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.

A Simple Message

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

```
message Test1 {
  required 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...

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
```

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.

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 tag 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
       → 2 + 4 + 16 + 128 = 150
```

More Value Types

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.

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.

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 {
  required string b = 2;
}
```

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

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

The red bytes are the UTF8 of "testing". The key here is 0x12 → tag = 2, type = 2. The length varint in the value is 7 and lo and behold, we find seven bytes following it – our string.

Embedded Messages

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

```
message Test3 {
  required 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).

Optional And Repeated Elements

If your message definition has `repeated` elements (without the `[packed=true]` option), the encoded message has zero or more key-value pairs with the same tag 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.

If any of your elements are `optional`, the encoded message may or may not have a key-value pair with that tag number.

Normally, an encoded message would never have more than one instance of an `optional` or `required` field. However, parsers are expected to handle the case in which they do. For numeric types and strings, if the same value 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.

Packed Repeated Fields

Version 2.1.0 introduced packed repeated fields, which are declared like repeated fields but with the special `[packed=true]` option. 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 tag 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        // tag (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.

Field Order

While you can use field numbers in any order in a `.proto`, when a message is serialized its known fields should be written sequentially by field number, as in the provided C++, Java, and Python serialization code. This allows parsing code to use optimizations that rely on field numbers being in sequence. However, protocol buffer parsers must be able to parse fields in any order, as not all messages are created by simply serializing an object – for instance, it's sometimes useful to merge two messages by simply concatenating them.

If a message has unknown fields, the current Java and C++ implementations write them in arbitrary order after the sequentially-ordered known fields. The current Python implementation does not track unknown fields.

Last updated 四月 *2, 2012.*