pbgen

(**p**roto**b**uf **gen**erator)

April 21, 2017

Contents

1 Introduction **2**

2 Defining protobuf representation **3**

2.1 Decoder, encoder 4
2.2 Field index 4
2.3 Default values 4
2.4 Pretty printing 4
2.5 Strings 5
2.6 Arrays 5
2.7 Integers and booleans 5
2.8 Callbacks 6
2.9 "Required" fields 6
2.10 "Rest" 6

3 Invoking `pbgen` **7**

1 Introduction

`pbgen` is a program that parses a C native structure decorated with "special" comments and generates C code necessary to encode and decode protobuf representation of the structure, and a pretty-printer to provide the structure in human-readable form. The generated code is kept small, fast, and does not allocate memory.

Main goal of `pbgen` is to provide a small (in size), fast, secure and easy way to work with protobuf-encoded messages in C embedded programs. It provides features that are absent in other libraries or generators which will be discussed later. The main difference is that instead of generating code from a `.proto` file, `pbgen` parses C code where a programmer defined how a protobuf-encoded message should look like natively, as a C native structure. This provides a huge benefit of a really simple API to work with such a structure in protobuf world — for a structure named X, `pbgen` generates just four (at most) functions that a programmer wants to use:

```
uint8_t *pb_encode_X(const struct X *in,
                     uint8_t *s, uint8_t *e);

const uint8_t *pb_decode_X(struct X *out,
                           const uint8_t *s, const uint8_t *e);

char *pb_seprint_X(const struct X *value,
                   char *s, char *e,
                   int start_indent, int indent);

void pb_reset_X(struct X *out);
```

Programmer doesn't need to look through generated code or make any wrong assumptions, since the native C structure is the only thing that matters. This becomes more apparent in the next section.

`pbgen` consists of three files to be copied into project that needs it:

a) `pb.c` — implementation.

b) `pb.h` — header used by generated code and (ocasionally) by the project itself.

c) `pbgen` — AWK[1] script to parse and generate code, to be incorporated into the project's build system.

[1] `http://en.wikipedia.org/wiki/AWK`

2 Defining protobuf representation

The "special comments" mentioned in the introduction are of the form `//#`. This way `pbgen` can tell when a comment should be parsed.

```
1  #include <stdint.h>
2  #include <stddef.h>
3
4  enum E {          //#
5      NoValue = -1,
6      ValueA,       //# pretty=A
7      ValueB,       //# pretty=B
8  };
9
10 struct X {        //# decoder encoder
11     int i;        //#1 required
12     enum E e;     //#2 default=NoValue print=no
13     uint8_t b[8]; //#3
14     uint8_t c[8]; //#4 num_at=num_c
15     char *s;      //#5 truncate
16
17     char *t;
18     uint64_t u;
19
20     int x;        //#
21
22     int num_c;
23     uint8_t buf[32];  //# buffer=here
24     uint8_t rest[32]; //# rest=here
25 };
26
27 struct Y { //# decoder encoder
28     int num_x;
29     struct X x[4]; //#1 num_at=^ callback=each_x
30     int all_i;
31 };
32
33 static int each_x(struct Y *y, struct X *x) { //
34     if(y->num_x == 0) y->sum_i = 0;
35     y->sum_i += x->i;
36     return PB_CALLBACK_OK;
37 }
38
39 #include "def0.pb.h"
```

On line 4 `enum E` is marked as protobuf-enabled by appending `//#`. This ensures `pbgen` will generate a pretty-printer for this data type. This is needed since the enum is used later inside `struct X`.

2.1 Decoder, encoder

Either (or both) `decoder` or `encoder` need to be specified for each structure one wants to use `pbgen` with. Not specifying `decoder`, for example, will make `pbgen` not generate a decoder. This is useful if the structure is only going to be **sent** over the wire, but never received.

2.2 Field index

`#1 ...#5` specifies protobuf index of a field.

Omitting the number (line 20) makes the field used just for pretty-printing and resetting. This is useful when the value for a field is calculated with a specific code, depending on values of other fields, and the pretty-printed structure needs to have it as well.

2.3 Default values

The "default value" for any field is `0` for numbers, `NULL` for pointers, and all zeroes for arrays. Default value may be overridden with `default=...` (line 12) and has several effects:

a) a field set to its default value is skipped when encoding

b) default values aren't printed by pretty-printers

c) when decoding, fields that aren't set in the protobuf-encoded message will be set to their default values

The generated `pb_reset_X` function may be used to set fields of the structure to their default values without touching any fields that aren't used by `pbgen` , like `t` and `u`. This may be used prior to setting fields to their values and calling `pb_encode_X`, so that only explicitly changed fields are encoded.

2.4 Pretty printing

`pretty=...` (lines 6 and 7) is optional and is used to override the enumeration value or field name when pretty-printing. If it's not set, the original value or name is used.

The `print=no` (line 12) means field is hidden when pretty-printing.

A custom pretty-printing function name may be specified as the value as well. The type of the pretty-printer is:

```
char *(*seprint)(const void *in,
                 char *s, char *e,
                 int left, int indent,
                 const struct pb_field *field);
```

The arguments are:

in	Pointer to the value that needs to be pretty-printed.
s	Pointer to the beginning of the destination buffer.
e	Pointer to past the end of the destination buffer.
left	Left intentation for multi-line values.
indent	Indentation for additional "levels" of a tree-like multi-line values.
field	Should be ignored.

The returned value must be pointer past the last character written to the destination buffer.

The header (`pb.h`) provides a useful function to correctly print formatted output to the destination buffer:

```
char *seprintf(char *s, char *e, char *fmt, ...);
```

`pbgen` is able to generate pretty good pretty-printers by default, so one rarely needs to write their own.

Note: pretty-printing is enabled in compile-time by defining `PB_PRINTERS` macro. All custom pretty-printers should be guarded with `#ifdef PB_PRINTERS`.

2.5 Strings

With `pbgen` , strings are either a `char` array or a pointer.

If a string field has a non-`const` pointer type, there needs to be a "buffer" field (line 23) added to the structure that will hold all strings packed tightly. This helps with saving crucial memory space on embedded systems and most importantly allows an easy way to check if a value is set: it's not set when it's `NULL`.

On the other hand, if a string field **is** `const char *` **and** has `num_at=...` set (required), the string may be stored outside of the structure. When decoding, it will point into the original protobuf message buffer while field specified with `num_at=...` will have its value set to the length of the string (obviously, it would be wrong for `pbgen` to modify the protobuf message while decoding).

When string doesn't fit into a `char` array, or into a separate "buffer" field, the protobuf message parsing will fail with an error unless **truncate** is set for the field for that string. Automatic truncation of strings isn't enabled by default and should only be enabled for information that is not required for the program's correct functioning.

2.6 Arrays

There are two types of arrays in `pbgen` : fixed-size and variable-size. When `num_at=...` points to a field where the number of valid elements is stored (line 14), the array is said to be of variable size **unless** `size=<number>` is set. If an array doesn't have `num_at=...`, the array has a constant size (line 13).

If a value being parsed has a size different from the one specified for a fixed-size array, parsing will fail. This is useful when some kind of a fixed-size hash needs to be decoded or encoded: `pbgen` makes sure the size is alway correct.

`num_at=^` (line 29) is a shortcut to point `pbgen` at the field above.

`pbgen` supports packed integers when decoding.

Byte arrays must be of type `uint8_t`.

2.7 Integers and booleans

The integer types `pbgen` expects one to use are standard ones, including the ones provided by `stdint.h`, like `int`, `uint32_t`, `int16_t`, `long`, `bool`, etc, but **not** `uint8_t` (only for byte arrays). Boolean is just an integer in protobuf encoding scheme, so `pbgen` doesn't differentiate either. If the type name begins with "u" character, it will be assumed `unsigned`. `pbgen` checks if the value to

be stored in a field cannot be represented by the type, in which case parsing fails.

Because of how protobuf encoding scheme was designed, numbers have two different ways of being represented on the wire. By default, in `pbgen` all integers are **not** ZigZag[2] encoded. If you need a field representing `sint32` or `sint64` in the `.proto` file, you have to specify `signed` for the field.

2.8 Callbacks

Each time a new value is decoded for a field, a callback function (line 33) pointed by `callback=...` (line 29) will be called for custom processing.

In general case, callback has type `int (*cb)(Parent*, Child*)`, where the value decoded is a second argument, always as a pointer. The first argument points to the structure that holds the field for which the value is decoded. In case the value is a byte array (or a string), additional third argument is added: `int (*cb)(Parent*, uint8_t*, int)`. It specifies the size in bytes (length of the string).

If the parsing needs to be stopped with an error, callback may return a negative number. To throw away the value parsed, zero should be returned. For anything else, a positive number should be returned. For convenience, `"pb.h"` defines the following:

```
/* callback return codes */
enum {
    PB_CALLBACK_ERROR = -1, /* parsing should fail */
    PB_CALLBACK_THROW_AWAY, /* throw away the value */
    PB_CALLBACK_OK,
};
```

In the example given, `each_x` function (line 33) calculates the sum of all `x->i` numbers and stores it in the parent structure.

2.9 "Required" fields

Specifying `required` (line 11) does exactly what is expected:

a) decoding a structure will fail unless the field is set.

b) the field is **always** encoded, no matter the value.

c) the field is **always** pretty-printed, no matter the value.

2.10 "Rest"

Sometimes there is a need to parse a protobuf message, store it in memory, and later transfer it back (or somewhere else). If this needs to be done in a "future-proof" way so that one can add more fields to the structure on the server, and client is still required to transfer all the fields back, `rest=...` (line 24) may be used. The "rest" of the protobuf message (which is not pointed to by any of the fields of the native structure) will be stored in this buffer. This information will be stored unchanged to the destination buffer when encoding the native structure to protobuf message.

[2] `https://developers.google.com/protocol-buffers/docs/encoding#signed-integers`

3 Invoking pbgen

The two ways of running `pbgen` are:

```
$ awk -f pbgen <a.c >a.pb.h
$ awk -f pbgen -v header=a.pb.h <a.c >a_data.pb.h
```

When no "header" is specified (the first method), all generated functions are defined as `static` and the generated file `"a.pb.h"` is supposed to be included exactly once, in a single C file which uses these functions.

Second method, on the other hand, makes `pbgen` generate two files. One (`"a_data.pb.h"`) containing all the functions and static "data", to be included once in a C file. And the other one (`"a.pb.h"`) containing the prototypes of the generated functions. This method is used when different C files need to work with protobuf representations of the same data structures.