RTI Protocol Buffers Extension

User's Manual

Version 7.7.0

Contents

1 About Protocol Buffers Extension **1**
- 1.1 Use Cases . . . 1
  - 1.1.1 Protocol buffers definition language support . . . 1
  - 1.1.2 DDS-XTYPES type generation . . . 1
  - 1.1.3 Protocol Buffers C++ language binding . . . 2
  - 1.1.4 Integration with DDS . . . 2
- 1.2 Components . . . 3
- 1.3 System Requirements . . . 3
- 1.4 Paths Mentioned in Documentation . . . 3

2 Protocol Buffers Extension Components **5**
- 2.1 IDL4 Converter Plugin . . . 5
- 2.2 C++ Code Generator Plugin . . . 6
- 2.3 DDS Options for Protocol Buffers . . . 7
- 2.4 Accessing the RTI Plugins . . . 8
- 2.5 Limitations . . . 8
  - 2.5.1 Platforms and compatibility . . . 8
  - 2.5.2 Recursive types . . . 8

3 Integrating Protocol Buffers with Connext **9**
- 3.1 Two-Phase Code Generation . . . 10
- 3.2 Single-Phase Code Generation . . . 10
- 3.3 Code Generation Commands . . . 10
- 3.4 Generating Code for Multiple Files . . . 12
  - 3.4.1 Integrating .proto files with dependencies . . . 12
  - 3.4.2 Managing dependencies . . . 14

4 Tutorial: Generating Code from Protocol Buffers Messages **15**
- 4.1 Before You Begin . . . 15
- 4.2 Define Protocol Buffers Message Types . . . 16
- 4.3 Convert Protocol Buffers Data . . . 17
- 4.4 Exchange Protocol Buffers Data Over DDS . . . 17
- 4.5 Communicate with Other DDS Applications . . . 19
- 4.6 Build and Run the Example . . . 21

5 DDS Options for Protocol Buffers **23**
- 5.1 Message Options . . . 24
- 5.2 Field Options . . . 25

5.3 RTI Protocol Buffers Descriptor File . . . 25

6 Protocol Buffers to DDS-XTYPES Mapping 27
6.1 User Types . . . 28
6.1.1 Protocol Buffers messages . . . 28
Mutability . . . 29
Nested messages . . . 29
6.1.2 Protocol Buffers enumerations . . . 30
Enumeration literals . . . 30
Nested Enumerations . . . 30
6.2 Primitive Types . . . 31
6.3 Collections . . . 31
6.3.1 Repeated fields . . . 32
6.3.2 Map fields . . . 32
6.4 Packages . . . 33
6.5 Imported Files . . . 34
6.6 Field Presence . . . 35
6.6.1 Required fields . . . 35
6.6.2 Optional fields . . . 36
6.6.3 Implicit fields . . . 36
6.6.4 Field presence examples . . . 37
6.7 OneOf Fields . . . 38
6.8 Field Groups . . . 38
6.9 Unsupported Features . . . 40

7 Release Notes 41
7.1 Supported Platforms . . . 41
7.2 Compatibility . . . 41
7.3 What's New in 7.7.0 . . . 41
7.3.1 Protocol Buffers Extension no longer experimental . . . 42
7.3.2 RTI Code Generator enables Protocol Buffers integration with DDS . . . 42
7.3.3 Improved tutorial build files for Protocol Buffers Extension . . . 42
7.3.4 Third-Party Software Changes . . . 42
7.4 What's Fixed in 7.7.0 . . . 42
7.4.1 [Major] Use of certain Protocol Buffers messages could result in a crash when publishing or subscribing to data . . . 43
7.4.2 [Minor] Using fields with camel case naming in .proto files was not working . . . 43
7.4.3 [Minor] Code Generator did not detect incompatible IDL file generated from .proto file . . . 43

8 Copyrights and Notices 44

Chapter 1

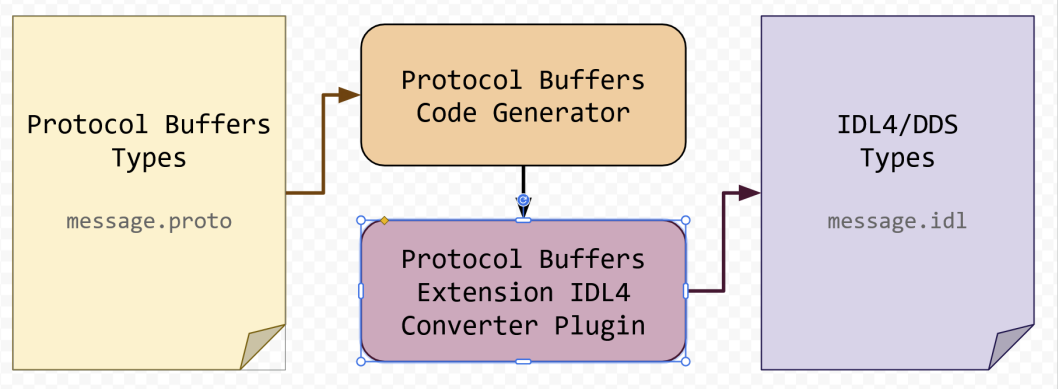
About Protocol Buffers Extension

RTI® Connext® Protocol Buffers Extension enables using Protocol Buffers™ data types in *Connext* applications.

1.1 Use Cases

1.1.1 Protocol buffers definition language support

Protocol Buffers Extension enables developers to use the Protocol Buffers (`protobuf`) message definition language to define the data types used by *Connext* applications. The toolchain automatically maps `protobuf` types to corresponding DDS-XTYPES types, which serve as the foundation for communication over DDS.

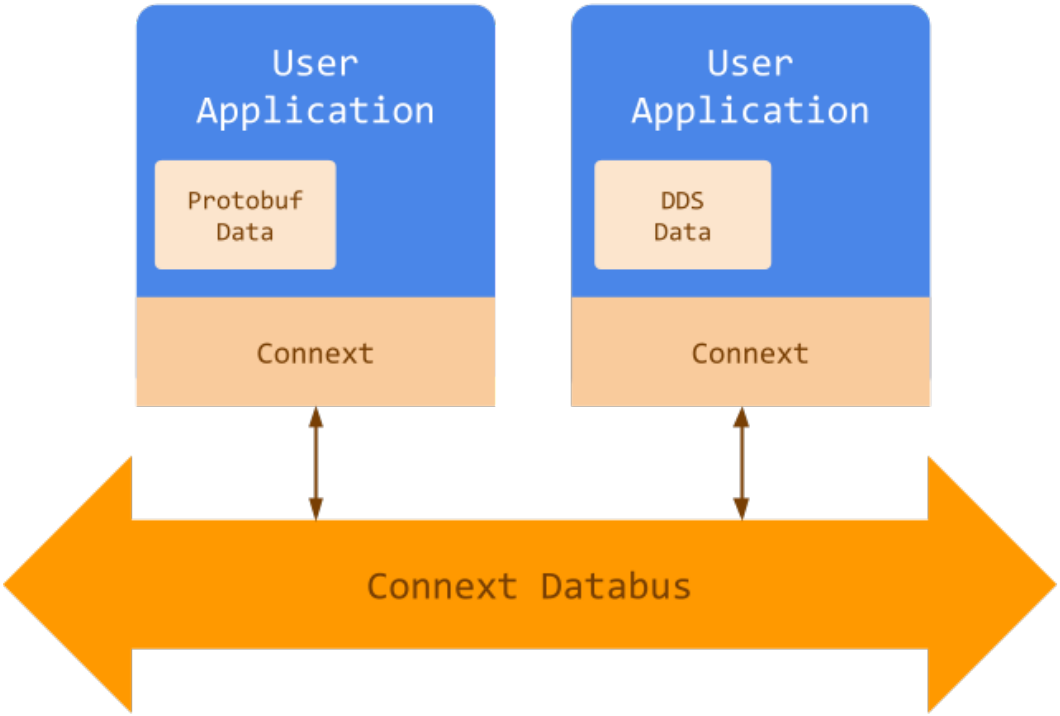


1.1.2 DDS-XTYPES type generation

Protocol Buffers Extension generates `.idl` files containing DDS-XTYPES types equivalent to the Protocol Buffers types defined in `.proto` files. This conversion enables seamless interoperability between *Connext* applications using Protocol Buffers data types and those using equivalent DDS-XTYPES, regardless of the programming language used.

1.1.3 Protocol Buffers C++ language binding

Protocol Buffers Extension allows developers to use the C++ message types generated by the protocol buffers compiler (`protoc`) in combination with *Connext*'s Modern C++ API. Applications can create *DataWriter* and *DataReader* endpoints to exchange objects of the Protocol Buffers types directly over DDS *Topics*. Existing components that use these types can be easily and efficiently integrated onto the DDS databus.



1.1.4 Integration with DDS

The data types defined using Protocol Buffers are first-class citizens in the *Connext* platform and can be used just like other DDS types. Full support is provided for features such as type discovery, type matching, type evolution, dynamic types, dynamic data, content filtering, and more.

The `.proto` files can be enriched with custom *Protocol Buffers Extension* options defined to control DDS-specific properties, such as indicating key members. Other options allow developers to take advantage of DDS-XTYPES features not natively available in Protocol Buffers. These features enhance the integration of *Protocol Buffers Extension* components with other *Connext* applications.

1.2 Components

Protocol Buffers Extension consists of two RTI components and a set of DDS-specific options:

- The *IDL4 Converter Plugin* converts `.proto` files into equivalent `.idl` files.
- The *C++ Code Generator Plugin* generates C++ source code from `.proto` files.
- *DDS Options for Protocol Buffers* types customizes the generated IDL4 types using DDS-specific options defined in `.proto` files.

1.3 System Requirements

To use *Protocol Buffers Extension* successfully, your system must meet the following requirements:

- *Connext Professional* is installed on a supported operating system (see *Supported Platforms*) and the RTI environment variables are set.
- The protocol buffer compiler (`protoc`) is installed and available in your system's `PATH`. *Protocol Buffers Extension* supports `protoc` versions 3.12.0 through 33.0.
- `<NDDSHOME>/bin` is included in your system's `PATH`.
- The RTI `<NDDSHOME>/resource/proto/omg/dds/descriptor.proto` file is available to `protoc`. This file is included in the *Connext* installation.

For more detail about these requirements, see *Before You Begin*.

Protocol Buffers Extension is pre-installed as part of *Connext Professional*. For instructions to install *Connext*, see the RTI Connext Professional Installation Guide.

1.4 Paths Mentioned in Documentation

This documentation refers to *Connext* installation and example paths using the following conventions:

- `<NDDSHOME>` refers to the installation directory for *Connext*. The default installation paths are:
  - macOS® systems: `/Applications/rti_connext_dds-<version>`
  - Linux® systems, non-root user: `/home/your user name/rti_connext_dds-<version>`
  - Linux systems, root user: `/opt/rti_connext_dds-<version>`
  - Windows® systems, user without Administrator privileges: `<your home directory>\rti_connext_dds-<version>`
  - Windows systems, user with Administrator privileges: `C:\Program Files\rti_connext_dds-<version>`

  Whenever you see `<NDDSHOME>` used in a path, replace it with your installation path. You may also see `$NDDSHOME` or `%NDDSHOME%`, which refers to an environment variable set to the installation path.

- <path to examples> refers to the directory for *Connext* example files. By default, examples are copied into your home directory the first time you run *RTI Launcher* or any script in <NDDSHOME>/bin. This document refers to the location of the copied examples as <path to examples>.

  The default paths to the examples are:

  – macOS systems: /Users/your user name/rti_workspace/<version>/examples

  – Linux systems: /home/your user name/rti_workspace/<version>/examples

  – Windows systems: your Windows documents folder\rti_workspace\<version>\examples. Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 7, the folder is C:\Users\your user name\Documents; on Windows Server 2003, the folder is C:\Documents and Settings\your user name\Documents.

  Wherever you see <path to examples>, replace it with the appropriate path.

Chapter 2

Protocol Buffers Extension Components

Protocol Buffers Extension consists of two RTI plugins for `protoc` and a set of DDS-specific options for Protocol Buffers types.

2.1 IDL4 Converter Plugin

The *Protocol Buffers Extension* IDL4 Converter Plugin (`idl4`) converts `.proto` files into equivalent `.idl` files using the `--idl4_out protoc` option. For example:

```
protoc --idl4_out=<outputdir> message.proto
```

- Example input (`message.proto`):

```
syntax = "proto3";

import "other.proto";

package myapp;

message MyMessage {
    int32 foo = 1;
    Other bar = 2;
}
```

- Example output (`message.idl`):

```
#ifndef myapp_message_proto_IDL4_
#define myapp_message_proto_IDL4_

#include "other.idl"

module myapp {
    @mutable
    struct MyMessage {
        @id(1) int32 foo;
```

(continues on next page)

(continued from previous page)

```
        @id(2) @optional Other bar;
    };
}; // module myapp

#endif // myapp_message_proto_IDL4_
```

See *Protocol Buffers to DDS-XTYPES Mapping* for a detailed description of the mapping between Protocol Buffers and IDL4 types.

2.2 C++ Code Generator Plugin

RTI Code Generator (`rtiddsgen`) includes the *Connext* C++ Code Generator Plugin (`connext-cpp`) to generate C++ source code from IDL files. This plugin works together with the Protocol Buffers built-in C++ code generator to create C++ classes for each data type in a `.proto` file. For example:

```
protoc --cpp_out=<outputdir> \
       --connext-cpp_out=<outputdir> \
       message.proto
```

Example output (`message.pb.h`, abridged):

```
#include "other.pb.h"

// Injected by RTI code generator
#include "rti/topic/cdr/ProtobufInterpreter.hpp"

namespace myapp {

class MyMessage final : public ::google::protobuf::Message {
public:
    MyMessage();
    ~MyMessage() override;

    bool has_foo() const;
    void clear_foo();
    int32_t foo() const;

    bool has_bar() const;
    void clear_bar();
    const ::myapp::Other& bar() const;
    ::myapp::Other* mutable_bar();
    void set_allocated_bar(::myapp::Other* bar);

    // Injected by RTI code generator
    template <typename MessageType>
    friend struct ::rti::topic::interpreter::detail::protobuf_message_access;

private:
    int32_t foo_;
```

(continues on next page)

(continued from previous page)

```
    ::myapp::Other* bar_;
}

} // namespace myapp
```

2.3 DDS Options for Protocol Buffers

Custom options are available to control IDL4 types derived from Protocol Buffers definitions. These DDS-specific options are defined using Protocol Buffers syntax and become available after importing RTI's supporting `omg/dds/descriptor.proto` file.

- Example input (`message.proto`, annotated):

```
syntax = "proto3";

import "other.proto";

import "omg/dds/descriptor.proto";

package myapp;

message MyMessage {
    option (.omg.dds.type) = {
        default_id: DDS_DEFAULT_ID,
        auto_id: HASH
    };

    int32 foo = 1 [ (.omg.dds.member).key = true ];
    Other bar = 2 [ (.omg.dds.member) = {
      optional: true,
      hash_id: "baz"
    }];
}
```

- Example output (`message.idl`, customized):

```
#ifndef myapp_message_proto_IDL4_
#define myapp_message_proto_IDL4_

#include "other.idl"

module myapp {
    @mutable
    @autoid(hash)
    struct MyMessage {
        @key int32 foo;
        @hash_id("baz") Other bar;
    };
}; // module myapp
```

(continues on next page)

(continued from previous page)

```
#endif // myapp_message_proto_IDL4_
```

See *DDS Options for Protocol Buffers* for detailed information about the RTI descriptor file.

2.4 Accessing the RTI Plugins

The Protocol Buffers plugins, `idl4` and `connext-cpp`, are located in `<NDDSHOME>/bin`. To make them accessible to `protoc`, add this directory to your `PATH` environment variable. For example:

```
export PATH=<NDDSHOME>/bin:$PATH
```

Alternatively, specify the explicit path of each plugin using the `protoc` command:

```
protoc --plugin=<NDDSHOME>/bin/protoc-gen-idl4 \
       --plugin=<NDDSHOME>/bin/protoc-gen-connext-cpp \
       ...
```

2.5 Limitations

The current versions of the RTI plugins for `protoc` have the following known limitations:

2.5.1 Platforms and compatibility

The RTI plugins are included in the installer for Linux and macOS. While these are the primary platforms where the plugins are installed, RTI has also tested the generated code's compatibility with QNX 8 builds. The generated code is expected to be compatible with Windows; however, this platform has not yet been formally tested by RTI.

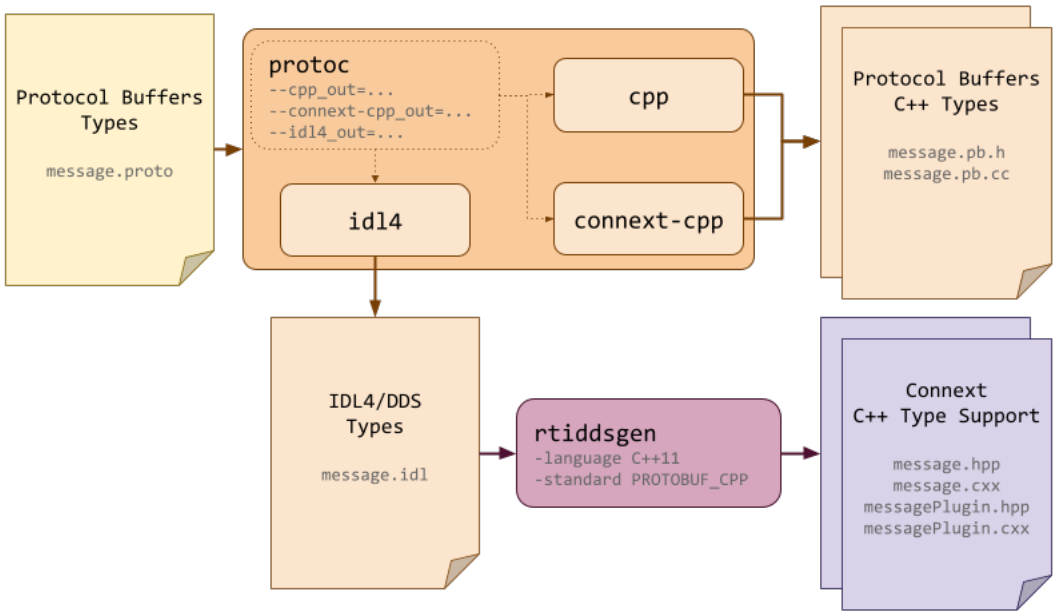
2.5.2 Recursive types

The IDL4 Converter Plugin does not support recursive types, such as a message that contains a field of its own type.

Chapter 3

Integrating Protocol Buffers with Connext

Protocol Buffers Extension allows your applications to use their existing Protocol Buffers messages with *Connext*. It uses a generative process to produce C++ source code from `.proto` files. The generated code enables *Connext* applications to use the Protocol Buffers types with DDS *Topics*.



There are two methods to generate the necessary source code from an input `.proto` file using *Protocol Buffers Extension*:

- *Two-phase generation*. Execute `protoc` using the RTI plugins and all necessary options, then run `rtiddsgen` on the resulting IDL file. We recommend this workflow because it simplifies the protoc configuration process.
- *Single-phase generation*. Invoke `rtiddsgen` using the `.proto` file as the direct input to generate all necessary files. This streamlined method is ideal for simple `.proto` files that do not contain includes or require specific `protoc` configuration options.

3.1 Two-Phase Code Generation

In phase one, the protocol buffers compiler (`protoc`) is invoked to:

- Convert each `.proto` file into an equivalent `.idl` file.
- Generate C++ source code for each Protocol Buffers type.

In phase two, all input `.proto` files are processed by three `protoc` plugins:

- The *Protocol Buffers Extension IDL4 Converter Plugin* (`idl4`) generates an `.idl` file for each input `.proto` file. The generated `.idl` files use the type system defined by the Extensible and Dynamic Topic Types for DDS specification (DDS-XTypes) from the Object Management Group (OMG).
- The built-in C++ code generator (`cpp`), part of the Protocol Buffers distribution, generates the C++ classes associated with each Protocol Buffers type.
- The *Protocol Buffers Extension C++ Code Generator Plugin* (`connext-cpp`) decorates the C++ code produced by `cpp` so that it can be efficiently integrated with *Connext*.

Phase two relies on *RTI Code Generator* (`rtiddsgen`) to process the `.idl` files generated by `protoc`. `rtiddsgen` produces the additional C++ source code necessary to use Protocol Buffers types as DDS *Topics* within *Connext* applications.

3.2 Single-Phase Code Generation

You can generate code from a `.proto` file in a single step using `rtiddsgen`. When a `.proto` file is used as input, `rtiddsgen` automatically invokes `protoc` internally using the *RTI plugins*.

Use the `-protocPath` option to specify the `protoc` executable location if it is not in your system path.

Use the `-protocOption` flag to pass specific parameters directly to `protoc`.

3.3 Code Generation Commands

Table 3.1 shows the two commands used to generate the necessary source code from an input `.proto` file using the *two-phase code generation method*. It lists the input files, the commands used to generate the output, and the resulting files.

Table 3.1: Protocol Buffers Extensions Code Generation Commands

Command	Input	Output

`protoc --idl4_out=<outputdir> \`
`  --cpp_out=<outputdir> \`
`  --connext-cpp_out=<outputdir> \`
`  <input>`
	`message.proto`	• `message.idl`
		• `message.pb.h`
		• `message.pb.cc`

`rtiddsgen -language C++11 \`
`  -standard PROTOBUF_CPP \`
`  <input>`
	`message.idl`	• `message.hpp`
		• `message.cxx`
		• `messagePlugin.hpp`
		• `messagePlugin.cxx`

In the example `protoc` command above, all plugins are run at once. This is not a strict requirement; each `protoc` plugin can be invoked independently. However, it is recommended to run `protoc` with all plugins in a single command because other options (for example, `#include` paths) should be kept consistent across different `protoc` calls.

Important: If the plugins are called independently, RTI's `connext-cpp` plugin must be run with (or after) the built-in `cpp` plugin.

If an input `.proto` file imports the RTI *omg/dds/descriptor.proto* file, the directory containing the file must be specified using the `-I` option in the `protoc` command line. For example:

```
protoc --idl4_out=<outputdir> \
       --cpp_out=<outputdir> \
       --connext-cpp_out=<outputdir> \
       -I <NDDSHOME>/resource/proto \
       <input>
```

When using the *single-phase code generation method*, a similar `rtiddsgen` command is used, but the input file is the original `.proto` file instead of the generated IDL file. The following example generates source code directly from `example.proto`:

```
rtiddsgen -language C++11 \
          -standard PROTOBUF_CPP
          example.proto \
          -protocPath <path to protoc installation> \
          -protocOption=--idl4_out=<outputdir>
```

3.4 Generating Code for Multiple Files

Most projects use multiple `.proto` files with dependencies between them. These dependencies are typically expressed using `import` statements in the `.proto` files. For these projects, you need to make sure the code generation process respects the dependency order.

Note: If your project has multiple `.proto` files with dependencies, it is recommended to use the *two-phase code generation method*. This method simplifies the management of dependencies between files.

3.4.1 Integrating .proto files with dependencies

As an example, consider the following file structure:

```
.src/
├── a.proto
├── foo
│   └── b.proto
└── bar
    └── c.proto
```

Here, `a.proto` imports `b.proto` and `c.proto` using `import` statements:

```
import "common/b.proto";
import "utils/c.proto";
```

After processing `a.proto` with the IDL4 Converter Plugin, the generated `a.idl` would then have corresponding `#include` statements:

```
#include "foo/b.idl"
#include "bar/c.idl"
```

To ensure the generated `#include` statements are valid:

- all imported `.proto` files must be converted to `.idl` files before `a.idl` can be processed by `rtiddsgen`.
- the generated `.idl` files must be placed in a similar directory structure. For example:

```
.build/       # generated by:
├── a.idl     # ??? protoc --idl4_out=...
├── foo
│   └── b.idl
└── bar
    └── c.idl
```

If a similar ordering requirement existed between `b.proto` and `c.proto`, `b.idl` would have to be generated before `c.idl` can be processed. If the two files were independent, they could be processed simultaneously for faster processing in parallel.

In this example, code generation can be executed as follows using the *two-phase code generation method*:

1. First, process all `.proto` files with `protoc`.

   ```
   protoc --idl4_out=build \
          --cpp_out=build \
          --connext-cpp_out=build \
          -I src \
          src/a.proto \
          src/foo/b.proto \
          src/bar/c.proto
   ```

   This command compiles multiple related `.proto` files, generating both standard C++ code and *Connext*-specific integration code, with all output going to a single build directory.

2. Then, process all `.idl` files with `rtiddsgen`.

   ```
   rtiddsgen -language C++11 \
             -standard PROTOBUF_CPP \
             -I build \
             build/a.idl \
             build/foo/b.idl \
             build/bar/c.idl
   ```

   This command generates C++ code from the related IDL files that were converted from Protocol Buffers types in step 1. It processes IDL files organized in different directories and resolves cross-file dependencies using the `#include` path.

When code generation is complete, the following C++ files are generated and made available for compilation in the output directory:

```
.build/                 # generated by:
├── a.pb.cc             # ??? protoc --cpp_out=... --connext-cpp_out=...
├── a.pb.h              # ??? protoc --cpp_out=... --connext-cpp_out=...
├── a.cxx               # ??? rtiddsgen -language C++11 -standard PROTOBUF_CPP
├── a.hpp               # ??? rtiddsgen -language C++11 -standard PROTOBUF_CPP
├── aPlugin.cxx         # ??? rtiddsgen -language C++11 -standard PROTOBUF_CPP
├── aPlugin.hpp         # ??? rtiddsgen -language C++11 -standard PROTOBUF_CPP
├── foo
│   ├── b.pb.cc
│   ├── b.pb.h
│   ├── b.cxx
│   ├── b.hpp
│   ├── bPlugin.cxx
│   └── bPlugin.hpp
└── bar
    ├── c.pb.cc
    ├── c.pb.h
    ├── c.cxx
    ├── c.hpp
    ├── cPlugin.cxx
    └── cPlugin.hpp
```

Warning: Both `protoc` and `rtiddsgen` can process multiple input files from a single command line. However, when automating code generation, it is recommended to use a single input file for each invocation of these plugins.

This approach ensures that the length of the command-line invocation does not exceed system command-line length limits, which is a risk as the number of `.proto` files used by a project grows.

3.4.2 Managing dependencies

Managing these precise dependencies becomes increasingly complex as the number of input files grows. This complexity typically introduces additional maintenance overhead and makes a fully parallelized solution difficult to implement.

It is recommended to forego a bit of parallelization efficiency in favor of simpler build rules. That's why we recommend using the *two-phase code generation* method when working with multiple `.proto` files with dependencies.

Chapter 4

Tutorial: Generating Code from Protocol Buffers Messages

This chapter uses an RTI example, `protobuf-sdk`, to demonstrate how *Protocol Buffers Extension* enables *Connext* applications to use Protocol Buffers message types as DDS *Topics*. It details the workflow for defining, converting, and using Protocol Buffers message types in *Connext* applications. The chapter concludes with instructions for building and running the example applications.

The `protobuf-sdk` example files reference the AddressBook example included in the Protocol Buffers documentation, which describes a simple data model composed of a single `.proto` file.

The complete source code for this example is available in your *Connext* installation at `<path to examples>/connext_dds/c++/protobuf-sdk`. See *Paths Mentioned in Documentation* for the path to the example.

4.1 Before You Begin

- Set up the *Connext* environment variables required to run *RTI Code Generator* (`rtiddsgen`) using the `rtisetenv` script for your architecture. For example:

```
source <NDDSHOME>/resource/scripts/rtisetenv_x64Linux4gcc8.5.0.bash
```

  This script sets up the environment variables for running `rtiddsgen`. For more information, see Setting up Environment Variables in the *Connext Getting Started Guide*.

- Install the protocol buffer compiler (`protoc`), then add the installation directory to your `PATH` environment variable. For example:

```
export PATH=<protoc install dir>/bin:$PATH
```

  See the Protocol Buffers documentation for installation instructions.

- Add `<NDDSHOME>/bin` to your `PATH` environment variable. For example:

```
export PATH=<NDDSHOME>/bin:$PATH
```

The `PATH` variable must include this directory to access the *Protocol Buffers Extension* plugins.

- Import RTI's `<NDDSHOME>/include/omg/dds/descriptor.proto` file into your `.proto` files to enable the *DDS Options for Protocol Buffers*.

4.2 Define Protocol Buffers Message Types

To integrate Protocol Buffers with *Connext*, first define your Protocol Buffers message types in a `.proto` schema file. This step is illustrated in the `addressbook.proto` file in the `protobuf-sdk` example.

Below is an excerpt of the `addressbook.proto` file, which defines the `Person` and `AddressBook` message types.

```
// addressbook.proto

syntax = "proto3";
package tutorial;

import "google/protobuf/timestamp.proto";

message Person {
  string name = 1;
  int32 id = 2;  // Unique ID number for this person.
  string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phones = 4;

  google.protobuf.Timestamp last_updated = 5;
}

message AddressBook {
  repeated Person people = 1;
}
```

Most Protocol Buffers data models can be used without modifications because *Protocol Buffers Extension* provides *detailed mapping* between Protocol Buffers and IDL4 definitions.

4.3 Convert Protocol Buffers Data

Next, convert the Protocol Buffers types into types for the programming language used by the application code.

To convert using two phases:

- Use `protoc` to convert the `.proto` file to an equivalent `.idl` file.
- Use *RTI Code Generator* (`rtiddsgen`) to generate the remaining source code needed to use the Protocol Buffers types as DDS *Topics* in *Connext*.

For example:

```
# First run the protocol buffers compiler with the RTI plugins
protoc --idl4_out=build \
       --cpp_out=build \
       --connext-cpp_out=build \
       -I . \
       addressbook.proto \
       google/protobuf/timestamp.proto

# Then use the Connext compiler to generate additional code from the␣
↪generated IDL
rtiddsgen -language C++11 \
          -standard PROTOBUF_CPP \
          -I build \
          build/addressbook.idl \
          build/google/protobuf/timestamp.idl
```

To convert using a single phase, use `rtiddsgen` to convert the `.proto` file and generate code in one step. For example:

```
# Run rtiddsgen to convert the .proto file and generate code in one step
```

4.4 Exchange Protocol Buffers Data Over DDS

After converting the Protocol Buffers types, applications can use the generated type support code to exchange Protocol Buffers data directly over DDS *Topics*.

Note: *Protocol Buffers Extension* supports the Protocol Buffers C++ language binding to exchange `protobuf` data.

The `protobuf-sdk` example includes the `addressbook_publisher.cxx` and `addressbook_subscriber.cxx` files to illustrate how to publish and subscribe to a DDS *Topic* using the converted Protocol Buffers types. These files are excerpted below:

```
// addressbook_publisher.cxx
```

(continues on next page)

(continued from previous page)

```
// Include the standard DDS C++ API.
#include <dds/dds.hpp>

// Include the generated header files for the Protocol Buffers types.
// addressbook.hpp is generated from addressbook.idl by rtiddsgen.
// It includes addressbook.pb.h, which is generated by protoc
// from addressbook.proto.
#include "addressbook.hpp"

// Instantiate DDS entities to publish an AddressBook topic on␣
↪domain 0.
dds::domain::DomainParticipant participant(0);
dds::topic::Topic<tutorial::AddressBook> topic(participant, "Example␣
↪tutorial_AddressBook");
dds::pub::Publisher publisher(participant);
dds::pub::DataWriter<tutorial::AddressBook> writer(publisher, topic);

// Create an instance of the AddressBook message and populate it.
tutorial::AddressBook data;

auto person = data.add_people();
person->set_name("John Doe");
person->set_id(1);
person->set_email("johndoe@example.org");

auto phone = person->add_phones();
phone->set_number("867-5309");
phone->set_type(tutorial::Person_PhoneType_HOME);

// Write the AddressBook message using the typed DataWriter.
writer.write(data);
```

```
// addressbook_subscriber.cxx

// Include the standard DDS C++ API.
#include <dds/dds.hpp>

// Include the generated header files for the Protocol Buffers types.
#include "addressbook.hpp"

// Instantiate DDS entities to subscribe to an AddressBook topic on␣
↪domain 0.
dds::domain::DomainParticipant participant(0);
dds::topic::Topic<tutorial::AddressBook> topic(participant, "Example␣
↪tutorial_AddressBook");
dds::sub::Subscriber subscriber(participant);
dds::sub::DataReader<tutorial::AddressBook> reader(subscriber,␣
↪topic);

// Take samples from the typed DataReader.
dds::sub::LoanedSamples<tutorial::AddressBook> samples = reader.
```

(continues on next page)

(continued from previous page)

```
↪take();

for (const auto &sample : samples) {
    if (sample.info().valid()) {
        // Samples are instances of the Protocol Buffers AddressBook␣
↪type.
        tutorial::AddressBook &data = sample.data();
        std::cout << data.DebugString() << std::endl;
    }
}
```

4.5 Communicate with Other DDS Applications

On running the `protoc` command, Protocol Buffers message types are converted into equivalent DDS-XTypes, which can then be used to communicate with other DDS applications. These DDS-XTypes are defined in the `.idl` files generated by the *Protocol Buffers Extension IDL4 Converter Plugin*.

In the `protobuf-sdk` example, the `addressbook.proto` file is converted into the `addressbook.idl` file, which defines the DDS-XTypes for the `AddressBook` and `Person` message types. The generated `addressbook.idl` file contains the following definitions:

```
// addressbook.idl

// ------------------------------------------------------------------------
↪--
// WARNING ----------------------------------------------------------------
↪--
// This file was automatically generated by RTI's IDL4 plugin for protoc,
// from Protobuf source file: addressbook.proto
// Do not edit this file manually. ALL CHANGES WILL BE LOST!
// ------------------------------------------------------------------------
↪--

#ifndef tutorial_addressbook_proto_IDL4_
#define tutorial_addressbook_proto_IDL4_
#include "google/protobuf/timestamp.idl"

module tutorial {
  @containing_type("Person")
  enum Person_PhoneType {
    @value(0) @default_literal Person_PhoneType_MOBILE,
    @value(1) Person_PhoneType_HOME,
    @value(2) Person_PhoneType_WORK
  }; // enum Person_PhoneType

  struct Person_PhoneNumber;

  struct Person;
```

(continues on next page)

(continued from previous page)

```
  struct AddressBook;

  @nested
  @containing_type("Person")
  @mutable
  struct Person_PhoneNumber {
    @id(1) @field_presence(implicit) string number;
    @id(2) @field_presence(implicit) ::tutorial::Person_PhoneType type;
  }; // struct Person_PhoneNumber

  @mutable
  struct Person {
    @id(1) @field_presence(implicit) string name;
    @id(2) @field_presence(implicit) int32 id;
    @id(3) @field_presence(implicit) string email;
    @id(4) sequence<::tutorial::Person_PhoneNumber> phones;
    @id(5) @optional ::google::protobuf::Timestamp last_updated;
  }; // struct Person

  @mutable
  struct AddressBook {
    @id(1) sequence<::tutorial::Person> people;
  }; // struct AddressBook

}; // tutorial
#endif // tutorial_addressbook_proto_IDL4_
```

The generated IDL4 types can be used in other applications that use any of the programming languages supported by *Connext* (see the RTI Connext Getting Started Guide). For example, to use the Modern C++ API with the latest language binding defined by the IDL4 specification:

```
mkdir build-dds

rtiddsgen -language C++11 \
          -standard IDL4_CPP \
          -I build \
          -d build-dds \
          build/addressbook.idl

mkdir -p build-dds/google/protobuf

rtiddsgen -language C++11 \
          -standard IDL4_CPP \
          -I build \
          -d build-dds/google/protobuf \
          build/google/protobuf/timestamp.idl
```

4.6 Build and Run the Example

After completing the above steps, you can build and run the `protobuf-sdk` example applications.

1. Set the *Connext* environment variables:

```
source <NDDSHOME>/resource/scripts/rtisetenv_<architecture>.bash
```

   Protocol Buffers Extension is included with *Connext*, so no additional *Connext* configuration is needed.

2. Build the example applications:

```
// Create a build directory and enter it
mkdir build
cd build

// Generate the CMake build files
cmake <NDDSHOME>/resource/template/rti_workspace/examples/connext_dds/
↪c++11/protobuf

// Build the example applications
cmake --build .
```

Note: If the Google `protobuf` libraries are not installed on your system, CMake may not be able to find all the required dependencies. Use the `-DCMAKE_PREFIX_PATH=<Path to protobuf build>/lib64/cmake` and `-DProtobuf_USE_CONFIG=ON` options when generating the CMake build files. These options will help CMake configure the dependencies and compile correctly.

---

3. Publish data samples to a DDS *Topic* using the Protocol Buffers types using one of the supported language bindings:

   a. Start a publisher using the Protocol Buffers C++ language binding:

```
./addressbook_publisher
```

   b. Start a publisher using the *Connext* C++ language binding:

```
./addressbook_publisher_dds
```

4. Subscribe to the DDS *Topic* using the Protocol Buffers types:

   a. Start a subscriber using the Protocol Buffers C++ language binding:

```
./addressbook_subscriber
```

   b. Start a subscriber using the *Connext* C++ language binding:

```
./addressbook_subscriber_dds
```

5. Start *RTI Spy* (`rtiddsspy`) to subscribe to all data in the DDS domain:

```
rtiddsspy -printSample
```

This command displays all data samples exchanged in the DDS domain, including the Protocol Buffers published data samples.

For information on using *Spy* for debugging and verifying the data exchanged in DDS domains, see the RTI Spy User's Manual.

Chapter 5

DDS Options for Protocol Buffers

The `omg/dds/descriptor.proto` file is a Protocol Buffers extension provided by RTI. This file acts as a bridge between Protocol Buffers and DDS, enabling you to use DDS-specific features in your Protocol Buffers definitions.

`omg/dds/descriptor.proto` defines rules that are expressed as custom options in your Protocol Buffers message definitions. When the definitions are compiled to IDL4, the DDS options are applied to the generated IDL4 code as specific annotations to the corresponding IDL4 types.

5.1 Message Options

Table 5.1: Message Options (`.omg.dds.type`)

Option	Description	Value	IDL4 Output
`.name`	Specifies a custom name to identify types used on a DDS *Topic*.	`"MyCustomName"` (non-empty string literal)	`@type_name("MyCustomName")`
`.extensibility`	Controls the extensibility annotation of the corresponding IDL4 `struct`. The default value is `MUTABLE`.	`MUTABLE`	`@mutable`
		`FINAL`	`@final`
		`APPENDABLE`	`@appendable`
`.default_id`	Specifies the default policy used to control the default `@id` annotation emitted for each member of an IDL4 `struct`.	`PROTOBUF_DEFAULT_ID`	`@id(N)` (on members)
		`DDS_DEFAULT_ID`	N/A (use default DDS id for each member)
`.auto_id`	Controls the `@autoid` annotation for the IDL4 `struct`.	`NO_AUTO_ID`	N/A (no annotation)
		`SEQUENTIAL`	`@autoid(sequential)`
		`HASH`	`@autoid(hash)`

5.2 Field Options

Table 5.2: Field Options (`.omg.dds.member`)

Option	Description	Value	IDL4 Output
`.key`	Specifies whether the associated member is part of the DDS key.	`true`	`@key`
		`false` (default)	N/A (no annotation)
`.optional`	Controls whether the associated member is annotated as `@optional`. If specified, this option takes precedence over the default annotation derived from the *field presence* setting.	`true`	`@optional`
		`false`	N/A (no annotation)
`.default_id`	Controls the default `@id` annotation emitted for the associated member. If specified, this option takes precedence over the default policy set at the message level in the option `.omg.dds.type.default_id`.	See `.omg.dds.type.default_id`.	
`.id`	Assigns and emits an explicit `@id` annotation for the associated member.	`N` (32-bit unsigned integer)	`@id(N)`
`.hash_id`	Generates the `@hashid` annotation for the associated member with the specified source string.	`"My String"` (non-empty string literal)	`@hashid("My String")`

5.3 RTI Protocol Buffers Descriptor File

The RTI Protocol Buffers descriptor file included with *Protocol Buffers Extension*, `omg/dds/descriptor.proto`, allows you to mark your Protobuf definitions with specific DDS instructions. The file is located at `<NDDSHOME>/include/omg/dds/descriptor.proto`. The complete file content is shown below.

Listing 5.1: omg/dds/descriptor.proto

```
1  syntax = "proto2";
2  import "google/protobuf/descriptor.proto";
3
4  package omg.dds;
5
6  enum ExtensibilityKind {
7    MUTABLE = 0; // "Mutable" extensibility, members can be reordered,  added,␣
   ↪or removed.
8    APPENDABLE = 1; // "Appendable" extensibility, members can be added or␣
```

(continues on next page)

(continued from previous page)

```
↪removed from the end.
  FINAL = 2; // "Final" extensibility, member cannot be added, removed or␣
↪reordered.
}

enum DefaultIdKind {
  PROTOBUF_DEFAULT_ID = 0; // Derive an @id annotation for each DDS member␣
↪from the corresponding protobuf field number.
  DDS_DEFAULT_ID = 1; // Do not emit @id annotations, unless explicitly␣
↪provided through the "omg.dds.member.id" option.
}

enum AutoIdKind {
  NO_AUTO_ID = 0; // Do not emit an @autoid annotation
  SEQUENTIAL = 1; // Emit @autoid(sequential) to use sequential IDs starting␣
↪from 0
  HASH = 2; // Emit @autoid(hash), to use a hash of the field name to␣
↪generate IDs
}

message TypeAnnotation {
  optional string name = 1;
  optional ExtensibilityKind extensibility = 2;
  optional DefaultIdKind default_id = 3;
  optional AutoIdKind auto_id = 4;
}

extend google.protobuf.MessageOptions {
  optional TypeAnnotation type = 7400;
}

message MemberAnnotation {
  optional bool key = 1;
  optional bool filterable = 2;
  optional bool optional = 3;
  optional DefaultIdKind default_id = 4;
  optional uint32 id = 5;
  optional string hash_id = 6;
}

extend google.protobuf.FieldOptions {
  optional MemberAnnotation member = 7400;
}
```

Chapter 6

Protocol Buffers to DDS-XTYPES Mapping

This chapter details how Protocol Buffers types are mapped to DDS-XTYPES via the *Protocol Buffers Extension* IDL4 Converter Plugin.

For every `.proto` file processed, the plugin generates an `.idl` representation. The resulting output contains DDS-XTYPES definitions that are functionally equivalent to the input Protocol Buffers types.

The generated IDL4 files contain type definitions for all Protocol Buffers messages, enumerations, and other constructs, which are mapped to IDL4 constructs.

A C preprocessor guard protects the definitions in each generated file, allowing them to be included by multiple files without causing redefinition errors.

Table 6.1: Protocol Buffers to IDL4 Mapping for .proto files

Protocol Buffers	DDS-XTYPES/IDL4
message.proto	message.idl

`message MyMessage {`
 	
`}`
	`#ifndef message_proto_IDL4_`
	`#define message_proto_IDL4_`
	
	`@mutable`
	`struct MyMessage {`
	
	`};`
	
	`#endif // message_proto_IDL4_`

myapp/message.proto	myapp/message.idl

`package myapp;`
 	
`message MyMessage {`
 	
`}`
	`#ifndef myapp_message_proto_IDL4_`
	`#define myapp_message_proto_IDL4_`
	
	`module myapp {`
	
	`  @mutable`
	`  struct MyMessage {`
	
	`  };`
	
	`}; // module myapp`
	
	`#endif // message_proto_IDL4_`

6.1 User Types

Protocol Buffers supports two user-defined types:

- *Protocol Buffers messages* define the schema for structured data fields.
- *Protocol Buffers enumerations* define the discrete set of values, or literals, allowed for a field.

6.1.1 Protocol Buffers messages

Protocol Buffers messages are mapped to IDL4 structs.

Mutability

Each mapped Protocol Buffers message uses the `@mutable` extensibility. This setting enables type modifications without breaking communication and supports declaration of `@optional` members. As a result, the XCDR serialization of these types behaves similarly to Protocol Buffers serialization behavior.

To change the default extensibility setting for the mapped IDL4 types, use the `.omg.dds.type.extensibility` DDS option.

Table 6.2: Protocol Buffers to IDL4 Mapping for Messages

Protocol Buffers	DDS-XTYPES/IDL4
`message MyMessage { }`	`@mutable` `struct MyMessage { };`

Nested messages

IDL4 does not support nested type declarations. Therefore, Protocol Buffers messages declared inside other messages are mapped to top-level IDL4 types using a specific naming convention and dedicated annotations.

The name of the IDL4 type associated with a nested message encodes the names of all containing types, concatenated by underscores. This naming convention matches that followed by Protocol Buffers when deriving the names of types associated with nested messages in several programming languages.

Nested types are annotated with `@nested` to prevent them from being used as top-level types in DDS *Topics*. This convention better matches the visibility of nested types in Protocol Buffers, which is limited to the `.proto` file where they are declared.

Nested types are also annotated with `@containing_type` to further encode their relationship to the containing type.

Table 6.3: Protocol Buffers to IDL4 Mapping for Nested Messages

Protocol Buffers	DDS-XTYPES/IDL4

`message MyMessage {`
`  message NestedMessage { }`
`}`
	`@nested`
	`@containing_type("MyMessage")`
	`@mutable`
	`struct MyMessage_NestedMessage {`
	
	`};`

6.1.2 Protocol Buffers enumerations

Protocol Buffers enumerations are mapped to IDL4 `enum` types.

Enumeration literals

The value of each literal is propagated to IDL4 using the `@value` annotation. The default literal, which Protocol Buffers typically defines as the first value, is always marked in IDL4 with `@default_literal` for greater robustness. The literals are exposed as top-level symbols without any prefix, unless the enumeration is nested inside a message.

Table 6.4: Protocol Buffers to IDL4 Mapping for Enumerations

Protocol Buffers	DDS-XTYPES/IDL4

`enum MyEnum {`
`  HELLO = 0;`
`  WORLD = 1;`
`}`
	`enum MyEnum {`
	`  @value(0)`
	`  @default_literal`
	`  HELLO,`
	`  @value(1)`
	`  WORLD`
	`};`

Nested Enumerations

Nested enumerations follow the same naming conventions as *nested messages*. The only difference is that enumeration literals are also prefixed with the nested type's name.

Consistently with message handling, this naming convention mirrors how Protocol Buffers transforms nested enumeration names when generating code for languages such as C++.

Table 6.5: Protocol Buffers to IDL4 Mapping for Nested Enumerations

Protocol Buffers	DDS-XTYPES/IDL4

`message MyMessage {`
`  enum NestedEnum {`
`  HELLO = 0;`
`  WORLD = 1;`
`  }`
`}`
	`@containing_type("MyMessage")`
	`enum MyMessage_NestedEnum {`
	`  @value(0)`
	`  @default_literal`
	`  MyMessage_NestedEnum_HELLO,`
	`  @value(1)`
	`  MyMessage_NestedEnum_WORLD`
	`};`

6.2 Primitive Types

Each Protocol Buffers primitive (or scalar) type is mapped to the appropriate IDL4 counterpart. This type mapping results in very similar representations in both Protocol Buffers and the other language bindings supported by *Connext*.

The Protocol Buffers `bytes` type is the only notable primitive mapping exception. `bytes` is mapped to a sequence of octets in IDL4 (`sequence<octet>`) rather than a primitive type. Protocol Buffers typically represents *bytes* as a string; this structural difference may cause some inconsistencies when accessing these values from different language bindings. In C++ the `bytes` type is mapped to:

- `std::string` when using the Protocol Buffers C++ API.
- `std::vector<uint8_t>` when using the standard DDS C++ API.

Table 6.6: Protocol Buffers to IDL4 Mapping for Primitive Types

Protocol Buffers	DDS-XTYPES/IDL4
`double`	`double`
`float`	`float`
`int32`	`int32`
`int64`	`int64`
`uint32`	`uint32`
`uint64`	`uint64`
`sint32`	`int32`
`sint64`	`int64`
`fixed32`	`uint32`
`fixed64`	`uint64`
`sfixed32`	`int32`
`sfixed64`	`int64`
`bool`	`boolean`
`string`	`string`
`bytes`	`sequence<octet>`

6.3 Collections

Protocol Buffers offers two ways to represent collections of data:

- *Repeated fields* represent a sequence of values
- *Map fields* represent key-value pairs

6.3.1 Repeated fields

Protocol Buffers repeated fields are mapped to unbounded IDL4 sequences. Use the `.omg.dds.member.optional` DDS option to mark the corresponding IDL4 member as @optional.

When a message includes a repeated `bytes` field, the generated IDL4 defines an alias for `sequence<octet>` scoped within the message type. This alias is required for the definition of the associated `struct` member because IDL4 does not support sequences of anonymous sequences. For example, syntax like `sequence<sequence<T>>` is not valid IDL4.

Table 6.7: Protocol Buffers to IDL4 Mapping for Repeated Fields

Protocol Buffers	DDS-XTYPES/IDL4
`repeated TYPE my_field ... ;`	`sequence< TYPE > my_field;`

`message MyMessage {`
`  repeated bytes my_field = 1;`
`}`
	`typedef sequence<octet> MyMessage_OctetSeq;`
	
	`struct MyMessage {`
	`  @id(1)`
	`  sequence<MyMessage_OctetSeq> my_field;`
	`};`

6.3.2 Map fields

Protocol Buffers map fields are mapped to IDL4 sequences of `key` and `value` pairs. These pairs are represented using an automatically generated MapPair `struct` type.

A MapPair `struct` is defined for each pair of key and value types used by map fields in a Protocol Buffers message using the `<message>_MapPair_<key-type>_<value-type>` naming convention. The same type is reused by all map fields within a message that share the same key-value definitions.

MapPair `struct` types are marked with `@map_pair`, to indicate their use, and with `@containing_type`, to trace their relationship to the containing message type. The `@nested` annotation is also used to prevent their use as top-level types for DDS *Topics*.

Members associated with a map field are annotated with `@map` to make sure they are represented as map constructs in the language bindings that support them.

Table 6.8: Protocol Buffers to IDL4 Mapping for Map Fields

Protocol Buffers	DDS-XTYPES/IDL4

`message MyMessage {`
`  map<K, V> my_field = 1;`
`}`
	`@nested`
	`@final`
	`@map_pair`
	`@containing_type("MyMessage")`
	`struct MyMessage_MapPair_K_V {`
	`  K key;`
	`  V value;`
	`};`
	
	`struct MyMessage {`
	`  @id(1)`
	`  @map`
	`  sequence< MyMessage_MapPair_K_V >`
	`  my_field;`
	`};`

Warning: The `@map` and `@map_pair` annotations are a temporary solution to allow the Protocol Buffers C++ language binding to support map fields. The annotations are ignored by language bindings where maps must be accessed as linear collections rather than as associative containers.

Map field representations will be updated to use the native IDL4 `map` type in a future *Connext* release.

6.4 Packages

Each Protocol Buffers package is mapped to an IDL4 `module`.

The name of the Protocol Buffers package is split into segments by the dot (`.`) character, and each segment is mapped to a nested IDL4 module. To convert qualified Protocol Buffers `package` names to IDL4 `module` names, replace dots (`.`) with double colons (`::`).

The name of the Protocol Buffers package is also included as part of the C preprocessor guard used to wrap all definitions in the generated IDL4 file.

Table 6.9: Protocol Buffers to IDL4 Mapping for Packages

Protocol Buffers	DDS-XTYPES/IDL4

`package my.messages.package;`
 	
`// Type definitions...`
	`#ifndef my_messages_package_proto_IDL4_`
	`#define my_messages_package_proto_IDL4_`
	
	`module my {`
	`module messages {`
	`module package {`
	
	`// Type definitions...`
	
	`}; // module package`
	`}; // module messages`
	`}; // module my`
	
	`#endif // my_messages_package_proto_IDL4_`

6.5 Imported Files

Each `import` statement in a Protocol Buffers file is converted to an `#include` statement in the generated IDL4 file. The `#include` uses the same path, but with the `.idl` extension instead of `.proto`.

All imported `.proto` files must be independently converted to `.idl` for the generated `#include` statements to be valid. Any builtin Protocol Buffers file imported by a `.proto` file (for example, `google/protobuf/timestamp/proto`) also needs to be explicitly converted to IDL4.

Typically, Protocol Buffers applications do not need to generate artifacts from these files, because the associated code is already part of the Protocol Buffers binary distribution.

Protocol Buffers Extension does not include a pre-generated IDL4 version of the built-in Protocol Buffers files. This approach allows developers to precisely match the interfaces included in their preferred version of Protocol Buffers.

Table 6.10: Protocol Buffers to IDL4 Mapping for Imported Files

Protocol Buffers	DDS-XTYPES/IDL4
`import "another.proto";`	`#include "another.idl"`

6.6 Field Presence

Based on whether a field has `implicit presence` or `explicit presence`, `protoc` generates a different API for the message type:

- Fields with `explicit presence` have an associated `has` method, which can be used to check whether the field has been set.
- Fields with `implicit presence` do not have an associated `has` method, and their value can only be checked against the default value for their type (or the default value assigned via options).

Protocol Buffers Extension relies on this classification to determine whether to map a field to an `@optional` member in IDL4.

Every field with a `has` method (that is, with `explicit presence`) not marked as `required` is mapped to an `@optional` member in IDL4.

The behavior of fields with `explicit presence` depends on the Protocol Buffers language revision used in the `.proto` file. For details about field presence, see the Protocol Buffers documentation.

To override the default mapping of the `@optional` annotation, use the `.omg.dds.member.optional` DDS option.

6.6.1 Required fields

The `proto2` syntax allows the declaration of `required` fields. This functionality is also available when using the `edition = "2023"` syntax via the `features.field_presence = LEGACY_REQUIRED` option.

Because a required field must be present in every serialized instance of a message, *Protocol Buffers Extension* maps required fields to IDL4 `struct` members without the `@optional` annotation.

Table 6.11: Protocol Buffers to IDL4 Mapping for Required Fields

File Syntax	Protocol Buffers	DDS-XTYPES/IDL4
`proto2`	`required TYPE my_field ... ;`	`TYPE my_field;`
`proto3`	N/A	N/A
`edition = "2023"`	`TYPE my_field ... [ features.field_presence = LEGACY_REQUIRED ];`	`TYPE my_field;`

6.6.2 Optional fields

All Protocol Buffers syntaxes allow the declaration of `optional` fields, which represent singular values that may be omitted from a serialized message.

`proto2` and `proto3` provide the `optional` keyword to denote such fields, while `edition = "2023"` automatically marks every singular field as `optional`. With `edition = "2023"`, fields can be further controlled using the `features.field_presence = EXPLICIT` option.

Fields with `optional` semantics are mapped to IDL4 members with the `@optional` annotation.

Table 6.12: Protocol Buffers to IDL4 Mapping for Optional Fields

File Syntax	Protocol Buffers	DDS-XTYPES/IDL4
`proto2`	`optional TYPE my_field ... ;`	`@optional` `TYPE my_field;`

`proto3`	`optional TYPE my_field ... ;`
	
	`MESSAGE_TYPE my_msg_field;`
		`@optional`
		`TYPE my_field;`
		
		`@optional`
		`MESSAGE_TYPE my_msg_field;`

`edition = "2023"`	`TYPE my_field ... ;`
	
	`TYPE my_field ... [`
	`  features.field_presence =`
	`↪EXPLICIT`
	`];`
		`@optional`
		`TYPE my_field;`
		
		`@optional`
		`TYPE my_field;`

6.6.3 Implicit fields

The `proto3` revision introduced the concept of `implicit presence`. This revision does not provide `Hazzer` methods for singular fields of non-message types that are not explicitly marked as `optional`.

In `edition = "2023"`, `implicit presence` is enabled by setting the `features.field_presence = IMPLICIT` option.

In addition to omitting the `@optional` annotation, IDL4 members derived from fields with `implicit presence` are annotated with `@field_presence(implicit)`. This annotation encodes the API behavior in the IDL4 data model.

Warning: Protocol Buffers discourage using *implicit presence*.

Table 6.13: Protocol Buffers to IDL4 Mapping for Implicit Fields

File Syntax	Protocol Buffers	DDS-XTYPES/IDL4
`proto2`	N/A	N/A
`proto3`	`NON_MESSAGE_TYPE my_field ... ;`	`@field_presence(implicit)` `NON_MESSAGE_TYPE my_field;`

`edition = "2023"`	`NON_MESSAGE_TYPE my_field ... [`
	`  features.field_presence = IMPLICIT`
	`];`
		`@field_presence(implicit)`
		`NON_MESSAGE_TYPE my_field;`

6.6.4 Field presence examples

Table 6.14: Example use of the `.omg.dds.member.optional` DDS option

Example Element	Protocol Buffers	DDS-XTYPES/IDL4

Explicitly mark any field as `@optional`	`TYPE my_field ... [`
	`  .omg.dds.optional = true`
	`];`
		`@optional`
		`TYPE my_field;`

Prevent default mapping to `@optional`	`optional TYPE my_field ... [`
	`  .omg.dds.optional = false`
	`];`
		`TYPE my_field;`

Mark a `repeated` field as `@optional`	`repeated TYPE my_field ... [`
	`  .omg.dds.optional = true`
	`];`
		`@optional`
		`sequence< TYPE > my_field;`

Mark a `map` field as `@optional`	`map<K, V> my_field ... [`
	`  .omg.dds.optional = true`
	`];`
		`@map`
		`@optional`
		`sequence<`
		`  MyMessage_MapPair_K_V`
		`> my_field;`

6.7 OneOf Fields

Protocol Buffers uses the `oneof` construct to group multiple optional fields, allowing only one field in the group to be set at any given time.

Every field included in a `oneof` is treated as an `optional` message field. The associated IDL4 members are annotated with `@optional` and `@oneof` to encode the name of the `oneof` group.

The `@oneof` annotation is most useful when the types are used with the Protocol Buffers C++ language binding. Applications using other language bindings must enforce the `oneof` semantics manually by setting/resetting fields in the same group.

Table 6.15: Protocol Buffers to IDL4 Mapping for OneOf Fields

Protocol Buffers	DDS-XTYPES/IDL4

`message MyMessage {`
 	
`  // other fields...`
 	
`  oneof oneof_field {`
`  A oneof_a ...;`
`  B oneof_b ...;`
`  C oneof_c ...;`
`  }`
 	
`}`
	`@mutable`
	`struct MyMessage {`
	
	`  // other fields...`
	
	`  @optional`
	`  @oneof("oneof_field")`
	`  A oneof_a;`
	
	`  @optional`
	`  @oneof("oneof_field")`
	`  B oneof_b;`
	
	`  @optional`
	`  @oneof("oneof_field")`
	`  C oneof_c;`
	`};`

6.8 Field Groups

Field groups are a Protocol Buffers feature that allows grouping fields within a message without requiring a separate message declaration.

In IDL4, each field group is mapped to a nested `struct` associated with the declaring message type.

> **Warning:** Protocol Buffers has deprecated field groups; they are only available when using the older `proto2` syntax.

Table 6.16: Protocol Buffers to IDL4 Mapping for Field Groups

File Syntax	Protocol Buffers	DDS-XTYPES/IDL4

proto2	`message MyMessage {`
	`  required group my_req_group = 1 {  }`
	`  optional group my_opt_group = 2 {  }`
	`  repeated group my_rep_group = 3 {  }`
	`}`
		`@nested`
		`@containing_type("MyMessage")`
		`@mutable`
		`struct MyMessage_MyReqGroup {  };`
		
		`@nested`
		`@containing_type("MyMessage")`
		`@mutable`
		`struct MyMessage_MyOptGroup {  };`
		
		`@nested`
		`@containing_type("MyMessage")`
		`@mutable`
		`struct MyMessage_MyRepGroup {  };`
		
		`struct MyMessage {`
		`  @id(1)`
		`  MyMessage_MyReqGroup my_req_group;`
		
		`  @id(2)`
		`  @optional`
		`  MyMessage_MyOptGroup my_opt_group;`
		
		`  @id(3)`
		`  sequence<MyMessage_MyRepGroup> my_rep_group;`
		`};`

proto3	N/A	N/A
edition = "2023"	N/A	N/A

6.9 Unsupported Features

The following Protocol Buffers features are not supported by *Protocol Buffers Extension*:

- Message extensions
- Self-referencing messages

Chapter 7

Release Notes

7.1 Supported Platforms

For a list of all platforms supported in *Connext Professional*, see the Core Libraries Platform Notes.

7.2 Compatibility

Protocol Buffers Extension has been tested with and supports:

- protocol buffer compiler (`protoc`) versions 3.12.0 through 33.0
- Protocol Buffers language specifications proto2, proto3, and Edition 2023

For backward compatibility information between the current and previous releases of *Protocol Buffers Extension*, see the *Migration Guide* on the RTI Community Portal.

7.3 What's New in 7.7.0

This section describes what's new in *Protocol Buffers Extension* 7.7.0.

Connext 7.7.0 is the first production release in the 7.7 long-term support (LTS) series. It is built upon and combines all of the fixes in releases 7.4.0, 7.5.0, and 7.6.0. See the Connext Versions and Lifecycle page for more information on RTI's software release model.

For what's new and fixed in other products in the *Connext* suite, see those products' release notes on the RTI Community Portal or in your installation.

7.3.1 Protocol Buffers Extension no longer experimental

Release 7.6.0 introduced *Protocol Buffers Extension*, an extension to Protocol Buffers that enables efficient serialization of Protocol Buffers messages in DDS applications.

In release 7.6.0, this product was experimental. In release 7.7.0, *Protocol Buffers Extension* is ready for production use.

7.3.2 RTI Code Generator enables Protocol Buffers integration with DDS

Code Generator 7.7.0 includes a new feature that allows developers to generate C++ source code from Protocol Buffers definitions. For more information, see *Integrating Protocol Buffers with Connext*.

7.3.3 Improved tutorial build files for Protocol Buffers Extension

The `USER_QOS_PROFILES.xml` file created by the *Protocol Buffers Extension* tutorial build files is now generated in a more intuitive location to improve usability. For details about the tutorial, see *Generating Code from Protocol Buffers messages*.

7.3.4 Third-Party Software Changes

The following third-party software used by *Protocol Buffers Extension* has been upgraded:

Table 7.1: Third-Party Software Upgrades

Third-Party Software	Old Version	New Version
Protocol Buffers	25.9	33.0

For information on third-party software used by *Connext* products, see the "3rdPartySoftware" documents in your installation: <NDDSHOME>/doc/manuals/connext_dds_professional/release_notes_3rdparty.

7.4 What's Fixed in 7.7.0

This section describes bugs fixed in *Protocol Buffers Extension* 7.7.0.

Connext 7.7.0 is the first production release in the 7.7 long-term support (LTS) series. It is built upon and combines all of the fixes in releases 7.4.0, 7.5.0, and 7.6.0. See the Connext Versions and Lifecycle page for more information on RTI's software release model.

For what's fixed in other products in the *Connext* suite, see those products' release notes on the RTI Community Portal or in your installation.

7.4.1 [Major] Use of certain Protocol Buffers messages could result in a crash when publishing or subscribing to data

Connext applications using the C++ Protocol Buffers (protobuf) language binding may have crashed when trying to read or write data on *Topics* associated with specific protobuf message types. (The C++ Protocol Buffers language binding was an experimental feature in release 7.6.0 and is now productized in the *Protocol Buffers Extension* in release 7.7.0.)

The issue originated in the generated type-support code. It occurred only when a *Topic* type was, or referenced (directly or indirectly through a nested field), a protobuf message with one or more fields containing:

- `repeated` fields (collections) of `message`, `string`, or `byte` elements
- `maps` of any kind

Topics using any of the previously affected types are now correctly handled by the middleware.

[RTI Issue ID CORE-16524]

7.4.2 [Minor] Using fields with camel case naming in .proto files was not working

The protocol buffer compiler (`protoc`) converted camel case file names to lowercase for C++. However *Code Generator* did not. That discrepancy caused a mismatch between the code generated by *Code Generator* and the code generated by `protoc`.

Code Generator's conversion of `.proto` files to IDL now converts the field names to lower case to generate code that will compile with the code generated by `protoc`.

[RTI Issue ID CODEGENII-2349]

7.4.3 [Minor] Code Generator did not detect incompatible IDL file generated from .proto file

Protocol Buffers `.proto` files support some features such as recursive types and a member ID higher than that supported by the Protocol Buffers mapping generated by *Code Generator*.

Code Generator now detects those issues after converting the `.proto` file to IDL and throws an error instead of generating code that will not work.

[RTI Issue ID CODEGENII-2348]

Chapter 8

Copyrights and Notices

© 2025 - 2026 Real-Time Innovations, Inc. All rights reserved. June 2026

Trademarks

RTI, Real-Time Innovations, Connext, Connext Drive, NDDS, the RTI logo, 1RTI and the phrase, "Your Systems. Working as one." are registered trademarks, trademarks or service marks of Real-Time Innovations, Inc. All other trademarks belong to their respective owners.

Copy and Use Restrictions

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form (including electronic, mechanical, photocopy, and facsimile) without the prior written permission of Real-Time Innovations, Inc. The software described in this document is furnished solely under and subject to RTI's standard terms and conditions available at https://www.rti.com/terms and in accordance with your License Acknowledgement Certificate (LAC) and Maintenance and Support Certificate (MSC), except to the extent otherwise agreed to in writing by RTI.

Third-Party Software

RTI software may contain independent, third-party software or code that are subject to third-party license terms and conditions, including open source license terms and conditions. Copies of applicable third-party licenses and notices are located at community.rti.com/documentation. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF THIRD-PARTY SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS AND CONDITIONS.

Notices

Deprecations and Removals

Deprecated means that the item is still supported in this release, but will be removed in a future release. *Removed* means that the item is discontinued or no longer supported.

Any deprecations or removals noted in RTI's documentation serve as notice under the Real-Time Innovations, Inc., Maintenance Policy #4220 and/or any other agreements by and between RTI and customer regarding maintenance and support of RTI's software. RTI's current standard terms and support and maintenance policies are available at https://www.rti.com/terms.

Technical Support Real-Time Innovations, Inc. 232 E. Java Drive Sunnyvale, CA 94089 Phone: (408) 990-7444 Email: support@rti.com Website: https://support.rti.com/