

$$P_{e=2} = (10^{-7})^2 * (1 - 10^{-7})^{10} = 9,99 * 10^{-15} = 9,99 * 10^{-17}\% \tag{21}$$

For an error to occur a code word has to be falsified into another one. Since the data bits of the code have been reduced, the code is no longer dense.
From the Matlab code word generator A.1 and the script in A.2 the residual error proba-

bility can be calculated with the following formula (see [Mildenberger and Werner(1998), 6.4.3]):

$$P_R = \sum_{i=d_{min}}^{n} A_i * p^i * (1-p)^{n-i} \tag{22}$$

Where $A$ is the number per bit equal to 1 in a code word, also called weight:

| Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 1 | 0 | 0 | 16 | 39 | 48 | 48 | 48 | 39 | 16 | 0 | 0 | 1 |

Table 3: Count of code words with the weight A

Summing up the probabilities in Equation 22 and starting with the minimum distance $d_{min} = 3$ (as the distance where errors are no longer detectable) result in a residual error probability of:

$$P_R = 1.6 * 10^{-27} \tag{23}$$

For the error correction we can simply calculate the possibility of two or more errors, which results in a different valid code word:

$$P_{e\geq 2} = \sum_{i=k}^{n} \binom{n}{i} * p^i (1-p)^{n-i} \tag{24}$$

$$P_{e\geq 2} = \sum_{i=2}^{12} \binom{12}{i} * (10^{-7})^i (1-10^{-7})^{12-i} \tag{25}$$

$$P_{e\geq 2} = 6.60 * 10^{-13} \tag{26}$$

Where the probability factor reads $p = 10^{-7}$ and the count of bits in the word is $n = 12$. This results in a possibility of $P_{e\geq 2} = 6.60 * 10^{-11}\%$ for a falsely corrected error with a transmission experiencing the worst Bit Error Rate.

This means that it is very reliable to detect errors but correcting errors is more likely to be wrong. However, if a 5 Mbit transmission is assumed, $5 * 1024 * 1024 = 5242880$ bit will be transmitted per second. So in one second $5242880 * 6.60 * 10^{-13} = 3.46 * 10^{-6}$ errors will be corrected to another code word. This implies one uncorrected error appears every 80.27 hours in a laboratory environment without additional external disturbances. Such a single error during the duration of a flight is not very likely and can easily be corrected by mechanisms in higher OSI levels.

## 3.2 Software Architecture

In this chapter an overview of the derived software architecture is given for an understanding how and in which way the data is processed. The realization of different merging modes is shown together with difficulties that were encountered during the process. Furthermore, this chapter shall not only show the design decisions but also give a fundamental understanding of the processing. The most important parts of the code framework are shown in the text.

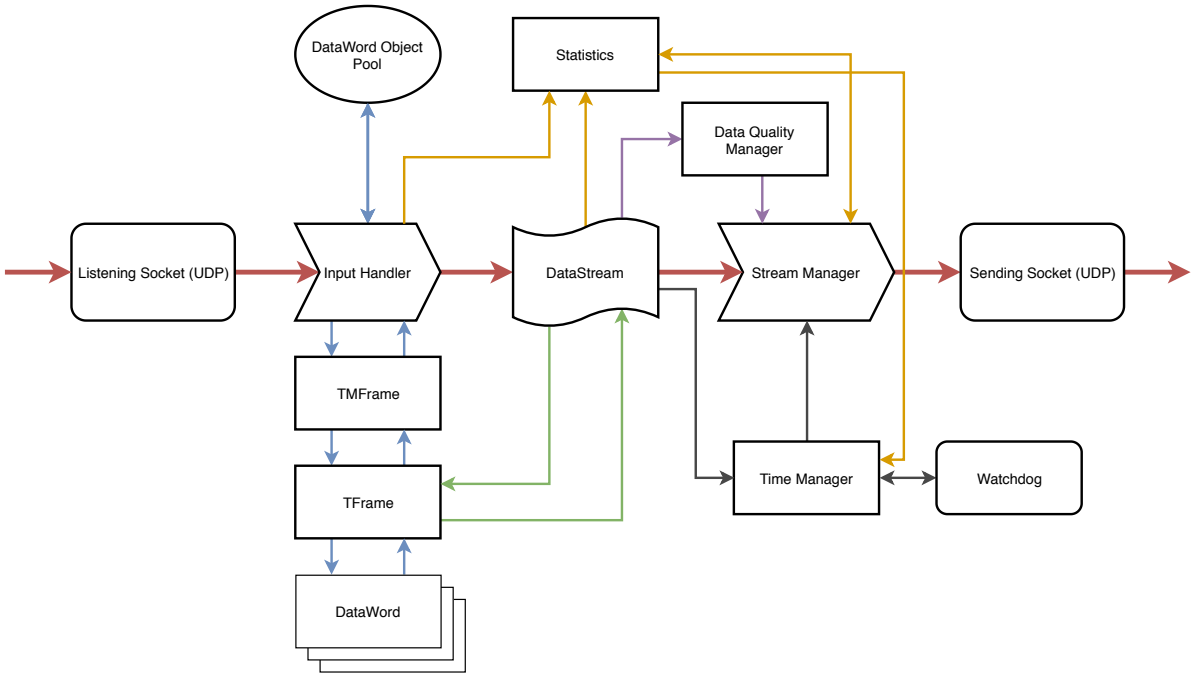


Figure 13: Schematic representation of the data flow inside the software. Red: data flow, blue: DataWord object flow, green: DataWord with filled objects flow, yellow: metadata flow, gray: time data flow

Figure 13 shows a schematic drawing of data paths through the system. The red big arrows indicate the logic path of the data through different processing steps. UDP binary data streams are received simultaneously and spitted in single frames as they are sent from a telemetry station. This is done inside the input handler (see Section 3.2.3). The frames are atomized in their constituent parts. The telemetry frame TMFrame (see Section 3.2.6), the transfer frame TFrame (see Section 3.2.5) and the data in single DataWord objects (see Section 3.2.1). The DataWord objects are pre-generated and collected inside a pool (see Section 3.2.2). During this process the error correction is executed as it was explained in Section 3.1.2 and statistical data is collected for each stream in the Statistics class (see Section3.2.11). In the end, the streams are merged into one inside the StreamMerger class (see Section 3.2.9) and send to at UDP output socket. The merging process is based on information collected in the Statistics class and the DataQualityManger which is responsible for matching the streams against each other (see Section 3.2.8). For this process to stay in certain time limits a TimeManager class is introduced to observe the run-time (see Section 3.2.12).

### 3.2.1 Data Word

A class called DataWord is used to handle code words with additional meta information about whether the word has errors of if it had been corrected. An object of the class is created without data content (for more information refer to the next Section Object Pooling 3.2.2) and the data has to be filled during run-time. The word has to be corrected and during the merging process it has to be provided to the output. Afterwords the object needs to be cleaned so that it can be used again after it has been returned to the object pool.
Since the protocol standard supports endianness, both big and little endian has to be supported in the framework. For this reason there is an abstract class DataWord which contains all functions to provide the methods for the aforementioned tasks as well as two inherited implementations for both bit orders.

The error detection and error correction is done using a lookup table. For each 8—bit word there is a corresponding 4—bit FEC code (see Section 3.1.2). The 4—bit code can be looked up in the table as "static readonly" which is very similar to a constant. The modifier is used since a byte array as a constant can only have the null-type.

**Listing 1: ErrorDetectionTable**

```
internal static readonly byte[] ErrorDetectionTable = new byte[] {
    0x0,0x5,0x6,0x3,0x9,0xc,0xf,0xa,0xa,0xf,
    0xc,0x9,0x3,0x6,0x5,0x0,0x7,0x2,0x1,0x4,
    0xe,0xb,0x8,0xd,0xd,0x8,0xb,0xe,0x4,0x1,
    0x2,0x7,0xb,0xe,0xd,0x8,0x2,0x7,0x4,0x1,
    0x1,0x4,0x7,0x2,0x8,0xd,0xe,0xb,0xc,0x9,
    0xa,0xf,0x5,0x0,0x3,0x6,0x6,0x3,0x0,0x5,
    0xf,0xa,0x9,0xc,0xd,0x8,0xb,0xe,0x4,0x1,
    0x2,0x7,0x7,0x2,0x1,0x4,0xe,0xb,0x8,0xd,
    0xa,0xf,0xc,0x9,0x3,0x6,0x5,0x0,0x0,0x5,
    0x6,0x3,0x9,0xc,0xf,0xa,0x6,0x3,0x0,0x5,
    0xf,0xa,0x9,0xc,0xc,0x9,0xa,0xf,0x5,0x0,
    0x3,0x6,0x1,0x4,0x7,0x2,0x8,0xd,0xe,0xb,
    0xb,0xe,0xd,0x8,0x2,0x7,0x4,0x1,0xe,0xb,
    0x8,0xd,0x7,0x2,0x1,0x4,0x4,0x1,0x2,0x7,
    0xd,0x8,0xb,0xe,0x9,0xc,0xf,0xa,0x0,0x5,
    0x6,0x3,0x3,0x6,0x5,0x0,0xa,0xf,0xc,0x9,
    0x5,0x0,0x3,0x6,0xc,0x9,0xa,0xf,0xf,0xa,
    0x9,0xc,0x6,0x3,0x0,0x5,0x2,0x7,0x4,0x1,
    0xb,0xe,0xd,0x8,0x8,0xd,0xe,0xb,0x1,0x4,
    0x7,0x2,0x3,0x6,0x5,0x0,0xa,0xf,0xc,0x9,
    0x9,0xc,0xf,0xa,0x0,0x5,0x6,0x3,0x4,0x1,
    0x2,0x7,0xd,0x8,0xb,0xe,0xe,0xb,0x8,0xd,
    0x7,0x2,0x1,0x4,0x8,0xd,0xe,0xb,0x1,0x4,
    0x7,0x2,0x2,0x7,0x4,0x1,0xb,0xe,0xd,0x8,
    0xf,0xa,0x9,0xc,0x6,0x3,0x0,0x5,0x5,0x0,
    0x3,0x6,0xc,0x9,0xa,0xf
};
```

As an example the word 13 from Section 3.1.2 is used. The data part is spitted from the

parity bits and used separately. The spiting can simply be done by bit shifting.

$$c_{data} \left[ \begin{array}{cccccccc} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{array} \right] = 0x2C \tag{27}$$

The value $0x2C$ is now looked up in the Table 1. The entry in the array at the location $0x2C$ is $0x8$. This is now compared to the FEC of the code word:

$$c_{fec} \left[ \begin{array}{cccc} 1 & 0 & 0 & 0 \end{array} \right] = 0x8 \tag{28}$$

So the FEC matches the entry in the ErrorDetectionTable which means that the code word is correct. The matching is done by a XOR calculation with the lookup result from the table and the FEC. If the result is greater zero there must be an error and the detection flag will be set inside the object.

**Listing 2: ErrorDetect**

```
1 public bool ErrorDetect()
2 {
3     byte diff = (byte)(ErrorDetectionTable[Data] ^ fecBits);
4     errorDetected = diff > 0 ? true : false;
5     return errorDetected;
6 }
```

This process is also needed for the error correction part. With a XOR-operation the looked up FEC is processed with the received FEC. The result is zero if the word is correct. If it is not zero, there is an error that will be corrected in the next step. With the result of an XOR-operation the error can be looked up from a second array from the ErrorSyndromTable.

**Listing 3: ErrorSyndromTable**

```
1 internal static readonly byte[] ErrorSyndromTable = new byte[] {
2     0x00,0x00,0x00,0x00,0x00,0x01,0x02,0x10,
3     0x00,0x04,0x08,0x20,0x00,0x40,0x80,0x00
4 };
```

With the syndrome the error position in the data can be determined. By using XOR again, the flipped bit is flipped back in its original position (see Listing 4). Remember that the correction might result in the next valid data word instead of the original one if there were two bit errors. The variables are then overwritten with the resulting code word and a boolean is returned if a correction took place. Also the corrected-flag inside the object is set to true to be used in the statistics later.

**Listing 4: ErrorCorrect**

```
1 public bool ErrorCorrect()
2 {
3     byte diff = (byte)(ErrorDetectionTable[Data] ^ fecBits);
4     if (diff == 0) return false;
```

```
5       int correctedWord = (int)((Data ^ ErrorSyndromTable[diff]));
6       data = (byte)((correctedWord >> 4) & 0xFF);
7       fecBits = (byte)(correctedWord & 0x0F);
8       corrected = true;
9       return true;
10  }
```

In case there was no error the run-time is not longer than the ErrorDetect function. This and the derived possibility of receiving the wrong code word from the error correction results in the decision of using only the detection in this implementation of the framework. Furthermore, two DataWord objects are assumed to be equal when the code words are equal. This can later be used for comparison in the merger process. A cleaning function resets all variables to their initial value for the use in the object pooling.

Two classes inherit from the DataWord class. DataWordLE and DataWordBE for little and big endian. They both only have a getter and a setter each to fill the data fields Data and FECBits inside the DataWord object.

For the DataWordLE:

**Listing 5: DataWordLE**

```
1  //Extract data and FEC from little endian DataWord
2  public override void setWord(int word)
3  {
4      Data = (byte)((word >> 4) & 0xff);
5      FECBits = (byte)(word & 0xf);
6  }
7
8  //Build little endian DataWord
9  public int getWord()
10 {
11     return (int)((Data << 4)|FECBits);
12 }
```

And for the DataWordBE:

**Listing 6: DataWordBE**

```
1  //Extract data and FEC from big endian DataWord
2  public override void setWord(int word)
3  {
4      Data = (byte)(word & 0xff);
5      FECBits = (byte)((word >> 8) & 0xf);
6  }
7
8  //Build big endian DataWord
9  public int getWord()
10 {
11     return (int)((FECBits << 8) | Data);
12 }
```

### 3.2.2 Object Pooling

Object pooling is a design pattern that can be used in case if a lot of objects have to be created in a short time or the creation of an object e.g. the procedure in the constructor takes a lot of time to initialize. In such a case objects can be pre-generated to gain a performance boost during the run time. To keep track of the available objects a pool is generated where objects can be taken from for further usage or placed back into the pool in case the object is no longer needed. This pick and place is much more efficient than creating a new object every time one is needed and even faster if objects do not have to be copied but reached around by a reference. There is, however, the downside that data is not available at the point of creation and data has to be filled into the empty objects after picking it up.
In this thesis object pooling was found to be a good way of handling code words also known as DataWords objects since they have to be processed very rapidly.

The class is called DataWordObjectPool and contains all functions necessary to keep track of the DataWord objects inside the pool and handle events like adding, returning and creation of new objects as well as the pre-allocation during startup.

As can be seen in Listing 7 the constructor of the class takes the object count as argument and uses function PreAllocateObjects with a parallel For-function to create new objects using the CreateNewDataWord function. A new object will be created and placed in the ConcurrentBag. A ConcurrentBag is an unstructured container in the C# concurrent collections which provide thread-safe collection classes that can easily be used in multitasking or multi threading environments. The automatic locking features of the collection allow safe access on the bag for multiple accesses at a time.

**Listing 7: DataWordObjectPool**

```
1  private ConcurrentBag<DataWord> _bag;
2  //New DataWordPool. Create new bag object, call pre-allocation
3  public DataWordObjectPool(int objectcount)
4  {
5      _bag = new ConcurrentBag<DataWord>();
6      PreAllocateObjects(objectcount);
7  }
8
9  //Pre-create DataWord objects in parallel tasks.
10 public void PreAllocateObjects(int count)
11 {
12     Parallel.For(0, count, ctr => { CreateNewDataWord(); });
13 }
```

A Get and a Release function shown in Listing 8 provides the previously described necessary access functionalities for getting empty and returning used objects to the bag for further use. In case the bag is empty because all objects have been taken and are in the process new objects are created and placed in the bag to provide locking the process. However, this should not happen since the time an object can be used is limited by the design. So only a sufficient pre-allocation has to be done to prevent time-consuming creation of objects.

**Listing 8: DataWordObjectPool**

```
1 //Get empty DataWord from the bag, if there are no more, create one
2 public DataWord Get()
3 {
4     DataWord word;
5     if (_bag.TryTake(out word)) return word;
6     return CreateNewDataWord();
7 }
8
9 // Release used word into the bag, clear data before
10 public void Release(DataWord word)
11 {
12    word.Clear();
13    _bag.Add(word);
14 }
```

This simple but time saving construct can now be used by the InputHandler class for reaching the objects down to the frames where they are used for data validation, error correction and storing additional information.

### 3.2.3 Input Handler

The main purpose of the Input Handler Class is to provide the functionality to listen on a network socked and handle the pre-treatment of the received data like header validation. There is an instance of the InputHandler for each incoming data stream. Each data stream needs to have an unique socket where the data can arrive. For a non-blocking processing the handler is launched in a separate thread so that it does not interfere with other parts of the software. This way different data sources can send data to the framework as they would send the data to the operator. This ensures that the system is optional insight of pipeline and can be placed and taken from the data path without the need of additional information apart from changing IP and port. The transmission is done as a UDP Ethernet data stream. This means the InputHandler might listen for data even if there is nothing connected.

In a first step it connects to the socket which is handed over by the main class. If an exception occurs during the connection e.g. a socket is blocked by a different application the system will try to reconnect until the connection can be successfully established. Data reaching the UDP Port will be buffered by the operating system's kernel outside the program until the data gets collected by the framework. This external buffer needs to be periodically read and emptied. The collection is done by the receiving function RcvData, which moves all available data in the external buffer to a linked byte list. A Linked List consists of single elements linked to a pre- and a successor object of the same type. By this constellation the size is logically unlimited, the only limitation is the working memory.

The runner, which contains the main loop of the class, constantly iterates over the list while new items are added. Bytes are moved into a temporary buffer until a value matches with the first byte of the telemetry frames sync word (preamble). See Table 2 and line 12 of Listing 8. Then the next three bytes are tested against the sync word. In

case they do not mach, no header has been found and the byte is assumed to be part of the data block of a frame. A frame's header can only be verified after the frame has fully been received. For this reason the temporary buffer is introduced. With an assumed correct stream it always contains valid frame data except from the first moment when the software is switched on. At this point most probably only fractional parts of the last frame transmitted are received. The temporary buffer is filled until a new sync word is found (lines 13 and 19 of Listing 8). The buffer contains now data that has to be verified if it is a valid frame, corrupted data or only fractional parts. For this a TMFrame object is filled with the data and the validation is executed. See Section 3.2.6 for more information on the validation process. If the process does not raise an error, a valid frame has been found. It is now added to the DataStream as described in Section 3.2.7 to be used inside the merging process. If an error occurs, the procedure starts again and the data in the temporary buffer is skipped since it seemed to be unrecoverable distorted. After a frame has been validated or no more bytes are available for processing the RevData function is called to get new data from the external buffer. If the runner is stopped the last frame will be processed and the thread comes to an end.

This process is also important since the length of a frame is not specified. By using the header information of a TMFrame discussed in Section 2.2.8 it would be possible to send 65536 byte long frames. Since the field containing the frame length does not have a CRC or FEC it cannot safely be used for predicting the data collection time.

**Listing 9: InputHandler**

```
1  while (run)
2  {
3      //Fill rawData with data from Network Socket
4      RcvData();
5
6      //Test if data received
7      if (rawData.First == null)
8      {
9          //If there is no data, collect more and come back afterwords.
10         continue;
11     }
12
13     //Add bytes to frame buffer until rawData is empty and new data
           has been received and a byte matches the first byte of the
           preamble.
14     while (rawData.Count > 0 && rawData.First.Value != preamble[0])
15     {
16         frame.Add(rawData.First.Value);
17         rawData.RemoveFirst();
18     }
19
20     //If there are 4 or more bytes and the first one matches the
           preamble, check if this is a header. If not add it to frame
           buffer.
21     if (rawData.Count >= 4 && rawData.First.Value == preamble[0])
22     {
```

```
23
24          //Check if next three bytes match the preamble. First byte
                has already been checked.
25          if ((rawData.First.Next.Value == preamble[1]) &
26              (rawData.First.Next.Next.Value == preamble[2]) &
27              (rawData.First.Next.Next.Next.Value == preamble[3]))
28              {
29
30                  //New header found!
31                  //If frame buffer is empty, obviously there cannot be
                        a frame. (e.g. on program start)
32                  if (frame.Count == 0)
33                  {
34                      frame.Add(rawData.First.Value);
35                      rawData.RemoveFirst();
36                      continue;
37                  }
38
39                  //Add old frame buffer to frame object (validity
                        check will be executed), if it fails frame will
                        not be added to stream
40                  try
41                  {
42                      //Fill TMFrame object and check if frame is valid
                            . Throws error if not.
43                      tmframe.SetRawBytes(frame.ToArray());
44
45                      // Update statistics about corrected errors and
                            other meta data.
46                      stats.LastFrameErrors = tmframe.
                            ErrorsCorrectedCnt;
47                      stats.LastFrameSize = tmframe.Length;
48                      stats.LastFrameQuality = tmframe.Quality;
49                      //Add the new found frame to stream DataStream
50                      dataStream.Add(tmframe);
51                   }
52                   catch
53                   {
54                       //Frame validation error. Nothing can be done to
                             repair it, go on and
55                       //search for next frame header.
56                       frame.Clear();
57                       frame.Add(rawData.First.Value);
58                       rawData.RemoveFirst();
59                       continue;
60                   }
61
62                   frame.Clear();
63                   //Since new frame has been found and last frame has
                         been added successfully,
64                   //putting all new bytes into frame buffer
65                   frame.Add(rawData.First.Value);
66                   rawData.RemoveFirst();
67               }
68               else
```

```
69                {
70
71                    //This is not a header, so add bytes to frame buffer
                          .
72                    frame.Add(rawData.First.Value);
73                    rawData.RemoveFirst();
74                }
```

As explained in the following Sections e.g. DataStream 3.2.7 Concurrent Queues are used to transport the objects created inside the Input Handler. This ensures the safe usage for the multi threading environment by internal blocking operations.

### 3.2.4 Frame

To cope with different frame types an abstract class Frame is introduced. A function SetRawBytes 10 is implemented which calls the validation method that is overwritten by the inherited classes. The function is used to set received bytes to the empty frame object as it can be seen in the InputHandler class in Section 3.2.3. When bytes are set, the validation is executed in a dedicated task to prevent a locking of the thread. The task will return a result containing a byte consisting of 8 flags that can be used to determine the occurred error. If the result is greater than zero the error does not match the requirements and the frame has to be dropped. The validation is dependent of the frame type and can of course be implemented individually. The error flags are reported inside an argument exception to the parent instance.

**Listing 10: Frame SetRawBytes**

```
1  public void SetRawBytes(byte[] RawBytes)
2  {
3      this.rawbytes = RawBytes;
4
5      try
6      {
7          byte validation = 0;
8          //Start a new validation task
9          ValTask.Start();
10         //Wait for the result
11         ValTask.Wait();
12         //Save the result for reporting and stream three and so on.
13         validation = ValTask.Result;
14
15         //If result is not equal zero there must have been an error.
               Report it
16         if (validation > 0)
17         {
18             System.Console.WriteLine(string.Format("Header not
                   matching design: {0} -> Argument Exception", (int)
                   validation));
19             throw new ArgumentException(string.Format("Header not
                   matching design: {0}", (int)validation));
20         }
21     }
```

```
22      catch (Exception e)
23      {
24          //Catch and report if there were errors thrown within the
                validation process
25          Console.WriteLine(e.ToString());
26      }
27  }
```

It also provides functionalities to compare two frame types. Two frames are equal when their bytes from the body match. The TMFrame header cannot be compared to each other since it could have a different version with additional information as well as another quality index since it traveled on a different data path. This overwrites the standard Equal function of C#. But there is also a second method shown in Listing 11 which allows comparing similar frames and get the percentage of congruence between the byte representation of these frames. Even frames with a different length can be compared.

**Listing 11: Frame EqualPercentage**

```
1  public float EqualPercentage(object obj)
2  {
3      //Check if object to be compared is of the same type
4      if (obj is Frame){
5          var frame = obj as Frame;
6          int count = 0;
7          byte[] bytes = frame.GetByteRepresentation();
8
9          //Lock into each byte
10         for (int i = 0; i < Length; i++)
11         {
12             try
13             {
14                 //Count matching bytes
15                 if (rawbytes[i] != bytes[i])
16                 {
17                     count += 1;
18                 }
19             }
20             catch
21             {
22                 continue;
23             }
24         }
25         //Calculate the ratio of matching bytes (range 0 to 1)
26         return (Length - count) / Length;
27     }else{
28         return 0.0f;
29     }
30 }
```

The Length method of the object is implemented as the length of the frame in its byte representation. The GetHashCode function is as well overwritten with a hash code over the raw byte representation of the whole frame. The object carries the ID value that is

assumed to exist in the used protocol. If it does not exist it has to be derived from e.g. the input counter or in a different manner, since the matching uses this value to compere two frames in a first fast step. However, the TFrame contains a counter that can be used for our purpose. The following two classes TFrame in Section 3.2.5 and TMFrame in Section 3.2.6 are derived from this. Frames are compared and transported through the whole process, while carrying all the information including the DataWords with their additional correction information.

### 3.2.5 TFrame

The TFrame class is used to implement the representation of a transfer data frame as it has been described in Section 2.2.7. The class inherits from the abstract Frame-class shown in Section 3.2.4. It keeps track of the DataWord-objects that came with the data header together with the header validation and of course the header data itself.

The validation functions implement the abstract validation method of the parent class. In a first step it is checked if the DataWordObjectPool is already created. Without it, there are no DataWords that can be filled. This check is necessary in case there is something wrong with the initialization. Then the header is checked whether it fits the pre-programmed preamble as shown in Table 1. Now the data collection process starts. Since a word is 12—bit long, it does not fit into a byte sized variable. For this reason a whole Integer has to be used. A For-loop iterates over the previous set rawbytes-array until it reaches the end of the data. Always three bytes are covered in one cycle. The first byte is data of the first word, the third byte is data of the second word and the byte in between consists of four bytes from the first and four bytes of the second word. This data has to be shifted together and filled into the according integer values.
An empty DataWord is taken from the DataWordObjectPool (see Section 3.2.2). Depending on the endianness it can now be filled with the data- and FEC-bits. As mentioned before in Section 3.2.1, an error correction is performed to ensure a valid word. A counter tracks the errors that where found in this frame. As it can be seen in Table 1 there is the noteworthiness that the frame counter of the TFrame is written into the first DataWord of the body, even so it does belong to the header. Due to this fact that the counter is needed in the later process of matching the data of the first word is written to a variable called ID.

**Listing 12: TFrame**

```
1 internal override byte Validation()
2 {
3     if (dataWordPool == null) throw new Exception("Wordpool not
          initialized. Use DataHandler.TFrame.SetDataWordObjectPool()
          beforehand");
4     int firstword = 0;
5     int secondword = 0;
6
7     //Check if preamble matches. This did not take place before or
          the TMFrame
8     if (rawbytes[0] != preamble[0]) return 0xFF;
9     if (rawbytes[1] != preamble[1]) return 0xFF;
```

```
10      if (rawbytes[2] != preamble[2]) return 0xFF;
11      int count = 0;
12      wordList.Clear();
13
14      //Separate data of the body. This contains the DataWord objects
15      for (int i = 3; i < rawbytes.Length; i += 3)
16      {
17          try
18          {
19              //Read always 3 bytes/2 DataWords at once
20              firstword = ((rawbytes[i] & 0xFF) << 4);
21              firstword |= ((rawbytes[i + 1] >> 4) & 0x0F);
22              secondword = ((rawbytes[i + 1] & 0x0F) << 8);
23              secondword |= ((rawbytes[i + 2] & 0xFF));
24          }
25          catch
26          {
27              //Throw an exception in case the frame is not complete
28              throw new IndexOutOfRangeException("RawBytes of Frame to
                    short. Last Word omitted");
29          }
30
31          //Take DataWord objects from the objects pool
32          dw1 = dataWordPool.Get();
33          dw2 = dataWordPool.Get();
34
35          //Check if little or big endian words are used
36          if (dw1 is DataWordLE)
37          {
38              dw1.Data = (byte)((firstword >> 4) & 0xFF);
39              dw1.FECBits = (byte)(firstword & 0x0F);
40              dw2.Data = (byte)((secondword >> 4) & 0xFF);
41              dw2.FECBits = (byte)(secondword & 0x0F);
42          }
43          else
44          {
45              dw1.FECBits = (byte)((firstword >> 8) & 0x0F);
46              dw1.Data = (byte)(firstword & 0xFF);
47              dw1.FECBits = (byte)((secondword >> 8) & 0x0F);
48              dw1.Data = (byte)(secondword & 0xFF);
49          }
50
51          //Call the error correction and save the result for the use
                in the statistics class
52          ErrorsCorrectedCnt += dw1.ErrorCorrect() ? 1 : 0;
53          ErrorsCorrectedCnt += dw2.ErrorCorrect() ? 1 : 0;
54
55          //Add words to the wordList containing all the DataWord
                objects
56          wordList.Add(dw1);
57          wordList.Add(dw2);
58
59          //The first DataWord of a TFrame is belonging to the header.
                It contains the frames ID
60          if (i == 3)
```

```
61          {
62              count = dw1.Data;
63              ID = count;
64          }
65      }
66      return 0x00;
67  }
```

### 3.2.6 TMFrame

A second frame, the telemetry frame (TMFrame), is wrapped around a transport frame to add additional information gained by the telemetry station. As described in Section 2.2.8 it contains quality information as well as the frame size, version of the header and length of the header for dynamic header extension. With the version and offset additional fields can be added. Here discussed is the initial version number 0.
Since we observe the data directly by extracting the frames, verifying the headers and doing error correction on the body we know the absolute quality of the data. This means the quality field, that is used for DQE otherwise can be ignored, it is even set to zero. In the end there will be an output stream from several input streams and the quality index value will be set again based on measurements that were done during the merging and the previous correction process.

The TMFrame class also implements the abstract functions from the parent Frame-class as described in Section 3.2.4. So it differs from TFrame mainly by the implementation of the validation method. In a first step the byte values of the fields are derived by shifting the appropriate bits together. The result is then converted to fit the data types. Secondly the content of the length field is matched against the actual length of the raw representation of the frame in bytes and the offset is tested to fit the size of a version 0 header. If the header does not match the criteria the frame has to be dropped since the data cannot be evaluated anymore.
After the validation the content of the TMFrame data block is set as raw input to the provided TFrame object where a second evaluation is done as described in Section 3.2.5.

**Listing 13: TMFrame**

```
1  internal override byte Validation()
2  {
3      if (_tframe == null) throw new Exception("TFrame not initialized.
           Use DataHandler.TMFrame.SetTFrame() beforehand");
4      byte validationbools = 0;
5
6      //Check if frame is longer than header's minimum size
7      if (rawbytes.Length < 16)
8      {
9          validationbools |= 0x11;
10     }
11     else {
12         //Cast values
13         _sync = (uint)((rawbytes[0] << 24) | (rawbytes[1] << 16) | (
               rawbytes[2] << 8) | (rawbytes[3] << 0));
```

```
14          _length = (short)((rawbytes[4] << 8) | (rawbytes[5] << 0));
15          _version = (byte)(rawbytes[6] << 0);
16          _offset = (byte)(rawbytes[7] << 0);
17      }
18
19      //Check minimum requirements
20      if (Length != Length) validationbools |= 0b0000_0001;
21      if (Version != 1) validationbools |= 0b0000_0010;
22      if (Offset != (2 * sizeof(int) + sizeof(double)))
23      {
24          validationbools |= 0b0000_0100;
25      }
26
27      //Return result if error occurred
28      if (validationbools != 0b0000_00000)
29      {
30          return validationbools;
31      }
32
33      //If no error, pass body to TFrame
34      byte[] words = new byte[Length - Offset];
35      for (int i = _offset; i < rawbytes.Length; i++)
36      {
37          words[i - _offset] = rawbytes[i];
38      }
39
40      _tframe.SetRawBytes(words);
41      ErrorsCorrectedCnt = _tframe.ErrorsCorrectedCnt;
42
43      return validationbools;
44  }
```

As can be seen the header is not tested, this is due to the fact that the header must have matched the preamble in the InputHandler already.

### 3.2.7 DataStream

The data transportation between the InputHandler and the StreamMerger is handled by the DataStream class. However, it is not only used for transmission, this could have been done with a concurrent list more easily. It also carries metadata like statistics to the DataQualityManager and the TimeManager which can than access the data periodically without interrupting the transmission process. The class implements the IList-interface for Frame data types. Internally it consists of a concurrent queue on the input side and a list on the output side. The splitting is done to prevent locking on the input and therefore an unnecessary block for the InputHandler during the data appending process while the StreamMerger (see Section 3.2.9) can operate on the stream simultaneously. Since the access by the merger is rather long compared to rapid appending process and the appending has to be done very frequently this is a sufficient workaround of the blocking problem. Each stream and input handler has its own unique instance of the class.

To move the Frame-objects from the input to the output there is a runner that periodically locks the output side and moves objects from the input. The time interval is

managed by the Timer-class of C# that executes the AddQueueToList function periodically. The runner is started from the main class on program startup. It takes the duration between two executions as an argument. The AddQueueToList checks if there is data to copy before acquiring the lock. The hole process is there to ensure that the lock time is reduced to a minimum and the merger can work with a minimum interruption time.

**Listing 14: DataStream Runner**

```
public void Runner(int ms)
{
    //Start timer for AddQueueToList
    Timer timer = new Timer(ms);
    timer.Elapsed += AddQueueToList;
    timer.Enabled = run;
}

private void AddQueueToList(Object source, ElapsedEventArgs e)
{
    Frame frame;

    //Check if there is data in the input queue
    if (inputQueue != null)
    {
        //Lock output, copy input to output
        lock (_lock)
        {
            while (inputQueue.TryDequeue(out frame))
            {
                frameList.Add(frame);

            }
        }
    }
}
```

### 3.2.8 DataQualityManager

Several duties are maintained by the DataQualityManger. It accesses the DataStream objects of each stream to inspect the data and provides information to the merger about the stream synchronization. Incoming streams arrive with a different delay which is dependent on the distance between rocket and ground station but also on the distance between ground station and this Stream Merger. Additional time is added by the Ethernet transmission e.g. by switches, routers or the scheduler of the kernel of the PC where this software is running on. The result is an asynchronous reception of the frames. Additionally the time information is lost in the receiving process since the packages are stored in the DataStream without detailed timing information. However, it is necessary to synchronize the streams before frames can be compared. The synchronization itself can be achieved by determining the offset in between. The offset is not calculated in time, but in frame count which is equivalent to the position inside the DataStream. Therefore, the offset is always an integer value.

Since it is not intended to store the offsets between very stream, the stream that connected first to the software is used as reference stream. Its position number is always 1, which does not mean that it is the first stream. The others streams can have a negative offset which means the stream is earlier or positive which means it has a higher latency.

The process is executed periodically since it is not expected that the latency will not change a lot between two frames. The worst case execution time for finding a corresponding frame in a second stream is $O(n)$. It is a good approach to save calculation time by not determining the difference periodically.

In a first step the runner has to find the first stream in a dictionary of streams that contains data. A dictionary entry consists of a DataStream object as value and the port number as key. The key value is used to identify a stream. Since this is an UDP connection a stream exists when it is created, not when a connection to the UDP socket is made. Also with an outage of a ground station a stream would be empty. If the check is done, streams containing data are counted. A single stream containing data results in a jump to the next cycle of probing, since a single stream cannot be merged. In this case only error correction would take place and all the data from the input will be sent to the output.

In case there are two or more streams with available data they have to be matched against each other. Always two streams are compared at a time. To iterate over the dictionary an enumerator (dsEnumberator) is used.

**Listing 15: DataQualityManager Runner**

```
1  //Search for next non-empty stream
2  while (dsEnumberator.Current.Value ==null)
3  {
4      emptycount += 1;
5      dsEnumberator.MoveNext();
6  }
7
8  //Check if non or only one stream has data, there is nothing to merge
9  if (emptycount >= dataStreamDict.Count-1)
10 {
11     continue;
12 }
13
14 stream1 = dsEnumberator.Current.Value;
15
16 //Reset of an enumerator not possible by design of C#, so re-
       initialize enumerator to start from scratch
17 dsEnumberator = dataStreamDict.GetEnumerator();
18
19 //Skip first entry, since we already have it saved to stream1
20 dsEnumberator.MoveNext();
21
22 //Match all streams against each other and save the resulting offset
       in the offsets array
23 while (dsEnumberator.MoveNext())
24 {
25     stream2 = dsEnumberator.Current.Value;
26     try
27     {
```

```
28            offsets[streamcounter] = MatchStreams(stream1, stream2);
29        }
30        catch
31        {
32            continue;
33        }
34        stream1 = stream2;
35    }
```

After the processing, the "offsets" array contains the offset between stream pairs in rising order. This means that the first entry is the offset of stream two to stream one, the second entry the offset between stream three to stream two and so on. As explained before it is important to know the offset of all streams regarding the first entry. This has to be solved by an equation system. However, a matrix can be derived from the problem to avoid the necessity to solve this linear equation system during run-time like using e.g. Cramer's Rule. The matrix describing the absolute positions looks like:

$$x' = \begin{bmatrix} 1 & 0 & 0 & 0 & \\ 1 & -1 & 0 & 0 & \dots \\ 0 & 1 & -1 & 0 & \\ 0 & 0 & 1 & -1 & \\ & & \vdots & & \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \end{bmatrix} \tag{29}$$

Which translates into an equation system like following:

$$\begin{aligned} x'_1 &= x_1 \\ x'_2 &= (-x_2) + x_1 \\ x'_3 &= (-x_3) + (-x_2) + x_1 \\ x'_4 &= (-x_4) + (-x_3) + (-x_2) + x_1 \\ x'_n &= (-x_n) + (-x_{n-1}) + (-x_{n-2}) + \ldots + x_1 \end{aligned} \tag{30}$$

That can be reduced down to:

$$\begin{aligned} x'_1 &= x_1 \\ x'_2 &= (-x_2) + x'_1 \\ x'_3 &= (-x_3) + x'_2 \\ x'_4 &= (-x_4) + x'_3 \\ x'_n &= (-x_n) + x'_{n-1} \end{aligned} \tag{31}$$

Where $x$ is the previous calculated value from Listing 15 and $x'$ is the absolute position to $x_1$. The start value for the equation system is defined as stream one at position one ($x_1 = 1$). With this solution we can derive an equation that calculates the absolute position of each stream in relation to stream number one from our offsets-array input.

It can be expressed with the term:

$$x'_N = \begin{cases} 1, & N = 1 \\ 1 + \sum_{i=2}^{N}(-x_i), & N > 1 \end{cases} \tag{32}$$

This term now can easily be implemented with a For-loop to calculate the corresponding $x'$ for each stream.

**Listing 16: DataQualityManager AbsolutStreamPosition**

```
foreach (int StreamID in dataStreamDict.Keys)
{
    int sum = 0;

    //Do not execute for x_1
    if (!first) {
        cnt += 1;

        //calculate x' for this stream
        for (int j = 1; j <= cnt; j++)
        {
            sum += (-1) * offsets[j];
        }
        sum += 1;

        //Add an entry to the AbsStreamPosition dictionary
        AbsStreamPosition.Add(StreamID, sum);
    }else{
        //Add the first stream with x'=1 to the AbsStreamPosition
           Dictionary
        AbsStreamPosition.Add(StreamID, 1);
        first = false ;
    }

}
```

When looking at the matching process to find the dependent stream positions between two streams, there is the assumption to be made that the package loss has the worst possible constellation. The frame counter (ID) is limited to one byte, this way it can only count 256 times until it overflows. This means that when two frames are compared and have the same ID they are not necessarily identical frames. In case the time shift in the transmission from the rocket to the merger is longer than 256—times it takes to transmit a single frame, a frame with the right ID is in the right position but still it is a frame from the previous cycle. This can be seen in Figure 14. In this schematic it is assumed that the ID-counter has a maximum of 5 instead of 256 for displaying purposes. Stream one is the reference stream which an offset should be calculated to. Stream two is a stream without frame loss but has a shift of five times the length of a frame. The green frame is the one that should be located in stream two. At the assumed position is a frame with the right ID but since the counter does not produce unique IDs within a flight it is still the wrong package. So a byte-by-byte comparison has to be done to find out if it really does match. This has to be done every time a matching frame is found to ensure the result.
At startup of the software, it is unknown what is the stream with the lowest latency. So it might be that it is not the previous cycle but a cycle earlier in case the stream that is compared has a lower latency. So the shift can be n-times in both directions.
In the absolute worst case every 255th frame is lost, this would mean the IDs in the buffer

have a descending order even, so they are in the right place. This means that it is not possible to assume any kind of order when only looking at the IDs and therefore no binary search algorithm can be used. On top of that, frame drops can occur any time so a jump of 256 frame into the past or future is not an applicable method to find a stream offset, since it can not be assumed that there is not a frame with the frame ID in-between the 256 frames. This scenario can also be found in Figure 14. Stream number three has three frames with ID 5 within a row of five frames. If the first package does not match, a jump to a frame five fields away might be not the next frame in the row. Stream 3 also has the lowest latency in this example.
Altogether this explains why no sorting can be assumed and therefore a binary search algorithm is not suitable for the correlation process.

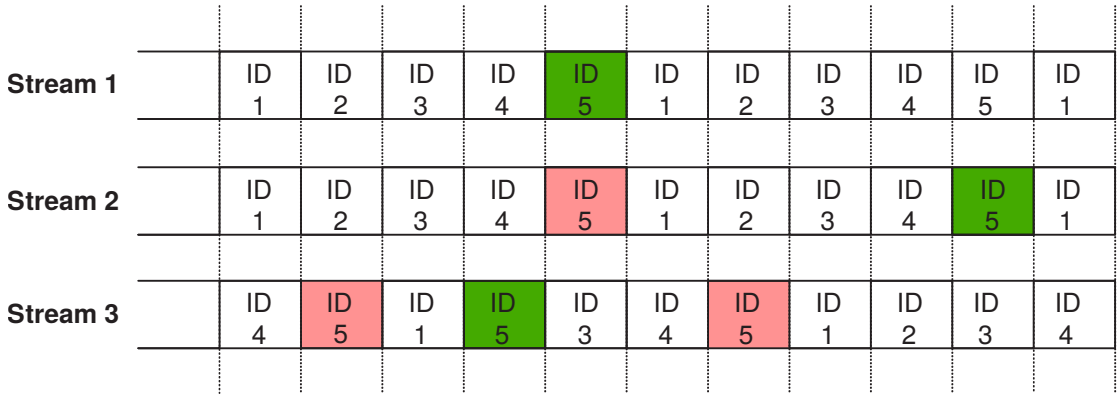


Figure 14: Three streams with a maximum ID count of 5. Green frames are matching, red frames do not match. The second stream is shifted by one cycle count compared to stream one. The third stream has frame loss.

The only solution to this problem without previous knowledge is, to search the first frame of one stream in the second stream and if it cannot be found, stream two has to have a lower latency than stream one. Then frame one of stream two has to be searched in stream one. The worst case run-time is $O(n^2)$ if the latency of one stream is so high, that they do not share any frames in the buffer. However, since the latency shift will not change rapidly it is okay to run the matching with a lower cycle rate and in a separate task. If there is a frame drop and the frame is only shifted by some frames this is handled in the merger class which also reports the offset back to the DataQualityManager.

**Listing 17: DataQualityManager MatchStreams**

```
1 private int MatchStreams(DataStream stream1, DataStream stream2)
2 {
3     //Break in case streams do not have data
4     if(stream1.Count==0 || stream2.Count == 0)
5     {
6         throw new IndexOutOfRangeException();
7     }
8     int countStream1 = 1;
9     int countStream2 = 1;
```

```
10      Frame firstFrame = stream1.First;
11      float congruence = 0;
12
13      //Try to find first frame of first stream in second stream
14      while (countStream2 <= stream2.Count)
15      {
16          Frame secondFrame = FindFrameWithID(stream2, countStream2-1,
                firstFrame.ID);
17          congruence = firstFrame.EqualPercentage(secondFrame);
18
19          //Check if frames are matching with more than 98\%
20          if (congruence > 0.98)
21          {
22              return countStream2;
23          }
24          else
25          {
26
27              //If they do not match search further
28              if(countStream2 == stream1.Count)
29              {
30                  countStream2 = 1;
31                  countStream1 += 1;
32                  firstFrame = stream1[countStream1];
33              }
34              countStream2 += 1;
35          }
36      }
37
38      //Reset conditions for reverse search
39      countStream1 = 1;
40      countStream2 = 1;
41      firstFrame = stream2.First;
42      congruence = 0;
43
44      //Try to find first frame of second stream in first stream
45      while (countStream1 <= stream1.Count)
46      {
47          Frame secondFrame = FindFrameWithID(stream1, countStream1 - 1
                , firstFrame.ID);
48          congruence = firstFrame.EqualPercentage(secondFrame);
49
50          //Check if frames are matching with more than 98\%
51          if (congruence > 0.98)
52          {
53              return -1*countStream1;
54          }
55          else
56          {
57
58              //If they do not match search further
59              if (countStream1 == stream2.Count)
60              {
61                  countStream1 = 1;
62                  countStream2 += 1;
```

```
63                      firstFrame = stream2[countStream1];
64                  }
65                  countStream1 += 1;
66              }
67          }
68          return 0;
69      }
```

As can be seen in the code, two frames match if they have a congruence of 98%. This means matching frames can even be found if errors in the correction occurred. This can then be corrected in the majority vote of the later described merging process. In case a match goes wrong, which is likely for empty frames, this can be corrected in the merging process as well.

### 3.2.9 StreamMerger

For now, it was discussed what had to be done to prepare the data for the merging process. With the StreamMerger class it will now be explained how the actual merging takes place. This is done in several steps which depend on the input data. Like other resource demanding classes the StreamMerger also runs in a dedicated thread. There is only a single instance of this class that takes a dictionary with the DataStream objects and a dictionary with the stream positions calculated in the DataQualityManager as reference parameters. The runner itself is kept very short and is running constantly without a pre-set cycle interval o timer. The Statistic object explained in Section 3.2.11 holds a StopWatch object additionally to the collected metadata of each stream. As every input stream, the output stream also has a particular statistic object to collect metadata like the processing time. The first step in the runner-function is to restart the stopwatch on the output stream-statistics. This is used for the watchdog that will later be explained in the TimeManager Section 3.2.12 to keep the operation time of the merging process within certain boundaries. Then the current quality index of each stream will be calculated. The order in which the streams are looked at is dependent on its quality index. If majority is found, streams with a low index will be skipped. This process is based on various criteria and will be explained later. For the merging it is necessary to always have a minimum amount of data inside the buffer. For this reason the TestSufficientData function is used. The offset between the fastest and slowest stream is calculated. At least 2/3 of the streams must have that amount of frames inside the buffer otherwise the runners loop is skipped for a cycle. To prevent a deadlock it can only be skipped five times before frames are processed again. Then the merger-function is called, which returns the next valid telemetry frame. The exact procedure for this is explained below. Statistics of this frame will be added to the output statistics and a function named CleanupProcessedFrames is called that removes processed frames from all streams. The frame is than handed over to a UDP output socket on default port 10000. With this the runner is completed and starts over again.

The whole process can be interrupted at any time and for this reason includes an interrupt handler. This works together with the aforementioned TimeManager to stay within the soft real-time boundaries.

**Listing 18: DataMerger Runner**

```
while (run)
{
    try
    {
        //Restart Stopwatch for the use with TimeManager
        outputStats.RestartFrameTicks();

        //Calculate quality index for each stream
        StreamQuality = calcQualityIndex();

        //Test if sufficient data is inside the buffers.
        if (TestSufficientData() == false & boundary < 5)
        {
            boundary += 1;
            continue;
        }
        boundary = 0;

        try
        {
            //Do the merging for the most recent frame
            outframe = (TMFrame)merger();

            //Save the statistics
            outputStats.LastFrameErrors = outframe.ErrorsCorrectedCnt
                ;
            outputStats.LastFrameSize = outframe.Length;
            outputStats.LastFrameQuality = outframe.Quality;

            //Cleanup
            CleanupProcessedFrames();
        }
        catch
        {
            //Insufficient data
            System.Console.WriteLine("No Input for Merger");
            continue;
        }

        //Send matched stream to the output
        try
        {
            byte[] outframebytes = outframe.GetByteRepresentation();
            UDPwriter.Send(outframebytes, outframebytes.Length,
                RemoteIpEndPoint);
        }
        catch
        {
            //Notify if there was an error during sending process
            System.Console.WriteLine("Could not send data to output
                socket");
            continue;
        }

```

```
52      }
53      catch (ThreadInterruptedException e)
54      {
55          //Handle interruptions
56          InterruptHandler();
57      }
58  }
```

The mentioned functions will now be explained in more detail. The CalcQualityIndex function is one of the core methods in the merging process. An index is calculated for each DataStream and will be used to rank the streams by quality for the next matching process. The index returned is a float with a maximum of 1. An index of 1 means that the stream is in a perfect condition without any errors occurred in the close past and no errors predicted for the future. The future quality of the stream is predicted by looking at the errors of the last 20 frames and by extrapolating those error counts. The process does not only include the average but also a trend. This extrapolation is done by a separate function simply called Extrapolation. The algorithm used for the extrapolation is called double exponential smoothing and is explained in Section 3.2.10.

Extrapolation is called with $\alpha = 0.8$ and $y = 0.2$ so that the smoothing and the trend enter the result with 80% and 20% respectively. With the predicted error the predicted quality index can be calculated. It is the quotient of the predicted errors and the length of the frame subtracted from 1. This way a prediction of zero errors results in a predicted quality index of 1. The average of the history of 20 error counts is calculated, divided by the length and subtracted from 1. With a quality index for the last values and an index for the future now both can be combined to be used for prioritizing the streams.
This is done calculating: 1 divided by the average of both indices which also leads to a maximum quality of 1 in case of no errors in the past and no predicted error in the future. This way a short burst error will not reduce the confidence significantly.

**Listing 19: DataMerger CalcQualityIndex**

```
1  public Dictionary<int, float> CalcQualityIndex()
2  {
3      StreamQuality = new Dictionary<int, float>();
4      float[] extrap = new float[15];
5      float prediction = 0.0f;
6      int length = 0;
7      float predQuality;
8      float lastQuality;
9      float sum = 0;
10     float qindex = 0;
11
12     //Calculate quality index for each stream
13     foreach (KeyValuePair<int, DataStream> dspair in dataStreamDict)
14     {
15         DataStream DS = dspair.Value;
16         Statistics dsstats = DS.Stats;
17
18         //Test if there are enough frames in the stream to do an
               extrapolation
19         if (DS.Count > 20)
```

```
20          {
21              length = DS.First<Frame>().Length;
22
23              //Extrapolate next error based on history
24              extrap = Extrapolation(dsstats.History, 0.8f, 0.2f);
25              prediction = extrap.Last<float>();
26
27              //Calculate predicted quality index
28              predQuality = 1 - (prediction / length); //if zero errors
                    . Quality == 1
29              for (int i = 0; i < 10; i++)
30              {
31                  sum += extrap[i];
32              }
33
34              //Calculate history quality index
35              lastQuality = 1 - (sum / length);
36
37              //Weight 1:1, average of history and prediction. If zero
                    errors. Quality == 1
38              qindex = 1 / ((lastQuality + predQuality) / 2);
39              StreamQuality.Add(dspair.Key, qindex);
40          }
41          else
42          {
43              StreamQuality.Add(dspair.Key, float.MinValue);
44          }
45      }
46      //Add results to a dictionary that has the streamid as reference,
            ordered by quality
47      var tmp = StreamQuality.OrderBy(x => x.Value);
48      Dictionary<int, float> StreamQualitySorted = new Dictionary<int,
            float>();
49      foreach (KeyValuePair<int, float> entry in tmp)
50      {
51          StreamQualitySorted.Add(entry.Key, entry.Value);
52      }asses
53      return StreamQualitySorted;
54  }
```

After a quality index is found and the streams have been sorted, the first frame of the stream with the best quality is taken and looked up in the other streams by a list containing the latency. A list with a sorting of the latency is received by the quality manager which is described in the list of absolute stream positions. The latency is not meant as a latency in time but an offset in frame count. The list streamID contains the ID of all available streams holding data sorted by the previous determined quality.
At first the assumed position of the frame in the stream is looked up by calculating the absolute offset between the streams which is based on the continuous calculation in the DataQualityManager. Then the FindMatchingFrame function is called to look up the frame. It returns the real position even when it is shifted because of a frame drop or a change in the overall stream offset. This can happen in between two cycles of the quality manager. The details of this process will be explained later. If a frame was found it will

be placed in a list for collation. This way there will be the frame with the right ID of each stream inside the compareList list.

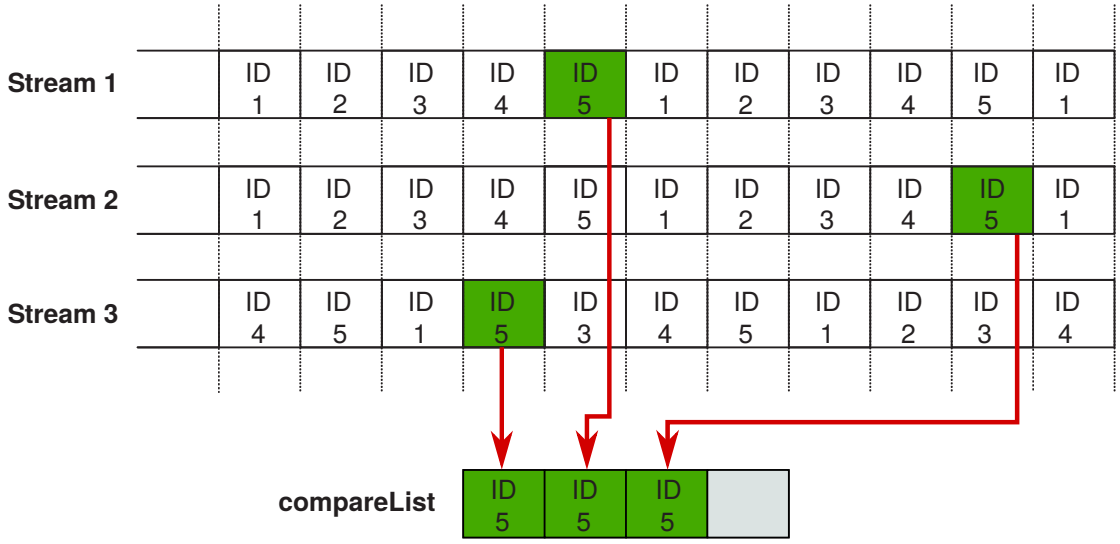


Figure 15: Matching frames of each stream are added to a list for comparison and majority vote

**Listing 20: DataMerger AddToComparison**

```
for (int i = 0; i < streamID.Count; i++)
{
    //Lookup the stream offset/latency
    int streamLatency = LatencyByStreamID(streamID[i]);
    DataStream DS;

    //Get the appropriate stream from the dictionary
    dataStreamDict.TryGetValue(streamID[i], out DS);

    //Calculate the expected positon
    int expectedPosition = streamLatency - firstStreamLatency;

    //Calculate the exact position
    int exactPositon = FindMatchingFrame(streamID[i],
        expectedPosition, DS[expectedPosition]);
    currentFramePos = exactPositon;

    //Add the frame into a list for later comparison.  If frame not
        found, skip
    if (exactPositon >= 0)
    {
        compareList.Add(DS[expectedPosition]);
    }

}
```

The list of frames can now be used to choose the best one. In a first step the frame with the least errors is chosen and if there are more than two streams a majority vote is executed with the selected word as basis. If there is no majority for this frame the next frame is chosen for the majority vote. If there are not enough frames for a majority vote the frame with the least corrected errors will be chosen. In case no majority can be found, the least defective frame will be returned. A majority is found if there are more than half of the frames $(0.5 * N_{Frames} + 1)$ matching the selected frame.

**Listing 21: DataMerger Frame Selection**

```
1  //Get and save errors for each frame in separate list
2  for (int i = 0; i < compareList.Count; i++)
3  {
4      errorAr[i] = ((TMFrame)compareList[i]).ErrorsCorrectedCnt; //
5  }
6
7  //Initialize the selection
8  int min = int.MaxValue;
9  int forcemin = 0;
10 int minIndex = 0;
11 int cyclecount = 0;
12 int absMinError = 0;
13
14 //Select the best frame
15 while (cyclecount <= compareList.Count - 1)
16 {
17     //Limit the selection process to the numbers of elements in the
           list
18     cyclecount += 1;
19
20     //Find frame with the least errors, save the error count and the
           position
21     for (int i = 0; i < errorAr.Length; i++)
22     {
23         if (errorAr[i] < min & errorAr[i] > forcemin)
24         {
25             min = errorAr[i];
26             minIndex = i;
27         }
28
29         //Save the frame with the absolute minimum error count for
               the use in case no majority can be found
30         if (cyclecount == 1)
31         {
32             absMinError = min;
33         }
34     }
35
36     //Do a majority vote based on the previous selected frame
37     if (compareList.Count > 2)
38     {
39         int matchcount = 0;
40
41         //Count the matches for a
42         for (int i = 0; i < errorAr.Length; i--)
```

```
43          {
44              if (compareList[minIndex].Equals(compareList[i]))
45              {
46                  matchcount += 1;
47              }
48          }
49          //If there are more matches than half of the entries a
                majority is found
50          if (matchcount > compareList.Count / 2)
51          {
52              return compareList[minIndex];
53          }
54          forcemin = min + 1;
55      }
56      else
57      {
58          return compareList[minIndex];
59      }
60  }
61  //If no majority could be found return the frame with the minimum
        corrected errors
62  return compareList[absMinError];
```

The last function of Stream merger that has to be discussed is the aforementioned FindMatchingFrame. As described in Section DataQualityManager 3.2.8 no binary search is applicable for the use inside a stream, but since we now have an expected position we can use this information to reduce the comparison operations.
The function takes three arguments. The ID can be used to look up the corresponding DataStream object in the stream dictionary, the expected position that was derived using the output of the offsets calculated in the DataQualityManager and the frame itself for verifying potential matches by frame ID. At first, it is checked if the frame is at the expected location, if it is, the function returns the expected position and quits. In case the stream ID that is been looked at has the lowest latency, we only have to search to the right which means younger frames than the chosen one. If not, then the matching frame can be younger or older and it has to be searched in both directions. Since it is more likely that the searched frame is closer to the expected position than far away, the search is executed in circles around this initial position. This means the next upper and lower member of the list is tested for a matching ID. If the ID matches at a frame, the content of the frame is compared to the frame given by argument. This way the run time can be reduced significantly in the best case. Of course in the worst case if there is no such frame the run-time is still $O(n)$ and the function will return -1. The found position will also be returned to the DataQualityManger, so that the next iteration might not have to search again.

**Listing 22: DataMerger FindMatchingFrame**

```
1  public int FindMatchingFrame(int streamid, int expectedPosition,
       Frame frame)
2  {
3      DataStream DS;
4      Frame thisFrame;
```

```
5       int upcounter;
6       dataStreamDict.TryGetValue(streamid, out DS);
7
8       //If the frame is at the expected position return the position
9       if (DS[expectedPosition].ID == frame.ID)
10      {
11          return expectedPosition;
12      }
13      else
14      {
15          //Check if this is the stream with the lowest latency.
16          //If yes, search only to the right
17          if (StreamAtPositionInLatencyList(0).Value == streamid)
18          {
19              upcounter = expectedPosition;
20
21              //Iterate over all entries in the stream
22              while (upcounter < DS.Count)
23              {
24                  upcounter += 1;
25                  thisFrame = DS[upcounter];
26
27                  //First just check if the ID matches to speed up the
                        process
28                  if (thisFrame.ID == frame.ID)
29                  {
30
31                      //If ID fits, compare whole frame to verify
                            result
32                      if (thisFrame.Equals(frame))
33                      {
34                          //Check if stream is listed in offset list (
                                streamPositionInLatencyList)
35                          if (streamPositionInLatencyList.ContainsValue
                                (streamid))
36                          {
37                              //Update offset list for
                                    DataQualityManager
38                              lock (streamPositionInLatencyList)
39                              {
40                                  //Remove the entry, and create the
                                        new one
41                                  streamPositionInLatencyList.Remove(
                                        expectedPosition);
42                                  streamPositionInLatencyList.Add(
                                        expectedPosition + upcounter,
                                        streamid);
43                              }
44                          }
45                          return (upcounter);
46                      }
47                  }
48              }
49          }
50          else
```

```
51          {
52              //Search in both directions
53              int downcounter = expectedPosition;
54              upcounter = expectedPosition;
55              int counter = 0;
56
57              //Stay inside the limits of the DataStream entries
58              while (downcounter > 0 || upcounter < DS.Count)
59              {
60                  counter += 1;
61                  downcounter -= counter;
62                  upcounter += counter;
63
64                  //Look to the right neighbor of the last tested frame
65                  if (upcounter < DS.Count)
66                  {
67                      thisFrame = DS[upcounter];
68
69                      //First just check if the ID matches to speed up
                            the process
70                      if (thisFrame.ID == frame.ID)
71                      {
72                          //If ID fits, compare whole frame to verify
                                result
73                          if (thisFrame.Equals(frame))
74                          {
75                              //Check if stream is listed in offset
                                    list (streamPositionInLatencyList)
76                              if (streamPositionInLatencyList.
                                    ContainsValue(streamid))
77                              {
78                                  //Update offset list for
                                        DataQualityManager
79                                  lock (streamPositionInLatencyList)
80                                  {
81                                      //Remove the entry, and create
                                            the new one
82                                      streamPositionInLatencyList.
                                            Remove(expectedPosition);
83                                      streamPositionInLatencyList.Add(
                                            expectedPosition + upcounter,
                                            streamid);
84                                  }
85                              }
86                              return (upcounter);
87                          }
88                      }
89                  }
90
91                  //Look to the left neighbor of the last tested frame
92                  if (downcounter > 0)
93                  {
94                      thisFrame = DS[downcounter];
95                      //First just check if the ID matches to speed up
                            the process
```

```
96                          if (thisFrame.ID == frame.ID)
97                          {
98                              //If ID fits, compare whole frame to verify
                                    result
99                              if (thisFrame.Equals(frame)) //if id fits,
                                    compare whole frame to verify result
100                             {
101                                 if (streamPositionInLatencyList.
                                        ContainsValue(streamid)) //check if
                                        entry is there
102                                 {
103                                     //Update offset list for
                                            DataQualityManager
104                                     lock (streamPositionInLatencyList)
105                                     {
106                                         //Remove the entry, and create
                                                the new one
107                                         streamPositionInLatencyList.
                                                Remove(expectedPosition);
108                                         streamPositionInLatencyList.Add(
                                                expectedPosition + downcounter
                                                , streamid);
109                                     }
110                                 }
111                                 return (downcounter);
112                             }
113                         }
114                     }
115                 }
116             }
117             return -1;
118         }
119 }
```

### 3.2.10 Double Exponential Smoothing

Exponential smoothing is a technique for calculating short term projections of a time series of values. Unlike the moving average function exponential smoothing decreases the weights for older observations in the process. This ensures that more recent events have a deeper impact on the prediction which is useful for our field of application since we have to cope with rapid changes in quality like burst errors. However, we want to observe a trend in our values to detect an overall increasing or decreasing quality. Nevertheless, this is not applicable with (single) exponential smoothing but with double exponential smoothing. See [NIST/SEMATECH(2012), 6.4.3.1] for the background and the following derived equation on single exponential smoothing.

The algorithm for single exponential smoothing consists of one expression. The first one defines the smoothing $S$ of the measured values $y$ in time period $t$. For $t = 1$ the value of $S_{t-1} = y_0$:

$$S_t = \alpha y_{t-1} + (1 - \alpha) S_{t-1} \tag{33}$$

The parameter $\alpha$ is called smoothing constant that can be used for weighting of the

current with the last measurement.
The exponent comes with the recursion. It will be more clear when looking at a substituted expression for the last two time periods:

$$S_t = \alpha y_{t-1} + (1-\alpha)[\alpha y_{t-2} + (1-\alpha)S_{t-2}] \tag{34}$$

$$S_t = \alpha y_{t-1} + \alpha(1-\alpha)y_{t-2} + (1-\alpha)^2 S_{t-2} \tag{35}$$

For double exponential smoothing a trend is introduced into the smoothing. The trend is calculated from the previous history. The difference of the last two smoothed values $S_t$ and $S_{t-1}$ is weighted with a trend constant:

$$b_t = \gamma(S_t - S_{t-1}) + (1+\gamma)b_{t-1} \tag{36}$$

Which is also an exponential equation like $S_t(y_t)$. As described the trend is added to the smoothing of the previous calculation. So the smoothing expression from the single exponential smoothing transforms into:

$$S_t = \alpha y_{t-1} + (1-\alpha)(S_{t-1} + b_{t-1}) \tag{37}$$

Since $S_t$ and $b_t$ are exponential functions this algorithm is called double exponential smoothing (see [NIST/SEMATECH(2012), 6.4.3.3]). The initial values for $S_{t-1}$ are zero and $b = y_2 - y_1$. The weighting parameters can be chosen by the operator and derived by testing data from previous flights and will change from mission to mission.

For the implementation the error of the past 20 frames will be looked at and one step will be predicted into the future.

**Listing 23: DataMerger Extrapolation**

```
1  public float[] Extrapolation(List<Statistics.ResetEvent> serie, float
       a, float y)
2  {
3      //Array predict keeps S for 21 frames: 20 from the past, 1
           predicted into the future
4      float[] predict = new float[21];
5
6      //Test if there are already 20 processed frames for this stream
7      if (serie.Count >= 20)
8      {
9          int[] errorAr = new int[20];
10
11         //Get the last 20 error counts out of the history from the
               statistic class
12         for (int i = serie.Count; i > (serie.Count - 20); i--)
13         {
14             errorAr[20 - i] = serie[i - 1].Errors;
15         }
16
17         //Set the start parameters
18         float S = (float)errorAr[0];
```

```
19          float St_1 = 0.0f;
20          float b = (float)(errorAr[1] - errorAr[0]);
21          float error = 0.0f;
22
23          //Predict one step into the future, based on the last 20
                steps
24          for (int i = 1; i <= errorAr.Length + 1; i++)
25          {
26
27              if (i < errorAr.Length)
28              {
29                  //The first 20 entries to be used as St_1 are from
                        measurements of the past
30                  error = (float)errorAr[i];
31              }
32              else
33              {
34                  //The last entry is based on the last prediction bt-1
35                  error = predict[i - 1];
36              }
37              St_1 = S;
38
39              //Execute the double exponential smoothing
40              S = a * error + (1 - a) * (S + b);
41              b = y * (S - St_1) + (1 - y) * b;
42              predict[i] = S;
43          }
44      }
45      return predict;
46  }
```

The function returns an array that can now be processed by the CalcQualityIndex method inside the merger.

### 3.2.11 Statistics

The Statistics object of each stream keeps track of meta information for each stream like last frame error, frames received, frames received per time period, frame quality, average frame quality per time period and overall corrected errors since startup. Quality and frame defined time period are kept as well as a history on errors corrected per time slot. For keeping track of the time intervals a DateTime is stored per object as well. It also has a stopwatch that is used by the time manager to keep track of the processing times in the system to keep them in defined bounds. Getters and Setters are defined so that errors just have to be added and the statistics will update automatically. Inside a statistic object there is a nested object called ResetEvent. The creation of a reset event can be triggered from outside periodically. Each reset event contains the errors that where corrected since the last event, the count of frames that have been processed and the average quality in the DQE of the incoming frames. The Statistics object keeps a list of all ResetEvents. These events will be used to calculate the prediction with double exponential smoothing like described in Section 3.2.10.
The ResetEvent class is very short since it just stores information:

**Listing 24: Statistics ResetEvent**

```
public class ResetEvent
{
    DateTime Time = DateTime.Now;
    public int Errors = 0;
    public int Frames = 0;
    public double AvgQuality = 0;
    public ResetEvent(int error, double avgQuality, int frames)
    {
        this.Errors = error;
        this.AvgQuality = avgQuality;
        this.Frames = frames;
    }
}
```

The introduction of ResetEvents allows a dynamic adaption of different data stream speeds. For a 1 kBit/s stream it is not useful to measure errors per ms and for several MBit/s it might be a way to find resolution. This way the operator can choose the intervals depending on the anticipated input.

The following fields are publicly accessible by the Statistics class and might also be used for further documentation purposes like post-processing of data or user interfaces.

**Listing 25: Statistics Data Fields**

```
public int LastFrameErrors { get => lastFrameErrors; set => AddError(
    value); }
public int ErrorsSinceReset { get => errorsSinceReset;}
public int ErrorsOverall { get => errorsOverall;}
public int FramesReceivedOverall { get => framesReceivedOverall;}
public int FramesSinceReset { get => framesReceivedSinceReset;}
public double LastFrameQuality { get => lastFrameQuality; set =>
    CalcAvgQuality(value); }
public double AvgQualitySinceReset { get => avgQualitySinceReset;}
public double AvgErrorSinceReset { get => avgErrorSinceReset; }
public List<ResetEvent> History { get => history;}
public DateTime ResetTime { get => resetTime; set => resetTime =
    value; }
public Stopwatch FrameWatch { get => frameWatch; set => frameWatch =
    value; }
public Int64 FrameTicks { get => FrameWatch.ElapsedTicks; }
private List<ResetEvent> history = new List<ResetEvent>();
```

### 3.2.12 TimeManager

The TimeManger is there to observe the duration of the processing time of a frame. It keeps track of the ongoing process as well as dynamically calculates the maximum allowed time based on the frame rate at the input side.
If the time limit for a frame in the merger is reached the process will be interrupted and the current frame is copied to the output omitting the merging process. This has to be done in order to guaranty that data cannot pile up in the system and that a frame reaches

the operator in reasonable time. On average the output rate has to be equal to the input rate. This is ensured by the Watchdog task inside the TimeManger class. The runner is executed in a dedicated task to work parallel to the other duties.

**Listing 26: TimeManager CalcMaxTics**

```
1  public void CalcMaxTics(Object source, ElapsedEventArgs e)
2  {
3      //Prevent a division by zero if no frames have been received yet.
4      if (stats.FramesSinceReset == 0)
5      {
6          ticksPerFrame = Double.MaxValue;
7      }
8      else
9      {
10         //Calculate the maximum allowed time in ticks per frame
11
12         //Seconds since last reset event
13         double sSinceReset = (double)((DateTime.Now.Subtract(stats.
               ResetTime)).TotalSeconds);
14         //Frames per second
15         double framesPerS = stats.FramesSinceReset / sSinceReset;
16         //Ticks per Frame
17         ticksPerFrame = Stopwatch.Frequency / (1 / framesPerS);
18     }
19 }
```

This function is called periodically by a timer task, the duration can be modified. The dynamic calculation of the time boundaries of a frame allows to adapt the time spend on the frame automatically. For slower streams the hard boundary is higher than for faster inputs.

The watchdog is a simple runner that checks the current frame statistics and the previously calculated boundary continuously. To have a 10% margin on top of the calculations the ticksPerFrame is reduced by the factor of 0.9. In case a boundary is reached the merger thread will be interrupted as explained in Section 3.2.9.

**Listing 27: TimeManager Runner**

```
1  public void Runner()
2  {
3      while (run)
4      {
5          //Check if maximum calculation time is elapsed. Add a 10%
               time margin to ensure soft-realtime
6          if (stats.FrameTicks >= ticksPerFrame*0.9)
7          {
8              mergerThread.Interrupt();
9          }
10
11     }
12 }
```

## 3.3 Verification

The functionality of the software was verified using unit tests together with a data generator that has been implemented for this reason. The generator is capable of generating DataWords, TFrames, TMFrames and Streams of various content. In the Appendix B the code can be found. DataWord generator has two constructors. The first one takes data and FEC bytes as arguments, which allows setting data and parity bits independently. With this a faulty word can be generated. The parity bits are limited to the maximum of 4—bit since only four party bits are allowed with this FEC algorithm. Constructor number two takes just the data bits as input. The FEC is generated accordingly for a valid DataWord. The words are generated in little endian representation. By using binary operations in the Forward Error Correction generator the generation speed is improved.
With the Frame generator TFrame objects are created. From inside the class the aforementioned DataWords are made. As arguments, it accepts a word count, a frame number and a signal quality. Using this information the FrameGen function is called. By utilizing the HostToNetworkOrder function from the System library the endianness of the host is adapted. In a first step the sync word $0xFAF320$ of the TFrame is placed inside a BitArray. Appended will be a DataWord containing the counter, followed by more words using a counter value as data for verification purposes.
A TMFrameGen method wraps the just generated TFrame into a second header. Data is returned in a byte array format. The preamble is added followed by the frame length, that has to be calculated based on the header and the TFrame length. Then the version is included followed by an offset containing the length of the header and the quality which is a double value. Finally, the data of the TFrame is added.
For creating a stream out of frames the stream generator function is used. As arguments to the constructor a frame and word count have to be given. From the StreamGen function the Frame and DataWord generators are called, subsequently filled and returned as a binary array. With this array unit testing can be done, by sending the array to the UDP socket of the framework. Each processing step is tested by comparing input against the processed output to verify its functionality. With it the DataWord and the parity bits for the Forward Error Correction as well as the error correction itself were checked. The data stream from input to output has been reviewed through the system.

Majority vote and frame selection worked as intended. Three data streams up to 717 kBit/s were tested without difficulties. The measurement was done with the network analysis tool WireShark. After reaching this bandwidth, copy instructions limited the transfer and the watchdog got activated on the system used. So the limitation of this implementation is not the algorithm itself but the implementation of data preparation. The system consisted of an Intel Core i7—6700K CPU and 16 GB RAM. However, due to the load on memory by copying objects, the full potential of the CPU could not be utilized during the tests, the load did not exceed 30%. Most certainly the utilization of a Linked List is not suitable for the use as buffer object. A possible solution would be to modify the InputHandler in a way so that it does not rely on a Linked List or use another FIFO object. The delay introduced by the system was measured with WireShark as well. It could be determined to be in the range of 0.1—0.2 seconds, which is acceptable. It has to be noted that the measurement is dependent on the hardware used.
Since optimization was explicitly not part of the requirements for this thesis there are

various options to improve the performance in the future. One approach would be to reduce the rapid object copying during the input procession or to make use of special CPU instructions in the error correction process like Intel's "POPCNT" for checking parities. Furthermore, the Statistics object inside the software can be utilized for example to build a graphical user interface showing the current stream stability and system health. For a maximum flexibility the software was written using the .Net framework which permits a support for Windows and Unix operating systems.

With the results and verification the functionality of the framework was tested successfully. The architecture combines various selection and improvement modes into a single framework, these have previously been used as standalone options. This approach allows to get the maximum quality of data filled into the algorithm. Additionally, through the detailed analysis of the Forward Error Correction algorithm now new developments for other use cases besides the conflating stream merger are made possible. Therefore, it can be concluded that the algorithm development and implementation was successful and met the requirements of showing a working implementation of a conflating best source selection with inter-stream error correction.

# 4 Conclusion

During the work presented in this thesis a large amount of effort had to be dedicated to analyze the existing error correction functions and to adapt the selection process to the special environment of sounding rockets. This was possible not only using best source detection but also by frame recombination enabled by the long overlapping time of telemetry reception. With this it became possible to achieve the maximum data quality for the operators in soft real-time. A functional framework with the approach of stream quality determination for pre-ranking and single frame selection were implemented.

From the known parity check matrix the forward error correction method was derived along with the statistical capabilities. With these values and the software implementation for the detection and correction of broken frames it was determined that each frame can be corrected without further investigation. This is possible because the computing overhead of the correction is not significantly higher than the detection process and the probability of a wrong correction is low, especially for the characteristic duration of a sounding rocket flight. Each stream is split up in its constituent parts for an educated reassembling into a single output stream. To achieve this, offsets have to be identified and eliminated and a majority vote on the pre-ranked frames is done if operable. By keeping the frame format during the process the output looks identical to a single stream as it would be sent by a telemetry station. This results in the possibility to use the framework as an optional add-on to the current process.

This altogether shows that it is possible to perform dynamic best data selection on several input streams while staying within a soft real-time boundary, so that the correction is not limited to the post processing of the recorded data. With this achieved, further developments can now focus on system performance, external monitoring and supervision options by the operator. The advantage of a tailor made system of sounding rocket applications running on a PC with a sophisticated selection process instead of using a FPGA based implementation comes with the downside of limited bandwidth. The decision which implementation fits best each campaign is, however, based on the custom criteria of each individual launch.

Overall this thesis lays a solid foundation for further custom-made best source selections for sounding rockets at MORABA.

# References

[McDowell(2019)] Jonathan McDowell. Sounding rocket launch database. http://planet4589.org/space/lvdb, September 2019. Retrieved 09.11.2019 14:30h.

[IN-SNEC(2009)] IN-SNEC. *Boardband Radio Telemetry Receiver Cortex XL Series*, ver.2 - ref. 3 edition, 2009.

[Nicolo(2018)] S. Nicolo. History and advantages of best source selection for today's modern telemetry applications along with the benefits of different approaches. In *Proceedings ettc2018: European Test and Telemetry Conference ettc2018*, pages 48 – 54. GDP Space Systems, AMA Service GmbH, The European Society of Telemetry, 2018. doi: 10.5162/ettc2018/2.3.

[Tran-Gia(2015)] Prof. Dr.-Ing. Phuoc Tran-Gia. Informationsuebertragung - fehlererkennung und fehlerkorrektur, 2015.

[DLR(2019)] Dlr at a glance. www.dlr.de/dlr/en/desktopdefault.aspx/tabid-10443, July 2019. Retrieved 01.09.2019 14:35h.

[RB(2019)] Dlr organisation. www.dlr.de/rb/en/desktopdefault.aspx/tabid-2711, September 2019. Retrieved 01.09.2019 15:46h.

[MOR(2019)] Welcome to moraba. www.moraba.de, September 2019. Retrieved 01.09.2019 15:48h.

[ISO(1994)] ISO. 7498-1:1994. Technical Report X.200, International Telecommunication Union (ITU), July 1994.

[Breed(2003)] Gary Breed. Bit error rate: Fundamental concepts and measurement issues. High Frequency Design, 2003.

[Butterfield et al.(2016)Butterfield, Ngondi, and Kerr] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr. *A Dictionary of Computer Science (Oxford Quick Reference)*. Oxford University Press, 2016. ISBN 978-019-968-8-975.

[Pallone et al.(2018)Pallone, Pontani, Teofilatto, and Minotti] Marco Pallone, Mauro Pontani, Paolo Teofilatto, and Angelo Minotti. Design methodology and performance evaluation of new generation sounding rockets. *International Journal of Aerospace Engineering*, 2018(1678709):16, 2018. doi: 10.1155/2018/1678709.

[Ast(2019)] Improved orion. www.astronautix.com/i/improvedorion.html, September 2019. Retrieved 17.09.2019 21:22h.

[Schuettauf et al.(2018)Schuettauf, Kirchhartz, et al.] Katharina Schuettauf, Rainer Kirchhartz, et al. *REXUS User Manual*. EuroLaunch, 7.16 edition, October 2018.

[Baggerman(2019)] Bob Baggerman. The standard for digital flight data recording. www.irig106.org, July 2019. Retrieved 18.09.2019 22:06h.

[Tan et al.(2019)Tan, Qin, Cong, and Zhao] Z. Tan, H. Qin, L. Cong, and C. Zhao. New method for positioning using iridium satellite signals of opportunity. *IEEE Access*, 7:83412–83423, June 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2924470.

[CML(2008)] *CMX909B GMSK Packet Data Modem*. CML Microcircuits Ltd., d/909b/2 edition, November 2008.

[Pless(1998)] Vera Pless. *Introduction to the Theory of Error-Correcting Codes*. Wiley-Interscience, 1998. ISBN 0-471-19047-0.

[Ling(2004)] San Ling. *Coding Theory: A First Course*. Cambridge University Press, 2004. ISBN 0-521-52923-9.

[Mildenberger and Werner(1998)] O. Mildenberger and M. Werner. *Nachrichtentechnik: Eine Einführung für alle Studiengänge*. uni-script. Vieweg+Teubner Verlag, 1998. ISBN 978-3-663-10867-2. doi: 10.1007/978-3-663-10867-2.

[NIST/SEMATECH(2012)] NIST/SEMATECH. *e-Handbook of Statistical Methods*. National Institute of Standards and Technology, April 2012. www.itl.nist.gov/div898/handbook/ Retrieved 04.09.2019 09:31h.

# A Matlab Code

## A.1 Hamming Distance Calculation

```
%% Hamming distance calculator for Blockcode used in CMX909
%  Result is dmin=3

CodeWords=genCode();
countmax=256;
countmin=0;
countarray=zeros(12,1);
count2=0;
for i=1:size(CodeWords)-1
    for j=i+1:size(CodeWords)
        a=CodeWords(i);
        b=CodeWords(j);
        count=0;
        if i~=j
            fillstring="";
            fillupsize=0;
            abin=dec2bin(a);
            bbin=dec2bin(b);
            fillup=(size(bbin)-size(abin));
            fillupsize=fillup(2);
            if fillupsize > 0
                for adsf=1:fillupsize
                    fillstring=fillstring+"0";
                end
                abin=strcat(fillstring,abin);
                abin=convertStringsToChars(abin);
            end
            sizeamatrix = size(abin);
            sizea = sizeamatrix(2);
            for x = 1:sizea
                if abin(x)~=bbin(x)
                    count=count+1;
                end
            end
            if count ==3
               count2=count2+1;
               abin;
               bbin;
            end
        end
        if count<countmax
            countmax=count;
            maxcnt=count;

        end
        if count>countmin
            countmin=count;
            mincnt=count;
        end
        i=i+1;
    end
```

```
52      countarray(countmax)= countarray(countmax)+1;
53      countmax=12;
54      countmin=0;
55  end
56  %Result:
57  "dmin="+mincnt
58  "dmax="+maxcnt
59
60  %Generate DataWords (eq. to code words)
61  function CodeWords=genCode()
62  H=[236 211 186 117];
63  P=[256:5];
64      for i=1:256
65          %count bits == 1
66          iAndH=bitand((i-1),H);
67          %calc parity
68          a=mod(sum(bitget(iAndH(1),1:8)),2);
69          b=mod(sum(bitget(iAndH(2),1:8)),2);
70          c=mod(sum(bitget(iAndH(3),1:8)),2);
71          d=mod(sum(bitget(iAndH(4),1:8)),2);
72          %join parity bits
73          Pbits=bin2dec([dec2bin(a) dec2bin(b) dec2bin(c) dec2bin(d)]);
74          CodeWords(i,:)=(i-1)*16+Pbits;
75      end
76  end
```

## A.2 Possibility of Undetectable Errors in Code Word

```
1  %% Calculate the possibility of undetectable errors in a code word
2  % calculate bits==1 sum per word
3  countarray=zeros(13,1);
4  CodeWords=genCode();
5  for x=1:size(CodeWords)
6      word=CodeWords(x);
7      out=sum(bitget(word,1:12));
8      countarray(out+1)=countarray(out+1)+1;
9  end
10
11 % calculate possibility per bitsum
12 posarray=zeros(13,1);
13 format longE;
14 for x=1:size(countarray)
15     posarray(x)=countarray(x)*(10.^-7).^x*(1-10.^-7).^(12-x);
16 end
17 posarray
18 sum=0;
19 % sum up possibilities
20 for x=4:size(posarray)
21     sum=sum+posarray(x);
22 end
23 "posiility of undetectable wrong code word: "+sum
```

# B DataGenerator Framework

## B.1 DataWord

**Listing 28: DataGenerator DataWord**

```
1  public class DataWord
2  {
3      private Int32 databits;
4      private Int32 fec;
5      private Int16 word;
6      private readonly Int32[] HammingBlock = {0b11101100,
7                                               0b11010011,
8                                               0b10111010,
9                                               0b01110101};
10
11     /// <summary>
12     /// Constructor of a new DataWord Object consiting of the data
           bytes and a given FEC parity
13     /// </summary>
14     /// <param name="databits">8bit data</param>
15     /// <param name="fec">4bit bits (FEC)</param>
16     public DataWord(byte databytes, byte fec)
17     {
18         //check if fec is longer than 4 bit
19         if ((fec & 11110000) > 0) throw new Exception("FEC has to be
               4 bits long");
20         this.databits = databytes;
21         this.fec = fec;
22         this.word = (Int16)((this.databits << 4) | this.fec);
23     }
24
25     /// <summary>
26     /// Constructor of a new DataWord Object consiting of the data
           bytes. FEC is generated
27     /// </summary>
28     /// <param name="databits">8bit Data</param>
29     public DataWord(byte databits)
30     {
31         this.databits = databits;
32         this.fec = FECGenerator(databits);
33         this.word = (Int16)((this.databits << 4) | this.fec);
34         //System.Diagnostics.Debug.WriteLine(this.word);
35     }
36
37     /// <summary>
38     ///  Constructor of a new DataWord Object from an existing word.
39     /// </summary>
40     /// <param name="word">12bit Word in binary format</param>
41     /// <param name="check">Do error detection</param>
42     public DataWord(Int16 word)
43     {
44         this.word = word;
45     }
46
```

```
47      /// <summary>
48      /// DataWord getter
49      /// </summary>
50      public Int16 Word { get { return this.word; } }
51
52      /// <summary>
53      /// Two DataWords are equal when their words are equal
54      /// </summary>
55      /// <param name="obj">Obj to compare with</param>
56      /// <returns>Equality</returns>
57
58      public override bool Equals(object obj)
59      {
60          DataWord dw = obj as DataWord;
61          if (dw == null) return false;
62          if (dw.Word == this.word) return true;
63          return false;
64      }
65
66      /// <summary>
67      /// Hash is generated on word itself
68      /// </summary>
69      /// <returns>Hash of Word</returns>
70      public override int GetHashCode()
71      {
72          return this.word.GetHashCode();
73      }
74
75      /// <summary>
76      /// Generate the fec for a 8-bit data input
77      /// </summary>
78      /// <returns>4-bit FEC code</returns>
79      public Int32 FECGenerator(Int32 databits)
80      {
81          Int32 fec = 0;
82          Int32 cAndd = 0;
83          foreach (Int32 code in HammingBlock)
84          {
85              cAndd = (code & databits);
86              fec = (fec << 1) | ParityCounter(cAndd);
87          }
88          return fec;
89      }
90
91      /// <summary>
92      /// Count parity, 1 if even 0 if uneven
93      /// </summary>
94      /// <param name="databits">databits to count on</param>
95      /// <returns>1 if even 0 if uneven</returns>
96      public Int32 ParityCounter(Int32 databits)
97      {
98          databits ^= databits >> 16;
99          databits ^= databits >> 8;
100         databits ^= databits >> 4;
101         databits ^= databits >> 2;
```

```
102             databits ^= databits >> 1;
103             return (databits & 1);
104         }
105 }
```

## B.2 Frames

**Listing 29: DataGenerator Frame**

```
1  public class Frame
2  {
3      // Using Sync word from IRIG106 Chapter 8 Section 8.4. 00 at the
           end has to stay there, to fill up the value even it is
           overwritten
4      private const UInt32 FrameSync = 0xFAF320;
5      private const UInt32 TMFrameSync = 0xFAFBFCFD;
6      private UInt16 wordcount;
7      private byte count;
8      int length = 0;
9      private byte[] frame;
10     private byte[] TMWrapperFrame;
11     private const byte TMFrameVersion=1;
12     public const byte offset = 2 * sizeof(Int32) + sizeof(double);
13     private double signalquality;
14
15     /// <summary>
16     /// Generates a single (TM/Transfer)Frame consisting #wordcount
           words, with rising numbers as data. Starting counter at 0,
           Signal quality is 0.
17     /// </summary>
18     /// <param name="wordcount">Number of words in frame</param>
19     public Frame(UInt16 wordcount)
20     {
21         if (wordcount % 2 != 1) throw new Exception("Wordcount has to
               be uneven");
22         this.wordcount = wordcount;
23         this.count = 0;
24         this.signalquality = 0;
25     }
26
27     /// <summary>
28     /// Generates a single (TM/Transfer)Frame consisting #wordcount
           words, with rising numbers as data. Signal quality is 0.
29     /// </summary>
30     /// <param name="wordcount">Number of words in frame</param>
31     /// <param name="framenumber">Frame number</param>
32     public Frame(UInt16 wordcount, byte framenumber)
33     {
34         if (wordcount % 2 != 1) throw new Exception("Wordcount has to
               be uneven");
35         this.wordcount = wordcount;
36         this.count = framenumber;
37         this.signalquality = 0;
```

```
38      }
39
40      /// <summary>
41      /// Generates a single (TM/Transfer)Frame consisting #wordcount
            words, with rising numbers as data
42      /// </summary>
43      /// <param name="wordcount">Number of words in frame</param>
44      /// <param name="framenumber">Frame number</param>
45      /// <param name="signalquality">Quality of the signal (0 to 1)</
            param>
46      public Frame(UInt16 wordcount, byte framenumber, double
            signalquality)
47      {
48          if (wordcount % 2 != 1) throw new Exception("Wordcount has to
                be uneven");
49          this.wordcount = wordcount;
50          this.count = framenumber;
51          this.signalquality = signalquality;
52      }
53
54      /// <summary>
55      /// Returns a transfer frame
56      /// </summary>
57      public byte[] TransferFrame => FrameGen();
58
59      /// <summary>
60      /// Returns a TM transfer frame
61      /// </summary>
62      public byte[] TMFrame=> TMFrameGen();
63
64
65      public static void PrintValues(IEnumerable myCollection)
66      {
67          foreach (Object obj in myCollection)
68              Debug.Write((bool)obj ? "1" : "0");
69          Debug.WriteLine("");
70      }
71
72      /// <summary>
73      /// Generates a transfer frame
74      /// </summary>
75      /// <returns>Transfer Frame</returns>
76      public byte[] FrameGen()
77      {
78
79          length = (int)(4.5 + wordcount * 1.5);
80          frame = new byte[length];
81
82          //Ad FrameSync with right endianness
83          unchecked
84          {
85              Array.Copy(BitConverter.GetBytes(System.Net.IPAddress.
                    HostToNetworkOrder((int)(FrameSync))), frame, 3);
86          }
87          DataWord counter = new DataWord(count);
```

```
88          BitArray counterbits = new BitArray(new Int32[] { counter.
               Word });
89          BitArray data = new BitArray(wordcount * 12+12);
90
91          for (int i = 0; i < 12; i++){
92                  data[((wordcount+1)*12)-i] = counterbits[i];
93          }
94
95          //Counts words, first word is counter so start at 2
96          for (int i = 2; i <= wordcount+1; i++)
97          {
98              DataWord dw = new DataWord((byte)128);
99              BitArray dwbit = new BitArray(new Int32[] { dw.Word });
100             for (int j = 0; j < 12; j++)
101             {
102                 data[wordcount*12-(j+1+(i-2)*12)]= dwbit[(12 - 1) - j
                       ];
103             }
104         }
105         byte[] output = new byte[data.Length / 8];
106         data.CopyTo(output, 0);
107         Array.Reverse(output);
108         Array.Copy(output, 0, frame, 3, output.Length);
109         return frame;
110     }
111
112     /// <summary>
113     /// Generates a TM transfer frame
114     /// </summary>
115     /// <returns>TM transfer frame</returns>
116     public byte[] TMFrameGen()
117     {
118         frame = FrameGen();
119         length = frame.Length + offset;
120         TMWrapperFrame = new byte[length];
121         unchecked
122         {
123             Array.Copy(BitConverter.GetBytes(System.Net.IPAddress.
                   HostToNetworkOrder((int)TMFrameSync)), TMWrapperFrame,
                    sizeof(Int32)); //frame sync
124             Array.Copy(BitConverter.GetBytes(System.Net.IPAddress.
                   HostToNetworkOrder((short)length)), 0, TMWrapperFrame,
                    4, sizeof(short));  //12bit per word + 3byte transfer
                    header+ 3byte counter + 18 byte tm header //length
125             TMWrapperFrame[6] = TMFrameVersion; //version
126             TMWrapperFrame[7] = offset; //offset
127             Array.Copy(BitConverter.GetBytes(signalquality), 0,
                   TMWrapperFrame, 8, sizeof(double)); //quality#
128         }
129         Array.Copy(frame, 0, TMWrapperFrame, offset, frame.Length);
               //data
130         return TMWrapperFrame;
131     }
132
133     /// <summary>
```

```
134     /// Two Frames are equal when their TM Frame data is equal
135     /// </summary>
136     /// <param name="obj">Obj to compare with</param>
137     /// <returns>Equality</returns>
138     public override bool Equals(object obj)
139     {
140         Frame otherframe = obj as Frame;
141         if (otherframe == null) return false;
142         return otherframe.TMFrame.SequenceEqual(TMFrame);
143     }
144 
145     /// <summary>
146     /// Hash is generated on TMFrame
147     /// </summary>
148     /// <returns>Hash of Word</returns>
149     public override int GetHashCode()
150     {
151         return this.TMFrame.GetHashCode();
152     }
153 
154 }
```

## B.3 Stream

**Listing 30: DataGenerator Stream**

```
1  public class Stream
2  {
3      private int framecount = 0;
4      private UInt16 wordcount = 0;
5      private byte[] stream;
6      public const byte offset = 2 * sizeof(Int32) + sizeof(double);
7      public Stream(int framecount, UInt16 wordcount)
8      {
9          this.framecount = framecount;
10         this.wordcount = wordcount;
11     }
12 
13     /// <summary>
14     /// Generate stream data
15     /// </summary>
16     /// <returns>Hash of Word</returns>
17     public byte[] StreamGen()
18     {
19         Int32 length = framecount * (int)(offset + 4.5 + wordcount *
               1.5);
20         stream = new byte[length];
21         for (UInt16 i = 0; i < framecount; i++)
22         {
23             Frame frame = new Frame(wordcount, (byte)((i * framecount
                   ) % sizeof(byte)));
24             Array.Copy(frame.TMFrame, 0, stream, i * frame.TMFrame.
                   Length, frame.TMFrame.Length);
```

```
25          }
26          return stream;
27      }
28
29      public byte[] TMStream => StreamGen();
30
31  }
```

# List of abbreviations

**BEP** Bit Error Probability

**BER** Bit Error Rate

**BSS** Best Source Selector

**CBSS** Correlating Best Source Selector

**CCSDS** Consultative Committee for Space Data Systems

**CRC** Cyclic Redundancy Check

**CRC** Cyclic Redundancy Check

**DLR** German Aerospace Center

**DQE** Data Quality Encapsulation

**DQM** Data Quality Matrix

**FEC** Forward Error Correction

**FM** Frequency Modulation

**FPGA** Field Programmable Gate Array

**GSOC** German Space Operation Center

**IRIG** Inter Range Instrumentation Group

**ISS** International Space Station

**MORABA** Mobile Rocket Base

**MUSC** Microgravity User Support Center

**OSI** Open Systems Interconnection

**PCM** Pulse Code Modulation

**POE** Probability Of Error

**RCC** Range Commanders Council

**UDP** User Datagram Protocol