Fast Decoding of ARINC 717 Flight Data Recordings

Conference Paper · January 2021

DOI: 10.2514/6.2021-1982

2 authors:

Florian Schwaiger
Technische Universität München
14 PUBLICATIONS **50** CITATIONS
SEE PROFILE

Florian Holzapfel
Technische Universität München
561 PUBLICATIONS **3,089** CITATIONS
SEE PROFILE

Fast Decoding of ARINC 717 Flight Data Recordings

Florian Schwaiger*, Florian Holzapfel†
Institute of Flight System Dynamics, Technical University of Munich, Garching b. München, 85748, Germany

To support big data applications for flight data analysis, we present an improved method for the decoding of binary ARINC 717 recorder files. Our tool transcribes flight parameters from any given dataframe layout of the raw file into one of several tabular open data standards (e.g. .parquet, .csv). Our improved method cuts execution time by a factor of up to 61 and memory utilization by a factor of up to 15 – compared to the previous implementation presented in 2016. It basically removes the performance bottleneck from the data intake stage for big data cloud platforms, as demonstrated in the SafeClouds.eu project. It can further decode much larger files than before, e.g. 2 GB on a desktop PC system, thanks to several performance optimization techniques. Thus, we can more easily parallelize and distribute the tool to remote compute clusters. Further, the tool provides new features, such as the estimation of the most important flight parameters from an undefined dataframe layout. Beyond simple detection of corrupt subframes, the tool can repair some parameters on bit level, e.g. fixing the alignment of multipart parameters (e.g. GPS position) or unrolling wrapped values (e.g. radio altitude).

Keywords: ARINC 717, FDM, FOQA, QAR, FDR, Matlab, Flight Data Analysis

I. Introduction

Inspecting recorded flight data is an important aspect of Flight Data Monitoring (FDM), also known as Flight Operations Quality Assessment (FOQA) programmes. Data are typically recorded and downloaded from the Quick Access Recorder (QAR). For most aircraft, its file format follows the ARINC 717 standard [1]. There are only few commercial tools to read these files that we are aware of, typically provided in the context of service contracts directly from FDM providers. To support flight data analysis in the scientific context, Wang et. al. have previously implemented a decoding tool [2] that has been used in several flight data analysis studies since.

Since then, we have been progressing on computational efficiency and robustness to make sure that data intake will not be a bottleneck in future data-centric applications. Our proposed new tool has proven to be fast enough that we no longer need to cache decoded flight data in temporary files. We can now load flight parameters from the raw ARINC 717 files directly into memory when needed.

In the form of a containerized application, our decoding tool can also be used as an enabler for open data sharing platforms around flight data. We have demonstrated this in the scope of the `SafeClouds.eu` project (section IV.C). Overall our new tool brings the following improvements that will be described in this paper:

- speed improvement of up to 61× and support for parallelization (section IV.A)
- support for a variety of file format variants (section III.A)
- robust design to deal with subframe corruption (section III.C)
- a concise way to configure dataframe layouts and features (section III.D)
- a model-based process to read files, without having any dataframe descriptor (section III.E)
- error detection and repair for misaligned parameter parts (section III.F)
- error detection and repair for incomplete data (section III.G)

II. The ARINC 717 File Format

The ARINC 717 standard [1] describes how flight data must be captured and recorded on board of an aircraft. Among others, the document describes the format of the data stream that will be written to disk / solid state memory. For recording, the official standard captures only basic features – in the past we have identified some more. We therefore refer to other publications describing further specifics of the data format. These publications include ATR 2016 [3],

*Research Associate, Institute of Flight System Dynamics, f.schwaiger@tum.de
†Professor, Head of the Institute of Flight System Dynamics, florian.holzapfel@tum.de, AIAA Associate Fellow

Wang et. al. 2016 [2], Wang 2019 [4] and Sembiring 2020 [5]. The following paragraphs not only describe the data structure from ARINC 717 but also include findings made throughout our studies.

ARINC 717 files can be understood as a binary tabular format. Each table cell is a 12 bit word that contains data $\in \{0, ..., 4095\}$. The columns are determined by *words per second* $\in \{64, 128, 256, 512, 1024\}$. Each row is formed by a *subframe*, that captures one second of recording time. In the file, these form a large block of (unaligned) binary data. To align the columns properly, the standard defines four distinct 12 bit *synchronization words* with maximal Hamming distances* of 11–12, namely `0x247`, `0x5B8`, `0xA47` and `0xDB8`. The first word of each subframe contains one of the four synchronization words (in the figures printed in bold font). Each subsequent subframe contains the next sync word, so they are used in a circulating pattern. They are named subframes 1 (`0x247`) to 4 (`0xDB8`) for that reason. Each group of subframes 1 to 4 aggregates in what is known as *frame*. This implies, that each frame records 4 seconds. This pattern is visualized in Fig. 1.

	1	2	3	4	...	wps − 3	wps − 2	wps − 1	wps	
Frame 1	**0x247**	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
	0x5B8	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
	0xA47	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
	0xDB8	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
Frame 2	**0x247**	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
	0x5B8	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
	0xA47	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
	0xDB8	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
...	**0x247**	0x???	0x???	0x???	...	0x???	0x???	0x???	0x???	1s
	...	...	...	...	...	...	...	...	...	

Fig. 1 12 bit *words* form *words-per-second* (wps) columns and aggregate row by row into *subframes*, and according to their four sync words into *frames*

Another feature that is not captured in the official ARINC 717 standard [1], but in other publications such as a training document from ATR [3, p. 35], is an overarching *superframe* structure. If a file has a superframe structure, it typically exposes what we call a *frame counter word* $\in \{0, ..., 4095\}$. The frame counter is commonly stored in one of the last words but can really be anywhere except the first word that is reserved for sync words. It can also be in any of the four subframes. An example is shown in Fig. 2. If the file has a superframe structure, values can also be sampled at every 16th frame (i.e. every 64th subframe, so roughly every minute). Only the frame counter word tells us reliably how to align the frames into superframe blocks. When the modulo operation on the frame counter yields zero (i.e. $framecounter \,\%\, 16 = 0$), a new superframe starts. We have not seen any larger repetition pattern over superframes so far.

Since the largest occurring pattern is the *superframe* structure, no parameter can repeat less often than this pattern. That means the lowest possible sampling rate for a parameter is $1/64$ Hz. In all other cases, parameters can store values either in each subframe in the same word (1 Hz), or leave out every other subframe (e.g. in subframes 1 & 3, or 2 & 4, making it 0.5 Hz). Parameters at 0.25 Hz only store values in a single word in each frame. Parameters with a sampling rate larger than 1 Hz can store values in repeated words that are equally spaced across each subframe. Their sampling rate must be a power of two (2^n Hz, $n \in \mathbb{N}$).

In some cases, storing parameters in 12 bit words does not provide sufficient accuracy. An example is the aircraft position in WGS84 coordinates, that must cover a range of 180 degrees for the latitude (respectively 360 degrees for the longitude). Recording these ranges across 12 bit provides 4096 unique values each, and the max position round-off error is no more than $1/2$ the distance between any discrete values, which at the equator would be approximately 5.5 kilometers, see Eqs. (1)–(3), where the radius of the earth is approximated by $r_E \approx 6371$km.

$$\Delta\lambda_{\text{error,max}} = \frac{1}{2}\frac{360^\circ}{2^{12}}\frac{\pi}{180^\circ} \tag{1}$$

*the "Hamming Distance" is the number of differing bits between two binary words

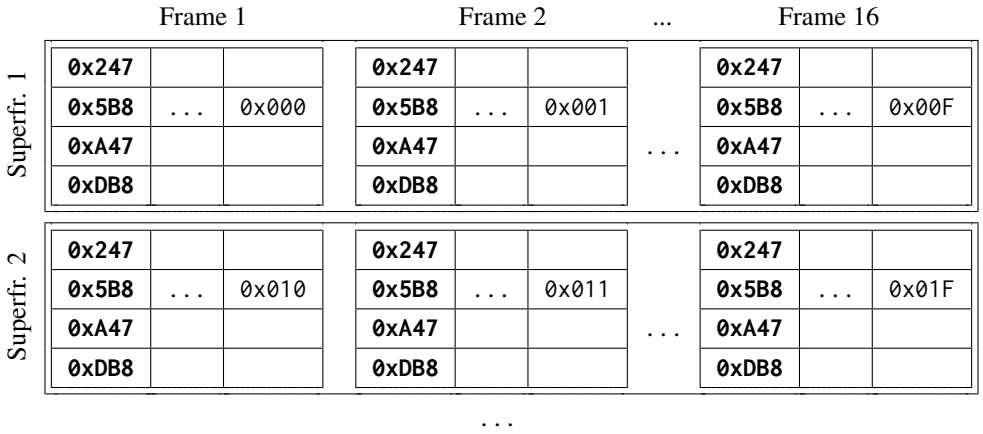

Fig. 2 ***frames*** **aggregating into** ***superframes*** **(shown horizontally `0x??0` .. `0x??F`) according to the modulo of the** ***frame counter word*****, shown in bold font. Here, the frame counter is in subframe 2, just as an example.**

$$\Delta\mu_{\text{error,max}} = \frac{1}{2}\frac{180^\circ}{2^{12}}\frac{\pi}{180^\circ} \tag{2}$$

$$\Delta p_{\text{error,max}} = r_E\sqrt{\Delta\lambda^2_{\text{error,max}} + \Delta\mu^2_{\text{error,max}}} \approx 5.5\text{km} \tag{3}$$

To improve recording accuracy, the dataframe developer can split a parameter value across multiple words, e.g. 12 + 6 = 18 bits (latitude) or 19 bits (longitude) in neighbouring locations. Following the same error estimation, the maximum position error can be reduced to 53 meters, sufficient to associate the location of the aircraft with a specific runway or taxiway, but not quite close to pinpoint the touchdown location on the runway. Our tool supports the aggregation of multiple "parameter parts" into combined values via a unified configuration structure, see section III.D.

With the same configuration structure, we also support the decoding of binary coded decimals (BCDs), where each digit 0–9 typically spans 4 bit, but we have seen less bits used for special cases. Such a special case could be the encoding of an ILS frequency, which spans only 108.10–111.95 MHz. The 100 MHz digit is fixed and does not need to be recorded, the 10 MHz digit can only be 0 or 1, needing a single bit to record. Conclusively, the frequency can be recorded in 1 + 4 + 4 + 3 = 12 bits in BCD notation and fits into a single word.

Finally, our tool further recognizes 7 bit ASCII characters in the data and will decode strings from characters spread across multiple word and bit locations.

III. Improvements on Decoding

This section lays out the various improvements to our decoding algorithm since its conception in [2].

A. Memory Layout

In ARINC 717, data are stored in 12 bit words. However, CPUs can only handle word sizes that are a power of two. The shortest adressable unit is the *byte* (8 bit). Recorded 12 bit data can thus fit easily into a 16 bit unit. However, in our previous method, data were indexed at the bit level. Bits were stored as bytes, allocating 7/8ths of unused memory. Encoded values where then indexed as individual bits, and assembled in a linear combination with powers of two (section III.B). Instead, we now handle "words" as the smallest unit, reducing memory consumption by a factor of at least six[†] (section IV.A) and making the whole process more robust. While reducing peak memory, we could also reduce memory bandwidth alongside, speeding up the process. With the reduced memory footprint, a single machine can now also decode multiple flights in parallel.

We have seen from experience, that recorded data might already come stored in a 16 bit word layout. That is, the 12 bit content is padded with four zeros above the most significant bit (MSB) – e.g. `0x247` will be stored as `0x0247`. Depending on the endianness of the file, byte order might also be inverted like `0x47 0x02`. On these files, we use bit masking to `0x0FFF` to make sure no erroneous values in the 4 MSBs pollutes the data, but keep the overall 16 bit words.

[†] $6\times \mathrel{\widehat{=}}$ (8 alloc/1 used)/(16 alloc/12 used), but can be more because of further changes to the implementation

In other files we have seen a packed dataframe where no padding bits are inserted. We can read these "12 bit packed files" byte-by-byte, store bytes into 16 bit `words`, and regroup every three bytes into two 16 bit words of 12 bit data (Algorithm 1). Using a scripting language that supports vectorization, this operation is more efficient than our previous approach of storing and expanding individual bits as bytes.

Algorithm 1: unpacking 12 bit packed data, $\ll$ and $\gg$ indicate bit shifts, `0x??` hexadecimal values

Data: words $w_i \in$ `int16`, containing 8 bit data, $i \in \{1, ..., N\}$
Result: words $w_i \in$ `int16`, containing 12 bit data

$w_{k+0} \leftarrow ((w_{k+0} \wedge$ `0xFF`$) \gg 0) \vee ((w_{k+1} \wedge$ `0x0F`$) \ll 8), \quad \forall k \in \{1, 4, 7, ..., N\};$

$w_{k+1} \leftarrow ((w_{k+1} \wedge$ `0xF0`$) \gg 4) \vee ((w_{k+2} \wedge$ `0xFF`$) \ll 4), \quad \forall k \in \{1, 4, 7, ..., N\};$

$w_{k+2} \leftarrow [\,], \quad \forall k \in \{1, 4, 7, ..., N\}$; `% drop every 3rd element`

B. Vectorized Indexing

Once we have data in 16 bit words, we reshape and sort data into a 4D data structure as was done in [2]. In the latest version, writing the algorithm out in *Matlab*, pure vectorized commands are used, speeding up the process. In general, for execution performance, our main goal was to minimize the occurrence of explicit loops and branching instructions.

The 4D data structure is used to index parameter parts easily. Using one-based indexing notation, the dimensions are:

1) word (index $\in \{1, ..., \text{words per second}\}$)
2) subframe (index $\in \{1, ..., 4\}$)
3) frame (index $\in \{1, ..., 16\}$)
4) superframe (index $\in \mathbb{N}$)

When indexing into this data structure, the order of these dimensions is important and depends on which language is used. In *Python* with `numpy`, values are stored in a column-major layout. That implies that the "fastest-changing," or innermost dimension (i.e. " word") is last and "superframe" first. In *Matlab*, matrix values are row-major. Here, "word" is the first dimension. If done right, one can linearily index all dimensions in a single instruction. Some examples are shown in Algorithm 2.

Algorithm 2: examples for the linear indexing of parameters with the 4D data structure

Data: words $\in$ `int16[M, 4, 16, N]`

```
words([64, 128, 192, 256], :, :, :) ;                    % parameter with 4 Hz
words([128, 256], :, :, :) ;                             % parameter with 2 Hz
words(256, :, :, :) ;                                    % parameter with 1 Hz
words(256, [1, 3], :, :) ;                              % parameter with 0.5 Hz
words(256, 3, :, :) ;                                   % parameter with 0.25 Hz
words(256, 3, 7, :) ;                          % superframe parameter with 1/64 Hz
```

The algorithm can also deal with files that do not have a superframe structure at all. In any case, it reshapes data into a fake 4D structure, where only dimensions 1 and 2 are assigned. 3 and 4 are generated from counters and their indexers are omitted when accessing data.

C. Masking Corrupted Data

Data can be corrupted on a subframe level. With each subframe having a distinct synchronization word, we can keep track if there are *missing*, *corrupted* or *duplicate* subframes. According to Jesse [6] we must expect all of those errors.

When we bring data to its 4D shape, we first determine how many valid subframes there are and into how many frames they aggregate. We then allocate just enough uninitialized memory to store everything, including all missing subframes. Then, we assign only the valid subframes, according to their sync-words and leave corrupted sections in their unknown state. Instead, we track an "integrity mask" alongside, that marks which subframes, frames and superframes have been assigned successfully. These again are only vectorized operations.

With the `words` array and integrity mask `valid`, we can easily extract parameter values and fill in corrupt values with *not-a-number*, see Algorithm 3.

Algorithm 3: masking corrupt values with *not-a-number*

Data: words ∈ `int16[M, 4, 16, N]`, valid ∈ `boolean[4, 16, N]`
Result: values: double
1 values ← (double) words(42, 3, 7, :);
2 mask ← ¬valid(3, 7, :);
3 values(mask) ← NaN;

Experience shows that many parameters have two or more distinct locations in the dataframe (e.g. aircraft position in 12 + 6 bits). In that case, the masking procedure will be applied to each parameter part individually. Then the joint parameter value will only be valid, if all parts are valid. The mask can span multiple subframes, and it is not uncommon that particular subframes are corrupt, invalidating values with parts from other, valid subframes, see Algorithm 4.

Algorithm 4: masking joint parameter corrupt values with *not-a-number*

Data: words ∈ `int16[M, 4, 16, N]`, valid ∈ `boolean[4, 16, N]`
Result: values: double
1 part1 ← (double) words(42, 3, 7, :);
2 part2 ← (double) words(42, 4, 7, :);
3 mask1 ← ¬valid(3, 7, :);
4 mask2 ← ¬valid(4, 7, :);
5 values = part1 + part2;
6 values(mask1 ∧ mask2) ← NaN;

D. Dataframe Specification

In the initial design for the tool [2], we used a relational database server to store the dataframe layouts. There are tables for each dataframe, for each parameter and for each parameter part location (because of multipart parameters). Since the overall structure is very much tree-like, we decided to abandon this relational form of dataframe description in favor of hierarchical markup languages such as XML or YAML. These text-based dataframe descriptor documents can also be easily stored in our version control system.

We load the dataframe document into an object-oriented data structure, where each object has many sensible default properties already. The configuration structure looks as illustrated in Fig. 3. On the top level, the user specifies a `Dataframe` object, that defines the overall file structure through `WordsPerSecond`, the memory layout and a list of `Parameter` definitions. Each `Parameter` again consists of 1..∗ `Parts`. The acutal `Word`, `Subframe`, `Frame` and decoding configuration is linked with `Location` only. Most parameters will have exactly one location, such as can be seen with the `FrameCounter` property.

You can see many default values specified in Fig. 3 block `Location`. From experience most parameters start out as unsigned 12 bit scaled values recorded at 1 Hz. In the dataframe document, you only have to specify any deviations from the default. You can use the properties given here, e.g. `Bits` which is a 12 bit bitmask, or use the dependent accessor properties that simplify some config tasks. From experience again, dataframe documents typically specify a least and most significant bit for the recorded range. However, our code runs faster using bit masking logic, so the single source of thruth is `Bits`. `LSB` and `MSB` are calculated on-the-fly when you build the dataframe structure.

The same goes for the `Polynomial` configuration. Raw bit values are extracted from the data stream first, then typically linearily scaled and offset. We have seen dataframes however, that specify a nonlinear polynomial scaling on the raw value. For a quadratic profile, you would specify `Poly = [1, 0, 0]` to scale a binary value b by $1b^2 + 0b + 0$. You can also use this feature to invert logical values specifying `Poly = [-1, 1]`, e.g. for air/ground switches or spoiler extension.

Some more examples are given in Listing 1, as they would be specified in a YAML configuration file. Here we demonstrate how the setup would look like for an inverted parameter `LDG_GEAR_NOS_UP`, a superframe parameter

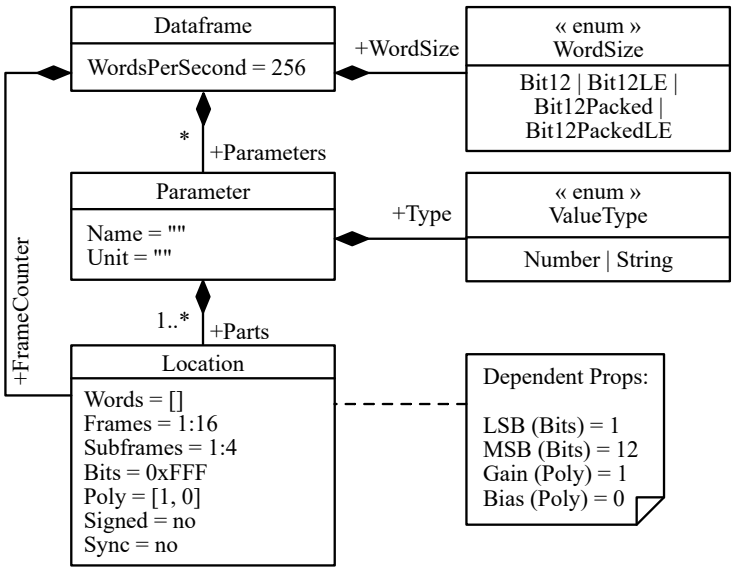

Fig. 3 Object-oriented design of the dataframe configuration – this is the minimal set we found most useful to specify almost any parameter with ease. We found that any added properties would only add redundancy.

`GROSS_WEIGHT`, a parameter with 8Hz sampling rate `ANGLE_OF_ATTACK`, a BCD parameter `ILS_FREQUENCY` and a parameter with our bit synchronization feature `GPS_POS_LAT` (section III.F).

Listing 1 Examples for some parameters in YAML configuration, exemplifying the different options

```
- name: LDG_GEAR_NOS_UP                                          # single inverted bit
  part: {word: 42, bits: 0b000000000100, poly: [-1, 1]}
- name: GROSS_WEIGHT                                                # every 16 frames
  unit: lbs
  part: {word: 43, subframe: 3, frame: 7, poly: [40, 0]}
- name: ANGLE_OF_ATTACK # with offset from zero
  unit: deg
  part: {word: '44:64:256', poly: [0.0023, 0.3], sign: yes}
- name: ILS_FREQUENCY                                               # BCD as 1xx.xx
  unit: MHz
  part:
    - {word: 46, bits: 0b100000000000, poly: [10.00, 100]}
    - {word: 46, bits: 0b011110000000, poly: [01.00,   0]}
    - {word: 46, bits: 0b000001111000, poly: [00.10,   0]}
    - {word: 46, bits: 0b000000000111, poly: [00.01,   0]}
- name: GPS_POS_LAT                                                 # using bit sync
  unit: deg
  part:
    - {word: 127,          poly: [0.00068665, 0]}
    - {word: 128, LSB: 7, poly: [2.8125,     0], sign: yes, sync: yes}
```

E. New Feature: Dataframe Estimation

Dataframe layouts can be loaded either from YAML, XML, INI or directly from code. We are also able to estimate the dataframe for some typical parameters (position, altitude, ground speed, air speed, track, vertical acceleration, vertical speed) from an arbitrary ARINC 717 file [7]. The process of estimating the dataframe is iterative, where partial solutions contribute to the estimation of other file features:

1) First, we estimate the block structure and encoding of the given ARINC 717 file. We can do this by quickly brute-forcing combinations of `WordSize` and `WordsPerSecond` on a small subset of the file. Knowing the

structure, we can already bring the file into 3D structure (M, 4, N).

2) We can then quickly brute-force the detection of the `FrameCounter`, because we can expect a value at the same subframe / frame position to increment every 16th frame. With the frame counter detected, we can bring data into 4D (M, 4, 16, N) shape.
3) From the 4D shape, we look at all the words in the first dimension M. We compute the four different statistical moments expectation $E[X]$, variance $Var[X]$, skew $S[X]$ and kurtosis $K[X]$ of each data point, assuming they repeat every superframe (64 subframes) only, because this is the minimal repetition pattern. This gives us four indicators for $M * 4 * 16$ data arrays of length N each.
4) We apply a clustering method on the data points from the previous step. If a parameter has a recording frequency larger than 1/64 Hz, we can identify point clusters.
5) Knowing about the probable repetition patterns, we can write a temporary dataframe to extract consistent parameters, though we do not know the contents of any parameter yet.
6) Using the structure as a basis, we use pattern detection from a pre-trained algorithm to identify basic parameters such as ground speed, air speed, latitude, longitude and pitch angle.
7) In a second (and third) pass, we can use successfully decoded parameters to create models for others, e.g. track from position or accelerations from speed.
8) We can also identify the jump bits (if existent) that complement a wrapped 12 bit maxed out parameter range if the parameter was originally split across two words, see Fig. 4.
9) We can finally fix the scaling of the detected parameters under the assumption that angular parameters are encoded in fractions of 360 degrees and others typically in powers of two. For latitude / longitude in particular, we use a global database of airport locations to find the best match of scaling factors so that the route is anchored at existing airports. This does not affect precision if the dataframe was setup properly with the scaling being a real power fraction $360/2^n, n \in \mathbb{N}$.

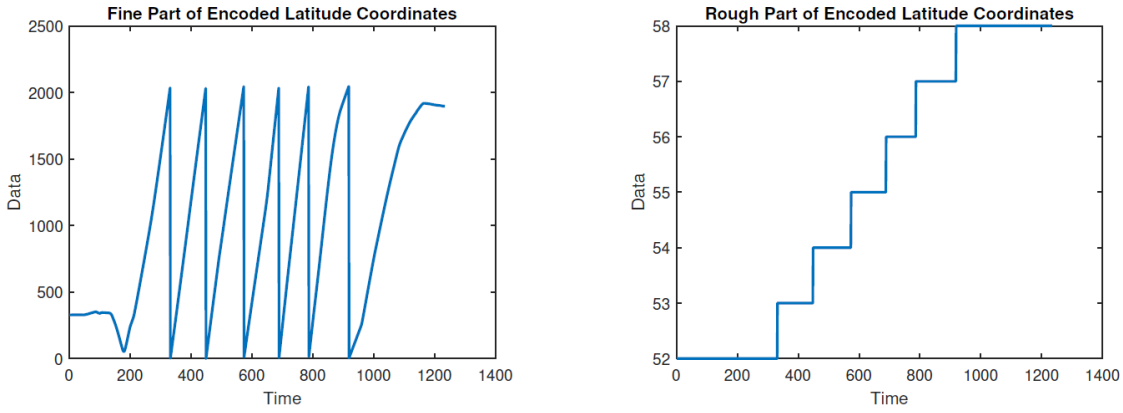

Fig. 4 Automatically detected coarse data (right side) going along with the fine data (dashed line left side), blue line is raw data over time [7]

With that feature, we can perform simple studies without recieving the proper dataframe specification from the airline alongside the data. A possible scenario could be to package and distribute an algorithm to airlines, that computes e.g. weather information from basic aircraft movement based on raw data, without the need of configuration or maintenance. Finally, this feature has also already proven useful when setting up dataframes with incomplete descriptions from the original documentation.

F. New Feature: Alignment of Parameter Parts

From experience, some parameters such as GPS latitude and longitude are recorded with high precision requiring more than 4096 distinct values (12 bit). Those parameters are typically split into a coarse and fine part, as shown before in the previous section in Fig. 4. We have seen, that in some dataframes the coarse part has a reduced sampling rate (e.g. 0.25 Hz) to save memory, since it does not contribute to the overall precision (fine part with 1 Hz). If we multiply both the coarse and fine parts according to their dataframe specification, we can identify outliers, see Fig. 5 left.

We could now proceed applying outlier detection and replacing these values with *not-a-number*. However, since

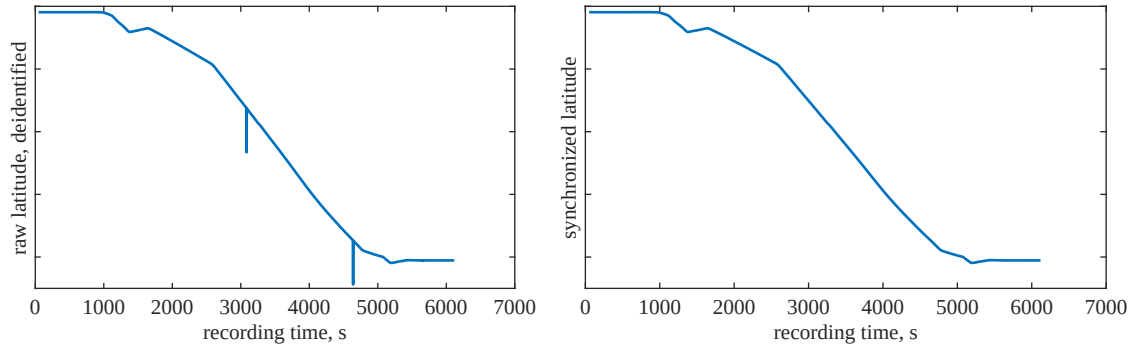

Fig. 5 left: sum of misaligned parameter parts leading to outliers, right: synchronizing the alignment of parameter parts resolves the outliers

we know that these are caused by a slight misalingment of both parts, we can shift individual offending data points in the coarse part in time, so that the reset in the fine signal corresponds precisely to the jumps in the coarse signal (Fig. 5 right). Our method is described in Algorithm 5. This feature can be enabled for any part individually using the `sync:yes` option in the YAML configuration (section III.D).

Algorithm 5: Updating the coarse part to match the jumps in the fine part of the signal, $\ll$ and $\gg$ indicate bit shift operations, `0x??` numericals in hexadecimal format

Data: coarse part $c_i \in$ `int16`, fine part $f_i \in$ `int16`, $i \in \{1, ..., N\}$
Result: updated coarse part c_i

$n \leftarrow \text{MSB}_{fine} - 2$;
$j \leftarrow \{i \; \exists \; |(f_i \gg n \wedge 0\text{x}03) \neq (f_{i-1} \gg n \wedge 0\text{x}03)|, \quad \forall i\}$; `% index where fine part resets`
$w \leftarrow \{k \; \exists \; c_{k+1} = c_k, \quad \forall k \in j\}$; `% index where jump is misaligned`
$r \leftarrow \{k \; \exists \; (k > 1) \wedge (k < N - 1), \quad \forall k \in j\}$; `% index where jump is repairable`
for $k \in j \cap w \cap r$ **do**
  if $c_k \neq c_{k-1}$ **then**
    $c_k \leftarrow c_{k-1}$; `% correct jump backwards`
  end
  else if $c_{k+1} \neq c_{k+2}$ **then**
    $c_{k+1} \leftarrow c_{k+2}$; `% correct jump forwards`
  end
end

G. New Feature: Unrolling of Wrapped Parameters

We also found from experience that some parameters are cut off after 12 bit with no coarse part existing. Typically, this affects the radio altitude. Radio altitude needs to be precise only near-ground, so it appears many dataframes are configured to record $\{0, ..., 4095\}$ feet of altitude and then wrap back around. We can unroll those parameters by detecting jumps larger than half their maximum (e.g. 2048) value, then offsetting them by their max. range (e.g. 4094) to minimize the jump.

For the radio altitude, we also have to take into account the possibility, that the parameter wraps into negative values, though it is recorded as an *unsigned* parameter over the full $\{0, ..., 4095\}$ range. This is the reason why in some CSV exports, you might see outliers of the value while the aircraft is on the ground. However we do know about radio altitude that it should be close to zero when on ground – and since files are usually cropped according to flights, they start and end recording at parking. In Algorithm 6 we introduce zero at the beginning and end of the signal timeseries, describing our assumption of the radio altitude being close to zero. Then we unroll half the signal from the beginning to the center and the second half backwards from the end to center. This ensures proper unrolling at take-off and landing, but allows a signal jump during cruise where we don't read the signal anyway. Figure 6 demonstrates how the process affects the

radio altitude.

Algorithm 6: Unrolling the radio altitude in two parts, from zero at the beginning to the center and from zero at the end backwards to center

Data: radio altitude $h_i \in$ `double[N]`, $i \in \{1, ..., N\}$
Result: unrolled radio altitude h_i

$h_0 \leftarrow 0$; `% insert zero at beginning and end`
$h_{N+1} \leftarrow 0$;
$\Delta h_i \leftarrow h_i - h_{i-1}, \quad \forall i$; `% determine offset for the correction`
$\Delta h_i \leftarrow 0 \quad \exists \quad |\Delta h_i| \leq 2048 \vee \Delta h_i \equiv NaN, \quad \forall i$;
$\Delta h_i \leftarrow \text{sign}(\Delta h_i) \cdot 2048, \quad \forall i$;
$h_i \leftarrow h_i - \sum_0^i \Delta h_k, \quad \forall i \in \{1, ..., N/2\}$; `% apply from both ends towards center`
$h_i \leftarrow h_i + \sum_N^i \Delta h_k, \quad \forall i \in \{N, ..., N/2\}$;

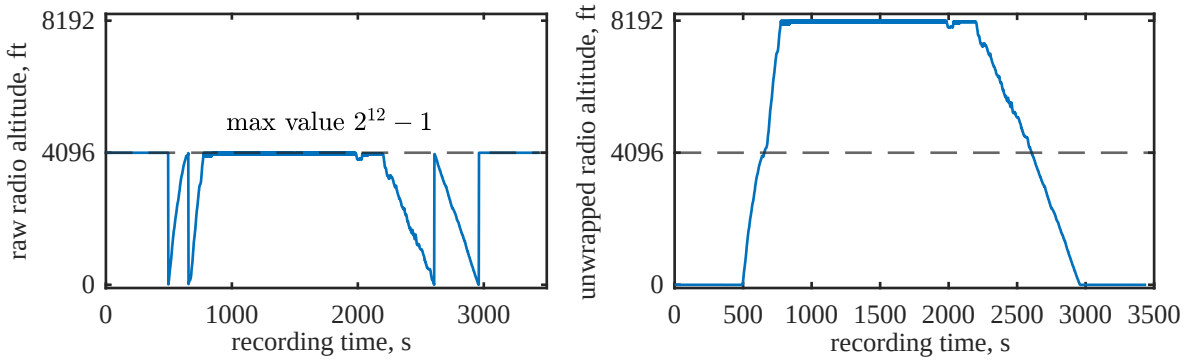

Fig. 6 left: radio altitude wrapping around at 4096, right: unwrapped radio altitude based on ground condition at beginning and end of flight, i.e. values are constrained to be close to zero

IV. Application

From the original tool [2] we have since progressed with two revisions. Version 2 has been a rewrite in *Python* from the original binary file handling to the improved memory layout. This version has since been in use with the *DataBeacon* infrastructure from the SafeClouds project [8], see section IV.C. Now we have version 3 in *Matlab* that has proven very efficient in our local data analysis jobs.

The current tool can also be containerized, so it can run on compute clouds. This would enable us to provide the tool in a software-as-a-service (SaaS) context. So far we have demonstrated a successful integration as a Docker[‡] container.

A. Performance Benchmark with Previous Implementation

We have compared the execution time and memory utilization for both the original and updated tools. The results can be seen in Table 1. The table shows "old" (previous implementation) versus "new" (current) decoding time and memory utilization for each dataframe. The values are the mean per flight, computed over the same 10 flights each.

We can see how the overall runtime benefit depends on file size and can be up to 61 times. At the same time memory consumption drops to acceptable levels. We show both allocated and peak memory, because memory bandwith has a direkt influence on runtime. For some measurements marked (*), the memory profiler produced unreliable results for "peak memory" (larger than 32 bit max value), so we do not trust "allocated memory" without closer inspection yet either. Looking at the profiling reports closer makes it seem likely that physical memory was maxed out and swap memory was triggered. This would explain the large speedup factors.

[‡] `https://www.docker.com/`

Table 1 Performance comparison for the original and revised decoder tool. Measurements include the processing of 10 arbitrary flights from our database per dataframe, to average results. Lower execution time / memory allocation is better. Measurements marked (*) were unreliable because slower swap memory appears to have been used.

Dataframe	Size avg. / max.	Version	Average Time	Allocated Memory	Peak Memory	Speedup
A319	4.2 MB / 6.7 MB	old	2.70 s	29.1 Mb	10 Mb	**6**×
		new	0.49 s	24.7 Mb	9.2 Mb	
A320	4.7 MB / 6.8 MB	old	22.3 s	418 Mb	446 Mb	**37**×
		new	0.60 s	27.0 Mb	9.4 Mb	
A321	5.1 MB / 7.5 MB	old	22.1 s	455 Mb	489 Mb	**29**×
		new	0.76 s	29.2 Mb	10.2 Mb	
A330	22 MB / 28 MB	old	88.0 s	1,114 Mb (*)	??? (*)	**61**×
		new	1.45 s	170 Mb	37.9 Mb	
A343	18 MB / 32 MB	old	65.8 s	800 Mb (*)	??? (*)	**48**×
		new	1.38 s	143 Mb	43.4 Mb	
A346	25 MB / 32 MB	old	96.1 s	112 Mb (*)	??? (*)	**61**×
		new	1.58 s	191 Mb	43.4 Mb	
A380	46 MB / 60 MB	old	128 s	1,194 Mb (*)	??? (*)	**8**×
		new	16.2 s	322 Mb	91.5 Mb	
B748	60 MB / 67 MB	old	79.3 s	50.5 Mb (*)	??? (*)	**33**×
		new	2.40 s	228 Mb	81.4 Mb	

Unfortunately, due to data confidentiality agreements, as of the time of writing we do not have access to any files larger than 1 GB containing multiple flights. We thus cannot compare the data intake capacity of the new tool based on any real recordings. We can however concatenate a single smaller file multiple times, until an arbitrary capacity is reached. Table 2 compares how the algorithms scale with file size. We found that our proposed new implementation enables the user to decode far larger files than before, where we marked file sizes leading to out-of-memory errors as (**).

Table 2 Similar to Table 1, but measurements include the processing of one arbitrary flight in a 512 words-per-second dataframe, its contents repeated multiple times to reach the desired target file size shown in the first column. Measurements marked (*) were unreliable in the profiling report. Measurements marked (**) failed due to an out-of-memory (OOM) error.

Dataframe	File Size	Version	Time	Allocated Memory	Peak Memory	Speedup
A319	5 MB	old	41 s	475 Mb	454 Mb	**24** ×
		new	1.7 s	7.7 Mb	6.8 Mb	
	50 MB	old	1,100 s	428 Mb (*)	??? (*)	**333** ×
		new	3.3 s	71 Mb	68 Mb	
	500 MB	old	OOM (**)	OOM (**)	OOM (**)	**enabled**
		new	19 s	698 Mb	683 Mb	
	1000 MB	old	OOM (**)	OOM (**)	OOM (**)	**enabled**
		new	36 s	??? (*)	??? (*)	
	2000 MB	old	OOM (**)	OOM (**)	OOM (**)	**enabled**
		new	71 s	??? (*)	??? (*)	

B. Flight Splitting

Since the raw QAR data files are mostly just dumps from recorder memory, it is not uncommon to encounter raw files that include more than a single flight. In our work on flight data analyses however, we usually want to process flights one-by-one. With our previous decoder generation, reading raw files containing possibly many flights was a challenge, hence Wang et. al. proposed in [2] a method to split large files into smaller files containing a single flight each, prior to fully decoding all parameters. With the improved performance of our new tool however, the limitation of having to split files prior to decoding is gone.

Hence, we propose to remove file splitting logic from the decoding stage altogether, and give the user access downstream to all the decoded parameters, to decide where to split the parameter timeseries into individual flights. In our case, we use a state machine to trace flight phases, where transitions are given by certain parameter conditions. We can then split flights on every transition from the "engine shutdown" to the "startup" phase. Or, as in our most recent work on Unstable Approach (UA) analysis [9], pick out all "approach" and "final approach" phases to process individual landing attempts, where we do not care about the specific flight instances.

C. Flight Data Collection in SafeClouds

The decoder tool has been integrated so far in the FDM data pipelines for the SafeClouds.eu project [8]. The concept for the "DataBeacon" platform developed in this project was that the decoding runs locally on airline premises (local computers) and only decoded and deidentified data are transferred to the cloud. So we packaged the tool as precompiled binaries to run remotely. Overall, we have decoded and analysed roughly 600,000 flights from various aircraft vendors and FDM providers, with a total of 20 different dataframes and 100 parameters on average. We refer the interested reader to [8–11] for more information on the case studies we've enabled through this tool.

V. Conclusion

In summary, we have created a fast and reliable tool for scientific studies on raw flight data. The tool has saved us valuable time, working with large amounts of flight data from several airlines. This is especially useful to label many flights to be used as a training set for machine learning algorithms. We have demonstrated this with the SafeClouds.eu project and ongoing student projects. Beyond the usual application as a tool for data ingestion, the process for the estimation of unknown dataframes enables us to offer basic case studies on QAR data that we haven't even seen before.

References

[1] Aeronautical Radio Inc, *Flight Data Acquisition and Recording System: ARINC Characteristic 717-15*, 2011.

[2] Wang, C., Mohr, N., and Holzapfel, F., "Decoding of Binary Flight Data in ARINC 717 Format," *1st International Conference in Aerospace For Young Scientists (ICAYS)*, Beijing, 2016.

[3] ATR, "Flight Data Monitoring on ATR Aircraft," 2016. URL https://www.easa.europa.eu/sites/default/files/dfu/16T0153_ATR_FDM_2016.pdf, retrieved on 2020/10/20.

[4] Wang, C., "Model-Based Quantification of Accident Probabilities from Operational Flight Data," Dissertation, Technical University of Munich, Munich, 2019. URL http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20190820-1472237-1-7.

[5] Sembiring, J., "A Heuristic Parameter Estimation Approach for FDM Data," Dissertation, Technical University of Munich, Munich, 2020. URL http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20200630-1519249-1-0.

[6] Jesse, D., "Eight Repeated Subframes or Going Under the Knife," 2014. URL http://www.flightdatacommunity.com/eight-repeated-subframes-going-under-the-knife/, retrieved on 2018/06/28.

[7] Hink, N., and Schubert, S., "Decoding Flight Data Recordings without Schema," Interdisciplinary Project, Technical University of Munich, Munich, 2019. Supervised by F. Schwaiger. Unpublished.

[8] Schwaiger, F., "FDM Decoding and Data Analysis: Techniques and lessons learnt from SafeClouds.eu," *SAFE360 2019*, Brussels, 2019. URL https://www.easa.europa.eu/newsroom-and-events/events/safety-aviation-forum-europe-safe#group-easa-event-proceedings.

[9] Walter, D., "Development of an Unstable Approach Detection and Prediction Model," Master's Thesis, Technical University of Munich, Munich, 2020. Supervised by F. Schwaiger and L. Beller. Unpublished.

[10] Martínez, D., Fernandéz, A., Hernandéz, P., Cristóbal, S., Schwaiger, F., Nunez, J. M., and Ruiz, J. M., "Forecasting Unstable Approaches with Boosting Frameworks and LSTM Networks," *9th SESAR Innovation Days*, 2019. URL https://www.sesarju.eu/sesarinnovationdays.

[11] Martínez, D., Fernandéz, A., Hernandéz, P., Cristóbal, S., Schwaiger, F., Nunez, J. M., and Ruiz, J. M., "Flight Data Monitoring (FDM) Unknown Hazards detection during Approach Phase using Clustering Techniques and AutoEncoders," *9th SESAR Innovation Days*, 2019. URL https://www.sesarju.eu/sesarinnovationdays.