# Introduction

## Purpose

This document contains the design of utility components for Google Protocol Buffers–formatted messaging and data serialization. Protocol Buffers itself is a binary protocol for serializing structured data. It is available for public use with open source licenses that have been cleared with Philips Intellectual Properties and Standards [XJV031020.13].

## Intended Audience of Document

The intended audience of this document is software developers.

## Scope

The utility components to be described reside in infra/basicsw/basicarchive/protobuf building block.

Bit-level implementation of the data structure serialization and deserialization is handled by third party libraries and is outside the scope of this document. Different libraries are needed to support multiple programming languages, most notably C# (Protobuf.NET) and C++ (Protobuf).

## Google Protocol Buffers Primer

Google Protocol Buffers use a concept of messages to transmit or store data in binary format. Despite the name, messages are actually static structures of simple data and not limited to communication protocols. The data inside the structures is arranged in tag-value pairs. Tags are integer values that are restricted to a following range:

]0-19000[, [20000- 536870911]

Values are restricted to the following types (and their array forms):

* double, float
* int32, int64, uint32, uint64, sint32, sint64, int32 enums
* fixed32, fixed64, sfixed32, sfixed64 (non-compressed integers)
* bool
* string (UTF-8)
* bytes

Both tag values and normal integers undergo simple compression for maximal efficiency for small values.

The data is typically declared in platform-agnostic format with human-readable text files. These files are known as .proto files due to the filename extension convention. The data structures can be declared with a package namespace, which comprises structures in a single proto-file.

### Known Limitations

The maximum message size is limited to 64MB by default (2GB theoretical limit). Large data entries should be chunked to several messages to fit into the default size limit.

Due to the C++ implementation of enumerators, where the protobuf enumerators are directly converted to C++ enumerators, the enumerator names need to be unique within the data model namespace to prevent ambiguous usage.

Data Model Rules:

1. Messages can never be removed
   1. they can be marked as obsolete, e.g., by a recommended naming convention of prefixing the old name with OBSOLETE\_ keyword, but the Id should remain reserved to support backwards compatibility (new messages with the same id shall not be introduced)
2. Fields can never be removed
   1. they can be marked as obsolete, but the Id should remain reserved to support backwards compatibility (new fields with the same id shall not be introduced)
3. Enumerator names must be unique within the package namespace. The enumerator values do not have this requirement.

## Terms and Abbreviations

CORBA XTC: Deprecated, **C**ommon **O**bject **R**equest **B**roker **A**rchitecture E**XT**ernally **C**ontrolled Scanner interface.

Message Class: A C# class that describes a protobuf message contents and provides a unique message id.

.NET : Microsoft-provided software framework.

POD: Plain-Old-Data. A data structure without methods, i.e., a passive collection of field values.

.proto: A file extension for a text file that contains Google Goolge Protocol Buffers message definitions in a cross-platform format.

Protobuf: an open-source C—library for serializing data in Goolge Protocol Buffers-format.

Protobuf.NET: an open-source .NET library for serializing data in Goolge Protocol Buffers -format.

## Design Specification

Protobuf building block provides functionality for

* Means to create a protobuf datamodel using C# Plain-Old-Data (POD) classes and reflection.
* Means to import a protobuf datamodel from a .proto file.
* Fast serialization and deserialization of Google Protobuf messages into .NET class instances.
* Serializer class for type-safe serializing with .NET streams.
* Proto-file generation for exporting datamodels to a language-neutral .proto-file format.

### Datamodel Creation

For simple data models and for a rapid development, the protobuf data model can be declared as a collection of publicly accessible C# classes in a .NET assembly, instead of using .proto files. Because the classes are implemented in protobuf as protobuf messages, the classes are named in this document as ”Message Classes”. Protobuf building block provides an implementation for reflecting message classes into a Protobuf.NET data model, which is optimized for fast binary serialization and deserialization at runtime.

Declaring .NET Assembly

Message classN

Message classN

Message classN

Philips.PmsMR.Protobuf\_cs

Protobuf.NET

User-of-datamodel

Another Assembly

Local user of datamodel

Data Model Tag

The Plain-Old-Data (POD) C# message classes can consist of public fields of simple types, that is:

* Booleans
* Bytes
* Enumerators
  + Global uniqueness of enumerator names is enforced with explicit checks.
* 32bit integers
* 64bit integers
* 32bit floating point values
* 64bit floating point values
* Strings
* Other POD classes
* Arrays of the above

**Example Data Model**

public class SampleChild

{

public Byte[] BytesValue;

}

public class Sample

{

public bool Int32 Int32Value;

public string StrValue;

[ForcedId(123)]

public double OBSOLETE\_DoubleValue;

public SampleChild[] ArrayOfChildren;

}

In addition to the primitive types, globally unique identifiers (GUIDs), and DateTime/TimeSpan values can be used, but these result in an additional .proto file called ”BCL.proto” to cater for .NET specific extensions. For maximal portability, these should be avoided and the contents provided in other, message-specific formats that are platform-agnostic. For example, UTC DateTime values can be provided as 64bit integer tick values (100ns clunks). Note that 16bit integers are mapped to 32bit integers—this also applies to the array formats.

In addition to message classes, the declaring implementation needs to provide a data model tag class, which identifies the message class types that form the data model. The data model tag class does this identification by implementing IDataModelTag interface. The tag class can also provide an optional .NET namespace string and explicit message type id integer field that is to be present in polymorphic classes (see Polymorphism chapter below) via INamespaceTag – all message classes can be declared inside this namespace and unique integer ids are guaranteed for all parentless message classes.

In general, it is recommended to declare all message classes under a single data model-specific namespace for consistency.

### Importing of a Data Model

If the data model is an already existing .proto-file, it can be converted to C# files with a command-line ProtoGen-executable that comes with Protobuf.NET. By providing a proto-file tag class that identifies generated message types and their integer ids via a IProtoFileTag interface, the message type id mapping can be created for serialization and deserialization.

For example, ProtoGen.exe can be executed as a part of the build process to produce C# files, which are then included an assembly (here: SampleAssembly). A ProtoFileTag implements IProtoFileTag and identifies MessageId as the enumerator that lists message classes and their unique ids. In this example, the ProtoFileTag implementation also uses reflection and the a-priori known package namespace to automatically retrieve all message classes in the SampleAssembly.

package SampleNS;

enum MessageId {

SampleMessageId = 1234;

}

message SampleMessage {

optional Int32 sample\_value;

}

Proto-file

ProtoFileTag Impl.

SampleNS reflection

Sample Assembly

Source files generated (ProtoGen)

### Fast Serialization and Deserialization

Fast serialization operations for .NET are based on a third-party Protobuf.NET library. This library is able to compile, also at runtime, converters for the given set of .NET types. After the compilation is finished, the converters can efficiently serialize instances of the given .NET types into byte arrays, or instantiate new instances from byte arrays.

#### Type Mapping

Protobuf building block contains an auxiliary TypeMapper component for creating a Protobuf.NET converters at runtime. It is used for those cases where the data model has been declared with C# message classes. TypeMapper uses .NET reflection to initialize itself, and delegates lengthy reflection of C# declared data models to background worker threads to reduce delays. The public interface of TypeMapper is inherently thread-safe. To be able to reflect a correct set of types into a data model, it requires the calling component to provide following:

* An array of public types to be used in the data model
  + Types need to be accessible to the Protobuf assembly
* An enumerator for message type to id mapping
  + Used only if .proto file is used for the data model declaration

This is done by providing a data model tag instance as an input parameter to the type mapper.

As a part of C# declared data model reflection, TypeMapper generates unique integer ids to the message class types and the fields in the message classes. To end up with the same ids each time the reflection is carried out, the names of the classes and fields are used to create valid Google Protocol Buffers integer values (]0-19000[, [20000- 536870911]). The hash id is calculated from ASCII names with a simple Fowler–Noll–Vo formula:

int length = nameId.Length;

int hash = length;

for (int i = 0; i != length; ++i) hash = (hash ^ nameId[i])\*16777619;

In the case the hash id is already taken or needs otherwise to be fixed to a particular number, an id override can be provided via a custom .NET meta attribute ”ForcedIdAttribute”. This attribute can be applied to both the message classes and the fields of the said classes.

For .proto declared data models, the message-id enumerator is used to deduce the ids. For example, the default implementation expects to get an enumerator type that has message class names postfixed with “Id” string, and where the enumerator value is a valid Google Protocol Buffers integer value.

#### Polymorphism

Google Protobuf does not provide direct support for polymorphism of messages. Protobuf.NET utilizes the optional fields to achieve polymorphism: all direct child classes are declared on the base class as individual optional fields, but only one of the fields can be non-null in an instantiated class.

Namespace tag class can be utilized to create automated, C++-reflection based type-casting also for cross-platform code: the tag instance can declare that all polymorphic classes are to be derived from a single root class, and that root class can contain an explicit type id field that provides the exact type information of class instance to the C++ code. Namespace tag provides reflection information about the type id field, and Philips.PmsMR.Protobuf\_cs provides a Factory class for creating message class instances with the type id field automatically set to the correct value:

RootClass (5678)

+int TypeId

B1 : RootClass (9876)

+int Foo

C : RootClass (123)

+int Bar

B2 : RootClass (432)

+int FooBar

enum MessageId {

B1Id = 9876;

B2Id = 432;

C = 123;

RootClassId = 5678;

};

message B1 {

optional in foo =..

optional C C = 123;

};

message B2 {

optional int foobar..

};

message C {

optional int bar =..

};

message RootClass {

optional int type\_id =..

optional B1 B1 = 9876;

optional B2 B2 = 432;

};

In the example above, instantiating a class ”C” and converting it to a protobuf message results in the following protobuf message contents:

var myC = TypeMapper.Factory.Create<C>();

myC.Bar = 1;

myC.Foo = 2;

RootClass(type\_id == 123, B1 == B1(Foo=2,C = C(Bar=1)), B2 = null)

#### Collapsed Namespaces

For legacy applications, most notably for the old CORBA XTC implementation, Namespace Tag can define additional ”collapsed” namespaces, under which all ids should be unique. This restriction concerns also fields. It allows mapping to other data formats, e.g., DICOM-like dictionaries, where tags of tag-value pairs need to be globally unique.

### Serializer

Google Protobuf does not specify the wire-format protocol needed for transmitting the structures in byte-format. Therefore a following, arbitrarily chosen convention is used:

|  |  |  |
| --- | --- | --- |
| **Length Bytes** as little-endian Int32 | **Message Type Id** as little-endian Int32 | structure bytes |

Length bytes include the four bytes from the field itself. That is, the content of length field is 8 bytes + number of bytes in the structure array. The minimalistic zero-content message is 4 bytes containing the value of 4.

Philips.PmsMR.Protobuf\_cs provides a Serializer class for serializing message class instances into .NET streams and deserializing stream bytes into new message class instances. It also provides support for deserializing keepalive messages for those cases where the underlying stream is implemented with sockets and the protocol requires active detection of socket communication problems: the keepalive messages are periodically dispatched zero-content messages. They are handled in the deserialization code and reported to the caller of the deserializing method.

### Proto-File Generation

Protobuf building block contains an auxiliary executable, protogenerator\_cs.exe, for generating a .proto file (or protofiles, if BCL.proto extensions have been used) from the C# data model. The rules of conversion are detailed below, in Type Mapping chapter. The resulting file contains following elements:

* MessageId-enumerator.
  + All parentless C# message classes are listed by appending ”Id” to the class names. The naming convention of the enumerator names does not follow the proto-file style guide, where enums are to be written with CAPITALS\_WITH\_UNDERSCORES for value names.
* Package declaration.
  + Protobuf namespace that does not need to be the same as .NET namespace
* Messages and enumerators of the datamodel in alphabetical order.

## Future Extensions

### Reflection Delay

The delay related to reflection can be removed by compiling the runtime type model to the disk as a separate assembly and by storing the related data structures to cache files. The datamodel assembly checksums can be used to detect the need for re-creating the assembly and the associated data structures.