Leveraging Open Source in Embedded Software Projects

Google's "Protocol Buffers" on a Medical Device

Morgan Kita, Zühlke Engineering

As engineering tools and approaches continue to mature, customers require more and more facets to their product development projects. They need custom software and/or hardware delivered to their specifications on time and within budget, while also expecting appropriate measures regarding safety, security, testability, continuous integration, delivery, and any other number of domain specific aspects. It has rapidly become apparent that the integration of third-party specifications and solutions for any number of these aspects are key in achieving project success; regardless of whether they be middleware proprietary solutions, free/libre, or open source in nature. Device firmware has remained one element of systems that is often treated as tied to customer IP and therefore mainly bespoke or relegated to established middleware solutions. In this case study, the successful integration in a medical device project of the open source communication protocol "Protocol Buffers" from Google as well as "NanoPB", a complementary third-party C-library implementation thereof, is presented. Along with a discussion of the technology space and the integration details, an analysis of the cost and benefits of such an approach will be examined.

Problem Domain

Data serialization is a key aspect of software engineering that has existed since the very first computers began to generate data. The need to efficiently and reproducibly transfer data across machine boundaries applies to any number of hardware scenarios: whether it be in large distributed networks, short term and long-term data storage, or between tightly coupled microprocessors. Over the years a large variety of specifications and data formats have been developed to address both the general problem space as well as individual subset domains. For many of these solutions, there exists a multitude of corresponding middleware and open source implementations.

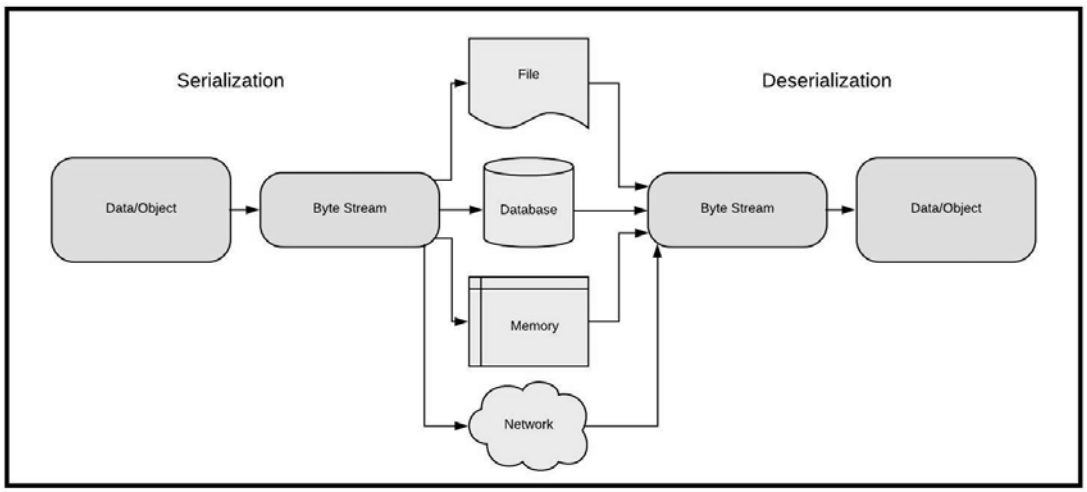

Figure 1: Flow of Data Serialization

Protocol Buffers Characterization

One such generic solution is “Protocol Buffers” [1], an open source portable binary data exchange format from Google. Though an official implementation that covers a variety of popular languages is available under the BSD license from Google, at its core “Protocol Buffers” is a well-defined “wire” exchange format, where the “wire” refers to any potential machine boundary. While developed with Google’s distributed network technologies in mind, “Protocol Buffers” is applicable to any number of domains. The specification is itself language and platform portable and covers all possible scalar data types, allowing the composition and translation of any possible data set. It was developed with a balance between wire size efficiency and runtime performance in mind and focuses on having a comprehensive solution that remains easy to comprehend. To this effect, Google has provided excellent documentation on both the format specification itself as well as its associated open source tooling and language APIs.

“Protocol Buffers” can be integrated as a robust backbone for an “Open Systems Interconnection (OSI)” style stack; however, it does not address various important common features at the upper layers of presentation and application logic. It provides no security encryption features and does not specify any data compression methods beyond the efficient packing of raw scalar data. No API for a “Remote Procedure Call” system is provided, and character encoding is limited to ASCII 7-bit or UTF-8. However, all these features are layers that can be added on top of the core format and are not restricted by its specification. The data encoded by “Protocol Buffers” may be passed through other pipelines that provide these upper layers. However, the restriction that “Protocol Buffers” is not self-describing is core to the specification; “the contract” which defines the semantics of the data messages to be exchanged must be known to all system actors ahead of time.

Alternatives

Embedded projects often define their own tailor-made domain specific specifications for serialization in order to define only the needed features and minimize overhead. While this approach has the possibility to be optimal for a particular project, it also bears the risk of increased budget costs due to bugs in the implementation and unforeseen specification limitations.

Over the years there have been various general-purpose solutions developed for active communication between machines such as Soap, Corba, and COM/DCOM. Though often comprehensive and designed to handle a full RPC system out of the box, they tend to be heavy weight in terms of integration and resource usage, making them potentially unattractive for embedded projects.

Text based data serialization formats such as XML and JSON have dominated sectors such as the web and application spaces for years due to their highly portable nature and reasonable level of human readability. However, they are also heavy weight in many aspects, including parsing, encoding, and storage, and thus are often unsuited to embedded projects.

Since its first public release, “Protocol Buffers” has spawned a renaissance in this development space and as such there are various peer binary based alternatives now available. Similar in terms of features and performance, “Apache Thrift™” [2]

originally developed from an ex-google employee at Facebook, is one such open source candidate. Though the documentation is not as mature as Google's, the "Apache 2.0" licensed specification and tooling provide an out of the box RPC framework suitable for many applications.

In the same vein is the more recent "Apache Avro™" [3] which allows for both an efficient binary packed data format as well as raw JSON which aides in debugging. Besides also offering a RPC solution, "Apache Avro" ensures upfront data contract exchange, allowing endpoints to receive the data without knowing the content ahead of time. Though similar in terms of performance to "Protocol Buffers", it is also not at the same level of maturity in terms of documentation and support.

Another class of efficient binary data serialization is defined by the new wave of open source "zero copy" formats. The major players in this space are "Cap'n Proto" [4] from the main author of "Protocol Buffers" v2, "SBE" [5] for financial trading, and "Flat Buffers" [6] from Google for game development. They do away with the encoding/decoding data transformation steps altogether, instead favoring an increased wire size in exchange for faster performance by intelligently mapping the data in a platform neutral way directly onto the wire through portable padding and alignment.

Protocol Buffer Specification and Runtime

Googles has released two versions of "Protocol Buffers" to the public, respectively known as "Proto2" and "Proto3". Proto2 was released in 2008 already full featured with official tooling supporting C++, Java, Python, and Javascript. Third party implementations have been released over the years for most other popular languages such as C, C#, and ruby. Proto3 was released mainly to streamline the existing specification and interface language as well to add support for more target languages. Proto2 is largely compatible with Proto3. Since Proto2 was the only option available at the time of the decision to use "Protocol Buffers" in the medical device project discussed in this paper, the rest of the relevant descriptions and details will be focused solely on Proto2. Since the release of Proto3, no feature or modification added was deemed important enough, to trigger a transition to Proto3 for the project.

Besides the core specified wire syntax, Google also specifies an "Interface Definition Language or IDL" for specification of the data messages to be broadcast across machine boundaries. This IDL allows for the definition of the "data contract" and is processed by a compiler to generate the glue boilerplate code in the target application language. Runtime libraries then use this boilerplate code to encode and decode new messages for exchange.

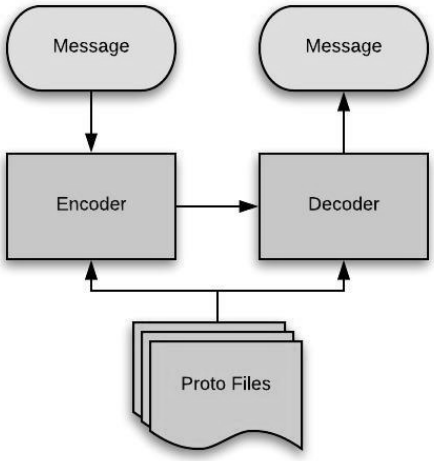

Figure 2: Interrelation of IDL and Runtime Libraries

The IDL itself is designed to be very straightforward, allowing a natural definition of data that should be packaged together through “messages”, where each message contains various numerically tag identified fields that have assigned primitive data types such as “bool” or “int”. Enumeration types can be defined for readability and type safety, and messages themselves can be nested to allow composability and reusability. A full description of the language can be easily understood through the documentation provided by Google and the associated examples.

The wire syntax format has an overhead of one byte per field. This overhead byte encodes the numeric tag for the associated data, and a wire type to allow the runtime library to correctly address each field. The four main wire types used are

VARINT - a variable length packed minimum encoding of all integer types
FIXED32 – a fixed 4-byte value used for floats and other fixed sized types
STRING – a variable length string encoding
REPEATED – a list like encoding of contiguous elements of the identical type

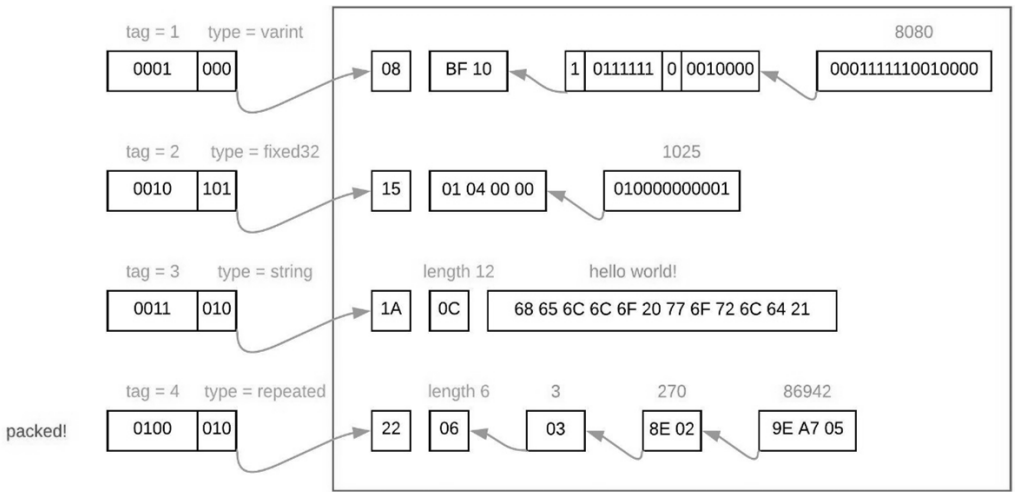

Figure 3: Supported Wire Types

Google provides a robust open source implementation of the runtime libraries needed to integrate into programs written in various popular languages, as well as the tooling needed to read and translate the “Protocol Buffers” IDL. However, the C++ implementation makes heavy use of the C++ “Standard Template Library”, dynamic memory allocation, and starts at about 100 kb of ROM space. This makes it fully unsuited for embedded projects running C/C++ code on constrained microprocessors.

Over the years various third-party open source libraries written in C have come and gone. The currently best suited choice “NanoPB” [7] was chosen for the medical device project in question. The implementation is fully static by default and has a focus on code size over performance, using just between 2 and 10 kb of ROM, depending on configuration, and only 300 bytes of RAM. Despite its static nature, custom sized data fields can be processed either via a callback API or custom compile-time sizing options. The documentation, like Google’s, is robust and extremely mature.

Project Details

The aforementioned medical device project required a portable solution which included at a minimum support for C/C++, C#, and Java and that could run both on PCs and midrange microprocessors. We decided from the very beginning to avoid creating a custom specification/implementation that would require duplicate effort to support the required languages/platforms, and potentially lead to unexpected schedule and/or budget bloat. Regarding our need to minimize the memory and performance foot print on the required microcontrollers, we did not want to use a portable yet expensive solution such as JSON or XML. When considering what open source alternatives were available, we had to strongly consider the customer’s licensing concerns, as they had at that time not yet integrated an open source library directly into the firmware of one of their device projects. Considering all these aspects and restrictions, NanoPB was chosen as the best candidate, and thanks to the quality of the documentation and sample code, we were quickly able to verify its suitability.

The hardware structure of the project consists of two microcontrollers running C++-code bound by a UART connection, one of which is additionally bound by a UART over USB connection to a host system. The main controller acts as a soft real-time constrained orchestration component managing the state of defined use cases, controlling various non-time critical hardware components, and maintaining communication/synchronization with the host system. The secondary controller acts the core calculation controller running the customer’s IP with strict hard real time requirements. We required a simplified RPC system allowing any one component in the system to remotely call code running on modules on another component and potentially asynchronously receive related data or state triggers.

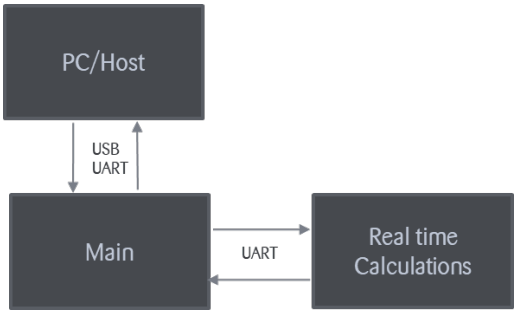

Figure 4: Hardware Overview

Project Integration Tooling

Although no RPC framework is included with NanoPB, our requirements were quite straightforward with only limited prerequisites for addressing data on remote endpoints. As such we choose a light weight DSL based code generation approach for defining our own boilerplate application layer code for remote calls and responses between system components. We defined our DSL according to the approach specified by “Diesel” [8], which uses a simple LL1 parsing strategy in ruby to quickly define and integrate bespoke DSLs. We leveraged the power and flexibility of “Gaudi” [9], a build system extension for ruby’s own build system Rake, to implement the various tasks and helpers needed to perform the parsing, generation, and build work required.

With this approach we defined system wide constants and “service” definitions in our custom DSL; each service representing an external interface for a logical subsystem of a specific hardware/software component. These definitions are mapped to Google’s own IDL and passed through the supplied compiler to generate appropriately modified NanoPB C-Structs for our runtime code. Separately C++ boilerplate classes are generated by our ruby tooling to aid in encoding and decoding the data as well as for addressing the data to the correct subsystem.

Communication Stack Overview

To ensure that the feature set of our communication stack contained only what we needed, we chose to specify and implement a custom light weight stack using “Protocol
uffers” as the data medium. Individual packet structure consists of two consecutive “protocol buffer” encoded messages with no separator. The first part provides common header data for routing and data integrity, and the second specifies the domain specific payload.

The header is encoded field by field using library functions available in every language API allowing for custom manual encoding and decoding according to the specification. In comparison the data payload can be encoded with a direct serialization library call and the associated message data contract. When decoding a full data packet, the boundary between header and payload can be identified by reading out the consecutive “protocol buffer” sequence tags. The payload is identified by the first field with a numerically smaller tag than previous fields, or rather when all expected header fields have been processed.

Figure 5: Packet Structure

Packets are separated using a clear zero-byte terminator, which requires the re-encoding of data relevant zero bytes. For this purpose, we selected the straightforward “Consistent Overhead Byte Stuffing” [10] algorithm, which re-encodes zero bytes into length delimiters of consecutive non-zero byte runs. This allows packets to be of unlimited size with a minimum overhead of 1 byte and a maximum of 1 byte for every 254 bytes of payload data.

Packet integrity is secured using a standard “cyclic redundancy check”. A CRC32 algorithm encodes a CRC for every full packet (header + payload), which is encoded into the header, and read out as the first step of the data decode process on the remote target endpoint. To ensure that the CRC is calculated over the entire packet data, without the CRC present, we again use the manual encode/decode functions for fields available in the runtime library, to calculate the CRC with the CRC bytes initialized to 0. With the fixed location and size of the “Fixed32” based header field, we can reliably identify the CRC and ensure integrity in a packet regardless of a data corruption’s position.

Stream integrity, as defined by the order of packets sent and received, can be ensured by way of a simple cyclic numeric packet identifier stored in each header between the values of 1 and 127. To minimize overhead, this identifier is then acknowledged or not acknowledged using a secondary custom “protocol buffer” packet type transmitted over the same channel. This simplified packet includes a header consisting of only a CRC and a payload with a single field for the acknowledged identifier, or zero in the case of a NACK. This second class of data packet can be efficiently differentiated by the field count discovered in the header.

Conclusion

Although traditionally not necessarily the first choice, the potential advantages of integrating popular and well-maintained open source libraries (free/libre and/or OSS) are immediately clear upon consideration for embedded software applications. Although custom protocols will always have their place depending on the scope and resources of a project, the ability to introduce a mature and well-defined serialization protocol from day one can save a massive amount of time. In the case of NanoPB and the corresponding specification of Googel’s “Protocol Buffers”, any candidate project would benefit from a solid, efficient, and well tested implementation, together with comprehensive and highly understandable documentation. All of this of course needs to be weighed against the complex nature of the suitability of an OSS license today, the risk of alterations to licensing in the future, and the long-term costs of software verification.

Bibliography

[1] Google “Protocol Buffers”
https://developers.google.com/protocol-buffers/

[2] Apache Thrift™
https://thrift.apache.org/

[3] Apache Avro™
https://avro.apache.org/

[4] Cap’n Proto
https://capnproto.org/

[5] Simple Binary Encoding
https://github.com/real-logic/simple-binary-encoding

[6] Flat Buffers from Google
https://google.github.io/flatbuffers/

[7] NanoPB – protocol buffers with a small size
https://jpa.kapsi.fi/nanopb/

[8] Diesel - A Plea for a Lightweight Approach to DSLs
https://www.zuehlke.com/blog/en/diesel-part-1/
https://www.zuehlke.com/blog/en/diesel-part-2/

[9] Gaudi – A builder
https://github.com/damphyr/gaudi

[10] CBOE – Consistent Overhead Byte Stuffing
https://en.wikipedia.org/wiki/Consistent_Overhead_Byte_Stuffing

Author

Morgan Kita has been one of Zühlke Engineering GmBH's expert software engineering consultants for nearly 4 years. There he focuses on the development of embedded firmware for their clients as well as infrastructure for improving the cycle of testing and deployment. In conjunction with 10 years in the biotechnology and game programming sectors, he is a developer with a broad range of software expertise and enjoys both learning and sharing ideas on how to get the most out of today's technologies.

Contact
Email: morgan.kita@zuehlke.com
Zühlke Engineering GmbH, Düsseldorfer Str. 38, 65760 Eschborn, Germany