Version Control

Ver.	Date	Summary of changes
1.0	2026-05-20	Initial release
1.01	2026-05-29	Fix “alt” field description in Section 4.1

Scope and Summary

This document describes the the FLARM FAMP Public Protocol (FFPP). FAMP provides the traffic awareness and safety functions for the FLARM network. Use of this document is subject to the Terms and Conditions set forth in Annex 1.

Table of Contents

1 Introduction

This document describes the FLARM FAMP Public Protocol. FAMP supports the following applications:

- Situation awareness
- Traffic monitoring
- Collision avoidance
- Tracking
- Identification

Other uses, payloads, and supplementary protocols of the FLARM radio system are outside of the scope of this document but may depend on this.

Use of this document is subject to the Terms and Conditions set forth in Annex 1.

1.1 Bit and Byte Order

The following principles apply unless noted otherwise.

- In alignment with standard notation, hexadecimal numbers are prefixed with '0x', binary literals with '0b'.
- Byte size: A byte consists of 8 bits.
- Byte addressing: Bytes are indexed in the order of transmission, i.e. byte 0 is transmitted first.
- Bit addressing: The least significant bit has index 0 and a weight of 0x01, while the most significant bit has the index 7 and a weight of 0x80 (128 decimal). Bit 26 thus is bit 3 (0x04) of byte 3.
- Byte order: Little Endian byte order is used unless noted otherwise. That is, values overlapping byte boundaries are transmitted on the RF with the **least significant part first.**
- Bit order: Individual bytes are transmitted **most significant bit first**. This is usually a property or configuration of the transceiver chip.

Example 1: The integer 1328922 (0x14471A) is transmitted as 0x1A, 0x47, 0x14 on the radio (from left to right). The exact bit sequence on the RF is: 00011010, 01000111, 00010100 (again left to right).

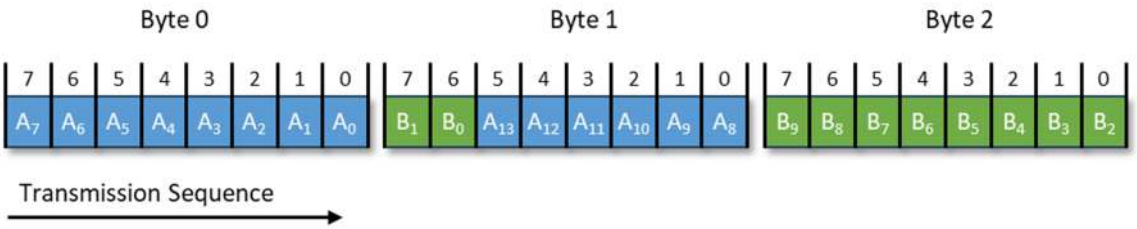
Example 2: The 6-bit value of 0x35 (0b111001) is to be encoded at position 28 of the RF packet. Since this overlaps a byte boundary, it is encoded with the least significant part first: Byte 3: 0b1001xxx, Byte 4: 0bxxxxx111, where 'x' is used for bits not affected by the value.

Example 3: Transmitting the sequence of a 6- and 10-bit values of 0b101101, 0b1100011111 (hex: 0x2D, 0x31F) yields the byte sequence 0b10110111, 0b00011111 (hex: 0xB7, 0x1F).

Example 4: Consider the following payload block, taking up a total size of 3 bytes / 24 bits:

Offset	*Description*	*Width (bits)*	*Encoding*
0	A	14	Unsigned integer
14	B	10	Unsigned integer

The structure of the payload, as transmitted on the radio is as follows:



Consider the example data {A = 0x12AA, B = 0x355}. Using the above construction rule, the payload is transmitted on the radio as Byte 0 = 0xAA, Byte 1 = 0x52, and Byte 2 = 0x55. The exact bit sequence is: 10101010 01010010 11010101 (left to right).

1.2 Coordinates Datum

The WGS-84 standard is used throughout. Elevation is referenced to the WGS-84 ellipsoid surface (i.e. not the geoid, not MSL). Longitude and latitude are encoded in degrees, scaled 1E-7. South and West are negative.

1.3 Code Fragments

Several C code fragments are given in this document since they provide a very clear and succinct description of the algorithms used in FAMP. These fragments are provided for documentation purposes, they are not production code. It is the responsibility of the implementor to add error handling and other defensive coding practices. Best practices for software design must be applied when implementing the FLARM FAMP protocol.

The fragments use the C99 language flavor. An int is assumed to have a size of 32 bits and byte ordering is little-endian. Other programming languages may have different implementations of numbers, types, and operators.

1.4 Limitations

This document describes only the FAMP payload of the FLARM radio network; other payloads are excluded. Only the frequency regime used in Europe is described; for other regions, contact FLARM Technology.

2 Radio System

The FLARM network uses a cost-effective, license-free, low-power radio system based on the 868 MHz SRD or 915 MHz ISM bands, depending on the region where it is operated. Data is sent on the radio in discrete packets with the following structure:

The individual blocks are described below.

2.1 Modulation and Encoding

A digital modulation scheme (2-GFSK) is employed to transmit data. The key parameters are given in the table below.

Parameter	*Value*
Frequency	868.2 / 868.4 MHz (Europe)
Channel Bandwidth	200kHz
Modulation	GFSK
Max. Power (ERP)	14dBm / 25mW
Bitrate	100kbps
Frequency Deviation	0: -50kHz, 1: +50kHz
Gauss Filter BT	0.5

Manchester encoding is applied to the Sync Word, Payload, CRC, and signature for better clock synchronization (lower error rate and more reliable transmission), halving the effective bit rate to 50kbps. The encoding is such that a '1' is encoded as '01' and a '0' as '10'.

Manchester encoding is not applied to the Preamble.

2.2 Timing

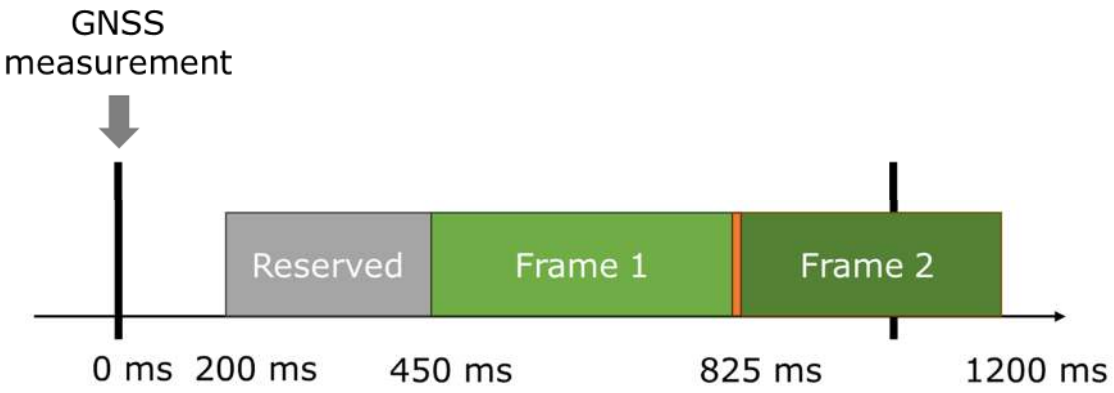
FLARM clients are time-synchronized, using their internal GNSS receivers as a precise time reference. Coordinated Universal Time (UTC) is used throughout. Every transmitting FLARM client shall have a time base with an accuracy of 5 ms or better. This typically requires the availability of the GPS time pulse event.

Access to the FLARM network is organized in cycles of one second duration, referenced to the full second. For technical reasons (legacy hardware with limited compute power), the cycle is delayed by 200 ms vs. the full second.

Each cycle comprises two radio frames: Frame 1 and Frame 2. Each frame uses a different frequency to increase resilience to interference and spectrum efficiency. The timing and frequencies are given in the table below.

	Frame 1	*Frame 2*
Start	450 ms	830 ms
End	820 ms	1200 ms
Frequency	868.2 MHz	868.4 MHz

Note the guard time of 10 ms in between the two frames: FAMP clients shall not transmit radio packets during this time.



The nominal send rate for the FAMP protocol is one packet per frame, i.e. two packets per second.

The contents of a packet (time, position, motion, …) always refer to the beginning of the cycle, i.e. the contents do not change between Frame 1 and Frame 2.

2.3 Medium Access Control (MAC)

The MAC protocol is Carrier-Sense Multiple Access (CSMA). Listen-before-talk (LBT) must be implemented according to the Polite Spectrum Access rules laid out in ETSI EN 300 220-1.

Devices must ensure that the channel is clear before sending. If the channel is busy, transmission of the packet may be rescheduled. The back-off time shall be chosen randomly but shall be at least 15ms. Rescheduling shall remain within the current frame. If rescheduling is not possible, the packet shall be discarded.

2.4 Preamble

The Preamble is sent by the transmitter to allow receivers to detect radio activity and synchronize their clocks. It consists of two parts:

1. A sequence of zeroes and ones of at least length 10 symbols, maximum 24 symbols. The sequence must end with a 1, and
2. the sequence 1001, transmitted twice.

Manchester encoding is not applied to the preamble. The shortest valid preamble thus is (sent from left to right): 01 0101 0101 1001 1001.

2.5 Sync Word

The Sync Word is 3 bytes long. The Sync Word content is given below as arrays, transmitted left to right.

Hex	0x31 0xFA 0xB6
Binary	00110001 11111010 10110110

2.6 Payload

The Payload is transmitted after the sync word. The length is fixed to 24 bytes. The layout of the payload is described in Section 4.

2.7 CRC

A 16-bit cyclic redundancy check (CRC) for message integrity is transmitted after the Payload. The algorithm used is a CRC-16/CCITT with a polynomial of 0x1021 and an initial value 0xFFFF.

A sample implementation is given below. The CRC accumulation is applied to the bytes of the Sync Word, then the Payload, in the order with which these data are sent over the RF. The resulting 16-bit code is then sent with the least significant byte first.

```
uint16_t crc_FLARM(uint16_t crc, uint8_t data){
  crc = crc ^ ((uint16_t)data << 8);
  for (int i=0; i<8; i++) {
    crc = (crc & 0x8000) ? (crc << 1) ^ 0x1021 : crc << 1;
  }
  return crc;
}
```

3 Algorithms and Methods

3.1 Extended Range Encoding

Extended Range Encoding (ERC) is a nonlinear encoding technique to encode a value with a large input range efficiently, using less bandwidth (bits) compared to a simple linear encoding. Due to the nonlinearity, it sacrifices (absolute) precision at higher values (i.e. larger quantization interval), but achieves a much larger input range. The relative precision (i.e. the ratio between quantization interval and encoded value) can be tuned to be sufficient for the application.

The method is comparable to floating point representations. The difference is that ERC uses integers and is flexible to adjust optimally to the individual values and fields in the protocol. ERC is parametrizable:

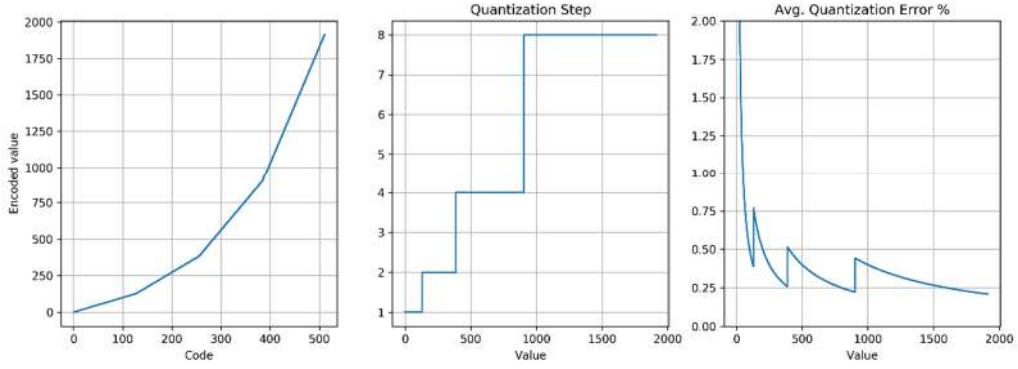
- Total number of bits *N* to use for code points.
- Number of bits N_m to use for mantissa, and, derived from *N* and N_m, the number of bits N_e to use for the exponent.
- Whether negative values are allowed, or only non-negative.

Assume the pair *c = (e, m)* is a code point comprising exponent and mantissa. The mantissa *m* has a bit width of N_m The total bit width of *c* is *N*, thus the width of *e* is $(N\text{-}N_m)$. Then, the value encoded by *c* is given by:

$$v = 2^e(2^{N_m} + m) - 2^{N_m}$$

Code points are represented as the binary concatenation of the exponent and mantissa, yielding the binary representation of *c*:

$c_{\text{binary}} = e \ll \text{N\_m} \mid m$ where "|" denotes the "bitwise or" and "<<" the shift left operator. An example for *Ne = 2, N = 9* is given below. Note how the encodable value range increases from 0.512 to 0..1912, but the precision decreases (or the quantization interval increases) at *v = 128*, *v = 128+256*, and so forth.



The parametrization in N, N_e defines the range of encodable values and how quickly the precision degrades with larger values. In addition to these parameters, the FAMP protocol also applies a linear scaling to every field. The used parameters are given in Section 4.1.

3.1.1 Implementation

For reference, the following code fragment implements decoding:

```
/// Type to use for code points. This can be of course
/// less wide, but 32 bits should be ok to demonstrate all
/// aspects without further thinking
typedef uint32_t Code_t;

/// Unsigned value type. As with Code_t, this
/// is a conservative (easy) choice for the sake of the
/// demo. In practice, you would choose this as small as needed.
typedef uint32_t UValue_t;

/// Signed value type.
typedef int32_t Value_t;

/// Macro for computing 2 to the power of X
/// Reason: Shift operators have lower precedence than +-/*, so they
/// are sometimes hard to read.
#define POW2(_x) (1 << (_x))

/// Decodes a code point and returns the original value.
/// \param code Input code point. Assumed the code point is valid (
///     (i.e. code  < 1 << N). No error checking is applied.
/// \param N Number of bits the code point uses in total
/// \param N_m Number of bits that are used by the mantissa. N_m must
///     be smaller than N
/// \return Corresponding value.
UValue_t decode(const Code_t code, const size_t N, const size_t N_m) {
  UValue_t exponent = (UValue_t)(code >> N_m);
  UValue_t mantissa = (UValue_t)(code & (POW2(N_m) - 1));
  return ((POW2(N_m) + mantissa) << exponent) - POW2(N_m);
}
```

This algorithm requires only integer operations (shift, add, sub, and). It can thus be used even on resource-constrained embedded systems.

For encoding, the following fragment implements a valid algorithm:

```
/// Encode a value into a code point.
/// \param val Input value. No range checks are performed.
///     If the input is out of range, the result will be, too.
///     It is safe to inject out-of-range values and then check the result.
/// \param N Number of bits the code point uses in total
/// \param N_m Number of bits that are used by the mantissa. N_m must
///     be smaller than N
/// \return Resulting code point. If res >= 1 << N, then the input
///     was too large.
Code_t encode(const UValue_t val, const size_t N, const size_t N_m) {
  // The largest possible exponent is 2^(N-N_m)-1. Start with this
  // and iteratively reduce until `val` can be expressed using this
```

```
  // exponent.
  for (Code_t exponent = POW2(N - N_m) - 1;; exponent--) {
    UValue_t thresh = (UValue_t)(POW2(N_m + exponent) - POW2(N_m));
    if (val >= thresh) {
      // Correct exponent is identified, now express val. Note the rounding
      // term for the mantissa.
      Code_t mantissa = (val - thresh + POW2(exponent) / 2) >> exponent;
      return (exponent << N_m) | mantissa;
    }
  }
  // Never reached because thresh is zero
  // in the final iteration step
}
```

Note that this required looping to find the proper exponent. Since the protocol uses small exponents (2..4), this is still very efficient. The fragments above do not check for valid code points and values.

3.1.2 Signed Values

Signed values are encoded by using the most significant bit of the exponent as a minus sign. The following fragments demonstrate the concept:

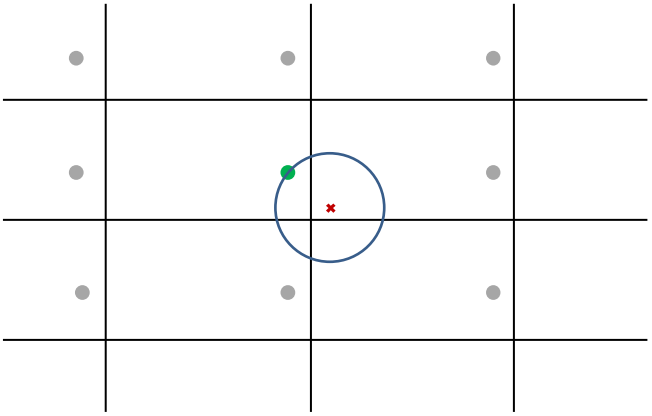
```
Value_t decode_signed(const Code_t code, const size_t N, const size_t N_m) {
  if (code & POW2(N - 1)) {
    // Minus big set. Clear it and delegate
    return -(Value_t)decode(code & (~POW2(N - 1)), N - 1, N_m);
  } else {
    return (Value_t)decode(code, N - 1, N_m);
  }
}

Code_t encode_signed(const Value_t val, const size_t N, const size_t N_m) {
  if (val < 0) {
    // Negative value, set minus bit
    return encode((UValue_t)(-val), N - 1, N_m) | POW2(N - 1);
  } else {
    return encode((UValue_t)val, N - 1, N_m);
  }
}
```

3.2 Adaptive Coordinate Compression

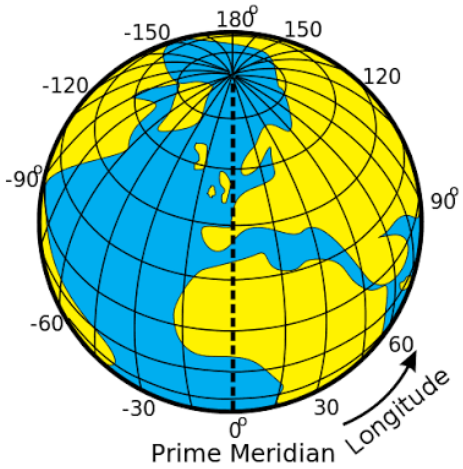
Adaptive Coordinate Compression (ACC) is a system to reduce the required bandwidth for transmitting (via radio) a global position, expressed in degrees latitude and longitude, by exploiting that sender and receiver are necessarily local. This is done by "compressing" the coordinates. Compression introduces ambiguity and reduces the resolution of the transmitted position. ACC balances these tradeoffs with the benefits carefully.

The WGS-84 geodesic system is used throughout for expressing global coordinates. The base units are longitude and latitude, scaled to 1E-7° (before compression, "raw format"). Signed integers are used, north and east are positive, respectively. Altitude is relative to the ellipsoid (i.e. not the geoid) and not compressed, i.e. not part of this algorithm.

In ACC, the WGS-84 coordinate space is divided into grid cells, where the grid cell dimensions ("grid widths") are chosen to be much larger than the maximum expected radio range. A sender transmits its position relative to the local grid cell origin. A receiver can determine the grid cell that results in the lowest distance to the sender. By the principle of locality, this must then be the true solution, since other solutions are not physically possible due to the lower radio range. The situation is depicted below: A receiver (red) determines the correct position (green) by proximity. Due to properties of the grid, any of the grey positions are correct decodings in principle and can only be discarded due to the principle of locality:



ACC uses a grid that is not uniform: If it was, the effective grid width would contract towards the pole for the longitude dimension, as illustrated below. As a consequence, the longitudinal cell width would drop below the radio range at some point; a receiver could then no longer unambiguously determine the sender's position. Conversely, to maintain a sufficient longitudinal grid width, longitude would require more bits in the transmission.



ACC addresses this by dynamically adapting the longitude grid width with higher/lower latitudes. Hence the "A" in ACC.

3.2.1 The Grid

The grid is created for longitude and latitude separately, i.e. the gridding is decomposed into two one-dimensional operations. Each 1-D compresses raw coordinates in three sequential operations: Scaling, rounding, and gridding. Integer operations are used throughout. Two parameters define the grid:

- The resolution divider (*D)*, defining the scaling operation
- The modulus (*m),* defining the grid width after scaling

Both parameters shall by convention be positive integers. Separate parameters D_{lon}, m_{lon}, D_{lat}, m_{lat} are defined below for each dimension. Let the one-dimensional compression algorithm be defined as:

$$c_{comp} = \frac{\left(c_{raw} + \frac{D}{2}\right)}{D} \; mod \; m$$

where c_{raw} is the input raw coordinate (either lat or lon) expressed as signed integer with a defined scaling factor (e.g. 1°E-7 per step), c_{comp} is the resulting, compressed coordinate, the fractions represent integer divisions, and *mod* is the modulo operator. Note how the rounding is achieved by adding *D/2* before the division.

Note: The modulo function here is used in its mathematical definition where the result is always a non-negative integer, as used e.g. in Python. For instance:

$$(-2) \; mod \; 3 = 1$$

Certain programming environments use different conventions. In particular, the "%" operator in C yields the remainder (not modulo), which then requires a special treatment for negative raw coordinates.

Note: The integer division here is defined as truncating to the next lower integer, i.e. -7/5 = -2 (*floored division*). As with mod, other programming environments use different definitions of the integer division, i.e. truncating towards zero (C, *truncated division*)[1].

The modulus *m* defines the range of the gridding, i.e. the compressed coordinate can take values in the range *0..m-1*. For practical purposes, the modulus *m* is thus often chosen as a power of 2 (e.g. 2^{20}), thereby using the available storage space (e.g. 20 bits) maximally and also decreasing computational effort since an efficient bit-mask operation can be used for the gridding operations, simply removing a number of leading bits.

Example: Given a resolution divider of *D=10* and a modulo of *m=32*, compressing the non-negative coordinate c_{raw} yields:

$$c_{comp} = \frac{c_{raw} + 5}{10} \; mod \; 32$$

Next we derive the parametrization *(m, D)* of the grid given the physical requirements (radio range). First we note that the grid width is given by the parametrization as follows:

$$G = m \; D \; res_{raw}$$

Where res_{raw} is the resolution of the raw coordinates in meters at the equator. Assuming a simplified Earth model (perfect sphere, 40'000km circumference) and the standard scaling of raw coordinates used throughout FLARM, this is given by

$$res_{raw} = \frac{40'000 \; km \; 10^{-7\circ}}{360^\circ} = 1/90 \; m \; = \; 0.0111 \ldots m$$

For the latitude, this is the actual, local resolution throughout. For longitude, the local resolution decreases continuously towards the poles. Note that the simplified Earth model does not lead to inaccuracies in FAMP – it is only used to determine the grid parameters.

Next, the resolution divider can be calculated from the available storage space and the minimum required grid width:

[1] In fact, the behaviors of mod and div coincide:
https://en.wikipedia.org/wiki/Modulo_operation

$$D = ceil\left(\frac{G_{min}}{m\, res_{raw}}\right)$$

Where G_{min} is the required minimum grid width chosen to be at least twice the expected maximum radio range. The ceiling function is used to retrieve a grid width G that is not smaller than G_{min}.

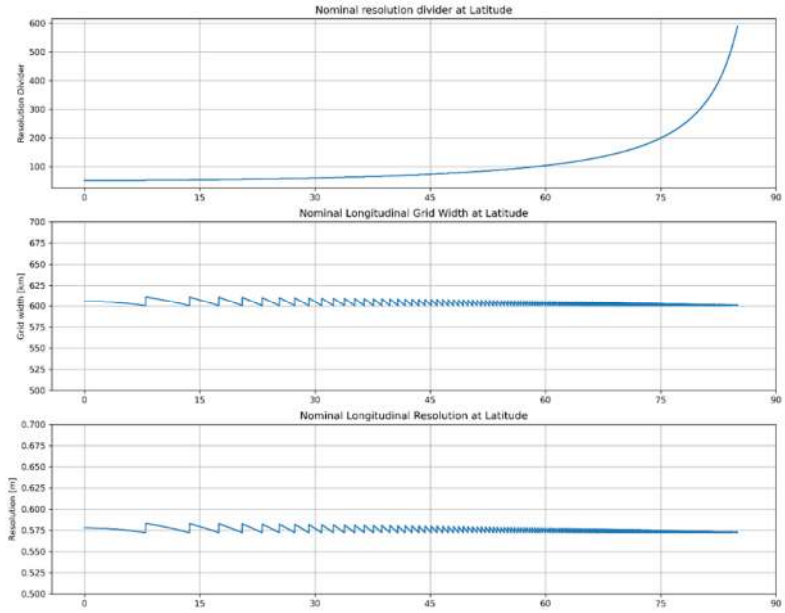
For latitude encoding in FAMP, we select $G_{min} = 600km$ and $m = 2^{20}$, thus yielding $G = 605.844km$ and $D_{lat} = 52$.

3.2.2 Longitude

For longitude encoding, the resolution divider needs to adapt to the latitude to keep the grid size above G_{min} towards the poles. Since the meridians converge with *1/cos(lat)* towards the poles, the optimal adaptation is straightforwardly achieved by adopting the above formula:

$$D_{lon}(lat) = ceil\left(\frac{G/cos(lat)}{m\, res_{raw}}\right)$$

This achieves an (almost) uniform resolution and grid width throughout all latitudes. Small deviations are due to the truncation to integer, as depicted below.



3.2.3 Approximating D_{lon}

The above solution is impractical on low-end embedded systems since both the floating-point division as well as the cosine function evaluation are computationally expensive (disambiguation needs to be calculated for every received aircraft).

Therefore, a piecewise-linear approximation of D_{lon} is developed, taking the form:

$$D_{lon}(lat) = D_0 + a_1 thr(|lat| - lat_1) + a_2 thr(|lat| - lat_2) + \cdots + a_N thre(|lat| - lat_N)$$

where $a_1, \ldots a_N$ are coefficients, $lat_1, \ldots, lat_N$ are called latitude thresholds, $D_0 = D_{lat} = 52$, and *thr()* is the threshold function

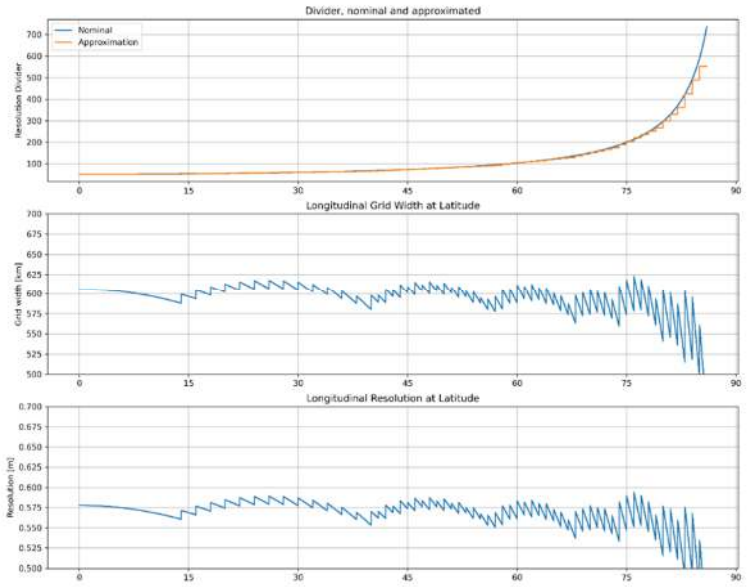
$$thr(x) = \begin{cases} x & if\ x > 0 \\ 0 & else \end{cases}.$$

The coefficients are restricted to powers of two (positive or negative) by convention, such that the multiplication can be efficiently expressed as bit shift operations. The thresholds are restricted to integers between 0 and 90 to further speed up computation without sacrificing substantially on the accuracy. Coefficients and thresholds are found through an optimization search:

n	*Threshold (L_n)*	*Coefficient (a_n)*
1	12	2^{-1}
2	39	2^{0}
3	56	2^{1}
4	67	2^{2}
5	73	2^{3}
6	79	2^{4}
7	82	2^{5}

This yields an approximation that is very accurate for latitudes up to about 85°. Above this threshold the approximation starts to fail, as expected from the optimization setup. This is considered acceptable.

Note: Using above approximation is not optional! Every FAMP user needs to use this exactly. The nominal derivation and plots are given here for instructional purposes only.



This algorithm is particularly well-suited for 8-bit systems as it can be implemented in pure 8-bit instructions up to 78.9999° degrees latitude. For latitudes of 79 degrees and higher, the divider is larger than 255, hence the algorithm can no longer be computed in 8-bit arithmetic. A practical consideration for such constrained systems is to simply ignore the terms 6 and 7, capping the maximum latitude at 79°. This is enough to cover all of Norway or Iceland.

3.2.4 Receiving ACC

Uncompressing is always performed using a reference position as input: The received, compressed code is ambiguous in the sense that the originating grid cell is unknown. By assuming a limited physical radio range, the receiver may choose the grid cell by simply choosing the one that minimizes the distance between his own and the received position.

For uncompressing an ACC lat/lon, pair, the receiver performs the following steps:

1. Uncompress the latitude
2. Determine longitude resolution divider by calculating $D_{lon}(lat)$ as defined by the approximation in Section 3.2.3.
3. Uncompress the longitude

Since the latitude grid is uniform, Step 1. is straightforward. With the latitude fixed, Steps 2. and 3. then become simple also.

3.2.5 Example Code

The following code fragments implement one-dimensional compression and decompression operations.

```
int32_t acc_compress_single(int32_t input, int32_t res_divider,
                            int32_t modulus) {
  if (input >= 0) {
    // The non-negative case is simple:
    return ((input + (res_divider / 2)) / res_divider) % modulus;
  } else {
    // The negative case adjusts the rounding:
    int32_t res = ((input - (res_divider / 2)) / res_divider) % modulus;

    // The % operator in C is not modulo, but remainder, and thus can
    // yield negative results. This is easy to catch below.
    // This part is not portable to other programming languages, i.e. Python
    // has a different definition of mod and this adjustment would not
    // be necessary.
    if (res < 0) {
      res += modulus;
    }
    return res;
  }
}

int32_t acc_uncompress_single(int32_t input_trunc, int32_t reference,
                              int32_t res_divider, int32_t modulus) {
  int32_t ref_reduced   = reference / res_divider;
  int32_t ref_grid_base = (ref_reduced / modulus) * modulus;
  int32_t result        = input_trunc + ref_grid_base;
  // If `reference` is negative and `input_trunc`
  // is one grid cell south or west of the reference, the initial `result` may
  // put the `input_trunc` north/east by *two* grid cells. To get the correct
  // result, we must thus subtract the `modulus` twice.
  //
  // This is due do how integer division is defined in C for negative arguments
  // and thus how `ref_reduced` is computed.
  for (size_t i = 0; i < 2; i++) {
    if (result >= ref_reduced + (modulus / 2)) {
      result = result - modulus;
    }
  }
  if (result <= ref_reduced - (modulus / 2)) {
    result = result + modulus;
  }
  return result * res_divider;
}
```

Note how negative values need special treatment in the C programming language. These samples are for instructional purposes only; actual implementations can improve efficiency.

A reference implementation for $D_{lon}(lat)$ is given by this code fragment for non-negative latitudes (negative latitudes can be used by taking the absolute value):

```
int32_t res_divider(int32_t lat_ref) {
  static const int params[] = {12, 39, 56, 67, 73, 79, 82};
  int32_t res               = 52;  // D0
  for (size_t i = 0; i < DIM(params); i++) {
    int p_i = params[i];
    if (lat_ref >= p_i) {
      int shift = i - 1;  // base shift is -1
      if (shift >= 0) {
        res += (lat_ref - p_i) << shift;
      } else {
        res += (lat_ref - p_i) >> (-shift);
      }
    }
  }
  return res;
}
```

3.3 Enhanced-Privacy Random Address

To improve the ability to conceal a sender's identity while maintaining consistency for collision avoidance, FAMP introduces Enhanced-Privacy Random Address (EPRA).

The method provides, from the perspective of the receiver, a stable address for every sender: A receiver can attribute signals originating from one sender correctly if it has continuous reception of the signal. If the receiver misses a certain number of transmissions, the attribution is no longer possible, and the sender is indistinguishable from any other sender. In other words, the history of a sender is lost if continuous reception is interrupted for a certain duration.

EPRA works by changing the sender address randomly over time, but only by "a bit", and by transmitting this randomness via radio to make it easy for the sender to follow.

Specifically, at every cycle, the sender generates a random number, either from a true random number generator or some sufficiently seeded PRNG. This random number is then mixed to the current address to get the next address. The random number is transmitted as part of the message, together with the current address. In certain time intervals, the sender switches to using the next address, regenerating a new random number, and thus the cycle continues.

By storing both the current address and the random number, a receiver can thus correctly attribute the sender without effort when the address update happens. If a receiver continuously receives at least one message per cycle, then attribution is perfect. If one or more cycles are missed, attribution becomes harder, since at least one random number must be "guessed". The capability of a receiver to

successfully calculate the correct sequence of random numbers is effectively limited by two effects:

1. Computational feasibility: Especially when tracking hundreds of targets, e.g. in a wide-area receiver network, or when limited processing power is available, as is often the case in embedded (on-board) systems.
2. Ambiguity: the more random bits are mixed to an address, the higher the likelihood of a collision: Different paths, starting from different addresses, lead to the same result. When this happens, the receiver can no longer be unambiguously attributed.

Thus, privacy (anonymity) is established for the sender since it is hard for any single receiver to achieve uninterrupted reception. For large-scale networks, this might become feasible, though attribution is possible in this case anyhow simply through correlating positions.

When an EPRA is used, this is marked in the `address_type` field. The random address has a width of 32 bits. An 8 bit value is used for the random number, but chosen from a subset range of $0..2^{Ne} - 1$, where N_e is the number of random bits used for generating the random number. The amount of randomness in the nonce can be varied, we here chose 2 bits, stored as least significant bits.

3.3.1 Prerequisites

The following mixing functions are used:

```
static uint32_t mix32(uint32_t x) {
  uint8_t r[4] = {x, x >> 8, x >> 16, x >> 24};
  uint8_t a[4];
  uint8_t b[4];
  uint8_t c;
  uint8_t h;
  // The array 'a' is simply a copy of the input array 'r'
  // The array 'b' is each element of the array 'a' multiplied by 2
  // in Rijndael's Galois field
  // a[n] ^ b[n] is element n multiplied by 3 in Rijndael's Galois field
  for (c = 0; c < 4; c++) {
    a[c] = r[c];
    // h is 0xff if the high bit of r[c] is set, 0 otherwise
    h = (r[c] >> 7) &
        1;  // arithmetic right shift, thus shifting in either 0s or 1s
    b[c] =
        r[c] << 1;  // implicitly removes high bit because b[c] is an 8-bit
                    // char, so we xor by 0x1b (and not 0x11b) in the next line
    b[c] ^= h * 0x1B;  // Rijndael's Galois field
  }
  r[0] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1];  // 2 * a0 + a3 + a2 + 3 * a1
  r[1] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2];  // 2 * a1 + a0 + a3 + 3 * a2
  r[2] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3];  // 2 * a2 + a1 + a0 + 3 * a3
  r[3] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0];  // 2 * a3 + a2 + a1 + 3 * a0
  return r[0] | ((uint16_t)r[1] << 8) | ((uint32_t)r[2] << 16) |
         ((uint32_t)r[3] << 24);
```

```
}
```

3.3.2 Sender

On the sender, EPRA comprises the following steps:

1. Initialize address by choosing a random unsigned 32-bit integer. Use this when sending FAMP.
2. Randomly choose a number from the set $0..2^{Ne}-1$
3. Use the pair (random number, address) in FAMP transmissions
4. After between 2 and 10 seconds, compute the new address from the old address and the random value

```
address = mix32(address ^ random);
```

5. Go to step 2.

The duration between to address updates (Step 4) and the number of bits to use for the random number (N_e) can be chosen by the sender to trade off privacy vs. trackability. The default value for the interval is 10 seconds; the minimum is 2 seconds. The default value for N_e is 2, the minimum 1, the maximum 8. The sender may adapt to the interval in flight (e.g. randomly). The address update may happen at any time, but only after the current cycle is completed.

3.3.3 Receiver

The receiver of a packet with a `(random value, address)` pair performs the following steps:

1. When adding a received packet to the internal memory store, store both `address` and `address_next = mix32(address ^ random)`, i.e. the address after the next update.
2. When receiving a `(random number, address)` pair, check the memory store if an entry with the same `address`, or with `address_next` exists. If a match is found, assume the new data originates from the same sender.

If no matching address is found in Step 2, a receiver may optionally employ a deeper (brute-force) search over multiple address updates, assuming it received EPRA-enabled data before.

3.4 Dynamic Message Versioning

Dynamic Message Versioning (DMV) is a method for eliminating the need for the hard firmware expiration mechanism in pre-FAMP: Devices that did not receive recent upgrades would stop operating at a predefined date. Thus, the active

firmware versions at any given date could be controlled, allowing a concerted, global, protocol update.

FAMP does not require a firmware expiration while still allowing the protocol to change and improve. Every FAMP client is backward-compatible indefinitely, i.e. capable of receiving and sending FAMP messages of any (older) version. This makes any older client visible to newer ones automatically. For vice-versa visibility, DMV dynamically balances the use of different versions of the protocol based on the capabilities of other receivers in the vicinity. The maximum version a FAMP client is capable of processing is disclosed in the `ver_max` field in the FAMP header and is thus transmitted with every FAMP packet.

DMV works as follows. Every FAMP sender:

- Updates and maintains a list of other FAMP senders and their published maximum protocol version (`ver_max`). This may deviate from the actual version used (`ver`).
- For every FAMP packet that is sent, it chooses the protocol version based on the above list. Heuristics are applied to determine the protocol version, maintaining a minimum connectivity with older clients. These parameters may develop over time: For instance, an old version may get higher priority immediately after a new release, to allow clients to catch up, then gradually be phased out.
- The most compatible protocol version is 3, compatible with all FAMP devices.

3.4.1 Implementation

Denote the device under consideration as "host" and nearby devices conversely as "clients". This section explains how the host selects the protocol version for sending based on packets received from clients.

Note that both roles (host and client) are usually present in any FLARM device, so this rule applies symmetrically to most clients as well. Non-senders (e.g. ground receiver networks) have no means to publish their capabilities. It is expected that they can be updated easily, i.e. it is safe to assume they can deal with all available protocol versions always. Non-receivers (e.g. paraglider beacons) shall transmit a 0xf for `ver_max` indicating they cannot receive. These clients are excluded from DMV.

Let i be an index for the list of a received FAMP client, as stored in memory, with $i=1..Nc$. Clients that are not currently received are removed from this list. Let ver_max_i be this client's associated (i.e. published in FAMP) maximum processable version, as last received in a FAMP packet. We assume ver_max_i does not change over time.

Let *m[i]* count the number of missed packets, i.e. packets that the client could not have received or parsed according to its declared *ver_max* being lower than the version sent. This is computed at the nominal FAMP transmit rate. If the host is unable to send a packet (e.g. due to RF collision or bandwidth management), this, too, counts as a miss for all clients.

The nominal transmit rate is 2 per second, one transmission per frame. The transmit version is determined before the start of a frame. Events during the frame do not influence the further transmissions within this frame.

m[i] shall be updated as follows:

1. At the beginning of any windows, increment *m[i]* for every client
2. If a FAMP packet is successfully sent in this window: For every client, set *m[i]* to zero if the sent version is smaller or equal to ver_max_i

Before the beginning of a frame, the protocol version to be used in this window is determined as follows:

1. Let the target update rate for each client be:

   $$t_{targ}[i] = f(D[i], ver\_max_i, M_i)$$

   where *D[i]* is a norm function equivalent to the distance and M_i the metadata available for the target, e.g. aircraft type, firmware version, or the type of the device. For a definition of the function *f()*, see below.
2. If the target is in conflict (i.e. an alarm is generated), t_i is set to 1.
3. Let the send gap describe the discrepancy between the target rate and the actual transmission. For any client, it is given by

   $$g[i] = m[i] - t_{targ}[i]$$

   Note that *g[i]* starts negative and continuously increases if no appropriate transmission is made. A gap of 0 or higher indicates an immediate need for transmitting to this client.
4. If no *g[i]* is 0 or higher, select the maximum protocol version. Break.
5. Select the version as the minimum of ver_max_i of all clients with *g[i]* nonnegative (0 or higher).

The function *f()* is not fixed but may be adapted with every update of the protocol, reflecting the current situation in the population of devices. The following basic construction shall apply:

- Base $t_{targ}[i]$ is based on vehicle type (in number of frames):

  - Hang glider, paraglider: 4
  - UAV: 2
  - All others: 1
- If a client signals no RX capabilities (*ver_max*$_i$ set to 0x0F), set $t_{targ}[i] = 20$
- If the horizontal distance is larger than 3km, multiply $t_{targ}[i]$ by 2
- If the vertical separation is larger than 500m, multiply $t_{targ}[i]$ by 2
- If the approach rate (distance divided by the relative speed vector projected on the relative position vector) is less than 30 seconds, set $t_{targ}[i]$ to 1. Computation may be limited to 2 dimensions and only if vertical separation is smaller than 200m.
- If *ver_max*$_i$ is far behind (e.g. a firmware update being available for longer than 2 years), multiple $t_{targ}[i]$ by 2

Note: Only one version of FAMP is currently published, hence *f()* is not defined concretely yet.

4 FAMP – Marshalling and Semantics

4.1 Structure

The radio packet comprises a header and a payload block. The size of the header part is 7 bytes, the payload part 17 bytes. The header is transmitted in clear; parts of the payload are encrypted.

The header is given by:

Field	*Width*	*Pos*	*Scaling, Unit*	*Description*
address_base	24	0	-	Depends on address_type, see below
msg_type	4	24	enum	Fixed to 2 (for FAMP payload)
address_type	2	28	enum	See below
urgency	2	30	enum	See below
address_ext	16	32		Depends on address_type, see below
msg_type_ext	6	48	enum	Extended message type,sub-type
stealth	1	54	flag	Stealth mode for competitive flying
no_track	1	55	flag	Request for increased privacy, disable tracking

Note: Entries for msg_type other than in the table above are reserved for FLARM products and shall not be used.

The FAMP payload is given by:

Field	*Width*	*Pos*	*Encr.*	*Scaling, unit*	*Description*
ver	4	0			FAMP protocol version used by sender in this packet. Set to 3.
ver_max	4	4			Maximum FAMP protocol version supported by the sender for receive, or 0xf if sender is not capable of receiving
time	6	8	yes	0.25 s	Least significant bits of Unix epoch timestamp, in quarter seconds, UTC. Refers to the time of generation of the navigation solution, i.e. GNSS fix
acft_type	5	14	yes	enum	See below
alt	13	19	yes	m	Altitude, above ellipsoid, offset -1000m, unsigned ERC, 12 bit mantissa
lat_acc	20	32	yes		Latitude, compressed
lon_acc	20	52	yes		Longitude, compressed
turnrate	9	72	yes	0.05 °/s	Turn rate. Signed ERC with 6 mantissa bits
speed	10	81	yes	0.1 m/s	Ground speed. Unsigned ERC with 8 mantissa bits

climb	9	91	yes	0.1 m/s	Climb rate, positive up. Signed ERC with 6 mantissa bits
track	10	100	yes	0.5 °	Ground track, 0..359.5
mov_mode	2	110	yes	enum	Movement mode
acc_pos_hor	6	112	yes	0.1 m	Horizontal accuracy estimate, 95% CEP. Unsigned ERC with 3 mantissa bits
acc_pos_ver	5	118	yes	0.25 m	Vertical accuracy estimate, 95% confidence level. Unsigned ERC with 2 mantissa bits
acc_vel	5	123	yes	0.1 m/s	Velocity accuracy estimate, 95% confidence level. Unsigned ERC with 3 mantissa bits.
sil	2	128	yes	enum	Source Integrity Level, see below
sda	2	130	yes	enum	System Design Assurance and supported failure condition, see below
nic	4	132	yes	enum	Navigation Integrity Category / Horizontal Radius of Containment

The fields are discussed in more detail in the following sections.

4.2 Encryption and Decryption

The payload is partly encrypted to ensure message integrity, system safety and provide protection for the relevant content against eavesdropping, namely by article 3 of the Budapest Convention on Cybercrime and Directive 2013/40/EU.

The XXTEA algorithm with a key size is 128 bits is used. The key is fixed and shared by all participants of the FLARM system. Only the payload block (see Section 4.3.1) – except for the version fields - is encrypted, the header is transmitted in clear.

The shared key is given in the table below.

Key Index	*Key Value*
0	0xA5F9B21C
1	0xAB3F9D12
2	0xC6F34E34
3	0xD72FA378

4.2.1 Nonce

A 128-bit nonce is mixed with the payload before encryption. It is created deterministically such that a sender can derive it from the unencrypted part of the radio packet. This connects the encrypted part tightly with the plaintext part. The nonce also contains a time stamp.

The nonce is constructed as follows:

Field	*Width*	*Pos*	*LSB*	*Description*

header	56	0	-	Verbatim copy of the header
ver	4	56		Copy from FAMP payload
ver_max	4	60		Copy from FAMP payload
time_utc_16s	32	64		Number of 16-second intervals since Unix Epoch (Jan 1 1970, 0:0:0), UTC. Same semantics apply otherwise than for time field in FAMP payload
constant	32	96		0x956F6C77

The transmitted, encrypted payload is then computed as

$$payload_E = XXTEA(mix\_nonce(nonce) \text{ ^ } payload, key)$$

where "^" denotes the bitwise-XOR operator and the `mix_nonce()` function is defined as

```
#define MX_ ((((z >> 5) ^ (y << 2)) + ((y >> 3) ^ (z << 4))) ^ (sum ^ y))

void mix_nonce(uint8_t* const v, uint32_t n, uint32_t round) {
  uint32_t z, y = v[0], sum = 0, DELTA = 0x9e3779b9ul;
  uint32_t p, q;
  z = v[n - 1];
  q = round;
  while (q-- > 0) {
    sum += DELTA;
    for (p = 0; p < n - 1; p++) {
      y = v[p + 1];
      v[p] += MX_;
      z = v[p];
    }
    y = v[0];
    v[n - 1] += MX_;
    z = v[n - 1];
  }
}
```

Two rounds (`round == 2`) are used for mixing.

4.3 Header Fields

4.3.1 Sender Address

The header fields *address_base*, *address_type* and *address_ext* together constitute the sender address, where *address_type* enumerates one of four supported methods:

address_type	*Description*
0	Enhanced-Privacy Random Address
1	ICAO
2	FLARM

3 | FLARM extended

The semantics of the other fields change depending on *address_type*. For *address_type* = 0 (Enhanced-Privacy Random Address), they are:

Field	*Width*	*Pos*	*Scaling, Unit*	*Description*
address_EPRA_base	24	0		Lower 3 bytes of EPRA
EPRA_random_val	8	32		EPRA random value
EPRA_MSB	8	40		Random element of EPRA

For *address_type* = 1 (ICAO):

Field	*Width*	*Pos*	*Scaling, Unit*	*Description*
address_ICAO	24	0		ICAO aircraft address
Reserved	16	32		

For *address_type* = 2 (FLARM address):

Field	*Width*	*Pos*	*Scaling, Unit*	*Description*
address_FLARM	24	0		FLARM address (derived from hardware type and serial number)
Reserved	16	32		

For *address_type* = 3 (Extended FLARM address):

Field	*Width*	*Pos*	*Scaling, Unit*	*Description*
address_FLARM_base	24	0		Extended FLARM address, lower 3 bytes
address_FLARM_ext	16	32		Extended FLARM address, upper 2 bytes

Note: Radio addresses must be unique for every FLARM client to avoid ambiguity. Ambiguous addresses may lead to a severe degradation of the safety of the FAMP protocol.

Note: Transmitting FAMP is not allowed under the FFPP license agreement. Please contact FLARM Technology for a license if you intend to transmit. Licensee will be assigned a range of radio addresses.

4.3.2 Urgency

Higher urgency settings shall be used to signal safety-relevant conditions, e.g. when an unusually high sink rate is detected. This shall be used conservatively, considering that it may increase system load on receiving devices, e.g. due to an increased recording rate.

urgency	*Description*
0	Normal
1	High
2	Pan-Pan
3	Mayday

4.3.3 Extended Message Type

The *msg_type_ext* field shall be used to encode sub-types or variants of payloads. This is defined for every payload individually. For FAMP, this field is not used (no FAMP sub-types exist) and shall be set to 0.

4.3.4 Flags

- Stealth Mode: If set this shall signal the intent of the sender to not give away her performance-specific parameters and the identity to airborne receivers. This is used in a (gliding) competition context.
- No Track: If set this shall signal the intent of the sender to not be tracked by ground receivers. Reception may be admissible for (local) safety purposes (e.g. ATC), but data shall not be stored or forwarded to other systems.

4.4 FAMP Fields

4.4.1 Version

Dynamic Message Versioning is used. This document describes version 3 of FAMP. *ver_max* shall be set to 0xf to indicate if a sender is not capable of receiving (e.g. paraglider devices) and should not be considered in the DMV algorithm of receiving peers.

4.4.2 Time

This field contains the 6 least significant bits of the Unix epoch timestamp (e.g. time elapsed since Jan 1st. 1970), in quarter seconds, UTC. This refers to the time of generation of the navigation solution (i.e. the GNSS fix), not the actual transmission. Since FAMP currently supports 1 Hz navigation updates only, this must always be a multiple of 4.

4.4.3 Aircraft Type

acft_type	*Description*
0	Undefined
1	Glider
2	Towplane
3	Helicopter
4	Parachute
5	Dropplane
6	Hangglider
7	Paraglider
8	Single-engine piston
9	Heavy/complex fixed wing
10	Gyrocopter
11	Balloon
12	Airship, blimp
13	UAV
14	Airfield
15	Static (Obstacle)
16	eVTOL / UAM
17	UAV Open
18	UAV Specific
19	UAV Certified

4.4.4 Movement Mode

The current mode of movement shall be determined by the sender depending on the available sensor data (e.g. wheel load sensor, airspeed, or GNSS).

mov_mode	*Description*
0	Undefined / Unknown
1	On Ground
2	Flying
3	Circling / Maneuvering

4.4.5 Accuracy Estimates

The three accuracy estimates, `acc_pos_hor` (horizontal position accuracy estimate), `acc_pos_ver` (vertical position accuracy estimate) and `acc_vel` (velocity accuracy estimate) are defined as the 95% percentile Circular Error Probable (CEP). In other words, the real navigation error should be smaller than this value in 95% of the cases, in a statistical sense.

In the case of u-blox receivers, the accuracy estimates are derived from internal filter states and are crude estimates of the accuracy, given as 1-sigma confidence

levels[2]. A scaling must thus be applied to calculate the 95% confidence levels. For the 1D accuracy estimates (vertical position accuracy, velocity accuracy), this is straight forward. Assuming a Gaussian distribution:

$$x_{95\%} = 1.96 \cdot x_{1\sigma}$$

where $x_{1\sigma}$ is the 1-sigma measurement from the GNSS receiver, and $x_{95\%}$ the value to be used for FAMP (`acc_pos_ver`, `acc_vel`).

For the horizontal position accuracy, the underlying error distribution is a multivariate Gaussian. Since only the norm of the horizontal error is considered in FAMP – i.e. not its vector components – the resulting error distribution is a Rayleigh distribution under some assumptions (not degenerate, circular). The scaling therefore[3] is the following:

$$x_{95\%} = 2.45 \cdot x_{1\sigma}.$$

4.4.6 Safety Integrity Level

Probability of exceeding the NIC Containment Radius, per flight hour:

sil	*Description*
0	Undefined / Unknown
1	SIL <= 1E-3
2	SIL <= 1E-5
3	SIL <= 1E-7

4.4.7 System Design Assurance

Supported failure condition / Design Assurance Level:

sda	*Description*
0	None
1	Minor / DAL-D
2	Major / DAL-C
3	Hazardous / DAL-B

4.4.8 Navigation Integrity Category

Horizontal Containment Radius Limit (Rc). Also See e.g. Table A-2 of DO-260B:

nic	*Description*
0	Rc >= 20 NM or Unknown
1	Rc < 20 NM
2	Rc < 8 NM

[2] https://portal.u-blox.com/s/question/0D52p00008HKCUyCAP/accuracy-haccvacc-vs-gst
[3] https://gssc.esa.int/navipedia//index.php/Accuracy

3	Rc < 4 NM
4	Rc < 2 NM
5	Rc < 1 NM
6	Rc < 0.6 NM
7	Rc < 0.2 NM
8	Rc < 0.1 NM
9	Rc < 75 m
10	Rc < 25 m
11	Rc < 7.5 m

4.4.9 Enhanced Range Coding Fields

For reference, the maximum and minimum encodable values are given for every field using ERC:

Field	*Width*	*Mantissa*	*Scaling, Unit*	*Signed?*	*Min*	*Max*
alt	13	12	1 m	no	-1000 m	11286 m
turnrate	9	6	0.05 °/s	yes	-47.6 °/s	47.6 °/s
speed	10	8	0.1 m/s	no	0 m/s	383.2 m/s
climb	9	6	0.1 m/s	yes	-95.2 m/s	95.2 m/s
acc_pos_hor	6	3	0.1 m	no	0 m	191.2 m
acc_pos_ver	5	2	0.25 m	no	0 m	223 m
acc_vel	5	3	0.1 m/s	no	0 m/s	11.2 m/s

Note that *alt* additionally receives an offset of -1000 m to encode the few locations on Earth that are beneath the WGS-84 ellipsoid.

Annex 1: License Terms and Conditions FFPP

1. Preamble

These Terms and Conditions concern the FLARM FAMP Public Protocol (FFPP). FLARM is a collision warning system for general aviation and drones. FLARM uses the FAMP (FLARM Aircraft Motion Prediction) Protocol for communication between participants. The FFPP includes a respective communication specification. FFPP underlies the present Terms and Conditions.

FFPP was developed by FLARM Technology AG (herein "Licensor"). FAMP, FFPP, the FFPP specification document itself, its included works and code and the technology described or referred to therein are subject to Licensor's IP rights. All FFPP specification documents underlie copyright of Licensor. Licensor reserves the full IP rights, except for the license as granted herein.

FFPP is accessible publicly under the present Terms and Conditions. Access thereto and use thereof by a party (herein "Licensee") is conditional upon compliance with the Terms and Conditions as specified herein.

2. License

Subject to the Terms and Conditions of this license, Licensor hereby grants to Licensee a perpetual, worldwide, non-exclusive, non-sublicensable, royalty-free license to access and use the FFPP for non-commercial purposes only. This license is limited to academic and research purposes as well as amateur/private uses of FFPP specification, limited to receive-only radio communication.

This license does not cover any radio sending/transmission and/or commercial or military uses, including (but not limited to) manufacturing of devices, developing products, developing software, providing commercial services, etc. This license neither covers any further or complementary protocols beyond the FFPP itself. Commercial or military use of FFPP and other FLARM technology is possible only with a separate written commercial license of Licensor. This license also does not cover any use, reproduction or distribution of the FFPP in case of knowledge or negligent ignorance of this supporting, enabling or otherwise furthering unlicensed use by a third party.

If you seek a license of broader scope, please contact licensing@flarm.com.

Generally, in the interest of flight safety and interference-free network operation, any sending under the FLARM protocols requires validation of the product by Licensor and assignment of a FLARM serial number. Any other radio activity potentially interfering with FLARM operation is also prohibited. Licensee may not use the FFPP to decrypt or interfere with the FLARM radio communication.

Licensee may not reverse-engineer FLARM protocols and/or software.

All FFPP specification documents underlie copyright and may not be used, particularly reproduced or in other way exploited, beyond what is necessary for and included by this above-defined license. FFPP reproduction or distribution, including technical implementations of the FFPP in, e.g., software code, to the extent allowed under this License, must always include these Terms and Conditions.

3. Trademarks

This License does not grant permission to use the trade names, trademarks, service marks, company names or product names of Licensor commercially or in any way interfering with Licensor's rights.

4. Miscellaneous

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, is Licensor liable for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the FFPP (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses). Licensor is not liable for any potential infringement of third party rights, particularly IP rights, arising out of or in connection with this use under this license. The limitation of liability does not apply to cases of intent or gross negligence, or for damages resulting from injury to life, body or health.

Any breach of these Terms and Conditions revokes any grant of license for past and future and will make the (former) Licensee liable accordingly. Licensor reserves any rights and claims in connection with or resulting from such breach. For the sake of clarity, using FFPP for working on Licensor's further protocols, such as reverse-engineering activities, is also a breach of these Terms and Conditions.

Should any individual provisions of this Agreement be found invalid or unenforceable, the remainder of this Agreement shall have full force and effect. The Parties undertake in good faith to replace any invalid provision by a new provision which shall approximate as closely as possible the economic and legal result intended by the Parties in entering into this Agreement.

The Agreement is governed by and construed in accordance with the laws of Germany and irrespective of any choice of law principle that might dictate a different Governing Law. The courts of Munich, Germany, shall have exclusive

jurisdiction for any and all disputes arising from, under, out of, or in connection with these Terms and Conditions.

Annex 2: Test Data

This section contains a small but representative test data set. The C type definition of the test data set is the following:

```
// Test case data structures
typedef struct {
  uint32_t time;
  struct {
    int32_t PN;  ///< degrees North Latitude scaled with 1e-7 (ie, PN/10,000,000
                 ///< is degrees)
    int32_t PE;  ///< degrees East Longitude scaled with 1e-7 (ie, PE/10,000,000
                 ///< is degrees)
  } reference_lat_lon;
  unsigned char packet_data[24];
  struct {
    uint8_t ver;      ///< Current packet version used for DMV
    uint8_t ver_max;  ///< Maximum version the host is capable to unmarshal
    struct {
      enum {
        FAMP_RADIO_ID_TYPE_EPRID             = 0,
        FAMP_RADIO_ID_TYPE_ICAO              = 1,
        FAMP_RADIO_ID_TYPE_FLARM_ID          = 2,
        FAMP_RADIO_ID_TYPE_FLARM_EXTENDED_ID = 3
      } id_type;
      union {
        uint32_t icao;
        uint32_t flarm;
        uint8_t extended[5];
        struct {
          uint32_t base;  ///< Enhanced-Privacy Random ID (EPRID)
          uint8_t rnd;  ///< Random number mixed with previous EPRID to generate
                        ///< next one
        } eprid;
      } id;
    } radio_id;
    enum {
      FAMP_URGENCY_NORMAL     = 0,
      FAMP_URGENCY_HIGH       = 1,
      FAMP_URGENCY_PAN_PAN    = 2,
      FAMP_URGENCY_PAN_MAYDAY = 3
    } urgency;
    uint8_t stealth;   ///< Flag: if 1 Stealth mode for competitive flying, 0
                       ///< otherwise
    uint8_t no_track;  ///< Flag: if 1 Request for increased privacy/disable
                       ///< tracking, 0 otherwise
    uint8_t time;      ///< Least significant bits of Unix epoch timestamp, in
                       ///< quarter seconds. [0, 63] range. [0.25 s] scaling
    enum {
      FAMP_AIRCRAFT_TYPE_UNDEFINED               = 0,
      FAMP_AIRCRAFT_TYPE_GLIDER                  = 1,
      FAMP_AIRCRAFT_TYPE_TOWNPLANE               = 2,
      FAMP_AIRCRAFT_TYPE_HELICOPTER              = 3,
      FAMP_AIRCRAFT_TYPE_PARACHUTE               = 4,
      FAMP_AIRCRAFT_TYPE_DROPPLANE               = 5,
      FAMP_AIRCRAFT_TYPE_HANGGLIDER              = 6,
      FAMP_AIRCRAFT_TYPE_PARAGLIDER              = 7,
      FAMP_AIRCRAFT_TYPE_SINGLE_ENGINE_PISTON    = 8,
      FAMP_AIRCRAFT_TYPE_HEAVY_COMPLEX_FIXED_WING = 9,
```

```
    FAMP_AIRCRAFT_TYPE_GYROCOPTER                = 10,
    FAMP_AIRCRAFT_TYPE_BALLOON                   = 11,
    FAMP_AIRCRAFT_TYPE_AIRSHIP_BLIMP             = 12,
    FAMP_AIRCRAFT_TYPE_UAV                       = 13,
    FAMP_AIRCRAFT_TYPE_AIRFIELD                  = 14,
    FAMP_AIRCRAFT_TYPE_STATIC_OBSTACLE           = 15,
    FAMP_AIRCRAFT_TYPE_EVTOL_UAM                 = 16,
    FAMP_AIRCRAFT_TYPE_UAV_OPEN                  = 17,
    FAMP_AIRCRAFT_TYPE_UAV_SPECIFIC              = 18,
    FAMP_AIRCRAFT_TYPE_UAV_CERTIFIED             = 19,
    FAMP_AIRCRAFT_TYPE_MAX
  } acft_type;
  int16_t altitude;  ///< Altitude, above ellipsoid. [-1000, 11286] range. [1
                     ///< meter] scaling.
  struct {
    int32_t PN;  ///< ACC Decompressed Latitude. [-900000000, 900000000]
                 ///< range. [1E-7 degree] scaling
    int32_t PE;  ///< ACC Decompressed Longitude. [-1800000000, 1800000000]
                 ///< range. [1E-7 degree] scaling
  } lat_lon;
  int16_t turnrate;  ///< Turn rate. [-952, 952] range. [0.05 deg/s] scaling
  uint16_t speed;    ///< Ground speed. [0, 3832] range. [0.1 m/s] scaling
  int16_t climb;     ///< Climb rate. [-952, 952] range. [0.1 m/s] scaling
  uint16_t track;    ///< Ground Track. [0, 719] range. [0.5 deg] scaling
  enum {
    FAMP_MOVEMENT_MODE_UNDEFINED_UNKNOWN    = 0,
    FAMP_MOVEMENT_MODE_ON_GROUND            = 1,
    FAMP_MOVEMENT_MODE_FLYING               = 2,
    FAMP_MOVEMENT_MODE_CIRCLING_MANEUVERING = 3
  } mov_mode;
  uint16_t acc_pos_hor;  ///< Horizontal accuracy estimate, 95% CEP. [0, 1912]
                         ///< range. [0.1 m] scaling
  uint16_t acc_pos_ver;  ///< Vertical accuracy estimate, 95% confidence
                         ///< level. [0, 892] range. [0.25 m] scaling
  uint8_t acc_vel;  ///< Velocity accuracy estimate, 95% confidence level. [0,
                    ///< 112] range. [0.1 m/s] scaling
  enum {
    FAMP_SIL_UNDEFINED_UNKNOWN               = 0,
    FAMP_SIL_LESS_OR_EQUAL_THAN_10_E_MINUS_3 = 1,
    FAMP_SIL_LESS_OR_EQUAL_THAN_10_E_MINUS_5 = 2,
    FAMP_SIL_LESS_OR_EQUAL_THAN_10_E_MINUS_7 = 3
  } sil;
  enum {
    FAMP_SDA_NONE            = 0,
    FAMP_SDA_MINOR_DAL_D     = 1,
    FAMP_SDA_MAJOR_DAL_C     = 2,
    FAMP_SDA_HAZARDOUS_DAL_B = 3
  } sda;
  enum {
    FAMP_NIC_RC_MAJOR_EQUAL_THAN_20_NM_OR_UNKNOWN = 0,
    FAMP_NIC_RC_MINOR_THAN_20_NM                  = 1,
    FAMP_NIC_RC_MINOR_THAN_8_NM                   = 2,
    FAMP_NIC_RC_MINOR_THAN_4_NM                   = 3,
    FAMP_NIC_RC_MINOR_THAN_2_NM                   = 4,
    FAMP_NIC_RC_MINOR_THAN_1_NM                   = 5,
    FAMP_NIC_RC_MINOR_THAN_0_POINT_6_NM           = 6,
    FAMP_NIC_RC_MINOR_THAN_0_POINT_2_NM           = 7,
    FAMP_NIC_RC_MINOR_THAN_0_POINT_1_NM           = 8,
    FAMP_NIC_RC_MINOR_THAN_75_M                   = 9,
    FAMP_NIC_RC_MINOR_THAN_25_M                   = 10,
    FAMP_NIC_RC_MINOR_THAN_7_POINT_5_M            = 11,
```

```
      FAMP_NIC_MAX
    } nic;
  } fampdata;
} FampRawDataTestCase_t;
```

The randomized set of ten test cases is presented here in form of an array of `FampRawDataTestCase_t` entries. More data is available upon request.

```
FampRawDataTestCase_t test_cases[] = {
    // Testcase 0
    {
        1996151268,  // .time
        {
            // .reference_lat_lon
            -273941486,  // .PN
            -67004173,   // .PE
        },
        {0xfb, 0x18, 0x0d, 0x22, 0x00, 0x00, 0xc0, 0x03,
         0x42, 0x1c, 0x8b, 0x57, 0xf5, 0x77, 0x38, 0xc2,
         0x0f, 0x29, 0xe5, 0x61, 0x15, 0x40, 0xa9, 0x52},  // .packet_data
        {
            // .fampdata
            3,  // .ver
            0,  // .ver_max
            {
                // .radio_id
                (FAMP_RadioIdType_t)2,  // .id_type
                {
                    // .id
                    .flarm = 0x0d18fb,
                },
            },
            (FAMP_Urgency_t)0,       // .urgency
            1,                       // .stealth
            1,                       // .no_track
            16,                      // .time
            (FAMP_AircraftType_t)8,  // .acft_type
            11240,                   // .altitude
            {
                // .lat_lon
                -274178008,  // .PN
                -67099756,   // .PE
            },
            304,                     // .turnrate
            1760,                    // .speed
            560,                     // .climb
            540,                     // .track
            (FAMP_MovementMode_t)2,  // .mov_mode
            152,                     // .acc_pos_hor
            764,                     // .acc_pos_ver
            14,                      // .acc_vel
            (FAMP_SIL_t)0,           // .sil
            (FAMP_SDA_t)2,           // .sda
            (FAMP_NIC_t)0,           // .nic
        },
    },
    // Testcase 1
    {
```

```
        1448385623,  // .time
        {
            // .reference_lat_lon
            -350742399,   // .PN
            -1008353120,  // .PE
        },
        {0x6f, 0x96, 0xdd, 0x02, 0x0f, 0xcd, 0x40, 0x33,
         0xb1, 0x8e, 0x48, 0xec, 0x30, 0x65, 0xb4, 0x26,
         0x83, 0x95, 0x92, 0xe4, 0x5a, 0x34, 0x13, 0x8a},  // .packet_data
        {
            // .fampdata
            3,  // .ver
            3,  // .ver_max
            {
                // .radio_id
                (FAMP_RadioIdType_t)0,  // .id_type
                {
                    // .id
                    .eprid =
                        {
                            0xcddd966f,  // .base
                            0x0f,        // .rnd
                        },
                },
            },
            (FAMP_Urgency_t)0,       // .urgency
            1,                       // .stealth
            0,                       // .no_track
            28,                      // .time
            (FAMP_AircraftType_t)4,  // .acft_type
            -466,                    // .altitude
            {
                // .lat_lon
                -350558468,   // .PN
                -1008705411,  // .PE
            },
            376,                     // .turnrate
            1268,                    // .speed
            164,                     // .climb
            309,                     // .track
            (FAMP_MovementMode_t)0,  // .mov_mode
            40,                      // .acc_pos_hor
            20,                      // .acc_pos_ver
            112,                     // .acc_vel
            (FAMP_SIL_t)3,           // .sil
            (FAMP_SDA_t)0,           // .sda
            (FAMP_NIC_t)0,           // .nic
        },
    },
    // Testcase 2
    {
        1917945369,  // .time
        {
            // .reference_lat_lon
            211847422,   // .PN
            -999236589,  // .PE
        },
        {0xb0, 0xf3, 0xf0, 0xa2, 0x00, 0x00, 0x80, 0x03,
         0x55, 0x77, 0x49, 0x0c, 0xc8, 0xfb, 0x15, 0x4c,
         0xa6, 0xa4, 0xfa, 0xde, 0x62, 0xc2, 0x72, 0xdc},  // .packet_data
        {
```

```
            // .fampdata
            3,  // .ver
            0,  // .ver_max
            {
                // .radio_id
                (FAMP_RadioIdType_t)2,  // .id_type
                {
                    // .id
                    .flarm = 0xf0f3b0,
                },
            },
            (FAMP_Urgency_t)2,        // .urgency
            0,                        // .stealth
            1,                        // .no_track
            36,                       // .time
            (FAMP_AircraftType_t)17,  // .acft_type
            1710,                     // .altitude
            {
                // .lat_lon
                211932032,   // .PN
                -999386640,  // .PE
            },
            -140,                    // .turnrate
            123,                     // .speed
            436,                     // .climb
            240,                     // .track
            (FAMP_MovementMode_t)2,  // .mov_mode
            168,                     // .acc_pos_hor
            10,                      // .acc_pos_ver
            24,                      // .acc_vel
            (FAMP_SIL_t)1,           // .sil
            (FAMP_SDA_t)1,           // .sda
            (FAMP_NIC_t)3,           // .nic
        },
    },
    // Testcase 3
    {
        2557412086,  // .time
        {
            // .reference_lat_lon
            -521067434,  // .PN
            229778893,   // .PE
        },
        {0xa2, 0x31, 0x67, 0x72, 0xa5, 0xdf, 0x80, 0x33,
         0x81, 0x67, 0xd8, 0x28, 0xa9, 0x5c, 0xe9, 0x0f,
         0x26, 0x18, 0xcf, 0xde, 0x3d, 0x6e, 0xfd, 0x39},  // .packet_data
        {
            // .fampdata
            3,  // .ver
            3,  // .ver_max
            {
                // .radio_id
                (FAMP_RadioIdType_t)3,  // .id_type
                {
                    // .id
                    .extended = {0xa2, 0x31, 0x67, 0xa5, 0xdf},
                },
            },
            (FAMP_Urgency_t)1,        // .urgency
            0,                        // .stealth
            1,                        // .no_track
```

```
            24,                         // .time
            (FAMP_AircraftType_t)14,    // .acft_type
            3236,                       // .altitude
            {
                // .lat_lon
                -521345084,  // .PN
                230040940,   // .PE
            },
            480,                      // .turnrate
            2184,                     // .speed
            -424,                     // .climb
            396,                      // .track
            (FAMP_MovementMode_t)0,   // .mov_mode
            888,                      // .acc_pos_hor
            124,                      // .acc_pos_ver
            8,                        // .acc_vel
            (FAMP_SIL_t)0,            // .sil
            (FAMP_SDA_t)0,            // .sda
            (FAMP_NIC_t)4,            // .nic
        },
    },
    // Testcase 4
    {
        3647407757,  // .time
        {
            // .reference_lat_lon
            39210529,    // .PN
            -748523182,  // .PE
        },
        {0x95, 0xa9, 0x8a, 0xe2, 0x00, 0x00, 0xc0, 0x33,
         0x21, 0xd1, 0x1f, 0xbf, 0x5b, 0x53, 0x88, 0x39,
         0x98, 0x7a, 0x67, 0xcf, 0x9b, 0x09, 0x5b, 0x3f},  // .packet_data
        {
            // .fampdata
            3,  // .ver
            3,  // .ver_max
            {
                // .radio_id
                (FAMP_RadioIdType_t)2,  // .id_type
                {
                    // .id
                    .flarm = 0x8aa995,
                },
            },
            (FAMP_Urgency_t)3,        // .urgency
            1,                        // .stealth
            1,                        // .no_track
            52,                       // .time
            (FAMP_AircraftType_t)10,  // .acft_type
            143,                      // .altitude
            {
                // .lat_lon
                39416364,    // .PN
                -748571096,  // .PE
            },
            656,                      // .turnrate
            1368,                     // .speed
            304,                      // .climb
            589,                      // .track
            (FAMP_MovementMode_t)0,   // .mov_mode
            22,                       // .acc_pos_hor
```

```
            10,                     // .acc_pos_ver
            3,                      // .acc_vel
            (FAMP_SIL_t)3,          // .sil
            (FAMP_SDA_t)0,          // .sda
            (FAMP_NIC_t)8,          // .nic
        },
    },
    // Testcase 5
    {
        1657145206,  // .time
        {
            // .reference_lat_lon
            737698753,   // .PN
            -551572165,  // .PE
        },
        {0x20, 0x83, 0x43, 0xc2, 0x6c, 0x83, 0xc0, 0x33,
         0x7b, 0xe7, 0x4b, 0x0e, 0xd1, 0xa0, 0x16, 0x78,
         0xee, 0x42, 0xa7, 0xcd, 0x5e, 0xbc, 0xb7, 0xbb},  // .packet_data
        {
            // .fampdata
            3,  // .ver
            3,  // .ver_max
            {
                // .radio_id
                (FAMP_RadioIdType_t)0,  // .id_type
                {
                    // .id
                    .eprid =
                        {
                            0x83438320,  // .base
                            0x6c,        // .rnd
                        },
                },
            },
            (FAMP_Urgency_t)3,       // .urgency
            1,                       // .stealth
            1,                       // .no_track
            24,                      // .time
            (FAMP_AircraftType_t)0,  // .acft_type
            2585,                    // .altitude
            {
                // .lat_lon
                737944480,   // .PN
                -552922410,  // .PE
            },
            236,                     // .turnrate
            472,                     // .speed
            248,                     // .climb
            604,                     // .track
            (FAMP_MovementMode_t)3,  // .mov_mode
            280,                     // .acc_pos_hor
            380,                     // .acc_pos_ver
            72,                      // .acc_vel
            (FAMP_SIL_t)2,           // .sil
            (FAMP_SDA_t)1,           // .sda
            (FAMP_NIC_t)1,           // .nic
        },
    },
    // Testcase 6
    {
        1008426509,  // .time
```

```
        {
            // .reference_lat_lon
            -166961921,  // .PN
            1500956331,  // .PE
        },
        {0xc0, 0x7f, 0xa4, 0x52, 0x00, 0x00, 0x80, 0x33,
         0xac, 0xa2, 0x3c, 0x90, 0xf1, 0x38, 0xdb, 0x4a,
         0x5c, 0x44, 0xf5, 0x0a, 0x08, 0x36, 0x61, 0x95},  // .packet_data
        {
            // .fampdata
            3,  // .ver
            3,  // .ver_max
            {
                // .radio_id
                (FAMP_RadioIdType_t)1,  // .id_type
                {
                    // .id
                    .icao = 0xa47fc0,
                },
            },
            (FAMP_Urgency_t)1,       // .urgency
            0,                       // .stealth
            1,                       // .no_track
            52,                      // .time
            (FAMP_AircraftType_t)4,  // .acft_type
            9028,                    // .altitude
            {
                // .lat_lon
                -167208288,  // .PN
                1500930972,  // .PE
            },
            -912,                    // .turnrate
            3088,                    // .speed
            9,                       // .climb
            603,                     // .track
            (FAMP_MovementMode_t)3,  // .mov_mode
            40,                      // .acc_pos_hor
            4,                       // .acc_pos_ver
            48,                      // .acc_vel
            (FAMP_SIL_t)2,           // .sil
            (FAMP_SDA_t)2,           // .sda
            (FAMP_NIC_t)7,           // .nic
        },
    },
    // Testcase 7
    {
        1891960921,  // .time
        {
            // .reference_lat_lon
            102113980,  // .PN
            605718156,  // .PE
        },
        {0x2c, 0xa6, 0x5a, 0x92, 0x00, 0x00, 0x40, 0x03,
         0xa7, 0x44, 0x1b, 0x2c, 0xaa, 0xfa, 0xf3, 0xf4,
         0x76, 0xce, 0x37, 0x72, 0x50, 0x25, 0x0a, 0x0b},  // .packet_data
        {
            // .fampdata
            3,  // .ver
            0,  // .ver_max
            {
                // .radio_id
```

```
            (FAMP_RadioIdType_t)1,  // .id_type
            {
                // .id
                .icao = 0x5aa62c,
            },
        },
        (FAMP_Urgency_t)2,        // .urgency
        1,                        // .stealth
        0,                        // .no_track
        36,                       // .time
        (FAMP_AircraftType_t)16,  // .acft_type
        5828,                     // .altitude
        {
            // .lat_lon
            102040224,  // .PN
            606113820,  // .PE
        },
        -146,                   // .turnrate
        614,                    // .speed
        60,                     // .climb
        536,                    // .track
        (FAMP_MovementMode_t)1,  // .mov_mode
        824,                    // .acc_pos_hor
        0,                      // .acc_pos_ver
        10,                     // .acc_vel
        (FAMP_SIL_t)2,          // .sil
        (FAMP_SDA_t)0,          // .sda
        (FAMP_NIC_t)6,          // .nic
    },
},
// Testcase 8
{
    3243042182,  // .time
    {
        // .reference_lat_lon
        -181928771,  // .PN
        1093943597,  // .PE
    },
    {0xa4, 0xe1, 0x2e, 0x02, 0xc4, 0xfe, 0x40, 0x33,
     0xb9, 0x8b, 0xc3, 0xb8, 0xa6, 0x55, 0xba, 0x1c,
     0xea, 0xf3, 0x0f, 0x28, 0x64, 0x3c, 0x60, 0xe0},  // .packet_data
    {
        // .fampdata
        3,  // .ver
        3,  // .ver_max
        {
            // .radio_id
            (FAMP_RadioIdType_t)0,  // .id_type
            {
                // .id
                .eprid =
                    {
                        0xfe2ee1a4,  // .base
                        0xc4,        // .rnd
                    },
            },
        },
        (FAMP_Urgency_t)0,        // .urgency
        1,                        // .stealth
        0,                        // .no_track
        24,                       // .time
```

```
            (FAMP_AircraftType_t)18,  // .acft_type
            1950,                     // .altitude
            {
                // .lat_lon
                -182132808,  // .PN
                1093637875,  // .PE
            },
            76,                      // .turnrate
            1388,                    // .speed
            800,                     // .climb
            311,                     // .track
            (FAMP_MovementMode_t)0,  // .mov_mode
            440,                     // .acc_pos_hor
            76,                      // .acc_pos_ver
            72,                      // .acc_vel
            (FAMP_SIL_t)0,           // .sil
            (FAMP_SDA_t)1,           // .sda
            (FAMP_NIC_t)6,           // .nic
        },
    },
    // Testcase 9
    {
        2426921248,  // .time
        {
            // .reference_lat_lon
            189800145,  // .PN
            -56924174,  // .PE
        },
        {0x28, 0xbb, 0x63, 0x22, 0x00, 0x00, 0x40, 0x03,
         0x7a, 0x2d, 0x60, 0xf2, 0x34, 0xf5, 0x0d, 0x2d,
         0x30, 0x8f, 0x7a, 0x36, 0x19, 0x6b, 0xda, 0x9f},  // .packet_data
        {
            // .fampdata
            3,  // .ver
            0,  // .ver_max
            {
                // .radio_id
                (FAMP_RadioIdType_t)2,  // .id_type
                {
                    // .id
                    .flarm = 0x63bb28,
                },
            },
            (FAMP_Urgency_t)0,        // .urgency
            1,                        // .stealth
            0,                        // .no_track
            0,                        // .time
            (FAMP_AircraftType_t)19,  // .acft_type
            1293,                     // .altitude
            {
                // .lat_lon
                189727616,  // .PN
                -56655885,  // .PE
            },
            76,                      // .turnrate
            256,                     // .speed
            -944,                    // .climb
            390,                     // .track
            (FAMP_MovementMode_t)2,  // .mov_mode
            44,                      // .acc_pos_hor
            2,                       // .acc_pos_ver
```

```
            3,                      // .acc_vel
            (FAMP_SIL_t)3,          // .sil
            (FAMP_SDA_t)3,          // .sda
            (FAMP_NIC_t)10,         // .nic
        },
    },
};
```