WebRTC size_t Conversion Design Doc

Summary

This discusses a CL (https://webrtc-codereview.appspot.com/23129004/) to convert various types in the WebRTC codebase to size_t, specifically for variables relating to packet or payload sizes, as well as a variety of other connected APIs, members, and variables which represent bit or byte counts in memory.

Proposal

The WebRTC codebase currently uses a wide mixture of types, often using different types in different parts of the codebase for the same concept. Specifically for payload/packet/padding sizes, existing code uses at least uint8_t, uint16_t, uint32_t, unsigned int, int, int64_t, and size_t in different places. This results in implicit and explicit type conversions. Standardizing on a single type in as many places as possible eliminates these conversions. I've chosen to standardize on size_t. The exception to this is where functions must be able to use negative values, e.g. to return error codes; in these cases I've elected to retain the existing int return types rather than e.g. use std::string::npos-style size_t "error constants".

Benefits

Type standardization brings a variety of benefits:

- Existing code can frequently be made slightly shorter and more readable by eliminating casts.
- Eliminating implicit truncations fixes a number of (currently disabled) warnings on MSVC which I'd like to enable across Chromium to help prevent unintentional value truncations in the future. Some of these truncations can represent real bugs where larger values can be truncated; using size_t as much as possible minimizes the likelihood of such truncations. (For example, existing code often stores the result of fread() and fwrite() calls in an int, even though these functions actually return size_t; hopefully reading 2+GB of data at once is rare, but better and more future-proof to not have to worry about it at all.)
- The Chromium style guide explicitly encourages use of size_t "for object and allocation sizes, object counts, array and pointer offsets, vector indices, and so on". Thus using size_t makes the code more style-guide-compliant.
- Using size_t allows authors to more easily reason about code, e.g. by knowing that callers can't pass a negative value, or knowing that a loop can't result in a byte count becoming negative, so such cases don't need to be explicitly handled.
- In the cases where negative values are actually used (e.g. to encode errors), the explicit use of a different, signed type calls attention to the possibility of error, and the use of

size_ts in other APIs makes authors more likely to have to consider how to convert the types (and thus likely to recognize that error-handling is necessary).
- No debate about what types to use is necessary when writing new code that interfaces with existing code: always use size_t.

Dangers

- Because size_t is an unsigned type, underflow causes a wraparound to large positive values. Depending on the algorithm in question, this can cause misbehavior. (This sort of wraparound is one of the reasons that, outside the use of size_t, the style guide normally recommends avoiding the use of unsigned types "just to convey that the value shouldn't be negative".) The danger is greatest in the conversion of existing code, which may have depended on the negative values possible in signed types, as compared to when new code is written; I've attempted to identify and correctly handle all cases where existing code used negative values.
- size_t can increase the size of containing objects, especially in 64-bit builds, though in most cases this size increase is immaterial (because there aren't hundreds of thousands of such objects live in memory).
- The fact that ints are still used in some places means that not all possible value truncation errors are eliminated (though the resulting code should at least be no less safe than before).

Alternates Considered

The obvious alternative is to simply not make this change at all. I think the benefits above outweigh the dangers, though. In particular, I'm keen to enable the MSVC warnings about possible value truncation, as fixing those warnings in Chrome has uncovered a number of cases that were either outright bugs or at least danger spots where the subtlety in the existing code wasn't fully clear.

Another alternative would be to standardize on some other type than size_t. I originally began this work trying to standardize on int, since it seemed like that might be the most common existing type. Unfortunately, using int everywhere seemed inferior to using size_t everywhere in terms of the total number of casts (there were more places that couldn't safely use int than couldn't safely use size_t) as well as in making code more error-prone (it often wasn't clear if code needed to handle negative values or not). Ultimately the int work become increasingly unappealing and I switched to size_t.

Another alternative to the current change would involve using size_t or ssize_t for functions which need to return an error code, which are currently left as int. I think this could be done, and would result in fewer net casts than today, as well as code less likely to have potential value truncations. However, it wasn't necessary to make the initial change here, which is already

enormous, and it would introduce the need to define constants like (size_t)-1 in various places, as well as have calling code check for those constants; code which failed to properly error-check would treat these as very large positive values, making bugs more likely than with the current patch. Such a change seems like it would be more controversial than the existing patch.

Review And Landing

Because the change is enormous (>330 files, thousands of lines touched), it's infeasible to review in a single piece. The obvious alternative would be to land the change as a series of isolated pieces converting different parts of the code, and inserting casts and the like at API boundaries. Unfortunately, this would be not only very time-consuming but more likely to introduce bugs: type changes like this, much like constness changes, tend to be viral, and breaking them into pieces can require a *lot* of casts/conditionals around the boundaries. All of these need to be authored, reviewed, and then later removed again when other pieces land, to be replaced with yet more casts and conditionals. Instead, I've elected to upload and test the patch as a single piece and split it into smaller pieces for review, and intend to land the monolithic patch once all pieces have approval. The pieces are split along directory lines, and with the exception of the rtp_rtcp changes and perhaps the video_coding changes, should be small enough for easy review. (For those other two CLs: sorry…)

Because of the interconnected, viral nature of type changes, I strongly encourage reviewers not to request a revert of some type away from size_t unless absolutely necessary, especially for struct/class members, especially in widely-used, core types. If using size_t is absolutely unacceptable for a particular member -- e.g. because there are many copies of the struct in question in memory and memory use is a critical concern -- the best alternative is likely to make the member private and force consumers to go through accessors which take and return size_t, and which do a runtime check that the provided values can be stored in a smaller underlying type. This sort of thing has its own costs, so I'm not eager to do it unless necessary.

In order to minimize the risk of this CL, after ensuring that all the trybots on the large CL were green, I manually re-reviewed all changes in each smaller CL, making followon fixes and updates where necessary, mirroring those changes back to the larger CL, and leaving comments on anything I felt especially noteworthy.