Messages ™

by **Xplicit Computing, Inc**

FILE & WIRE SCHEMA
LANGUAGE BINDINGS
for
SYSTEMS | GEOMETRY | PHYSICS | INTEGRATION

Download or Clone from Github

System	*Elements*	*Topology*	*Meta*
Model	*Indices*	*Zone*	*Visual*
Rule	*Vector64*		*Representation*
Script	*Vector32*		
Command	*Extrema*		
Function	*Revision*		
Variables			

I. "Rosetta Stone" for numerical systems

Messages is a schema designed for engineers and scientists that simply and efficiently enables numerical and related data to be shared across computers and programming languages.

Computer-generated code interfaces can deliver high-performance, elegant, and flexible messaging across most science and engineering fields, providing flexible encoding/decoding like XML and JSON but faster and denser. A machine-generated *Messages* library contains utilities to flatten and reconstruct object-oriented and vectorized data structures encountered in numerical simulation setup, expression, and results (e.g. CFD, FEA, EDA, and geometry processing).

Standard protocol buffer "protobuf" definitions offer a compact layout for data found in numerical computing storage and transmission. The *Messages* schema is passed into a *protoc* compiler to generate high-performance binary-encoded accessors for various languages. *Xplicit Computing*'s applications are built on these bindings – end-users can also apply those same standards to their own custom integration for improved workflow continuity and performance.

Four files** are central to the xcompute ecosystem, organized by context:

- ➔ **vector.proto** - numeric arrays using packed arena allocation (e.g. Vector64 ***.xco** files)
- ➔ **spatial.proto** - topology of elements and regions for a domain (e.g. Geometry ***.xcg** files)
- ➔ **system.proto** - domain setup, models, parameters, associations (e.g. Setup ***.xcs** files)
- ➔ **meta.proto** - meta-data and user-graphics for a specific system (e.g. Meta ***.xcm** files)

Benefits:

- binary file-and-wire formats for storage and transmission between computers
- native compatibility across *XCOMPUTE* and user applications
- universal accessors and serialization utilities for every OS and many languages
- efficient parallel read/write with reverse and forward compatibility
- free and open standard built with agnostic infrastructure

Limitations:

- protobuf 32-bit indexing, leading to a 2B element maximum per domain
- reading large messages greater than 64 MB requires special treatment (security measure)

Each system manages its own setup and members or references, including potential child sub-systems. It is up to each team to determine an appropriate fidelity for each study. Most messages are designed to meet the needs of a specific context, so messages files are separated for convenience.

Users and developers can use all or part of the provided messages as fit.

II. Adding messages to your project

Pre-built bindings are provided for the following languages: C++, Objective-C, C#, PHP, Python, Java, Javascript, and Ruby. Languages such as Dart and Go are available as official plugins. Several additional third-party bindings are available, some of which are listed in § V.

In your programming environment, import the relevant files as headers or libraries. Statically-compiled languages such as C++, Obj-C, and C# require linking to static library libxcmessages.a . Depending on usage, you may also need to install Google Protobuf 3 Libraries.

Message classes will become available for each of the four files imported. In C++ they can be found under the namespace Messages:: after including one or more *.pb.h header bindings. Other languages will name and organize the interfaces slightly differently. Refer to the *.proto definitions for all schema definitions, as access patterns are built from these assignments directly. *Get* and *set* functions provide accessors to members encoded in protocol buffer streams and files.

There are 18 useful messages that each relate to different concepts. Most users don't need to change any definitions, but the *.proto files can be handy references while constructing and assigning messages in a coding environment. The *vector.proto* set are well-suited for storing and transmitting vectorized data (e.g. space and time domain, frequency domain, scalar and vector quantities).

III. Serializing and parsing messages

In order to save or transmit, a program's associative data structures must be serialized into a format that permits representation in contiguous memory and/or storage. For instance, several built-in serialization functions are available for C++ including those that leverage a std output file stream.

```
std::ofstream outfile(path); // create an output file stream with given path
msg.SerializeToOstream(&outfile); // serialize a prepared Messages::Vector64
outfile.close(); // finish the file and release resource
```

Serialized data can then be transmit over file-and-wire to another computer session, where it is then de-serialized (parsed) using the conjugate pair load function such as std input file stream:

```
std::ifstream infile(path); // create an input file stream
Messages::Vector64 msg; // create an empty message container
msg.ParseFromIStream(&infile); // de-serialize binary file to fill message
infile.close(); // finish the file and release resource
```

Please refer to the Proto3 Tutorials for a comprehensive reference on patterns across languages.

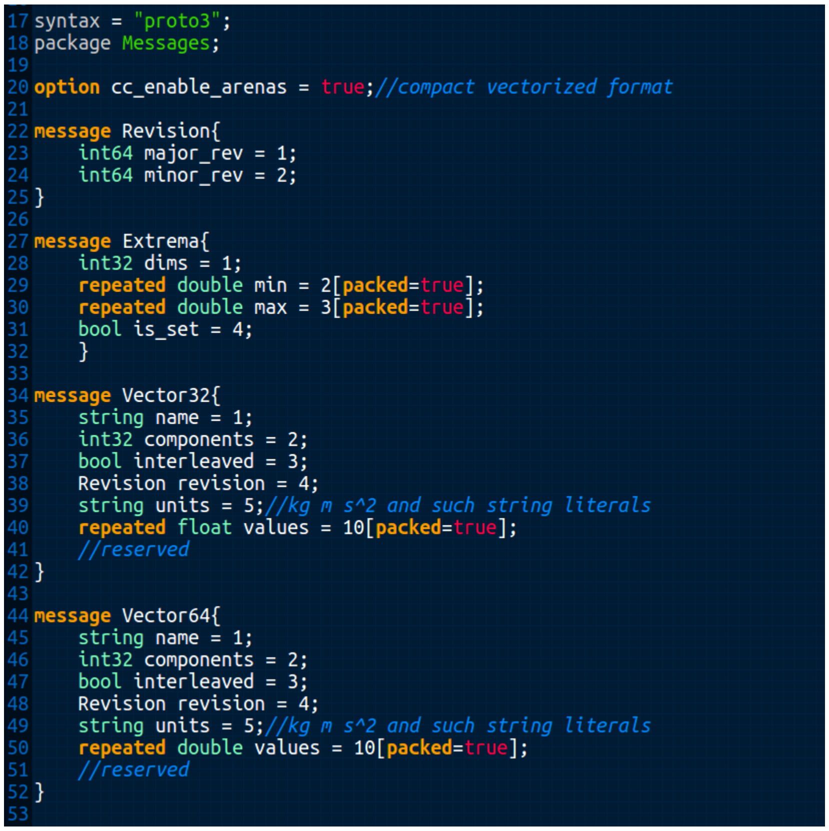
IV. C++ examples

To get started, explore the mini app in *examples/hello_vector* to demonstrate save and load floating-point data with the file system. It should compile and run out-of-box using build.sh, save.sh, and load.sh scripts using a C++ compiler such as gcc or clang. Be sure that the Google Protobuf 3 Libraries path and our schema-specific library are properly linked in the local CMakeLists.txt with either absolute or relative path (or else you'll get a linking error after compilation):

```
target_link_libraries(some_app /path/to/libprotobuf.a)
target_link_libraries(some_app /path/to/libxcmessages.a)
```

Or simply compile without cmake by specifying library paths with -L flag:

```
c++ save_msg.cpp -L/path/to/libprotobuf.a -L/path/to/libxcmessages.a
```



```
syntax = "proto3";
package Messages;

option cc_enable_arenas = true;//compact vectorized format

message Revision{
    int64 major_rev = 1;
    int64 minor_rev = 2;
}

message Extrema{
    int32 dims = 1;
    repeated double min = 2[packed=true];
    repeated double max = 3[packed=true];
    bool is_set = 4;
    }

message Vector32{
    string name = 1;
    int32 components = 2;
    bool interleaved = 3;
    Revision revision = 4;
    string units = 5;//kg m s^2 and such string literals
    repeated float values = 10[packed=true];
    //reserved
}

message Vector64{
    string name = 1;
    int32 components = 2;
    bool interleaved = 3;
    Revision revision = 4;
    string units = 5;//kg m s^2 and such string literals
    repeated double values = 10[packed=true];
    //reserved
}
```

Figure 1: Contents of vector.proto is human and machine-readable schema. Each message is defined from primitives, other messages, and/or repeated fields of primitives and messages.

Intro to Messages 2020.1

1. Assigning messages

Include the message header file containing the desired messages, such as:

```
#include "vector.pb.h"
```

a. Fill repeated fields (one-by-one)

```
Messages::Vector64 msg; // first, create an empty message container
msg.set_name("Position\tValue"); // set the name field with a string **
msg.set_components(3); // or however many vector dimensions, ie. 3 for xyz
msg.add_values(pos.x); //push back x value, e.g. from some glm::dvec
msg.add_values(pos.y); // push back y value
msg.add_values(pos.z); // push back z value
//add additional entries, only in triplets in this case...
```

** It is recommended to set the name using the *Property-Key* convention (using tab delimiter):

```
Property  or:  Property\tModifier  or key-chain:  Property\tModifier\tModifier...
```

b. Fill repeated fields (serial loop)

```
//given a std::vector other
for (auto val : other) // iterate through each element of other
      msg.add_values(some_function(val)); //add element one-by-one
```

c. Fill repeated fields (parallel loop)

```
auto N = other.size(); // get the source vector size
auto& values = *msg.mutable_values(); // get ref to mutable values
values.resize(N); // allocate memory in the destination
#pragma omp parallel for
for (auto n=0 ; n<N ; n++) // iterate concurrently
      values[n] = some_function(other[n]); //assign msg entries
```

2. Saving messages to file

```
std::ofstream outfile(path); // create an output file stream with given path
msg.SerializeToOstream(&outfile); // serialize a prepared Messages::Vector64
outfile.close(); // finish the file and release resource
```

3. Loading messages from file

Again, using the message header file containing the desired messages:

```
#include "vector.pb.h"
```

a. Use standard input file stream:

```
Messages::Vector64 msg; // create an empty message container
std::ifstream infile(path); // create an input file stream
msg.ParseFromIStream(&infile); // de-serialize binary file to fill message
infile.close(); // finish the file and release resource
```

b. See Example 7 for loading large messages (greater than 64MB)

4. Accessing messages

a. Access repeated fields (manually by entry):

```
auto& vec_name = msg.name(); // get string ref
auto C = msg.components(); // get int32 copy (as an alternative to ref)
auto& rev = msg.revision(); // get Messages::Revision ref
std::vector<double> output; // declare destination vector
if (msg.values_size() > 1 && rev.major_rev() < 42) // do something with size, rev
{
      output.push_back(msg.values(0)); // returns a double for first entry
      output.push_back(msg.values(1)); // returns a double for second entry
}
```

b. Access repeated fields (serial loop, better):

```
std::vector<double> output; // declare destination vector
for (auto value : msg.values())
      output.push_back(some_function(value)); //use the protobuf
```

c. Access repeated fields (parallel loop, best):

```
auto N = msg.values_size(); // get number of values
std::vector<double> output(N); // allocate destination
#pragma omp parallel for
for (auto n=0 ; n<N ; n++) // concurrent iteration
      output[n] = some_function(msg.values(n)); //access msg entries
```

5. Embedding messages

More complex data structures require a special getter prefixed with the mutable_ keyword to return pointer to an underlying mutable object.

a. For instance, Vector::revision is an embedded message inside Vector64, so it cannot be assigned as a primitive with the set_ keyword. However, once a mutable pointer has been retrieved, we can assign its member primitives:

```
auto rev = msg.mutable_revision(); // getting a mutable object returns a pointer
rev→set_major_revision(4); // access the pointer and set the member
rev→set_minor_revision(13); // access the pointer and set the member
```

b. A slight stylistic variation de-references the pointer and takes a ref to the underlying object to eliminate extra pointer access arrows for cleanliness:

```
auto& rev = *msg.mutable_revision(); // de-ref pointer and apply C-style ref
rev.set_major_revision(4); // access and set the member
rev.set_minor_revision(13); // access and set the member
```

Note: Forgetting the **&** in `auto&` on the first line in 1b will result in a copy of the object to be created on the stack and the assigned values will not touch the intended underlying msg object.

6. Copying messages

Messages can be explicitly copied from other messages of the same type:

a. `msg = other_msg;`

b. `msg.CopyFrom(other_msg);`

c. As repeated values exist in contiguous memory (i.e. packed arena allocation) and each conform to *IEEE-754 floating-point standards*, it is also possible to perform a C-style low-level copy to/from contiguous standard library and other containers, as desired:

```
msg.mutable_values()->Resize(other.size());
memcpy(msg.mutable_values()->data(), other.data(), other.size()*sizeof(double))
```

7. Loading large messages with Google::Protobuf::CodedFileStream

This process is necessary to bypass security restrictions in place within the Protobuf 3 libraries. A default limitation of 64MB is used to discourage buffer overflow attacks in internet infrastructure. The good news is that Google made it possible to increase and/or disable the limits up to int32.

The process requires use of the coded and zero-copy stream machinery used in parsing operations:

```
#include <google/protobuf/io/coded_stream.h>
#include <google/protobuf/io/zero_copy_stream_impl.h>
```

```
Messages::Vector64 msg; // create an empty message container
unsigned int IObyteLimit{ 1 << 31 }; // buffer size (0xffffffff max)
```

A coded input stream requires a raw file input stream with an open path:

```
int fd = open(path.c_str(), O_RDONLY); // open path and get file descriptor
FileInputStream raw(fd); // create a raw input stream buffer
CodedInputStream inputStream(&raw); // create a coded input stream
```

Set the stream byte limit at safe max and warn at half

```
auto limit = inputStream.PushLimit(-1); // limit work-around
inputStream.SetTotalBytesLimit(IObyteLimit >> 1, IobyteLimit);
```

Finally, do the actual parsing from input stream to message

```
if (msg.ParseFromCodedStream(&inputStream)) //if okay
{
      inputStream.PopLimit(limit); // limit work-around
      close(fd); // clean up
}
else
      // handle the error (e.g. message type mismatch, invalid assignments)
```

V. Building custom bindings

If you wish to use languages not provided or would like to customize the message schema, you can install a protoc compiler and 3rd-party protobuf plug-ins (including C, Perl, R, Rust, Scala, Swift, and Julia). Reuse in independent projects is encouraged as long as existing message definitions are not altered – only added and not distributed. It is recommended to enumerate custom extensions some number greater than 100 to minimize chance of conflict with official assignments.

To generate new bindings, install a Google Protobuf 3 Compiler from a package manager or sources. Check the bin path and version:

```
> which protoc
```

```
> protoc --version
```

After protobuf3 and protoc are installed, make a directory for each desired language and run the compiler from shell or script:

```
> mkdir -p cpp python java javascript ruby objc csharp go
```

Invoke the protoc compiler for the desired output language (by defining the name of the output folder following equals sign) against proto files to yield libmessages.a and bindings:

```
> protoc --cpp_out=cpp --csharp_out=csharp –objc_out=objc --ruby_out=ruby
      --python_out=python --java_out=java --js_out=javascript
      vector.proto system.proto spatial.proto meta.proto
```

Some languages such as Javascript require additional options for encoded serialization utilities:

```
--js_out=javascript,import_style=commonjs,binary:.
```

VI. License and fair-use agreement

We hope these nuggets will be useful and appreciate your feedback!

The Xplicit Computing Team

California, USA
info@xplicitcomputing.com