Nanopb: Basic concepts

The things outlined here are the underlying concepts of the nanopb design.

Proto files

All Protocol Buffers implementations use .proto files to describe the message format. The point of these files is to be a portable interface description language.

Compiling .proto files for nanopb

Nanopb uses the Google's protoc compiler to parse the .proto file, and then a python script to generate the C header and source code from it:

```
user@host:~$ protoc -omessage.pb message.proto
user@host:~$ python ../generator/nanopb_generator.py message.pb
Writing to message.h and message.c
user@host:~$
```

Modifying generator behaviour

Using generator options, you can set maximum sizes for fields in order to allocate them statically. The preferred way to do this is to create an .options file with the same name as your .proto file:

```
# Foo.proto
message Foo {
   required string name = 1;
}
```

```
# Foo.options
Foo.name max_size:16
```

For more information on this, see the Proto file options section in the reference manual.

Streams

Nanopb uses streams for accessing the data in encoded format. The stream abstraction is very lightweight, and consists of a structure (*pb_ostream_t* or *pb_istream_t*) which contains a pointer to a callback function.

There are a few generic rules for callback functions:

1) Return false on IO errors. The encoding or decoding process will abort immediately.
2) Use state to store your own data, such as a file descriptor.
3) *bytes_written* and *bytes_left* are updated by pb_write and pb_read.
4) Your callback may be used with substreams. In this case *bytes_left*, *bytes_written* and *max_size* have smaller values than the original stream. Don't use these values to calculate pointers.
5) Always read or write the full requested length of data. For example, POSIX *recv()* needs the *MSG_WAITALL* parameter to accomplish this.

Output streams

```
struct _pb_ostream_t
{
   bool (*callback)(pb_ostream_t *stream, const uint8_t *buf, size_t count);
   void *state;
   size_t max_size;
   size_t bytes_written;
};
```

The *callback* for output stream may be NULL, in which case the stream simply counts the number of bytes written. In this case, *max_size* is ignored.

Otherwise, if *bytes_written* + bytes_to_be_written is larger than *max_size*, pb_write returns false before doing anything else. If you don't want to limit the size of the stream, pass SIZE_MAX.

Example 1:

This is the way to get the size of the message without storing it anywhere:

```
Person myperson = ...;
pb_ostream_t sizestream = {0};
pb_encode(&sizestream, Person_fields, &myperson);
printf("Encoded size is %d\n", sizestream.bytes_written);
```

Example 2:

Writing to stdout:

```
bool callback(pb_ostream_t *stream, const uint8_t *buf, size_t count)
```

```
{
   FILE *file = (FILE*) stream->state;
   return fwrite(buf, 1, count, file) == count;
}

pb_ostream_t stdoutstream = {&callback, stdout, SIZE_MAX, 0};
```

Input streams

For input streams, there is one extra rule:

1) You don't need to know the length of the message in advance. After getting EOF error when reading, set bytes_left to 0 and return false. Pb_decode will detect this and if the EOF was in a proper position, it will return true.

Here is the structure:

```
struct _pb_istream_t
{
   bool (*callback)(pb_istream_t *stream, uint8_t *buf, size_t count);
   void *state;
   size_t bytes_left;
};
```

The *callback* must always be a function pointer. *Bytes_left* is an upper limit on the number of bytes that will be read. You can use SIZE_MAX if your callback handles EOF as described above.

Example:

This function binds an input stream to stdin:

```
bool callback(pb_istream_t *stream, uint8_t *buf, size_t count)
{
   FILE *file = (FILE*)stream->state;
   bool status;

   if (buf == NULL)
   {
       while (count-- && fgetc(file) != EOF);
       return count == 0;
   }

   status = (fread(buf, 1, count, file) == count);

   if (feof(file))
       stream->bytes_left = 0;
```

```
    return status;
}

pb_istream_t stdinstream = {&callback, stdin, SIZE_MAX};
```

Data types

Most Protocol Buffers datatypes have directly corresponding C datatypes, such as int32 is int32_t, float is float and bool is bool. However, the variable-length datatypes are more complex:

1) Strings, bytes and repeated fields of any type map to callback functions by default.
2) If there is a special option *(nanopb).max_size* specified in the .proto file, string maps to null-terminated char array and bytes map to a structure containing a char array and a size field.
3) If there is a special option *(nanopb).max_count* specified on a repeated field, it maps to an array of whatever type is being repeated. Another field will be created for the actual number of entries stored.

field in .proto	autogenerated in .h
required string name = 1;	pb_callback_t name;
required string name = 1 [(nanopb).max_size = 40];	char name[40];
repeated string name = 1 [(nanopb).max_size = 40];	pb_callback_t name;

repeated string name = 1 [(nanopb).max_size = 40, (nanopb).max_count = 5];	| size_t name_count;
	| char name[5][40];

required bytes data = 1 [(nanopb).max_size = 40];	| typedef struct {
	| size_t size;
	| uint8_t bytes[40];
	| } Person_data_t;
	| Person_data_t data;

The maximum lengths are checked in runtime. If string/bytes/array exceeds the allocated length, *pb_decode* will return false.

Note: for the *bytes* datatype, the field length checking may not be exact. The compiler may add some padding to the *pb_bytes_t* structure, and the nanopb runtime doesn't know how much of the structure size is padding. Therefore it uses the whole length of the structure for storing data, which is not very smart but shouldn't cause problems. In practise, this means that if you specify *(nanopb).max_size=5* on a *bytes* field, you may be able to store 6 bytes there. For the *string* field type, the length limit is exact.

Field callbacks

When a field has dynamic length, nanopb cannot statically allocate storage for it. Instead, it allows you to handle the field in whatever way you want, using a callback function.

The pb_callback_t structure contains a function pointer and a *void* pointer called *arg* you can use for passing data to the callback. If the function pointer is NULL, the field will be skipped. A pointer to the *arg* is passed to the function, so that it can modify it and retrieve the value.

The actual behavior of the callback function is different in encoding and decoding modes. In encoding mode, the callback is called once and should write out everything, including field tags. In decoding mode, the callback is called repeatedly for every data item.

Encoding callbacks

```
bool (*encode)(pb_ostream_t *stream, const pb_field_t *field, void * const *arg);
```

When encoding, the callback should write out complete fields, including the wire type and field number tag. It can write as many or as few fields as it likes. For example, if you want to write out an array as *repeated* field, you should do it all in a single call.

Usually you can use pb_encode_tag_for_field to encode the wire type and tag number of the field. However, if you want to encode a repeated field as a packed array, you must call pb_encode_tag instead to specify a wire type of *PB_WT_STRING*.

If the callback is used in a submessage, it will be called multiple times during a single call to pb_encode. In this case, it must produce the same amount of data every time. If the callback is directly in the main message, it is called only once.

This callback writes out a dynamically sized string:

```
bool write_string(pb_ostream_t *stream, const pb_field_t *field, void * const *arg)
{
    char *str = get_string_from_somewhere();
    if (!pb_encode_tag_for_field(stream, field))
        return false;

    return pb_encode_string(stream, (uint8_t*)str, strlen(str));
}
```

Decoding callbacks

```
bool (*decode)(pb_istream_t *stream, const pb_field_t *field, void **arg);
```

When decoding, the callback receives a length-limited substring that reads the contents of a single field. The field tag has already been read. For *string* and *bytes*, the length value has already been parsed, and is available at *stream->bytes_left*.

The callback will be called multiple times for repeated fields. For packed fields, you can either read multiple values until the stream ends, or leave it to pb_decode to call your function over and over until all values have been read.

This callback reads multiple integers and prints them:

```
bool read_ints(pb_istream_t *stream, const pb_field_t *field, void **arg)
{
    while (stream->bytes_left)
    {
        uint64_t value;
        if (!pb_decode_varint(stream, &value))
            return false;
        printf("%lld\n", value);
    }
    return true;
}
```

Field description array

For using the *pb_encode* and *pb_decode* functions, you need an array of pb_field_t constants describing the structure you wish to encode. This description is usually autogenerated from .proto file.

For example this submessage in the Person.proto file:

```
message Person {
   message PhoneNumber {
       required string number = 1 [(nanopb).max_size = 40];
       optional PhoneType type = 2 [default = HOME];
   }
}
```

generates this field description array for the structure *Person_PhoneNumber*:

```
const pb_field_t Person_PhoneNumber_fields[3] = {
   PB_FIELD(  1, STRING  , REQUIRED, STATIC, Person_PhoneNumber, number, number, 0),
   PB_FIELD(  2, ENUM    , OPTIONAL, STATIC, Person_PhoneNumber, type, number, &Person_Phone
   PB_LAST_FIELD
};
```

Oneof

Protocol Buffers supports oneof sections. Here is an example of `oneof` usage:

```
message MsgType1 {
    required int32 value = 1;
}

message MsgType2 {
    required bool value = 1;
}

message MsgType3 {
    required int32 value1 = 1;
    required int32 value2 = 2;
}

message MyMessage {
    required uint32 uid = 1;
    required uint32 pid = 2;
    required uint32 utime = 3;

    oneof payload {
        MsgType1 msg1 = 4;
        MsgType2 msg2 = 5;
        MsgType3 msg3 = 6;
    }
}
```

Nanopb will generate `payload` as a C union and add an additional field `which_payload`:

```
typedef struct _MyMessage {
  uint32_t uid;
  uint32_t pid;
  uint32_t utime;
  pb_size_t which_payload;
  union {
      MsgType1 msg1;
      MsgType2 msg2;
      MsgType3 msg3;
  } payload;
/* @@protoc_insertion_point(struct:MyMessage) */
} MyMessage;
```

`which_payload` indicates which of the `oneof` fields is actually set. The user is expected to set the filed manually using the correct field tag:

```
MyMessage msg = MyMessage_init_zero;
msg.payload.msg2.value = true;
msg.which_payload = MyMessage_msg2_tag;
```

Notice that neither `which_payload` field nor the unused fileds in `payload` will consume any space in the resulting encoded message.

Extension fields

Protocol Buffers supports a concept of extension fields, which are additional fields to a message, but defined outside the actual message. The definition can even be in a completely separate .proto file.

The base message is declared as extensible by keyword *extensions* in the .proto file:

```
message MyMessage {
    .. fields ..
    extensions 100 to 199;
}
```

For each extensible message, *nanopb_generator.py* declares an additional callback field called *extensions*. The field and associated datatype *pb_extension_t* forms a linked list of handlers. When an unknown field is encountered, the decoder calls each handler in turn until either one of them handles the field, or the list is exhausted.

The actual extensions are declared using the *extend* keyword in the .proto, and are in the global namespace:

```
extend MyMessage {
    optional int32 myextension = 100;
}
```

For each extension, *nanopb_generator.py* creates a constant of type *pb_extension_type_t*. To link together the base message and the extension, you have to:

1. Allocate storage for your field, matching the datatype in the .proto. For example, for a *int32* field, you need a *int32_t* variable to store the value.
2. Create a *pb_extension_t* constant, with pointers to your variable and to the generated *pb_extension_type_t*.
3. Set the *message.extensions* pointer to point to the *pb_extension_t*.

An example of this is available in *tests/test_encode_extensions.c* and *tests/test_decode_extensions.c*.

Message framing

Protocol Buffers does not specify a method of framing the messages for transmission. This is something that must be provided by the library user, as there is no one-size-fits-all solution. Typical needs for a framing format are to:

1. Encode the message length.
2. Encode the message type.
3. Perform any synchronization and error checking that may be needed depending on application.

For example UDP packets already fullfill all the requirements, and TCP streams typically only need a way to identify the message length and type. Lower level interfaces such as serial ports may need a more robust frame format, such as HDLC (high-level data link control).

Nanopb provides a few helpers to facilitate implementing framing formats:

1. Functions *pb_encode_delimited* and *pb_decode_delimited* prefix the message data with a varint-encoded length.
2. Union messages and oneofs are supported in order to implement top-level container messages.
3. Message IDs can be specified using the *(nanopb_msgopt).msgid* option and can then be accessed from the header.

Return values and error handling

Most functions in nanopb return bool: *true* means success, *false* means failure. There is also some support for error messages for debugging purposes: the error messages go in *stream->errmsg*.

The error messages help in guessing what is the underlying cause of the error. The most common error conditions are:

1) Running out of memory, i.e. stack overflow.
2) Invalid field descriptors (would usually mean a bug in the generator).
3) IO errors in your own stream callbacks.
4) Errors that happen in your callback functions.
5) Exceeding the max_size or bytes_left of a stream.
6) Exceeding the max_size of a string or array field
7) Invalid protocol buffers binary message.