Document Status

Published status	Confidentiality status
☐ Draft	☐ Internal
☒ Released	☒ NDA
☐ Canceled	☐ Public

Version Control

Ver.	Date	Summary of changes
1.0	2022-11-09	Initial version
1.1	2023-03-10	Re-work and add code samples
1.2	2023-05-16	Add 2.02 release notes
1.3	2023-06-29	Add AcftAlarm_t description
1.4	2024-01-16	Add 2.03 release notes
1.5	2024-10-07	Add 2.04 release notes
1.6	2025-02-05	Add 2.20 release notes
1.7	2025-12-16	Add 2.21 release notes, add FFS, ICAO address configuration description and navigation fix extrapolation.

Table of Contents

1 Abbreviations and Definitions

Term	Meaning/Explanation
CEP	Circular error probable
CPU	Central processing unit
CRC	Cycling Redundancy Check
FIFO	First-in, first-out
FPU	Floating-point unit
FFS	Force flight state
FSK	Frequency shift keying
GNSS	Global Navigation Satellite System
HIL	Hardware-in-the-loop
ISR	Interrupt Service Routing
LBT	Listen-before-talk
MCU	Microcontroller unit
Pub/Sub	Publish/Subscribe
RF	Radio Frequency
RX	Receive
SIL	Software-in-the-loop
TX	Transmit

2 Introduction

libFLARM is a software library implementing part of the FLARM communication and collision avoidance software stack. It takes timing and navigation data as inputs and transmits FLARM radio packets with the correct timing and on the correct frequency. It also receives incoming radio packets, computes collision threats and outputs aircraft data.

The library is deployed in binary form compiled to the customer's specifications. It can be integrated and linked with many typical software toolchains used for embedded development.

libFLARM is completely hardware-agnostic: All access to hardware is abstracted by device drivers and data structures. The host system implements these drivers and passes the driver objects, or the data produced by the hardware devices to the library at run-time. For instance, a radio driver object must be created, implementing functions for sending and receiving RF packets, but also setting channel properties such as the frequency or the transmit power. This is then passed to libFLARM at initialization.

A proprietary publish/subscribe (pub/sub) message broker is further used to pass data and events between library and host application. Data are advertised and subscribed to by name and type (size). Pub/sub includes a FIFO buffer with some (limited) storage in case data is not immediately consumed.

The library employs an embedded-friendly memory model: All memory is allocated statically or at initialization time. No heap allocations are needed (though supported), and memory is never freed. The host application may fine-tune memory allocation by providing a custom allocator, e.g., for putting libFLARM state into specific areas of RAM.

No RTOS is needed to run libFLARM: All state is managed internally, exposed to the host application through an (opaque) handle. The library is simply called periodically by the host system, thereby consuming incoming events, and generating outgoing data. Library calls never block on I/O, i.e., are guaranteed to return quickly.

libFLARM and pub/sub are not thread-safe or re-entrant, though. If running in an RTOS environment, access to these must be protected accordingly.

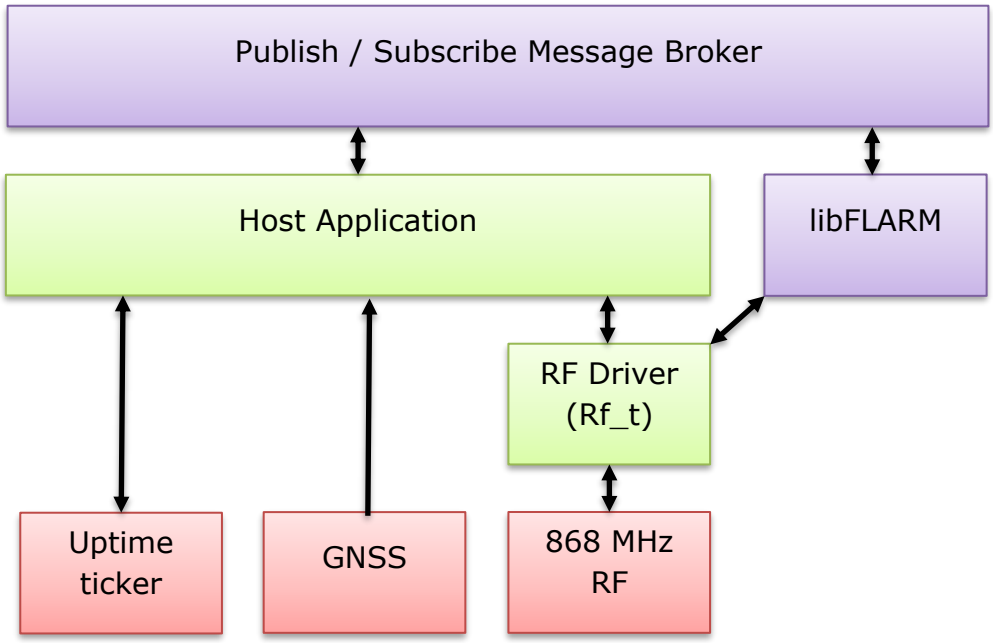
For energy-constrained systems, libFLARM returns information on when it needs to be called next (in absence of external inputs), thus enabling power-saving deep sleep states.

The basic timing unit is a system uptime counter with millisecond accuracy. This ticker is set up by the host application and preferably implemented in hardware, e.g. through a timer interrupt. The uptime value is passed as an argument to every call of the library. The library maintains its own global time base, established from GNSS time and timepulse data, and in sync with other FLARM units.

Runtime errors can occur during an invocation of the library. The host application should provide an error handler for such cases, providing a clear-text (in English) description of the error, as well as an indication of the severity, the domain, and a numerical code.

A transmit-only library is available which contains a reduced functionality set.

This document describes version 2.0 of libFLARM.



3 Requirements

This section describes the requirements on the host system, both hardware and software. This is not a complete list though, please apply own judgement.

3.1 GNSS

The host system must use a high-quality industrial grade GNSS navigation and timing receiver.

The GNSS must support at least a one second measurement update interval. Vertical and horizontal position accuracies as well as the velocity and track accuracies must be given as 1-sigma circular error probably (CEP) values. These values are not available on a standard NMEA-0183 interface, so a proprietary data interface must be used (not available on all GNSS products).

libFLARM requires a timing accuracy of at least 5ms, i.e. the internal time base derived from the GNSS receiver may not deviate from the true time by more than that. Preferably, this is achieved by wiring a time pulse line from the GNSS receiver to the host processor and detecting a logical step via an ISR. Other navigation and timing sources are feasible in principle if they achieve an equal level of accuracy.

At present, the GNSS configuration should use GPS only and no SBAS corrections. A suitable dynamic model must be selected, e.g., Airborne <2G is appropriate in many cases.

We recommend a u-blox gen 8 or newer GNSS receiver.

3.2 FSK Radio

FLARM transmissions are organized as discrete packets, usually of a length of 24 bytes (payload). The structure of such a packet is shown below, as it is transmitted from left to right.

Compliant FLARM packets can be generated with many off-the-shelf FSK transceiver chips. The preamble is not standard, so implementations must be mindful about this. Usually, the non-standard preamble can be worked around, e.g. by including part of it in the sync word etc. Implementations for some popular FSK chips (Semtech, SiLabs) are available on demand.

Implementors should be careful to measure the sensitivity of such setups: Modern FSK chips are good at receiving almost any data at sufficiently high signal strengths, even if the packet format is malformed. Thus, the reception at low signal strengths should be measured carefully to avoid spurious false alerts.

3.2.1 Modulation Parameters

A digital modulation scheme (Gaussian Frequency Shift Keying) is employed to transmit data. The key parameters are given in the table below.

Parameter	Value
Frequency	Europe: 868.2 and 868.4 MHz North America: 902.6 … 928.2 MHz
Modulation	GFSK
Max. Power (ERP)	Europe: 14d Bm / 25 mW North America: Up to 20 dBm / 100 mW
Bitrate	100 kbps
Frequency Deviation	±50 kHz
Gauss Filter BT	0.5

3.2.2 Manchester Encoding

Manchester encoding is applied to the Sync Word, Payload and CRC for better clock synchronization (lower error rate and more reliable transmission), halving the effective bit rate to 50kbps. Manchester encoding is not applied to the Preamble, see below. The encoding is such that a '1' is encoded as '01' and a '0' as '10'.

3.2.3 Bit Order

A data byte is transmitted with the most significant bit first, i.e. the byte 0x31 is transmitted on-air as 00110001 (left to right).

3.2.4 Preamble

The preamble is sent by the transmitter to allow receivers to 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.

3.2.5 Sync Word

The Sync Word is set by libFLARM. Usually, it is 3 bytes long, but this may change. Manchester encoding is to be applied when sending.

3.2.6 Payload

The Payload is transmitted after the sync word. The length is typically 24 bytes.

3.2.7 CRC

A 16-bit cyclic redundancy check (CRC) for message integrity is transmitted after the Payload. The implementation must be based on the C code algorithm below. Be aware that many CRC implementations exist[1], delivering different results.

The CRC state is initialized to `0xffff`. The function `crc_FLARM()` is applied to the bytes of the Sync Word, then the Payload, in the order these data are sent over the RF. The resulting CRC 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;
}
```

3.3 CPU/MCU

The choice of CPU/MCU will depend more on the requirements of the host application than libFLARM. Some of the following requirements are thus indicative only.

- Core: 20 MHz. Non-FPU CPUs may require more cycles.
- 32-bit registers and ALU
- Little endian memory layout

The memory footprint of libFLARM is as follows:

- Code (Flash): 20 kB
- RAM: 5.6 kB plus stack
- RAM (TX-only): 2.6 kB plus stack

3.4 Toolchain

The standard toolchain for building and linking libFLARM is gcc, typically in one of its variants for cross-compilation to embedded targets (e.g. arm-none-eabi-gcc). Other environments may be supported on request. `sizeof(int)` should be at least 4 for platforms where this can be selected. libFLARM specifies the size for all types

[1] Wikipedia: "Various CRC standards extend the polynomial division algorithm by specifying an initial shift register value, a final exclusive OR step and a bit ordering. As a result, the code seen in practice deviates confusingly from pure division, and the register may shift left or right."

specifically and does is insensitive to different handling of bitfields or packed structs.

When linking, the math library should be added (-lm flag for gcc) and a `printf()` function capable of outputting 32bit values (for error callback) must be enabled.

4 Using libFLARM

4.1 Hardware Abstraction

4.1.1 Time

libFLARM abstracts time by taking an `uptime` argument to all its invocation. This must represent the uptime (time duration the host system has been running) in milliseconds. Uptime time stamps are defined as:

```
/// Canonical type to depict the system uptime tick in milliseconds.
typedef uint64_t Milliseconds_t;
```

The resolution is milliseconds. The host system typically implements this using a hardware timer that automatically increments the uptime ticker. The accuracy should be better than 0.1%, i.e. 1ms deviation per second max. The offset of the ticker, i.e. when it started running, is irrelevant to libFLARM.

4.1.2 Radio Driver

libFLARM is completely abstracted from and agnostic to hardware. To let it operate the radio transceiver, the host application must construct a driver object and pass this to libFLARM during initialization. The driver is of type `Rf_t`, defined by

```
typedef struct {
  const RfOps_t *ops;
  // Implementor may add more state here,
  // For instance, add a
  // copy of current configuration object.
} Rf_t;
```

The implementor must thus define the method table (`RfOps_t`), instantiate an `Rf_t` (or equivalent) object, then pass the pointer to this to libFLARM.

The method table is given by

```
typedef struct {
  bool (*SetConfig)(Rf_t *self, const RF_Config_t *c);
```

```
  bool (*GetConfig)(Rf_t *self, RF_Config_t *c_out);
  int (*Available)(Rf_t *self);
  bool (*Ready)(Rf_t *self);
  bool (*GetFrame)(Rf_t *self, RfFrame_t *f_out, int16_t *rssi,
                   uint64_t *receivedAt);
  bool (*PeekFrame)(Rf_t *self, RfFrame_t *f_out, int16_t *rssi,
                    uint64_t *receivedAt);
  bool (*SendFrame)(Rf_t *self, const RfFrame_t *f);
} RfOps_t;
```

All these methods must be implemented except for `GetFrame()`, which can be omitted (set to NULL) for the TX-only variant. Note that the methods receive a reference to the object as the first argument, so the implementation may access the data attached to the object.

Calls to `GetFrame()` and `SendFrame()` must not block irrespective of the state of the radio. If no data is available (for receive) or sending is not possible, then the functions shall return false. We recommend using a small packet queue for reception to not lose packets if libFLARM processing is delayed (e.g. due to latency in the host application).

The `Ready()` function shall test the channel for an ongoing transmission according to the rules for Polite Spectrum Access in ETSI EN 300 220. libFLARM will autonomously re-schedule the transmission if the channel is busy.

The structure of `Rf_t` allows for additional methods and data to be added to the object for usage by the host application ("inheritance", derived object), as long as the layout of the resulting object remains compatible with `Rf_t`, e.g.:

```
// Deriving from Rf_t
typedef struct {
  Rf_t base;
  // Add data + methods
} MyRf_t;
```

The configuration struct is defined as:

```
typedef struct {
  /// Frequency in 0.1MHz resolution
  int freq;
  /// Transmit power in dBm -10..20
  int8_t tx_power_dbm;
  /// Size of syncword in bytes (1..4)
```

```
  int8_t syncword_size;
  /// The sync word, comprising up to four bytes.
  /// The sync word must be transmitted starting with array index 0
  /// (syncword[0]), most significant bit first.
  /// If syncword_size < 4, then the higher entries of
  /// syncword[] are to be ignored.
  uint8_t syncword[4];
  /// RX payload width, in bytes (1..32)
  int8_t payload_size;
} RF_Config_t;
```

The supported frequencies and transmit powers depend on the region where it is intended to operate the device. They are given in Section 3.2.1. For transmit power, the implementation must not use more power than requested by libFLARM, but it is safe to use less.

4.1.3 GNSS

Implementation of the GNSS as navigation and timing source is part of the host application. Navigation data is injected via pub/sub into libFLARM, using this data structure:

```
/// GNSS navigation data.
typedef struct {
  /// Degree Latitude scaled with 1e-7, north positive
  int32_t latitude;
  /// Degree Longitude scaled with 1e-7, east positive
  int32_t longitude;
  /// Meters Height above GPS Ellipsoid (not MSL)  scaled with 1e-0
  int16_t altitude;
  /// horizontal position accuracy in cm
  int16_t horizontalAccuracy;
  /// vertical position accuracy in cm
  int16_t verticalAccuracy;
  /// Northward (north = positive) Speed [cm/sec]
  int16_t northSpeed;
  /// Eastward  (east  = positive) Speed [cm/sec]
  int16_t eastSpeed;
  /// Vertical    (up  = positive) Speed [cm/sec]
  int16_t verticalSpeed;
  /// Horizontal (ground) Speed [cm/sec]
  int16_t horizontalSpeed;
  /// Heading in degrees scaled with 1e-1 (0-3600)
  int16_t track;
```

```
  /// Horizontal Speed accuracy cm/sec
  int16_t horizontalSpeedAccuracy;
  /// Heading accuracy in degrees scaled with 1e-1 (degrees times 10)
  int16_t trackAccuracy;
  /// Time, in seconds since Unix epoch
  uint64_t time;
  /// Current GPS fix quality
  GpsFix_t fix;
} Gps_t;
```

Horizontal position accuracies shall be given as a 1-sigma (i.e. ~68%) probability of being in a circle of the given radius around the indicated position (CEP). The other accuracies are 1-dimensional 1-sigma probabilities, i.e. standard deviations. For altitude, the WGS-84 ellipsoid is used throughout, i.e. not AMSL.

The time stamp has a one second resolution and references the time of measurement within the GNSS receiver. Only measurements at the full second shall be considered. The UNIX epoch is used as base (`time == 0`), which as January 1st 1970. UTC is used throughout (i.e. no time zones).

As another aspect, a precise time pulse event must be provided, coinciding with the full second. The time pulse is typically a physical (digital) signal transmitted by the GNSS, which can be captured and timed with respect to the uptime by the host system using an interrupt.

While other methods exist - such as deducing the exact time from the serial communication – these are not recommended. An accuracy of better than 5ms must be achieved for proper functioning of libFLARM in either case. This includes the absolute accuracy, i.e. errors vs. world time, as well as jitter / precision.

When receiving the time pulse, the host system shall store the uptime in milliseconds when the pulse happened. This information is passed to libFLARM to allow for some latency between the pulse happening and the next invocation of the library.

Both the navigation solution and the time pulse are passed to libFLARM via pub/sub, see below. The time pulse appears at the full second. Usually, the navigation solution is slightly after the pulse due to the processing time of the GNSS. For typical GNSS systems, the navigation solution appears 50 to 200 ms after the full second, depending on the configuration.

During operation, the navigation solution shall be applied to libFLARM no later than 450 ms after the full second.

4.2 Initializing libFLARM

4.2.1 Publish/Subscribe

Publish / Subscribe (pub/sub) is a module delivered with libFLARM. It constitutes a simple message broker whereby data entities ("topics") are referenced by their names and sizes. It provides a one-to-many distribution of data: Only one advertiser can publish a topic, but many subscribers can receive it. An advertiser/publisher may add a FIFO buffer – space which is allocated by the advertiser – to support subscribers catching up with missing data. The publisher may also only allocate space for one datum (no FIFO), or no data at all (event-only topic).

Pub/sub is initialized by the host application:

```
// Allocate memory for pubsub. Can be static, heap etc.
// Memory must be valid throughout usage of libFLARM.
static PubSubStore_t pubsub = {0};

// Initialize pubsub, watch for errors
PubSubError_t err = pubsub_Init(&pubsub);
if (err != PUBSUB_ERROR_NONE) {
  // ... deal with error
}
```

Note that we use the static keyword to indicate that this memory must be allocated and valid throughout the lifetime of the application. Other means to achieve this (beside static) exist, e.g., allocating on heap.

Next, the host application advertises its topics (TIMEPULSE, GPS). For this, it must reserve space for data storage within pub/sub, and for the publish handles that it can later use to publish data on pub/sub.

```
static struct pub_handle {
  PubSubHandle_t timepulse;
  PubSubHandle_t gps;
};
static struct pub_buf {
  Milliseconds_t pub_timepulse[1];
  Gps_t pub_navigation[1];
};
```

The use of structs is only instructional here. Note the array notation in pub_buf, indicating that these provide storage to the pub/sub. Since only one element is allocated, this is not strictly a FIFO.

Next the host advertises the topics:

```
PubSubError_t err = pubsub_Advertize(
    &pubsub,                // The broker
    &pub_handle.timepulse, // Use this handle to publish
    &(PubSubTopicIdentifier_t){"TIMEPULSE", sizeof(Milliseconds_t)},
    &pub_buf.timepulse,       // Storage to use for FIFO
    sizeof(pub_buf.timepulse) // Storage size
);
if (err != PUBSUB_ERROR_NONE) {
  // ... Deal with error
}

PubSubError_t err = pubsub_Advertize( //
    &pubsub,                          //
    &pub_handle.gps,                  //
    &(PubSubTopicIdentifier_t){"GPS", sizeof(Gps_t)},
    &pub_buf.gps,       //
    sizeof(pub_buf.gps) //
);
if (err != PUBSUB_ERROR_NONE) {
  // ... Deal with error
}
```

After libFLARM is initialized and has advertised its own topics (see below), the host application can subscribe:

```
static struct sub_handle {
  PubSubHandle_t acft_alarm;
  PubSubHandle_t acft_info;
};

PubSubError_t err = pubsub_Subscribe(
    &pubsub, sub_handle.acft_alarm,
    &(PubSubTopicIdentifier_t){"ACFT_ALARM", sizeof(AcftAlarm_t)});
if (err != PUBSUB_ERROR_NONE) {
  // ... Deal with error
}
PubSubError_t err = pubsub_Subscribe(
    &pubsub, sub_handle.acft_alarm,
```

```
    &(PubSubTopicIdentifier_t){"ACFT_INFO", sizeof(AcftInfo_t)});
if (err != PUBSUB_ERROR_NONE) {
  // ... Deal with error
}
```

No storage is needed for subscribing to topics apart from the subscribe handle.

Publishing to pub/sub during operation is now straightforward:

```
// Variable containing the time of the arrival
// of the time pulse. This can have a local scope.
Milliseconds_t ticker_timepulse = ...;

PubSubError_t err =pubsub_Publish(&pubsub, pub_handle.timepulse,
&ticker_timepulse);
if (err != PUBSUB_ERROR_NONE) {
  // ... Deal with error
}
```

Consuming data is almost equivalent. Since ACFT_ALARM and ACFT_INFO have a (small) FIFO buffer allocated, consuming is typically performed in a loop to retrieve all items:

```
// local storage for aircraft data
AcftAlarm_t acft = {0};
while (PUBSUB_ERROR_NONE ==
       pubsub_Next(&pubsub, sub_handle_acft_alarm, &acft)) {
  // do something with acft
}
```

Pub/sub will indicate to the subscriber if data was lost during calls to `pubsub_Next()` via the `PUBSUB_ERROR_OVERFLOW` error code. If no new data is available, the code `PUBSUB_ERROR_NO_DATA_AVAILABLE` is returned. Other pub/sub APIs are not needed to operate libFLARM.

4.2.2 Memory Allocator

The first phase of the initialization of libFLARM, `libflarm_Construct()`, requires allocating memory. The host application thus constructs a custom allocator object which provides an `malloc()` method, allowing the library to allocate memory. The allocator is not used in any other call to libFLARM. The allocator may use heap memory, or any other appropriate type of memory. The allocated memory must be preserved while libFLARM is being used.

Memory is never deallocated by libFLARM, not even in the case of an error during construction or initialization. If the host application wishes to recover the memory, it simply should not call libFLARM any further, thus the allocated memory can be re-used.

The allocator object is defined by:

```
typedef struct CasAllocator_s CasAllocator_t;
struct CasAllocator_s {
  void * (*malloc)(size_t size, CasAllocator_t* self);
};
```

As with `Rf_t`, the implementor may extend the structure by adding state data at the end ("inheritance"). A (useful) sample implementation of this is given by `FlarmCasAllocator_t`.

If for any reason the allocator cannot produce the required amount of memory the construction and subsequent initialization of the library will fail.

4.2.3 Random Generator

libFLARM uses a customizable random generator to both enable deterministic behavior when needed (e.g. SIL testing), or fully random behavior when preferable. The random generator is passed as a `Random_t` object, following the design pattern of the allocator discussed above:

```
typedef struct Random_s Random_t;
struct Random_s {
    uint32_t (*nextRandom)(Random_t* self);
};
```

Using a standard random generator (e.g. C's `random()`) is fine if it is properly seeded, and the width of the return values match `Random_t` (i.e. 32-bit)

4.2.4 Error Callbacks

A mechanism for returning error conditions within libFLARM to the caller is introduced by means of error callbacks. This is an error handler passed to the library which it calls when an error occurs. Its definition is given by:

```
typedef struct {
  void *context;
  void (*errorCallback)(void *context,       //
                        Severity_t severity, //
```

```
                        const char *domain,   //
                        int error,            //
                        const char *message);
} ErrorCallback_t;
```

The callback handler is called synchronously, i.e. during any invocation of libFLARM. The implementor can add a `context` pointer which is included in all callbacks. Using the callback is optional. A typical use case is to write error callbacks into a log file.

Also see Section 4.5.

4.2.5 Library Initialization

libFLARM is initialized in two stages:

The **construction stage** is started by calling `libflarm_Construct()`:

```
CasAllocator_t my_allocator = ...;
Rf_t my_radio_driver = ...;
Random_t my_random_generator = ...;
ErrorCallback_t my_error_hanlder = ...;

LibFLARMState_t *lib = libflarm_Construct(, &self->pubSubStore, //
                                          &my_radio_driver,     //
                                          &my_random_generator, //
                                          &my_error_handler);
if (NULL == lib) {
  // Error
}
```

The returned pointer (`lib`) is opaque and must be persisted while libFLARM is being used. In case of error (e.g., not enough memory), the function will return NULL.

During construction, libFLARM will advertise all its pub/sub topics, but not subscribe to any yet – this happens in the second stage. This allows for an orderly construction of the data flow within libFLARM.

The **initialization stage** is next started by calling `libflarm_Initialize()`. The host app needs to have its topics (GPS, TIMEPULSE) published before this:

```
if(false == libflarm_Initialize(lib, my_serial, my_aircrafttype) {
  // Erorr
}
```

The serial number is assigned by the host application. Serial number should be used contiguously, i.e. not introducing any holes in the range. Every instance of libFLARM must use a unique serial number.

After both initialization stages are complete, the host app can subscribe to the topics provided by libFLARM.

4.3 Configuration

A key-value configuration system is used:

```
bool libflarm_Set_config(LibFLARMState_t* self, const char* key, char* value);
bool libflarm_Get_config(LibFLARMState_t* self, const char* key, char* buf,
size_t bufSize);
```

All configuration items are set and read as strings, though their contents can be numeric. Available configuration items are:

- SERIAL_NO: Read-only, returns the serial number given in the initialization
- IDTYPE: Read-only. Address type used by libFLARM, can be "RANDOM", "ICAO", "FLARM", or "INVALID". Currently this is fixed to "FLARM".
- ID: Read-only. Address, depending on IDTYPE
- ACFT: Aircraft type, see FTD-014
- THRE: In-flight threshold, see FTD-014
- RANGE: Horizontal range, see FTD-014
- VRANGE: Vertical range, see FTD-014
- NOTRACK: Disable tracking, see FTD-014

For more options on address settings, see Section 4.6.2.

4.4 Operation

4.4.1 Process()

The basic execution model requires the host app to call `libflarm_Process()` preferably once per ticker increment (i.e. once per milliseconds) with a maximum interval of 5 tickers (5 milliseconds) in between consecutive invocations. As explained earlier, the current system uptime is passed as argument to the library. This call never blocks I/O, assuming a correct implementation of the `Rf_t` driver.

The following snippet clarifies the call order:

```
void mainloop() {
  while (true) {
    // retrieve the ticker value from the hardware timer
    uint64_t uptime_ms = ticker_ms();
    if (false == libflarm_Process(lib, uptime_ms, NULL)) {
      // Error
    }
    // Do other stuff
    // ...
    // Sleep until the next ticker
    sleep_ms(1);
  }
}
```

Note that this snippet does not show the passing of data and events through pub/sub.

For energy-constrained applications, `libflarm_Process()` provides a "deadline" for the next invocation, expressed in uptime. The host app may delay processing the library until that deadline to conserve energy. In absence of external signal such as navigation data, time pulse, or received radio packets, the library is guaranteed not to change its state within the indicated period.

The deadline has three states:

- `DEADLINE_INFINITE` (value UINT64_MAX) means the host can sleep until an external stimulus occurs (such as navigation data, time pulse, received radio packets).
- `DEADLINE_UNDEFINED` (value 0) means that no valid deadline is given, and the library must be called as soon as possible.
- Any other value is a valid deadline. The host may sleep until that uptime tick, or until an external stimulus occurs (whatever happens first).

The host application must thus detect such external inputs and invalidate the deadline, resuming processing of libFLARM.

The following snippet shows this alternative use:

```
void mainloop() {
  while (true) {
    uint64_t uptime_ms = ticker_ms();
    Milliseconds_t deadline = 0;
    if (false ==
        libflarm_Process(g_libFlarmState, uptime_ms, &deadline)) {
```

```
      // Error
    }
    // Perform tasks, such as consuming pub/sub data.
    if (deadline != DEADLINE_INFINITE) {
      // We can now sleep until the deadline.
      // Any incoming I/O (radio, nav data, time pulse)
      // should interrupt the sleep and thus initiate
      // the next call to  libflarm_Process().
      sleepUntil(&mutex, deadline);
    } else {
      // No deadline, libFLARM is idle. Can sleep
      // indefinitely, until something happens.
      sleepUntil(&mutex, getSystemTickerValue() + 10000);
    }
  }
}
```

The `sleepUntil()` function sleeps until a certain uptime tick has been reached, or an external input (RF packet received, GNSS solution or time pulse) requires `libFLARM_process()` to be called. The mutex argument is only exemplary, indicating that you will have to consider concurrency in your implementation.

4.4.2 Data to Library

The host application is required to advertise and publish the following topics on pub/sub:

Topic	Data type	Description
TIMEPULSE	Milliseconds_t	Time pulse event. Provide high-accuracy ticker timestamp as payload, created e.g. in the ISR. Publish as soon as the pulse arises, e.g. at the full seconds.
GPS	Gps_t	Navigation solution as determined by the GNSS receiver. Publish within 150 ms after the full second.

4.4.3 Data from Library

The host application may subscribe to the following topics:

Topic	Data type	Description
ACFT_ALARM	AcftAlarm_t	Aircraft traffic and alarm information. This is published by libFLARM at most once per second per target, and usually immediately when the target is received.

ACFT_INFO	AcftInfo_t	Aircraft static information such as device type or whether a transponder is present. This is broadcast by FLARM units in loose intervals of approximately five to ten seconds.

The AcftAlarm_t type is defined as follows:

```
/// Structure for publishing FLARM traffic information and collision
/// warning information. The name is misleading: It transports traffic
/// information even if no alarm situation is detected.
typedef struct {
  /// Type of alarm. Deprecated and redundant, please don't use.
  AlarmType_t alarmType;
  /// Radio ID (with ID type) of this traffic.
  /// If a traffic uses a randomized ID, then this is not necessarily
consistent
  /// over time for a given traffic.
  RadioId_t ID;
  /// The aircraft type of this traffic.
  AircraftType_t AcftType;

  /// Danger level, as calculated from the TTI.
  /// Note that mapping TTI -> level may vary depending on the configuration
  /// and parametrization of libFLARM.
  Danger_t level;
  /// Time-To-Impact in seconds, or TTI_NONE if no impact is predicted.
  uint8_t TTI;

  /// Traffic position, global coordinates.
  /// This is extrapolated into the future by 2 seconds to allow
  /// for some delay in the display system, while providing the
  /// intuitively correct position to the pilot.
  struct {
    /// Latitude, WGS-84, in 1E-7 degrees, northern hemisphere is positive.
    int32_t lat;
    /// Longitude, WGS-84, in 1E-7 degrees, eastern hemisphere is positive.
    int32_t lon;
    /// Altitude above the WGS-84 ellipsoid, in metres.
    int32_t alt;
  } pos_2sec;
```

```
  /// Ground track of traffic 1/10 degrees, or INT16_MIN if undefined.
  int16_t oppTrackDegrees;
  /// Ground speed of traffic in m/s.
  int16_t GroundSpeed;
  /// Traffic climb rate in cm/s
  int16_t ClimbRate;
  /// Traffic movement mode, see MovMode_t
  MovMode_t movMode;

  /// Traffic position, relative to ownship,
  /// in meters North / Easat
  int32_t RelativeN, RelativeE;
  /// Horizontal distance of traffic vs. ownship, in meters.
  int32_t rHdist;
  /// Vertical traffic position, relative to ownship..
  /// Positive values means traffic is higher than ownship.
  int16_t rVdist;

  /// Weighted distance of traffic for determining priority when displaying
  /// traffic to the pilot. Traffic with smaller NEARdist values should
  /// be displayed first if screen real estate is constrained.
  int32_t NEARdist;

  /// Relative bearing in degrees; angle in degrees under which ownship's
  /// pilot sees the traffic, from -180 to 180. Positive values are clockwise.
  /// E.g. 60 -> traffic is at 2 o'clock.
  int16_t direction;

  /// Traffic has stealth mode enabled; navigation data may
  /// be degraded. See FTD-014 for reference.
  bool StealthTarget;
  /// Traffic declares intent to not be tracked by ground-based
  /// tracking system. If set to True, this data MUST not be forwarded
  /// to such systems or stored permanently. Usage for safety concerns
  /// is acceptable.
  bool noTrack;
} AcftAlarm_t;
```

The full reference und the various sub types are given in cas_collision_types.h. The ACFT_INFO topic cannot currently be consumed.

4.5 Error Handling

4.5.1 Return Values

A common pattern in libFLARM is to use function return values to indicate error states. Most bool-valued functions will return false on error. Most pointer-valued functions will return NULL on error.

For instance, the `libflarm_Set_config()` and `libflarm_Get_config()` functions will signal success of the operation via the return value.

4.5.2 Error Callback

The optional error callback function can be provided at library construction time. libFLARM will call the handler on certain error conditions, providing detailed error descriptions. A typical implementation of the handler would e.g., write the error code, severity, domain, and message to a log file. This allows the implementor e.g., to inspect errors in the RF system. Error logs are particularly helpful to provide support during the R&D and integration phase.

4.5.3 Expiration (deprecated)

libFLARM used to have a hard-corded expiration date after which it stops working. The API still exists. The date can be queried with the following call:

```
time_t libflarm_Get_ExpirationDate(void);
```

4.6 Extensions

Optional extensions are available to extend the core libFLARM functionality. Extensions may be used individually or in combination.

4.6.1 Forced Flight State

A pub/sub topic `FFS` is available to override the built-in flight state detection algorithm. This can be used when the host system has other, more reliable means to detect the flight state, e.g. through a weight-on-wheel sensor.

This topic must be advertised after construction via `libflarm_Construct()`, but before initialization of libFLARM via `libflarm_Initialize()`. Otherwise, published items will be ignored.

The topic uses the `ForceFlightState_t` type. The library consumes data published to this topic and sets the flight state accordingly. Expected values are `FORCEFLIGHTSTATE_ON_GROUND` or `FORCEFLIGHTSTATE_IN_FLIGHT`. This forced flight state expires 20 seconds after being published, and needs to be renewed periodically, e.g., every 10 seconds. Publishing `FORCEFLIGHTSTATE_NONE` disables the override immediately, i.e. the library returns to the default algorithm for detecting the flight state.

```
// libFLARM construction
// ... libflarm_Construct(...);

// Advertize FFS publisher

ForceFlightState_t ffsBuffer;
PubSubError_t err = pubsub_Advertize( //
    &pubsub,                          //
    &pub_handle.ffs                   //
    &(PubSubTopicIdentifier_t){"FFS", sizeof(ForceFlightState_t)},
    &ffsBuffer,       //
    sizeof(ffsBuffer) //
);
if (err != PUBSUB_ERROR_NONE) {
  // Handle advertizement failure
}

// libFLARM initialization
// ... libflarm_Initialize(...);
```

To publish a force flight state request, use:

```
// publish in-flight state periodically, e.g. every 10 seconds
ForceFlightState_t ffs = FORCEFLIGHTSTATE_IN_FLIGHT;
pubsub_Publish(&pubsub, pub_handle.ffs, &ffs);
```

4.6.2 Aircraft (ICAO) Address

For installations in aircraft with transponders, libFLARM allows setting the aircraft (ICAO) address . If enabled, the following configuration items can be accessed:

- ICAOADDRESS: A six-digit hex string representing the ICAO address of the aircraft.
- IDTYPE: One of "FLARM" (default) or "ICAO". When setting this to "ICAO", the ICAOADDRESS configuration item must be set beforehand. `libFLARM_Set_config()` returns `false` if ICAOADDRESS is not a valid address.

For example, to configure an ICAO address of 4BABCD, first, configure the ICAOADDRESS with string "4BABCD", and second, set the IDTYPE to "ICAO".

For example, to configure an ICAO address of 4BABCD, first set ICAOADDRESS to "4BABCD", then set IDTYPE to "ICAO":

```
libflarm_Set_config(libFlarmState, "icaoaddress", "4BABCD");
libflarm_Set_config(libFlarmState, "idtype", "ICAO");
```

4.6.3 Extrapolation of Navigation Fix

When operating libFLARM, correct timing of GNSS data is critical. A `Gps_t` fix shall be supplied to the library within the timing constraints given in Section 4.1.3.

In system configurations where the GNSS fix is not available within this time window, the current navigation solution may be extrapolated from the previous one via the `cas_extrapolation` module.

The module operates independently of the main libFLARM processing chain and maintains an internal motion state and extrapolates the navigation fix to the next second using a constant turn rate and velocity (CTRV) model.

Once extrapolation is enabled, it must be used continuously. Conditional use (i.e. applying extrapolation only when a timely fix is missing) will result in an inconsistent internal state and incorrect extrapolation behavior and shall be avoided.

Detailed API documentation for the extrapolation module is provided in the corresponding header file.

4.6.3.1 Module Lifecycle

The extrapolation module follows the standard lifecycle as introduced in Section 4.2.5. The additional memory footprint introduced by the extrapolation module is 44 bytes.

Example (simplified):

```
// This block must be called during initialization.
static CasExtrapolationState_t* myExtrapolationState =
cas_extrapolation_Construct(myAllocator);
if (myExtrapolationState == NULL) {
  // Handle allocation failure
}
cas_extrapolation_Init(myExtrapolationState);

// In the main processing loop, for each GPS epoch:
if (!cas_extrapolation_Extrapolate(myExtrapolationState, &myGpsData)) {
  // Handle extrapolation failure (e.g., invalid parameters or GPS fix not 3D)
}
```

4.6.3.2 Operation

Extrapolation is performed by calling the extrapolation function `cas_extrapolation_Extrapolate()` once per second. The function operates directly on the provided `Gps_t` structure, modifying its contents in place while updating the internal extrapolation state. The modified `Gps_t` structure can then be published normally on the "GPS" topic.

A valid GNSS fix shall be provided for extrapolation to succeed. The fix field must indicate a valid 3D position (`GPSFIX_3D`). Fixes of other types are rejected.

When a valid GNSS fix is provided, the internal motion state is aligned to that fix before extrapolation. This ensures continuity of the internal model and prevents drift.

During initial operation, the extrapolation module may not yet have sufficient historical data to estimate turn rate. In this case, extrapolation is performed with zero turn rate until enough data has been accumulated.

4.6.3.3 Integration Considerations

Navigation fix extrapolation is intended for cases where navigation solution latency cannot reliably meet the required timing constraints. It does not replace proper GNSS synchronization.

Integrators shall ensure that:

- The extrapolation function is invoked at a fixed one-second interval
- All `Gps_t` data passed to libFLARM has been processed by the extrapolation module once it is enabled
- The extrapolation module remains active and is used for the lifetime of the GNSS data stream

4.7 Compliance Test

For a full, black-box, HIL verification of the libFLARM implementation, our Compliance Test can be used, see FTD-069: FLARM Compliance Test Manual.

The compliance test is especially useful in finding timing discrepancies, e.g., offset or jitter on the time pulse.

5 Version History and Release Notes

Version	Date	Description
2.00 (734f6c059)	12.09.2022	Initial library release Expires 1.2.2024
2.01 (698cc5197)	09.03.2023	Adapt deadline and example code Expires 1.7.2024
2.02 (050d97247)	16.05.2023	Fix decoding issues Expires 1.11.2024
2.03 (82ea2922d)	16.01.2024	Reduce library size Expires 1.3.2025
2.04 (75699e736)	07.10.2024	Maintenance release Expires 1.3.2026
2.20 (23bebb140)	05.02.2025	Remove expiration
2.21 (b0e434b34)	16.12.2025	Add force flight state topic, configurable ICAO address and navigation fix extrapolation.

5.1 Changes in libFLARM 2.21

Add force flight state topic, optionally available configurable ICAO address and navigation fix extrapolation.

5.2 Changes in libFLARM 2.20

Remove firmware expiration (for technical reasons, expiration still exists but is set to the year 2099).

5.3 Changes in libFLARM 2.04

Maintenance release.

5.4 Changes in libFLARM 2.03

Reduced library size by using more strict size optimizations.

5.5 Changes in libFLARM 2.02

Fixed an issue with received aircraft positions not decoded correctly on the southern hemisphere after March 2024.

5.6 Changes in libFLARM 2.01

The documentation and examples have been reworked. The deadline feature has been significantly simplified. A header file (`cas_pinvct_legacy.h`) is included to maintain backwards compatibility.

5.7 Changes in libFLARM 2.00

The library was renamed from libCAS to libFLARM. It now provides the main state structure as a pointer result of the construction function, and all memory is handled internally. Therefore, the implementor must provide a `CasAllocator_t` implementation.

This has substantial impact on the existing API in the following way:

- All API functions have a prefix change from `libcas*` to `libflarm*`
- The construction function returns a pointer to the state structure. The state structure is NOT public and cannot be defined or held by the implementor anymore.
- The initialization function requires a random abstraction provided by the implementor.
- Pub/sub topic definitions are wrapped in a new type (`PubSubTopicIdentifier_t`)
- Most API functions now require the state (`self`) structure as the first argument.
- `libflarm_Get_ExpirationDate()` now returns `time_t` instead of a `struct tm*`

Furthermore, the `libflarm_Process()` function now allows for two modes of operation, continuous (called as often as possible) or deadline (where the returned parameter deadline is used to determine the time to invoke it). The latter allows to implement significant power savings.

Appendix – End User License Agreement (EULA)

By purchasing or using a FLARM device or by downloading, installing, copying, accessing, or using any FLARM Technology Ltd, Zug, Switzerland (hereafter "FLARM Technology") software, firmware, license key, or data, you agree to the following terms and conditions. If you do not agree with the terms and conditions do not purchase or use the FLARM device and do not download, install, copy, access, or use the software, firmware, license key, or data. If you are accepting these terms and conditions on behalf of another person, company, or other legal entity, you represent and warrant that you have full authority to bind that person, company, or legal entity to these terms and conditions.

If you are purchasing or using a FLARM device, the terms "firmware", "license key", and "data" refer to such items installed or available in the FLARM device at time of purchase or use, as applicable.

1. **License and Limitation of use**
   - 1.1. **License**. Subject to the terms and conditions of this Agreement, FLARM Technology hereby grants to you a non-exclusive, non-transferable right to download, install, copy, access, and use the software, firmware, license key, or data in binary executable form solely for your own personal or internal business operations. You acknowledge that the software, firmware, algorithms, license key, or data and all related information are proprietary to FLARM Technology and its suppliers.
   - 1.2. **Limitation of use**. Firmware, license keys, and data may only be used as embedded in and for execution on devices manufactured by or under license from FLARM Technology. License keys and data may only be used in the specific devices, by serial number, for which they were sold or intended. Software, firmware, license keys, and data with an expiration date may not be used after the expiration date. Right to download, install, copy, access, or use software, firmware, license key, or data with an expiration date does not imply right to upgrade or extension of the license beyond the expiration date. No other licenses are granted by implication, estoppel or otherwise.
2. **Terms of use of FLARM**
   - 2.1. Every FLARM installation must be approved by licensed Part-66 certifying staff or the national equivalent. A FLARM installation requires an EASA Minor Change Approval or the national equivalent.
   - 2.2. FLARM must be installed according to the Installation Instructions and the EASA Minor Change Approval, or the national equivalent.
   - 2.3. FLARM cannot warn in all situations. In particular warnings may be incorrect, late, missing, not being issued at all, show other threats than the most dangerous or distract the pilot's attention. FLARM does not issue resolution advisories. FLARM can only warn of aircraft that are equipped with FLARM, SSR transponders (in specific FLARM devices), or of up-to-date obstacles stored in its database. The use of FLARM does not allow a change of flight tactics or pilot behavior. It is the sole responsibility of the pilot in command to decide upon the use of FLARM.
   - 2.4. FLARM may not be used for navigation, separation, or under IMC.
   - 2.5. FLARM does not work if GPS is inoperative, degraded, or unavailable for any reason.
   - 2.6. The most recent Operating Manual must be read, understood and followed at all times.
   - 2.7. The firmware must be replaced once per year (every 12 months). The firmware must also be replaced earlier if a Service Bulletin or other information is published with such instruction. Failure to replace the firmware may render the device inoperable or incompatible with other devices, with or without warning or notice thereof.
   - 2.8. Service Bulletins are published as a Newsletter by FLARM Technology. You are required to sign up for the Newsletter on www.flarm.com to ensure that you are informed of published Service Bulletins. If you are entering into this agreement in a form where your email address is available (e.g. online shop) you may be automatically signed up for the Newsletter.
   - 2.9. After power-up, FLARM performs a self-test which must be monitored by the pilots. If a malfunction or defect is observed or suspected, FLARM must be disconnected from the aircraft by maintenance before the next flight and the device inspected and repaired, as applicable.
   - 2.10. The pilot in command is solely responsible to operate FLARM according to applicable national regulations. Regulations might include, but are not limited to, airborne usage of radio frequencies, aircraft installation, safety regulations, or regulations for sports competitions.
3. **Intellectual Property**. No part of the software, firmware, license keys, data (including obstacle databases), the FLARM radio protocol and messages, and the FLARM hardware and design may be copied, altered, reverse engineered, decompiled or disassembled without an explicit and written approval by FLARM Technology. Software, firmware, license keys, data (including obstacle databases), the FLARM radio protocol and messages, the FLARM hardware and design, and the FLARM logos and name are protected by copyright, trademark and patent laws.
4. **Manipulation**. It is forbidden to intentionally feed artificially generated signals to the FLARM device, its GPS antenna or the external/internal GPS antenna connections, unless agreed with FLARM Technology in writing for limited R&D activities.
5. **FLARM Data and Privacy**
   - 5.1. FLARM devices receive, collect, store, use, send, and broadcast data to enable the system to work, improve the system, and to enable troubleshooting. This data may include, but is not limited to, configuration items, aircraft identification, own positions, and such data of other aircraft. FLARM Technology may receive, collect, store, and use this data for said or other purposes including Search and Rescue (SAR).
   - 5.2. FLARM Technology may share data with its partners for aforementioned or other purposes. FLARM Technology may in addition publicly make available data from a FLARM device (Flight Tracking). If a FLARM device has been configured to limit tracking, SAR and other services may not be available.
   - 5.3. Data sent or broadcast by FLARM devices may only be used at own risk and under the same conditions as the FLARM device itself, and is encrypted partially 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 as signed and ratified by most countries respectively its national implementations. FLARM Technology is not responsible for any third party device, software, or service receiving, collecting, storing, using, sending, broadcasting, or making publicly available data regardless of whether legally or illegally.

6. **Warranty, Limitation of Liability, and Indemnification**
   - 6.1. **Warranty**. FLARM devices, software, firmware, license keys, and data are provided on an "as is" basis without warranty of any kind — either expressed or implied — including, without limitation, any implied warranties of merchantability or fitness for a particular purpose. FLARM Technology does not warrant the performance of the device, software, firmware, license key, or data or that the device, software, firmware, license key, or data will meet your requirements or operate error free.
   - 6.2. **Limitation of Liability**. In no event shall FLARM Technology be liable to you or any party related to you for any indirect, incidental, consequential, special, exemplary, or punitive damages (including, without limitation, damages for loss of business profits, business interruption, loss of business information, loss of data or other such pecuniary loss), whether under a theory of contract, warranty, tort (including negligence), products liability, or otherwise, even if FLARM Technology has been advised of the possibility of such damages. In no event will FLARM Technology's total aggregate and cumulative liability to you for any and all claims of any kind arising hereunder exceed the amount of fees actually paid by you for the device, license keys or data giving rise to the claim in the twelve months preceding the claim. The foregoing limitations will apply even if the above stated remedy fails of its essential purpose.
   - 6.3. **Indemnification**. You will, at your own expense, indemnify and hold FLARM Technology, and all officers, directors, and employees thereof, harmless from and against any and all claims, actions, liabilities, losses, damages, judgments, grants, costs, and expenses, including reasonable attorneys' fees (collectively, "Claims"), arising out of any use of a FLARM device, software, firmware, license key, or data by you, any party related to you, or any party acting upon your authorization.
7. **General terms**
   - 7.1. **Governing Law**. This Agreement shall be governed by and construed in accordance with the internal law of Switzerland (to the exclusion of Swiss Private International Law and of international treaties, in particular the Vienna Convention on the International Sale of Goods dated April 11, 1980).
   - 7.2. **Severability**. If any term or provision of this Agreement is declared void or unenforceable in a particular situation, by any judicial or administrative authority, this declaration shall not affect the validity or enforceability of the remaining terms and provisions hereof or the validity or enforceability of the offending term or provision in any other situation. To the extent possible the provision will be interpreted and enforced to the greatest extent legally permissible in order to effectuate the original intent, and if no such interpretation or enforcement is legally permissible, shall be deemed severed from the Agreement.
   - 7.3. **No Waiver**. The failure of either party to enforce any rights granted hereunder or to take action against the other party in the event of any breach hereunder shall not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent actions in the event of future breaches.
   - 7.4. **Amendments**. FLARM Technology reserves the right, in its sole discretion, to amend this Agreement from time to time by posting an updated version of the Agreement on www.flarm.com, provided that disputes arising hereunder will be resolved in accordance with the terms of the Agreement in effect at the time the dispute arose. We encourage you to review the published Agreement from time to time to make yourself aware of changes. Material changes to these terms will be effective upon the earlier of (i) your first use of the FLARM device, software, firmware, license key, or data with actual knowledge of such change, or (ii) 30 days from publishing the amended Agreement on www.flarm.com. If there is a conflict between this Agreement and the most current version of this Agreement, posted at www.flarm.com, the most current version will prevail. Your use of the FLARM device, software, firmware, license key, or data after the amended Agreement becomes effective constitutes your acceptance of the amended Agreement. If you do not accept amendments made to this Agreement, then it is your responsibility to stop using the FLARM device, software, firmware, license key, and data.

Governing Language. Any translation of this Agreement is done for local requirements and in the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern