Project code: 1-10-DEV-1801 Classification: Not classified

**Number of pages: 36**

# FlashHawk software interface

**VERSIONS**

| IND. | DATE | AUTHOR | PURPOSE |
|---|---|---|---|
| 01 | 17-aug-2021 | GAU | Initial version |
| 02 | 11-oct-2021 | GAU | Switch to English<br>Additional explanations |

# SOMMAIRE

# 1. INTRODUCTION

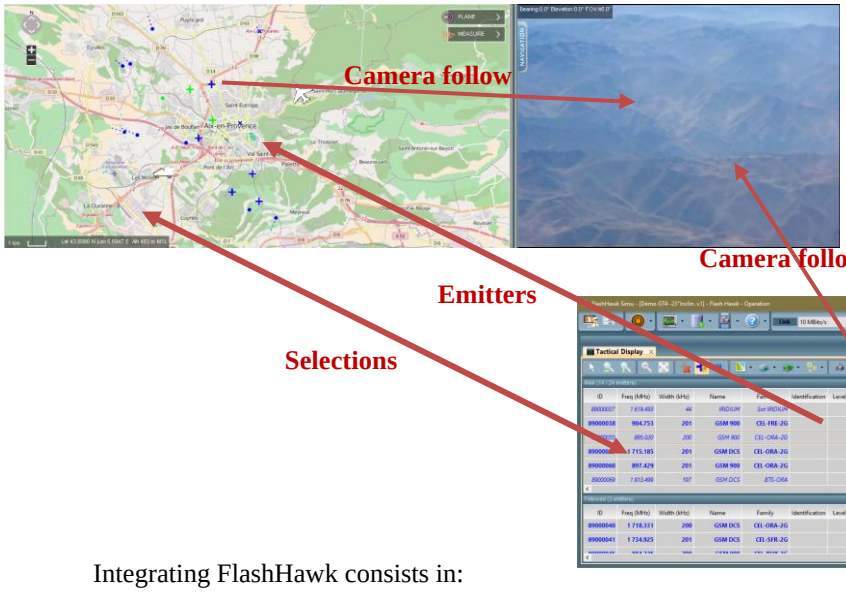
This document describes the software interface of the FlashHawk software.

This interface can be used to integrate FlashHawk with another application managing a multi-sensor-filled map and the optronic camera.

# 2. TERMINOLOGY

| | |
|---|---|
| Client | Refers to the external application integrating FlashHawk. |
| UTF-8 | Variable-width character encoding of the “Universal Coded Character Set”. |

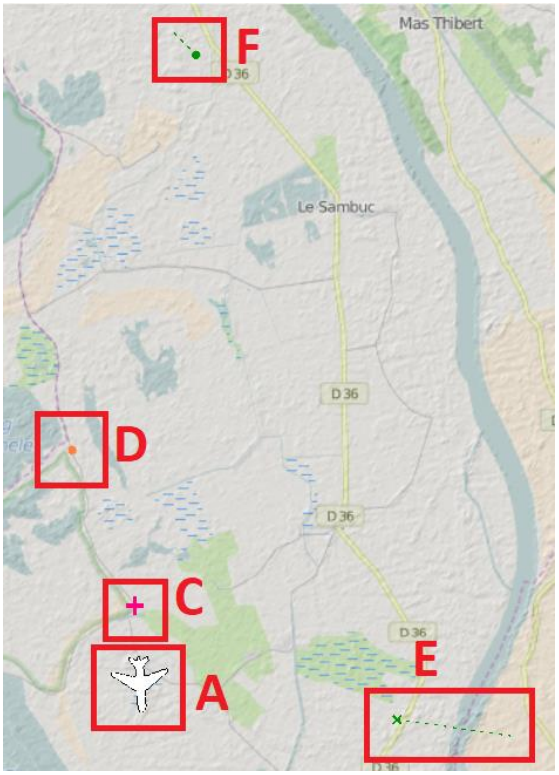
## 3. USER INTERFACE



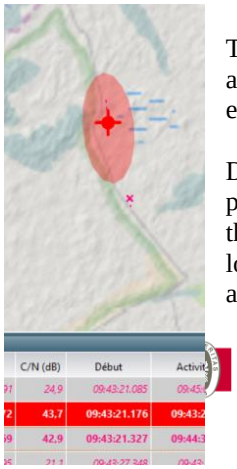

Integrating FlashHawk consists in:

- FlashHawk application running without map nor camera views.
- FlashHawk showing the emitter table and any other FlashHawk-specific monitoring or configuration.
- FlashHawk application occupying half of the screen, at least.
- External application showing map and camera.
- External application showing FlashHawk emitters on the map.
- External application may show FlashHawk emitters on a list, but with limited information.
- External application and FlashHawk fully integrated thanks to selection synchronization:
    - When the user selects a FlashHawk emitter on the external-application map (or on its table/list), it is automatically selected and shown on the FlashHawk table containing all the details.
    - When the user selects an emitter on the FlashHawk table, it is automatically selected on the external-application map.
    - When the user selects to follow an emitter from FlashHawk table, the external application controls the camera to point on its location.

The external application should display FlashHawk emitters with a specific symbol depending on the emitter states:
C: "**active**" emitter with "**good refining**" localization
D: "**inactive**" emitter with "**good refining**" localization
E: "**active**" emitter with "**horizon**" or "**refining**" localization
F: "**inactive**" emitter with "**horizon**" or "**refining**" localization
FlashHawk switches the emitter to inactive if no telecom burst is received during some time (internal tuning).



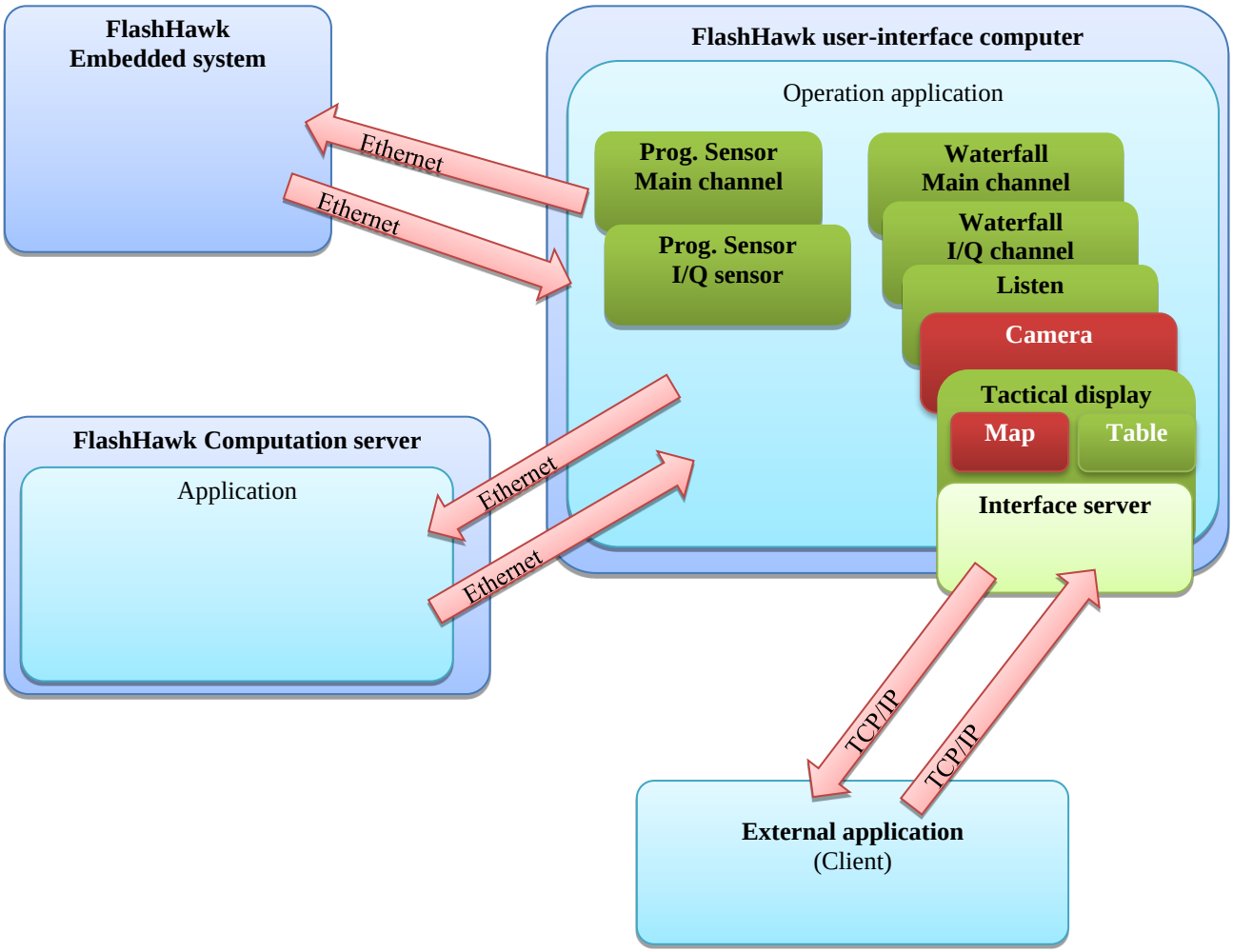

When the localization state is "**horizon**" or "**refining**", the external application should display a line segment showing the direction of the emitter. The segment length is a hint on the distance to the emitter real position. Both direction (azimuth) and length are provided by FlashHawk.




The client application should display the emitter localization accuracy on demand: for example, when the user selects the emitter.

Depending on the emitter's distance, FlashHawk may refine progressively the localization and the accuracy. Continuously, the camera should be automatically pointing the refined localization, and with the zoom (FOV/opening) englobing the accuracy area.

## 4. ARCHITECTURE




Interfacing with FlashHawk is done directly to the user-interface computer through TCP/IP link: using loopback or Ethernet, depending on the client (external) application location (the same computer or not).

## 5. TRANSPORT

The interface server is a TCP/IP server listening on port **9881**.

The client (external) application opens a connection on the interface-server port.

Only one connection at a time is allowed. In case of a new connection, the current connection is terminated and replaced with the new one.

The exchanged data reliability is assured by the TCP protocol.

# 6. APPLICATION PROTOCOL

The data exchanged between the client and the interface server is structured data in compliance to “protocol buffers” (“protobuf”) specifications. Theses specifications define the interface definition and the associated encoding.

The client can implement itself the format or use the free tool and the free library managing the “proto” file. See **8 PROTOCOL BUFFERS PRESENTATION** and **9 PROTOCOL BUFFERS ENCODING**.

The FlashHawk interface is specified by the file « **FlashHawk.proto** ». The last version is presented in this document, but evolutions may occur on this file before the release of the corresponding version of this document. Thus, the file « FlashHawk.proto », associated to the specifications of « protocol buffers » is the reliable reference to integrate with FlashHawk.

The file « FlashHawk.proto » is included in the **FlashHawk_toolkit** installation setup, with the same version number as the application installation setup.

# 7. INTERFACE

## 7.1. FLASHHAWK.PROTO

See 7.3 MESSAGES for further details.

```
//*********************************************************************//**
//* COMPANY : AVANTIX DEFENSE   - Project : FlashHawk
//*********************************************************************
//* Protocol specification to be used by external client to integrate with FlashHawk MMI
//* FlashHawk MMI implements the corresponding server for this purpose
//*********************************************************************

syntax = "proto3";
package Avantix.Elit.Mmi.Lib.Eclair;

// Action to do in the display of emitters
// id is mandatory
// other members optional
message EmitterAction
{
    fixed32 id = 1;         // unique identifier
    bool    selected = 2;   // whether selected or not in the display (table)
    bool    shown = 3;      // whether shown or not in the display (i.e.: not hidden by a filter)
}

// Position relative to earth
message GpsPos
{
    double  lonDeg = 1; // WGS84 longitude in decimal degrees
    double  latDeg = 2; // WGS84 latitude in decimal degrees
    double  altiM = 3;  // MSL altitude
}

// Emitter data for adding or updating
// id is mandatory
// other members are optional when updating
// directionPos and colorArgb are optional even when adding
message EmitterUpdate
{
    // Emitter behavior or display state
    enum State
    {
        INACTIVE = 0;
        ACTIVE = 1;
    }
    // Localization confidence
    enum LocState
    {
        HORIZON = 0;
        REFINING = 1;
        GOOD_REFINING = 2;
        BEST = 3;
    }
    // Identifies the peer on which the user has caused the change
    enum Origin
    {
        INTERNAL = 0;  // FlashHawk
        REMOTE = 1;    // External client application
        BOTH = 2;
    }

    fixed32         id = 1;                     // unique identifier
    GpsPos          position = 2;               // position on earth
    string          classificationUtf8 = 3;     // classification (emitter type)
    string          technicalSummaryUtf8 = 4;   // main characteristics
    double          startTimeUsec = 5;          // start time : number of microseconds since epoch
    double          endTimeUsec = 6;            // end time : number of microseconds since epoch
    repeated GpsPos accuracy = 7;               // ellipse of position/localization accuracy
    State           state = 8;                  // emitter state
    bool            shown = 9;                  // whether shown or not in the display (i.e.: not
hidden by a filter)
```

```
    Origin          hiddenOrigin = 10;          // cause, in case shown is false
    LocState        locState = 11;              // localization quality
    GpsPos          directionPos = 12;          // position to draw a line to, when locState is
HORIZON
    bool            selected = 13;              // whether selected or not in the display
    Origin          selectionOrigin = 14;       // cause, in case selected has changed
    fixed32         colorArgb = 15;             // specific color of the emitter display
}

// Exhaustive list of all existing emitters
// sent periodically for cleanup purpose
// Any ID not present in this list must be removed
// (Ensures that even if a message is lost wherever in client or server, trailing emitters are
removed)
message EmitterIds
{
    repeated fixed32 id = 1;
}

// all known classifications in the database
message Classifications
{
    repeated string classificationsUtf8 = 1;
}

// Confirm being alive
// Ease disconnection detection
message KeepAlive
{
};

// Incoming message from the client to FlashHawk
message CmdIn
{
    KeepAlive               keepAlive = 1;                      // to be sent every 10s (assumed
disconnected after 30s)
    repeated EmitterAction  emitterActions = 2;                 // action on an emitter
(acknowledgement is somehow in emitterUpdates)
    fixed32                 followEmitterWithCameraAck = 50;    // acknowledgement of camera lock on
emitter (or 0 for camera lock stop)
    fixed32                 followEmitterWithCameraReq = 51;    // request to lock camera on an
emitter, 0 to stop camera lock
}

// Outcoming message from FlashHawk to client
message CmdOut
{
    KeepAlive               keepAlive = 1;                      // to be sent every 10s (assumed
disconnected after 30s)
    repeated EmitterUpdate  emitterUpdates = 2;                 // add or update emitters
    repeated fixed32        emitterDeletions = 3;               // remove emitters
    EmitterIds              emitterIds = 4;                     // exhaustive list of emitter IDs
    Classifications         classifications = 5;                // all known classifications in the
database
    fixed32                 followEmitterWithCameraReq = 50;    // request to lock camera on an
emitter, 0 to stop camera lock
    fixed32                 followEmitterWithCameraAck = 51;    // acknowledgement of camera lock on
emitter (or 0 for camera lock stop)
}
```

## 7.2. BEHAVIOR

### 7.2.1. *General*

Once the client is connected, FlashHawk can start writing one or multiple messages "**CmdOut**", and the client can start writing one or multiple messages "**CmdIn**". The 2 peers write and read messages "**CmdOut**" or "**CmdInt**" continuously on the active connection.

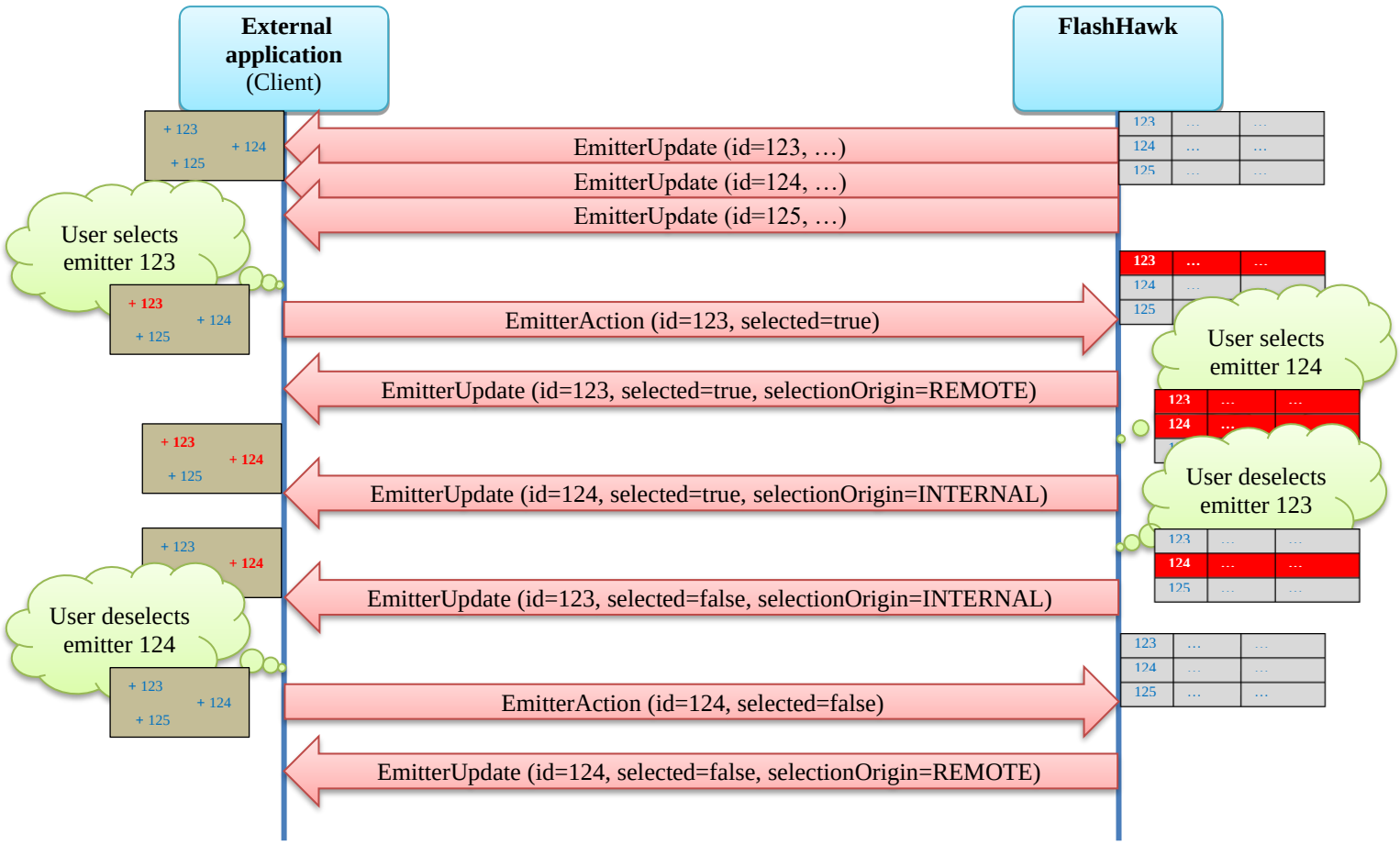
The "proto3" specification defines implicitly each member as optional. Thus, the members of "**CmdOut**" or "**CmdIn**" are the different functional messages exchanged by the 2 peers.

Each peer must send a "**keepAlive**" message every 10 seconds to help the other peer to manage any disconnection matter efficiently.

Any message ending with "Req" implies that the other peer acknowledges it with the corresponding message ending with "Ack". For example, "followEmitterWithCameraReq" (defined in "CmdOut"), sent by FlashHawk, must be acknowledged by the client by sending a "followEmitterWithCameraAck" message (defined in "CmdIn") with the same content.

### 7.2.2. *Selection synchronization*

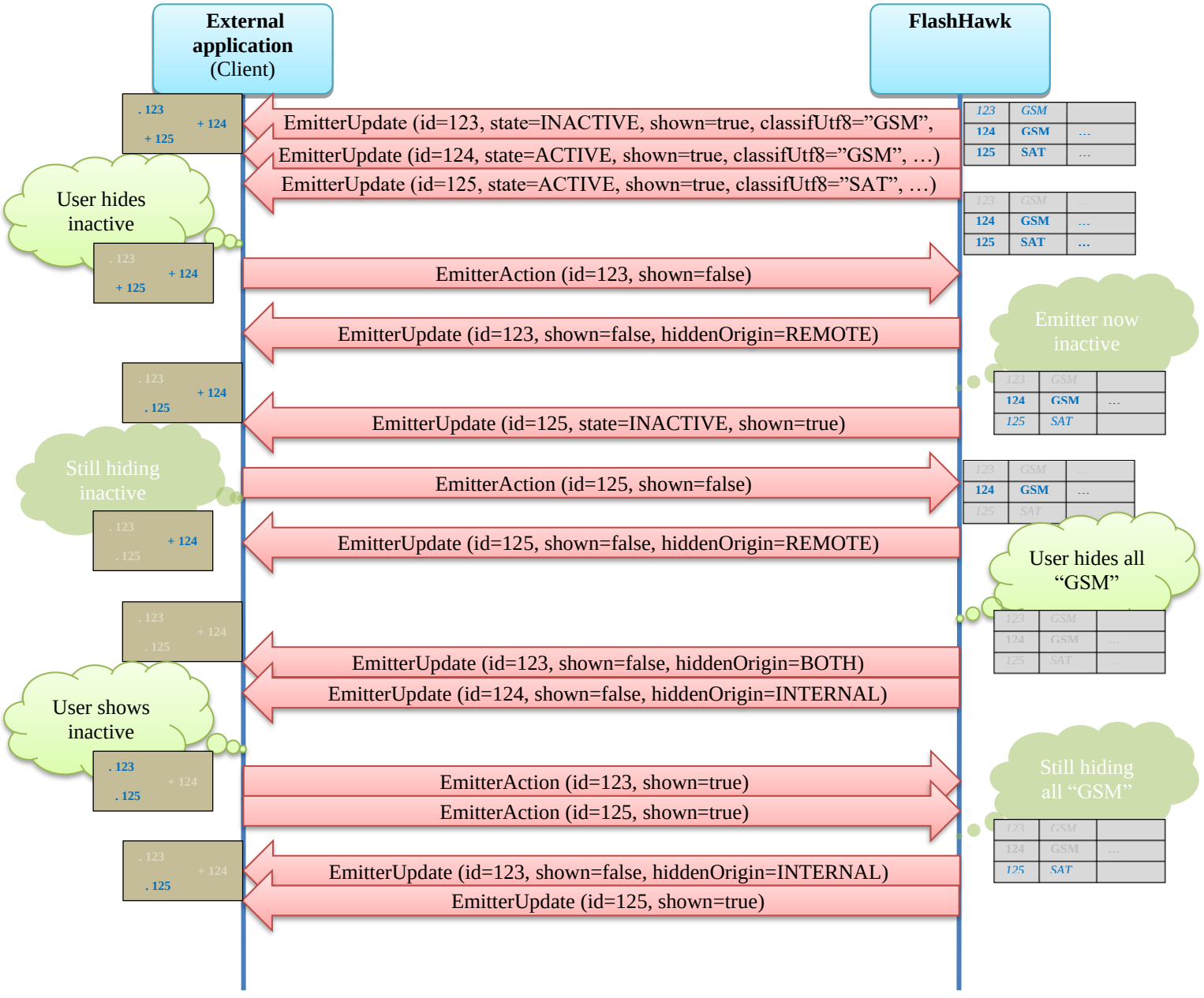
The selection synchronization gives a full-integration experience to the user, thanks to the messages CMDIN : EMITTERACTIONS and CMDOUT : EMITTERUPDATES.



### 7.2.3. *Masking/filtering synchronization*

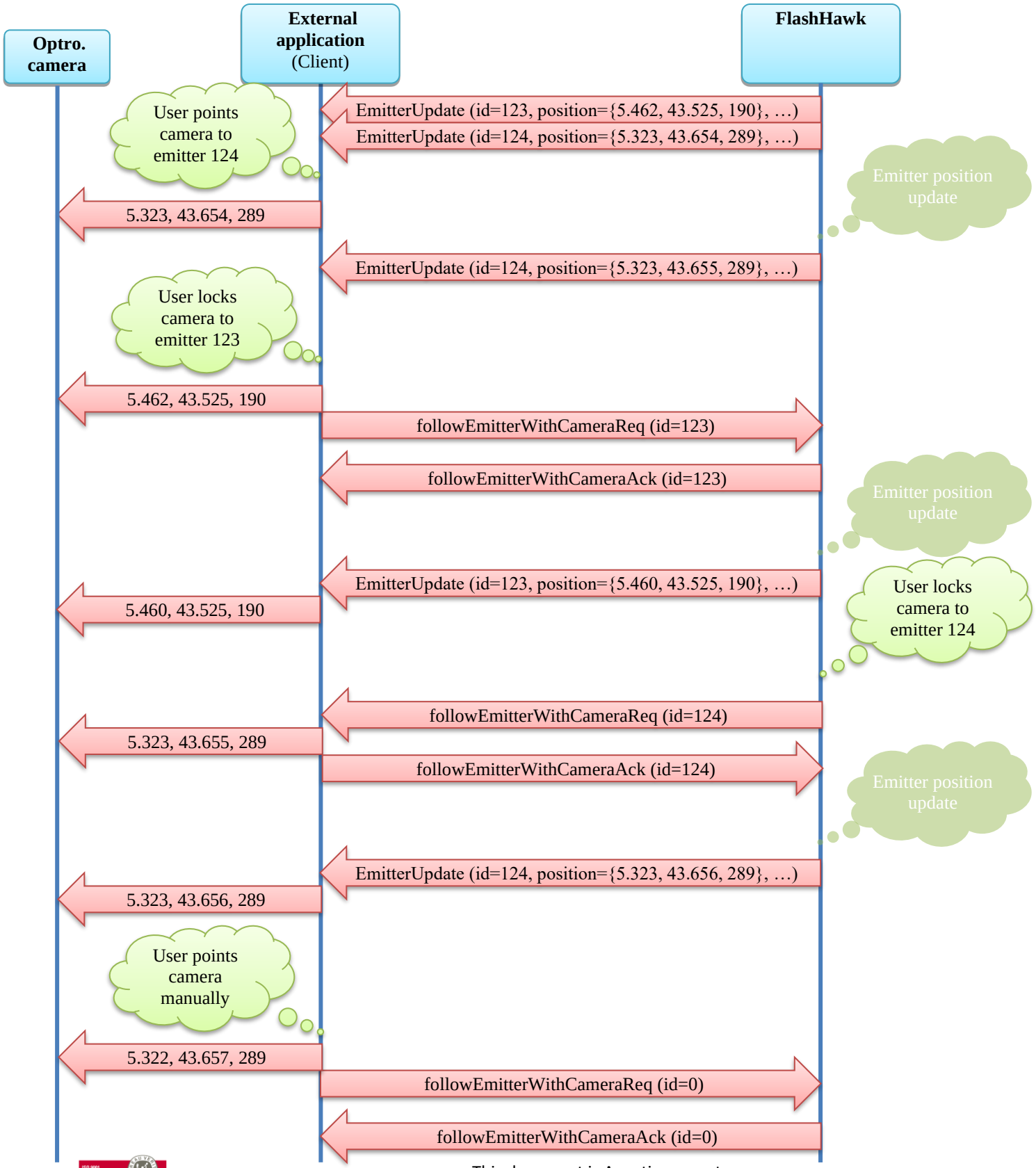
The masking/filtering synchronization gives a full-integration experience to the user, thanks to the messages CMDIN : EMITTERACTIONS and CMDOUT : EMITTERUPDATES.

("classificationUtf8" attribute contracted as "classifUtf8" in this drawing, to gain space)



### 7.2.4. *Camera follow*

The camera-follow synchronization gives a full-integration experience to the user, thanks to the messages CMDOUT : EMITTERUPDATES, CMDOUT : FOLLOWEMITTERWITHCAMERAREQ, CMDIN : FOLLOWEMITTERWITHCAMERAACK, CMDIN : FOLLOWEMITTERWITHCAMERAREQ and CMDOUT : FOLLOWEMITTERWITHCAMERAACK.



## 7.3. MESSAGES

### *7.3.1. CmdIn/CmdOut : keepAlive*

Message to manage any disconnection from the other peer. Sent every 10 seconds by the client application and by FlashHawk.

If none is received after 30 seconds, FlashHawk may close the connection.

The client should implement the same behavior.

### *7.3.2. CmdOut : emitterUpdates*

Message sent by FlashHawk as soon as an emitter is created. The unique identifier "**id**" will be used as reference for any change or action related to this emitter. The client must memorize the emitter and its attributes in its internal memory.

FlashHawk sends this message in case of any emitter attribute change, too. Only the modified attributes are mandatory in the message, in addition to the unique identifier "**id**".

The "**shown**" attribute is useful for the client application and FlashHawk, to display the same subset of emitters, according to filters set in the client application or FlashHawk. See 7.2.3 MASKING/FILTERING SYNCHRONIZATION.

If the emitter is hidden ("**shown**" set to false), this masking origin is maintained and provided by FlashHawk. FlashHawk can also switch "**shown**" to true if it is the only origin of this masking and if the user modifies the filtering to display it. Furthermore, the client may use this origin information to know if the emitter must be displayed when tis filtering is changing. Anyhow, the client application must notifies FlashHawk about its filtering in the message 7.3.6 CMDIN : EMITTERACTIONS.

Any emitter with attribute "**shown**" set to true must be displayed by the client application. On the map, it must be located according to the GPS coordinates and altitude (in the attribute "**position**").

The client application should adapt the emitter display on the map according to the attributes "**state**" and "**locState**". Depending on the state, the line segment to display starts from "**position**" and ends at "**directionPos**". See 3 USER INTERFACE.

According to some internal criteria, FlashHawk estimates the maximal localization accuracy error. The area delimiting this accuracy is provided as a polygon: a list of positions in the attribute "**accuracy**". The client should display this area on demand. E.g.: in case the user selects the emitter. See 3 USER INTERFACE.

To ease the use of the overall user interface of the client application and FlashHawk, the selection should be synchronized between the 2 peers in order have identical selections. Only, if the attribute "**selected**" is true, the client application should select this emitter on its display, too. To avoid any issue, the attribute "**selectionOrigin**" is used. For example, if "**selection**" switches to false and "**selectionOrigin**" is "INTERNAL", the client application knows that the user has just deselected the emitter in the FlashHawk application. Once the selection change is applied and notified, the attribute "**selectionOrigin**" is not included in the following updates. See 7.2.2 SELECTION SYNCHRONIZATION.

The attributes "**startTimeUsec**" and "**endTimeUsec**" provide time information to the client application if it needs this information. These times are the number of micro-seconds since EPOCH (January 1st, 1970 00:00 UTC). When the emitter is active, "endTimeUsec" is updated with the last telecom burst time (the current time, approximately).

The attribute "**classificationUtf8**" is an UTF-8-encoded string to provide the emitter type (e.g.: "PMR / APCO", "Satellite phone", "GSM phone", …). This important information might be displayed by the client application, directly or on demand.

The attribute "**technicalSumaryUtf8**" is and UTF-8-encoded string. It contains the main characteristics of the emitter, which the client might display directly or on demand.

The attribute "**colorArgb**" defines the color, the user might has customized in the FlashHawk application according to technical criteria. The attribute uses the classic encoding, by appending each 8-bit component into a 32-bit word: alpha on bits 24 to 31, red on bits 16 to 23, green on bits 8 to 15, blue on bits 0 to 7. The client application could use the color on its display.

### *7.3.3. CmdOut : emitterDeletions*

Message sent by FlashHawk, to notify that one or multiple emitters are deleted. The unique identifier of each emitter is provided to identify it.

The client application must remove these emitters from its display and delete them from its internal memory.

### *7.3.4. CmdOut : emitterIds*

Message sent by FlashHawk periodically to give and exhaustive list of all its emitters.

Any identifier which is in the client internal memory, but not in this list, must be removed from its display and deleted from its internal memory.

This overcomes any issue of data loss in FlashHawk or in the client application.

### *7.3.5. CmdOut : classifications*

Message sent by FlashHawk to list the known emitter classifications in its database. It is sent on client-application connection or in case of database change.

The strings are UTF-8 encoded.

Thanks to this information, the client application may implement a classification filter, which may require all the possible values.

### *7.3.6. CmdIn : emitterActions*

Message sent by the client application to notify a user action on one or multiple FlashHawk emitters.

The unique identifier of the related emitter is provided.

If the user changes the selection of a FlashHawk emitter in the client application, the attribute "**selected**" is provided with true or false.

Next to this message, FlashHawk will send an emitter-update message related to this emitter with the attribute "**selectionOrigin**" set to "REMOTE". See 7.2.2 SELECTION SYNCHRONIZATION.

In the client application, if the user changes a filter related to FlashHawk emitters, the attribute "**shown**" must be provided. It must be set to false if the filters mask it from the display. It must be set to true if the filters don't mask it from the display. Even if FlashHawk has masked the emitter from its display, the client application must set it to true.

Next to this message, FlashHawk will send an emitter-update message related to this emitter with the attribute "**hiddenOrigin**" set to "INTERNAL" or "BOTH". See 7.2.3 MASKING/FILTERING SYNCHRONIZATION.

### *7.3.7. CmdOut : followEmitterWithCameraReq*

Message sent by FlashHawk when the user starts a camera follow, in the FlashHawk application, from a selected emitter. The unique identifier of the related emitter is provided. If the user stops the camera follow from the FlashHawk application, 0 is set instead of the emitter identifier. See 7.2.4 CAMERA FOLLOW.

If no acknowledgement message is received within 2 seconds, FlashHawk could send the request message again.

### *7.3.8. CmdIn : followEmitterWithCameraAck*

Message sent by the client, upon receiving the message followEmitterWithCameraReq.

The unique identifier of the camera-followed emitter is provided. If no (FlashHawk) emitter is followed by the camera, 0 is set instead the emitter identifier. See 7.2.4 CAMERA FOLLOW.

### *7.3.9. CmdIn : followEmitterWithCameraReq*

Message sent by the client application when the user starts a camera follow on an emitter, in the client application. If it is a FlashHawk emitter, its unique identifier is provided. If the user stops the camera follow or if it does not follow a FlashHawk emitter, 0 is set instead of the emitter identifier. See 7.2.4 CAMERA FOLLOW.

The client application may implement to send the request again if no acknowledgement message is received within 2 seconds.

### *7.3.10. CmdOut : followEmitterWithCameraAck*

Message sent by FlashHawk, upon receiving the message followEmitterWithCameraReq.

The unique identifier of the camera-followed emitter is provided. If no (FlashHawk) emitter is followed by the camera, 0 is set instead the emitter identifier. See 7.2.4 CAMERA FOLLOW.

# 8. PROTOCOL BUFFERS PRESENTATION

https://developers.google.com/protocol-buffers/docs/proto3

## 8.1. DEFINING A MESSAGE TYPE

First let's look at a very simple example. Let's say you want to define a search request message format, where each search request has a query string, the particular page of results you are interested in, and a number of results per page. Here's the `.proto` file you use to define the message type.

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

- The first line of the file specifies that you're using `proto3` syntax: if you don't do this the protocol buffer compiler will assume you are using proto2. This must be the first non-empty, non-comment line of the file.
- The `SearchRequest` message definition specifies three fields (name/value pairs), one for each piece of data that you want to include in this type of message. Each field has a name and a type.

### *8.1.1. Specifying Field Types*

In the above example, all the fields are scalar types: two integers (`page_number` and `result_per_page`) and a string (`query`). However, you can also specify composite types for your fields, including enumerations and other message types.

### *8.1.2. Assigning Field Numbers*

As you can see, each field in the message definition has a **unique number**. These field numbers are used to identify your fields in the message binary format, and should not be changed once your message type is in use. Note that field numbers in the range 1 through 15 take one byte to encode, including the field number and the field's type (you can find out more about this in Protocol Buffer Encoding). Field numbers in the range 16 through 2047 take two bytes. So you should reserve the numbers 1 through 15 for very frequently occurring message elements. Remember to leave some room for frequently occurring elements that might be added in the future.

The smallest field number you can specify is 1, and the largest is $2^{29}$ - 1, or 536,870,911. You also cannot use the numbers 19000 through 19999 (`FieldDescriptor::kFirstReservedNumber` through `FieldDescriptor::kLastReservedNumber`), as they are reserved for the Protocol Buffers implementation - the protocol buffer compiler will complain if you use one of these reserved numbers in your `.proto`. Similarly, you cannot use any previously reserved field numbers.

### *8.1.3. Specifying Field Rules*

Message fields can be one of the following:

- singular: a well-formed message can have zero or one of this field (but not more than one). And this is the default field rule for proto3 syntax.

- `repeated`: this field can be repeated any number of times (including zero) in a well-formed message. The order of the repeated values will be preserved.

In proto3, `repeated` fields of scalar numeric types use `packed` encoding by default.

You can find out more about `packed` encoding in Protocol Buffer Encoding.

#### *8.1.4. Adding More Message Types*

Multiple message types can be defined in a single `.proto` file. This is useful if you are defining multiple related messages – so, for example, if you wanted to define the reply message format that corresponds to your `SearchResponse` message type, you could add it to the same `.proto`:

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}

message SearchResponse {
 ...
}
```

#### *8.1.5. Adding Comments*

To add comments to your `.proto` files, use C/C++-style `//` and `/* ... */` syntax.

```
/* SearchRequest represents a search query, with pagination options to
 * indicate which results to include in the response. */

message SearchRequest {
  string query = 1;
  int32 page_number = 2;  // Which page number do we want?
  int32 result_per_page = 3;  // Number of results to return per page.
}
```

#### *8.1.6. Reserved Fields*

If you update a message type by entirely removing a field, or commenting it out, future users can reuse the field number when making their own updates to the type. This can cause severe issues if they later load old versions of the same `.proto`, including data corruption, privacy bugs, and so on. One way to make sure this doesn't happen is to specify that the field numbers (and/or names, which can also cause issues for JSON serialization) of your deleted fields are `reserved`. The protocol buffer compiler will complain if any future users try to use these field identifiers.

```
message Foo {
  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

Note that you can't mix field names and field numbers in the same `reserved` statement.

#### *8.1.7. What's Generated From Your `.proto`?*

When you run the protocol buffer compiler on a `.proto`, the compiler generates the code in your chosen language you'll need to work with the message types you've described in the file, including getting and setting field values, serializing your messages to an output stream, and parsing your messages from an input stream.

- For **C++**, the compiler generates a `.h` and `.cc` file from each `.proto`, with a class for each message type described in your file.
- For **Java**, the compiler generates a `.java` file with a class for each message type, as well as a special `Builder` classes for creating message class instances.
- For **Kotlin**, in addition to the Java generated code, the compiler generates a `.kt` file for each message type, containing a DSL which can be used to simplify creating message instances.
- **Python** is a little different – the Python compiler generates a module with a static descriptor of each message type in your `.proto`, which is then used with a *metaclass* to create the necessary Python data access class at runtime.
- For **Go**, the compiler generates a `.pb.go` file with a type for each message type in your file.
- For **Ruby**, the compiler generates a `.rb` file with a Ruby module containing your message types.
- For **Objective-C**, the compiler generates a `pbobjc.h` and `pbobjc.m` file from each `.proto`, with a class for each message type described in your file.
- For **C#**, the compiler generates a `.cs` file from each `.proto`, with a class for each message type described in your file.
- For **Dart**, the compiler generates a `.pb.dart` file with a class for each message type in your file.

You can find out more about using the APIs for each language by following the tutorial for your chosen language (proto3 versions coming soon). For even more API details, see the relevant API reference (proto3 versions also coming soon).

### 8.2. SCALAR VALUE TYPES

A scalar message field can have one of the following types – the table shows the type specified in the `.proto` file, and the corresponding type in the automatically generated class:

| .proto Type | Notes | C++ Type | Java/Kotlin Type[1] | Python Type[3] | Go Type | Ruby Type | C# Type | PHP Type | Dart Type |
|---|---|---|---|---|---|---|---|---|---|
| double | | double | double | float | float64 | Float | double | float | double |
| float | | float | float | float | float32 | Float | float | float | double |
| int32 | Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead. | int32 | int | int | int32 | Fixnum or Bignum (as required) | int | integer | int |
| int64 | Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead. | int64 | long | int/long[4] | int64 | Bignum | long | integer/string[6] | Int64 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| uint32 | Uses variable-length encoding. | uint32 | int[2] | int/long[4] | uint32 | Fixnum or Bignum (as required) | uint | integer | int |
| uint64 | Uses variable-length encoding. | uint64 | long[2] | int/long[4] | uint64 | Bignum | ulong | integer/string[6] | Int64 |
| sint32 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s. | int32 | int | int | int32 | Fixnum or Bignum (as required) | int | integer | int |
| sint64 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s. | int64 | long | int/long[4] | int64 | Bignum | long | integer/string[6] | Int64 |
| fixed32 | Always four bytes. More efficient than uint32 if values are often greater than $2^{28}$. | uint32 | int[2] | int/long[4] | uint32 | Fixnum or Bignum (as required) | uint | integer | int |
| fixed64 | Always eight bytes. More efficient than uint64 if values are often greater than $2^{56}$. | uint64 | long[2] | int/long[4] | uint64 | Bignum | ulong | integer/string[6] | Int64 |
| sfixed32 | Always four bytes. | int32 | int | int | int32 | Fixnum or Bignum (as required) | int | integer | int |
| sfixed64 | Always eight bytes. | int64 | long | int/long[4] | int64 | Bignum | long | integer/string[6] | Int64 |
| bool | | bool | boolean | bool | bool | TrueClass/FalseClass | bool | boolean | bool |
| string | A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than $2^{32}$. | string | String | str/unicode[5] | string | String (UTF-8) | string | string | String |
| bytes | May contain any arbitrary sequence of bytes no longer than $2^{32}$. | string | ByteString | str | []byte | String (ASCII-8BIT) | ByteString | string | List |

You can find out more about how these types are encoded when you serialize your message in Protocol Buffer Encoding.

[1] Kotlin uses the corresponding types from Java, even for unsigned types, to ensure compatibility in mixed Java/Kotlin codebases.

[2] In Java, unsigned 32-bit and 64-bit integers are represented using their signed counterparts, with the top bit simply being stored in the sign bit.

[3] In all cases, setting values to a field will perform type checking to make sure it is valid.

[4] 64-bit or unsigned 32-bit integers are always represented as long when decoded, but can be an int if an int is given when setting the field. In all cases, the value must fit in the type represented when set. See [2].

[5] Python strings are represented as unicode on decode but can be str if an ASCII string is given (this is subject to change).

[6] Integer is used on 64-bit machines and string is used on 32-bit machines.

### 8.3. DEFAULT VALUES

When a message is parsed, if the encoded message does not contain a particular singular element, the corresponding field in the parsed object is set to the default value for that field. These defaults are type-specific:

- For strings, the default value is the empty string.
- For bytes, the default value is empty bytes.
- For bools, the default value is false.
- For numeric types, the default value is zero.
- For enums, the default value is the **first defined enum value**, which must be 0.
- For message fields, the field is not set. Its exact value is language-dependent. See the generated code guide for details.

The default value for repeated fields is empty (generally an empty list in the appropriate language).

Note that for scalar message fields, once a message is parsed there's no way of telling whether a field was explicitly set to the default value (for example whether a boolean was set to `false`) or just not set at all: you should bear this in mind when defining your message types. For example, don't have a boolean that switches on some behaviour when set to `false` if you don't want that behaviour to also happen by default. Also note that if a scalar message field **is** set to its default, the value will not be serialized on the wire.

See the generated code guide for your chosen language for more details about how defaults work in generated code.

### 8.4. ENUMERATIONS

When you're defining a message type, you might want one of its fields to only have one of a pre-defined list of values. For example, let's say you want to add a `corpus` field for each `SearchRequest`, where the corpus can be `UNIVERSAL`, `WEB`, `IMAGES`, `LOCAL`, `NEWS`, `PRODUCTS` or `VIDEO`. You can do this very simply by adding an `enum` to your message definition with a constant for each possible value.

In the following example we've added an `enum` called `Corpus` with all the possible values, and a field of type `Corpus`:

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  Corpus corpus = 4;
}
```

As you can see, the `Corpus` enum's first constant maps to zero: every enum definition **must** contain a constant that maps to zero as its first element. This is because:

- There must be a zero value, so that we can use 0 as a numeric default value.
- The zero value needs to be the first element, for compatibility with the proto2 semantics where the first enum value is always the default.

You can define aliases by assigning the same value to different enum constants. To do this you need to set the `allow_alias` option to `true`, otherwise the protocol compiler will generate an error message when aliases are found.

```
message MyMessage1 {
  enum EnumAllowingAlias {
    option allow_alias = true;
    UNKNOWN = 0;
    STARTED = 1;
    RUNNING = 1;
  }
}
message MyMessage2 {
  enum EnumNotAllowingAlias {
    UNKNOWN = 0;
    STARTED = 1;
    // RUNNING = 1;  // Uncommenting this line will cause a compile error inside
Google and a warning message outside.
  }
}
```

Enumerator constants must be in the range of a 32-bit integer. Since `enum` values use varint encoding on the wire, negative values are inefficient and thus not recommended. You can define `enum`s within a message definition, as in the above example, or outside – these `enum`s can be reused in any message definition in your `.proto` file. You can also use an `enum` type declared in one message as the type of a field in a different message, using the syntax `_MessageType_._EnumType_`.

When you run the protocol buffer compiler on a `.proto` that uses an `enum`, the generated code will have a corresponding `enum` for Java, Kotlin, or C++, or a special `EnumDescriptor` class for Python that's used to create a set of symbolic constants with integer values in the runtime-generated class.

**Caution:** the generated code may be subject to language-specific limitations on the number of enumerators (low thousands for one language). Please review the limitations for the languages you plan to use.

During deserialization, unrecognized enum values will be preserved in the message, though how this is represented when the message is deserialized is language-dependent. In languages that support open enum types with values outside the range of specified symbols, such as C++ and Go, the unknown enum value is simply stored as its underlying integer representation. In languages with closed enum types such as Java, a case in the enum is used to represent an unrecognized value, and the underlying integer can be accessed with special accessors. In either case, if the message is serialized the unrecognized value will still be serialized with the message.

For more information about how to work with message `enum`s in your applications, see the generated code guide for your chosen language.

#### *8.4.1. Reserved Values*

If you update an enum type by entirely removing an enum entry, or commenting it out, future users can reuse the numeric value when making their own updates to the type. This can cause severe issues if they later load old versions of the same `.proto`, including data corruption, privacy bugs, and so on. One way to make sure this doesn't happen is to specify that the numeric values (and/or names, which can also cause issues for JSON serialization) of your deleted entries are `reserved`. The protocol buffer compiler will complain if any future users try to use these identifiers. You can specify that your reserved numeric value range goes up to the maximum possible value using the `max` keyword.

```
enum Foo {
  reserved 2, 15, 9 to 11, 40 to max;
  reserved "FOO", "BAR";
}
```

Note that you can't mix field names and numeric values in the same `reserved` statement.

### 8.5. USING OTHER MESSAGE TYPES

You can use other message types as field types. For example, let's say you wanted to include `Result` messages in each `SearchResponse` message – to do this, you can define a `Result` message type in the same `.proto` and then specify a field of type `Result` in `SearchResponse`:

```
message SearchResponse {
  repeated Result results = 1;
}

message Result {
  string url = 1;
  string title = 2;
  repeated string snippets = 3;
}
```

#### *8.5.1. Importing Definitions*

**Note that this feature is not available in Java.**

In the above example, the `Result` message type is defined in the same file as `SearchResponse` – what if the message type you want to use as a field type is already defined in another `.proto` file?

You can use definitions from other `.proto` files by *importing* them. To import another `.proto`'s definitions, you add an import statement to the top of your file:

```
import "myproject/other_protos.proto";
```

By default you can only use definitions from directly imported `.proto` files. However, sometimes you may need to move a `.proto` file to a new location. Instead of moving the `.proto` file directly and updating all the call sites in a single change, now you can put a dummy `.proto` file in the old location to forward all the imports to the new location using the `import public` notion. `import public` dependencies can be transitively relied upon by anyone importing the proto containing the `import public` statement. For example:

```
// new.proto
// All definitions are moved here
// old.proto
// This is the proto that all clients are importing.
import public "new.proto";
import "other.proto";
// client.proto
import "old.proto";
// You use definitions from old.proto and new.proto, but not other.proto
```

The protocol compiler searches for imported files in a set of directories specified on the protocol compiler command line using the `-I`/`--proto_path` flag. If no flag was given, it looks in the directory in which the compiler was invoked. In general you should set the `--proto_path` flag to the root of your project and use fully qualified names for all imports.

#### *8.5.2. Using proto2 Message Types*

It's possible to import proto2 message types and use them in your proto3 messages, and vice versa. However, proto2 enums cannot be used directly in proto3 syntax (it's okay if an imported proto2 message uses them).

### 8.6. NESTED TYPES

You can define and use message types inside other message types, as in the following example – here the `Result` message is defined inside the `SearchResponse` message:

```
message SearchResponse {
  message Result {
    string url = 1;
    string title = 2;
    repeated string snippets = 3;
  }
  repeated Result results = 1;
}
```

If you want to reuse this message type outside its parent message type, you refer to it as `_Parent_._Type_`:

```
message SomeOtherMessage {
  SearchResponse.Result result = 1;
}
```

You can nest messages as deeply as you like:

```
message Outer {                   // Level 0
  message MiddleAA {  // Level 1
    message Inner {   // Level 2
      int64 ival = 1;
      bool  booly = 2;
    }
  }
  message MiddleBB {  // Level 1
    message Inner {   // Level 2
      int32 ival = 1;
      bool  booly = 2;
    }
  }
}
```

### 8.7. UPDATING A MESSAGE TYPE

If an existing message type no longer meets all your needs – for example, you'd like the message format to have an extra field – but you'd still like to use code created with the old format, don't worry! It's very simple to update message types without breaking any of your existing code. Just remember the following rules:

- Don't change the field numbers for any existing fields.
- If you add new fields, any messages serialized by code using your "old" message format can still be parsed by your new generated code. You should keep in mind the default values for these elements so that new code can properly interact with messages generated by old code. Similarly, messages created by your new code can be parsed by your old code: old binaries simply ignore the new field when parsing. See the Unknown Fields section for details.
- Fields can be removed, as long as the field number is not used again in your updated message type. You may want to rename the field instead, perhaps adding the prefix "OBSOLETE_", or make the field number reserved, so that future users of your `.proto` can't accidentally reuse the number.
- `int32`, `uint32`, `int64`, `uint64`, and `bool` are all compatible – this means you can change a field from one of these types to another without breaking forwards- or backwards-compatibility. If a number is parsed from the wire which doesn't fit in the corresponding type, you will get the same effect as if you had cast the number to that type in C++ (e.g. if a 64-bit number is read as an int32, it will be truncated to 32 bits).
- `sint32` and `sint64` are compatible with each other but are *not* compatible with the other integer types.
- `string` and `bytes` are compatible as long as the bytes are valid UTF-8.
- Embedded messages are compatible with `bytes` if the bytes contain an encoded version of the message.
- `fixed32` is compatible with `sfixed32`, and `fixed64` with `sfixed64`.
- For `string`, `bytes`, and message fields, `optional` is compatible with `repeated`. Given serialized data of a repeated field as input, clients that expect this field to be `optional` will take the last input value if it's a primitive type field or merge all input elements if it's a message type field. Note that this is **not** generally safe for numeric types, including bools and enums. Repeated fields of numeric types can be serialized in the packed format, which will not be parsed correctly when an `optional` field is expected.
- `enum` is compatible with `int32`, `uint32`, `int64`, and `uint64` in terms of wire format (note that values will be truncated if they don't fit). However be aware that client code may treat them differently when the message is deserialized: for example, unrecognized proto3 `enum` types will be preserved in the message, but how this is represented when the message is deserialized is language-dependent. Int fields always just preserve their value.

- Changing a single value into a member of a **new** `oneof` is safe and binary compatible. Moving multiple fields into a new `oneof` may be safe if you are sure that no code sets more than one at a time. Moving any fields into an existing `oneof` is not safe.

### 8.8. UNKNOWN FIELDS

Unknown fields are well-formed protocol buffer serialized data representing fields that the parser does not recognize. For example, when an old binary parses data sent by a new binary with new fields, those new fields become unknown fields in the old binary.

Originally, proto3 messages always discarded unknown fields during parsing, but in version 3.5 we reintroduced the preservation of unknown fields to match the proto2 behavior. In versions 3.5 and later, unknown fields are retained during parsing and included in the serialized output.

### 8.9. ANY

The `Any` message type lets you use messages as embedded types without having their .proto definition. An `Any` contains an arbitrary serialized message as `bytes`, along with a URL that acts as a globally unique identifier for and resolves to that message's type. To use the `Any` type, you need to import `google/protobuf/any.proto`.

```
import "google/protobuf/any.proto";

message ErrorStatus {
  string message = 1;
  repeated google.protobuf.Any details = 2;
}
```

The default type URL for a given message type is
`type.googleapis.com/_packagename_._messagename_`.

Different language implementations will support runtime library helpers to pack and unpack Any values in a typesafe manner – for example, in Java, the Any type will have special `pack()` and `unpack()` accessors, while in C++ there are `PackFrom()` and `UnpackTo()` methods:

```
// Storing an arbitrary message type in Any.
NetworkErrorDetails details = ...;
ErrorStatus status;
status.add_details()->PackFrom(details);

// Reading an arbitrary message from Any.
ErrorStatus status = ...;
for (const Any& detail : status.details()) {
  if (detail.Is<NetworkErrorDetails>()) {
    NetworkErrorDetails network_error;
    detail.UnpackTo(&network_error);
    ... processing network_error ...
  }
}
```

**Currently the runtime libraries for working with Any types are under development.**

If you are already familiar with proto2 syntax, the `Any` can hold arbitrary proto3 messages, similar to proto2 messages which can allow extensions.

## 8.10. ONEOF

If you have a message with many fields and where at most one field will be set at the same time, you can enforce this behavior and save memory by using the oneof feature.

Oneof fields are like regular fields except all the fields in a oneof share memory, and at most one field can be set at the same time. Setting any member of the oneof automatically clears all the other members. You can check which value in a oneof is set (if any) using a special `case()` or `WhichOneof()` method, depending on your chosen language.

### *8.10.1. Using Oneof*

To define a oneof in your `.proto` you use the `oneof` keyword followed by your oneof name, in this case `test_oneof`:

```
message SampleMessage {
  oneof test_oneof {
    string name = 4;
    SubMessage sub_message = 9;
  }
}
```

You then add your oneof fields to the oneof definition. You can add fields of any type, except `map` fields and `repeated` fields.

In your generated code, oneof fields have the same getters and setters as regular fields. You also get a special method for checking which value (if any) in the oneof is set. You can find out more about the oneof API for your chosen language in the relevant API reference.

### *8.10.2. Oneof Features*

- Setting a oneof field will automatically clear all other members of the oneof. So if you set several oneof fields, only the *last* field you set will still have a value.

- 
```
SampleMessage message;
message.set_name("name");
CHECK(message.has_name());
message.mutable_sub_message();   // Will clear name field.
CHECK(!message.has_name());
```

- If the parser encounters multiple members of the same oneof on the wire, only the last member seen is used in the parsed message.

- A oneof cannot be `repeated`.

- Reflection APIs work for oneof fields.

- If you set a oneof field to the default value (such as setting an int32 oneof field to 0), the "case" of that oneof field will be set, and the value will be serialized on the wire.

- If you're using C++, make sure your code doesn't cause memory crashes. The following sample code will crash because `sub_message` was already deleted by calling the `set_name()` method.

```
•  SampleMessage message;
SubMessage* sub_message = message.mutable_sub_message();
message.set_name("name");      // Will delete sub_message
sub_message->set_...            // Crashes here
```

- Again in C++, if you `Swap()` two messages with oneofs, each message will end up with the other's oneof case: in the example below, `msg1` will have a `sub_message` and `msg2` will have a `name`.

```
• SampleMessage msg1;
  msg1.set_name("name");
  SampleMessage msg2;
  msg2.mutable_sub_message();
  msg1.swap(&msg2);
  CHECK(msg1.has_sub_message());
  CHECK(msg2.has_name());
```

#### *8.10.3. Backwards-compatibility issues*

Be careful when adding or removing oneof fields. If checking the value of a oneof returns `None/NOT_SET`, it could mean that the oneof has not been set or it has been set to a field in a different version of the oneof. There is no way to tell the difference, since there's no way to know if an unknown field on the wire is a member of the oneof.

##### *8.10.3.1. Tag Reuse Issues*

- **Move fields into or out of a oneof**: You may lose some of your information (some fields will be cleared) after the message is serialized and parsed. However, you can safely move a single field into a **new** oneof and may be able to move multiple fields if it is known that only one is ever set.
- **Delete a oneof field and add it back**: This may clear your currently set oneof field after the message is serialized and parsed.
- **Split or merge oneof**: This has similar issues to moving regular fields.

### 8.11. MAPS

If you want to create an associative map as part of your data definition, protocol buffers provides a handy shortcut syntax:

```
map<key_type, value_type> map_field = N;
```

...where the `key_type` can be any integral or string type (so, any scalar type except for floating point types and `bytes`). Note that enum is not a valid `key_type`. The `value_type` can be any type except another map.

So, for example, if you wanted to create a map of projects where each `Project` message is associated with a string key, you could define it like this:

```
map<string, Project> projects = 3;
```

- Map fields cannot be `repeated`.
- Wire format ordering and map iteration ordering of map values is undefined, so you cannot rely on your map items being in a particular order.
- When generating text format for a `.proto`, maps are sorted by key. Numeric keys are sorted numerically.
- When parsing from the wire or when merging, if there are duplicate map keys the last key seen is used. When parsing a map from text format, parsing may fail if there are duplicate keys.
- If you provide a key but no value for a map field, the behavior when the field is serialized is language-dependent. In C++, Java, Kotlin, and Python the default value for the type is serialized, while in other languages nothing is serialized.

The generated map API is currently available for all proto3 supported languages. You can find out more about the map API for your chosen language in the relevant API reference.

#### *8.11.1. Backwards compatibility*

The map syntax is equivalent to the following on the wire, so protocol buffers implementations that do not support maps can still handle your data:

```
message MapFieldEntry {
  key_type key = 1;
  value_type value = 2;
}

repeated MapFieldEntry map_field = N;
```

Any protocol buffers implementation that supports maps must both produce and accept data that can be accepted by the above definition.

### 8.12. PACKAGES

You can add an optional `package` specifier to a `.proto` file to prevent name clashes between protocol message types.

```
package foo.bar;
message Open { ... }
```

You can then use the package specifier when defining fields of your message type:

```
message Foo {
  ...
  foo.bar.Open open = 1;
  ...
}
```

The way a package specifier affects the generated code depends on your chosen language:

- In **C++** the generated classes are wrapped inside a C++ namespace. For example, `Open` would be in the namespace `foo::bar`.
- In **Java** and **Kotlin**, the package is used as the Java package, unless you explicitly provide an `option java_package` in your `.proto` file.
- In **Python**, the package directive is ignored, since Python modules are organized according to their location in the file system.

- In **Go**, the package is used as the Go package name, unless you explicitly provide an `option go_package` in your `.proto` file.
- In **Ruby**, the generated classes are wrapped inside nested Ruby namespaces, converted to the required Ruby capitalization style (first letter capitalized; if the first character is not a letter, `PB_` is prepended). For example, `Open` would be in the namespace `Foo::Bar`.
- In **C#** the package is used as the namespace after converting to PascalCase, unless you explicitly provide an `option csharp_namespace` in your `.proto` file. For example, `Open` would be in the namespace `Foo.Bar`.

#### *8.12.1. Packages and Name Resolution*

Type name resolution in the protocol buffer language works like C++: first the innermost scope is searched, then the next-innermost, and so on, with each package considered to be "inner" to its parent package. A leading '.' (for example, `.foo.bar.Baz`) means to start from the outermost scope instead.

The protocol buffer compiler resolves all type names by parsing the imported `.proto` files. The code generator for each language knows how to refer to each type in that language, even if it has different scoping rules.

# 9. PROTOCOL BUFFERS ENCODING

https://developers.google.com/protocol-buffers/docs/encoding

This document describes the binary wire format for protocol buffer messages. You don't need to understand this to use protocol buffers in your applications, but it can be very useful to know how different protocol buffer formats affect the size of your encoded messages.

## 9.1. A SIMPLE MESSAGE

Let's say you have the following very simple message definition:

```
message Test1 {
  optional int32 a = 1;
}
```

In an application, you create a `Test1` message and set `a` to 150. You then serialize the message to an output stream. If you were able to examine the encoded message, you'd see three bytes:

```
08 96 01
```

So far, so small and numeric – but what does it mean? Read on...

## 9.2. BASE 128 VARINTS

To understand your simple protocol buffer encoding, you first need to understand *varints*. Varints are a method of serializing integers using one or more bytes. Smaller numbers take a smaller number of bytes.

Each byte in a varint, except the last byte, has the *most significant bit* (msb) set – this indicates that there are further bytes to come. The lower 7 bits of each byte are used to store the two's complement representation of the number in groups of 7 bits, **least significant group first**.

So, for example, here is the number 1 – it's a single byte, so the msb is not set:

```
0000 0001
```

And here is 300 – this is a bit more complicated:

```
1010 1100 0000 0010
```

How do you figure out that this is 300? First you drop the msb from each byte, as this is just there to tell us whether we've reached the end of the number (as you can see, it's set in the first byte as there is more than one byte in the varint):

```
 1010 1100 0000 0010
→ 010 1100  000 0010
```

You reverse the two groups of 7 bits because, as you remember, varints store numbers with the least significant group first. Then you concatenate them to get your final value:

```
000 0010  010 1100
→  000 0010 ++ 010 1100
→  100101100
→  256 + 32 + 8 + 4 = 300
```

### 9.3. MESSAGE STRUCTURE

As you know, a protocol buffer message is a series of key-value pairs. The binary version of a message just uses the field's number as the key – the name and declared type for each field can only be determined on the decoding end by referencing the message type's definition (i.e. the `.proto` file).

When a message is encoded, the keys and values are concatenated into a byte stream. When the message is being decoded, the parser needs to be able to skip fields that it doesn't recognize. This way, new fields can be added to a message without breaking old programs that do not know about them. To this end, the "key" for each pair in a wire-format message is actually two values – the field number from your `.proto` file, plus a *wire type* that provides just enough information to find the length of the following value. In most language implementations this key is referred to as a tag.

The available wire types are as follows:

| Type | Meaning | Used For |
|---|---|---|
| 0 | Varint | int32, int64, uint32, uint64, sint32, sint64, bool, enum |
| 1 | 64-bit | fixed64, sfixed64, double |
| 2 | Length-delimited | string, bytes, embedded messages, packed repeated fields |
| 3 | Start group | groups (deprecated) |
| 4 | End group | groups (deprecated) |
| 5 | 32-bit | fixed32, sfixed32, float |

Each key in the streamed message is a varint with the value `(field_number << 3) | wire_type` – in other words, the last three bits of the number store the wire type.

Now let's look at our simple example again. You now know that the first number in the stream is always a varint key, and here it's 08, or (dropping the msb):

```
000 1000
```

You take the last three bits to get the wire type (0) and then right-shift by three to get the field number (1). So you now know that the field number is 1 and the following value is a varint. Using your varint-decoding knowledge from the previous section, you can see that the next two bytes store the value 150.

```
96 01 = 1001 0110  0000 0001
       → 000 0001  ++  001 0110 (drop the msb and reverse the groups of 7 bits)
       → 10010110
       → 128 + 16 + 4 + 2 = 150
```

### 9.4. MORE VALUE TYPES

#### *9.4.1. Signed Integers*

As you saw in the previous section, all the protocol buffer types associated with wire type 0 are encoded as varints. However, there is an important difference between the signed int types (`sint32` and `sint64`) and the "standard" int types (`int32` and `int64`) when it comes to encoding negative numbers. If you use `int32` or `int64` as the type for a negative number, the resulting varint is *always ten bytes long* – it is, effectively, treated like a very large unsigned integer. If you use one of the signed types, the resulting varint uses ZigZag encoding, which is much more efficient.

ZigZag encoding maps signed integers to unsigned integers so that numbers with a small *absolute value* (for instance, -1) have a small varint encoded value too. It does this in a way that "zig-zags" back and forth through the positive and negative integers, so that -1 is encoded as 1, 1 is encoded as 2, -2 is encoded as 3, and so on, as you can see in the following table:

| Signed Original | Encoded As |
|---|---|
| 0 | 0 |
| -1 | 1 |
| 1 | 2 |
| -2 | 3 |
| 2147483647 | 4294967294 |
| -2147483648 | 4294967295 |

In other words, each value `n` is encoded using

```
(n << 1) ^ (n >> 31)
```

for `sint32`s, or

```
(n << 1) ^ (n >> 63)
```

for the 64-bit version.

Note that the second shift – the `(n >> 31)` part – is an arithmetic shift. So, in other words, the result of the shift is either a number that is all zero bits (if `n` is positive) or all one bits (if `n` is negative).

When the `sint32` or `sint64` is parsed, its value is decoded back to the original, signed version.

#### *9.4.2. Non-varint Numbers*

Non-varint numeric types are simple – `double` and `fixed64` have wire type 1, which tells the parser to expect a fixed 64-bit lump of data; similarly `float` and `fixed32` have wire type 5, which tells it to expect 32 bits. In both cases the values are stored in little-endian byte order.

#### *9.4.3. Strings*

A wire type of 2 (length-delimited) means that the value is a varint encoded length followed by the specified number of bytes of data.

```
message Test2 {
  optional string b = 2;
}
```

Setting the value of b to "testing" gives you:

```
12 07 [74 65 73 74 69 6e 67]
```

The bytes in [brackets] are the UTF8 of "testing". The key here is 0x12. It's parsed:

```
0x12
→ 0001 0010   (binary representation)
→ 00010 010   (regroup bits)
→ field_number = 2, wire_type = 2
```

The length varint in the value is 7 and lo and behold, we find seven bytes following it – our string.

### 9.5. EMBEDDED MESSAGES

Here's a message definition with an embedded message of our example type, Test1:

```
message Test3 {
  optional Test1 c = 3;
}
```

And here's the encoded version, again with the Test1's `a` field set to 150:

```
 1a 03 08 96 01
```

As you can see, the last three bytes are exactly the same as our first example (`08 96 01`), and they're preceded by the number 3 – embedded messages are treated in exactly the same way as strings (wire type = 2).

### 9.6. OPTIONAL AND REPEATED ELEMENTS

If a proto2 message definition has `repeated` elements (without the `[packed=true]` option), the encoded message has zero or more key-value pairs with the same field number. These repeated values do not have to appear consecutively; they may be interleaved with other fields. The order of the elements with respect to each other is preserved when parsing, though the ordering with respect to other fields is lost. In proto3, repeated fields use packed encoding, which you can read about below.

For any non-repeated fields in proto3, or `optional` fields in proto2, the encoded message may or may not have a key-value pair with that field number.

Normally, an encoded message would never have more than one instance of a non-repeated field. However, parsers are expected to handle the case in which they do. For numeric types and strings, if the same field appears multiple times, the parser accepts the *last* value it sees. For embedded message fields, the parser merges multiple instances of the same field, as if with the `Message::MergeFrom` method – that is, all singular scalar fields in the latter instance replace those in the former, singular embedded messages are merged, and repeated fields are concatenated. The effect of these rules is that parsing the concatenation of two encoded messages produces exactly the same result as if you had parsed the two messages separately and merged the resulting objects. That is, this:

```
MyMessage message;
message.ParseFromString(str1 + str2);
```

is equivalent to this:

```
MyMessage message, message2;
message.ParseFromString(str1);
message2.ParseFromString(str2);
message.MergeFrom(message2);
```

This property is occasionally useful, as it allows you to merge two messages even if you do not know their types.

#### *9.6.1. Packed Repeated Fields*

Version 2.1.0 introduced packed repeated fields, which in proto2 are declared like repeated fields but with the special `[packed=true]` option. In proto3, repeated fields of scalar numeric types are packed by default. These function like repeated fields, but are encoded differently. A packed repeated field containing zero elements does not appear in the encoded message. Otherwise, all of the elements of the field are packed into a single key-value pair with wire type 2 (length-delimited). Each element is encoded the same way it would be normally, except without a key preceding it.

For example, imagine you have the message type:

```
message Test4 {
  repeated int32 d = 4 [packed=true];
}
```

Now let's say you construct a `Test4`, providing the values 3, 270, and 86942 for the repeated field `d`. Then, the encoded form would be:

```
22        // key (field number 4, wire type 2)
06        // payload size (6 bytes)
03        // first element (varint 3)
8E 02     // second element (varint 270)
9E A7 05  // third element (varint 86942)
```

Only repeated fields of primitive numeric types (types which use the varint, 32-bit, or 64-bit wire types) can be declared "packed".

Note that although there's usually no reason to encode more than one key-value pair for a packed repeated field, encoders must be prepared to accept multiple key-value pairs. In this case, the payloads should be concatenated. Each pair must contain a whole number of elements.

Protocol buffer parsers must be able to parse repeated fields that were compiled as `packed` as if they were not packed, and vice versa. This permits adding `[packed=true]` to existing fields in a forward- and backward-compatible way.

### 9.7. FIELD ORDER

Field numbers may be used in any order in a `.proto` file. The order chosen has no effect on how the messages are serialized.

When a message is serialized, there is no guaranteed order for how its known or unknown fields should be written. Serialization order is an implementation detail and the details of any particular implementation may change in the future. Therefore, protocol buffer parsers must be able to parse fields in any order.

#### *9.7.1. Implications*

- Do not assume the byte output of a serialized message is stable. This is especially true for messages with transitive bytes fields representing other serialized protocol buffer messages.
- By default, repeated invocations of serialization methods on the same protocol buffer message instance may not return the same byte output; i.e. the default serialization is not deterministic.
    - Deterministic serialization only guarantees the same byte output for a particular binary. The byte output may change across different versions of the binary.
- The following checks may fail for a protocol buffer message instance `foo`.
    - `foo.SerializeAsString() == foo.SerializeAsString()`
    - `Hash(foo.SerializeAsString()) == Hash(foo.SerializeAsString())`
    - `CRC(foo.SerializeAsString()) == CRC(foo.SerializeAsString())`
    - `FingerPrint(foo.SerializeAsString()) == FingerPrint(foo.SerializeAsString())`
- Here're a few example scenarios where logically equivalent protocol buffer messages `foo` and `bar` may serialize to different byte outputs.
    - `bar` is serialized by an old server that treats some fields as unknown.
    - `bar` is serialized by a server that is implemented in a different programming language and serializes fields in different order.
    - `bar` has a field that serializes in non-deterministic manner.
    - `bar` has a field that stores a serialized byte output of a protocol buffer message which is serialized differently.
    - `bar` is serialized by a new server that serializes fields in different order due to an implementation change.
    - Both `foo` and `bar` are concatenation of individual messages but with different order.

## 10.PROTOCOL BUFFERS EXAMPLE

The source code bellow is an example on how to decode messages from a continuous stream, thanks to generated source code and Protocol-Buffers library:

(The proprietary source code receiving the bytes through TCP/IP must append them continuously at the end of the vector “buffer”, before calling ReadMessage)

```
class MyException
{
public:
    std::wstring    m_cause;
    MyException(const std::wstring& cause) : m_cause(cause) {};
};

bool ReadMessage(std::vector<uint8_t>& buffer, Out& message)
{
    CodedInputStream st0(buffer.data(), buffer.size());
    uint32 t = st0.ReadTag();
    if ((t & ~7) == 0)
    {
        // not enough data for tag
        return false;
    }
    if ((t & 7) != 2)
    {
        // not a structure ?
        throw MyException(L"Invalid data : not a tag");
    }
    int s = 0;
    if (!st0.ReadVarintSizeAsInt(&s))
    {
        // not enough data for size
        return false;
    }
    size_t chunkSize = buffer.size() - st0.BytesUntilLimit() + s;
    if (chunkSize > buffer.size())
    {
        // not enough data for message content
        return false;
    }
    CodedInputStream st(buffer.data(), chunkSize);
    if (!message.ParseFromCodedStream(&st))
    {
        throw MyException(L"Invalid data : failed to parse message");
    }
    buffer.erase(buffer.begin(), buffer.begin() + chunkSize);
    return true;
}
```