Trademarks

RTI, Real-Time Innovations, Connext, 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 accepted in writing by a corporate officer of RTI.

This is an independent publication and is neither affiliated with, nor authorized, sponsored, or approved by, Microsoft Corporation.

The security features of this product include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Notices

Deprecations and Removals

Any deprecations or removals noted in this document 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.

Deprecated means that the item is still supported in the release, but will be removed in a future release. Removed means that the item is discontinued or no longer supported. By specifying that an item is deprecated in a release, RTI hereby provides customer notice that RTI reserves the right after one year from the date of such release and, with or without further notice, to immediately terminate maintenance (including without limitation, providing updates and upgrades) for the item, and no longer support the item, in a future release.
Early Access Software

“Real-Time Innovations, Inc. (“RTI”) licenses this Early Access release software (“Software”) to you subject to your agreement to all of the following conditions:

(1) you may reproduce and execute the Software only for your internal business purposes, solely with other RTI software licensed to you by RTI under applicable agreements by and between you and RTI, and solely in a non-production environment;

(2) you acknowledge that the Software has not gone through all of RTI’s standard commercial testing, and is not maintained by RTI’s support team;

(3) the Software is provided to you on an “AS IS” basis, and RTI disclaims, to the maximum extent permitted by applicable law, all express and implied representations, warranties and guarantees, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, satisfactory quality, and non-infringement of third party rights;

(4) any such suggestions or ideas you provide regarding the Software (collectively, “Feedback”), may be used and exploited in any and every way by RTI (including without limitation, by granting sub-licenses), on a non-exclusive, perpetual, irrevocable, transferable, and worldwide basis, without any compensation, without any obligation to report on such use, and without any other restriction or obligation to you; and

(5) TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL RTI BE LIABLE TO YOU FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, EXEMPLARY OR PUNITIVE OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR FOR LOST PROFITS, LOST DATA, LOST REPUTATION, OR COST OF COVER, REGARDLESS OF THE FORM OF ACTION WHETHER IN CONTRACT, TORT (INCLUDING WITHOUT LIMITATION, NEGLIGENCE), STRICT PRODUCT LIABILITY OR OTHERWISE, WHETHER ARISING OUT OF OR RELATING TO THE USE OR INABILITY TO USE THE SOFTWARE, EVEN IF RTI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.”

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/
# Contents

Chapter 1 Introduction ................................................................. 1
Chapter 2 Paths Mentioned in Documentation ................................ 2
Chapter 3 Command-Line Arguments for rtiddsgen ............................ 4

Chapter 4 Generating Example Code
  4.1 Input Files (IDL, XML, XSD) ............................................... 13
    4.1.1 IDL Language .......................................................... 13
    4.1.2 XML Language ....................................................... 13
    4.1.3 XSD Language ....................................................... 14
  4.2 C++ Example ........................................................................ 14

Chapter 5 Generated Files ................................................................. 18

Chapter 6 Customizing the Generated Code ........................................ 24

Chapter 7 Optimizing the Code Generation Process
  7.1 Optimization Levels ............................................................. 27
  7.2 How the Optimizations are Applied .......................................... 28
    7.2.1 Inline expansion of nested types .................................. 28
    7.2.2 Serialization of consecutive members with a single copy ...... 29
    7.2.3 Rules for Inline Expansion .......................................... 29
    7.2.4 Guidelines ............................................................. 32

Chapter 8 Boosting Performance
8.1 Using Server Mode ................................................................. 33
8.2 Using JVM Optimization .......................................................... 35
Chapter 1 Introduction

RTI® Code Generator creates the code needed to define and register a user data type with RTI Connext® DDS. It takes a language-independent specification of the data (in IDL, XML, or XSD notation) and generates supporting classes and code to distribute instances of the data over Connext.

Using Code Generator is optional if:

- You are using dynamic types (see Managing Memory for Built-in Types (Section 3.2.7) in the RTI Connext Core Libraries Users Manual).
- You are using one of the built-in types (see Built-in Data Types (Section 3.2) in the RTI Connext Core Libraries Users Manual).

To use Code Generator, you will need to provide a description of your data type(s) in an IDL or XML file. You can define multiple data types in the same type-definition file. For details on these files, see the "Data Types and DDS Data Samples" chapter in the RTI Connext Core Libraries User's Manual.
Chapter 2 Paths Mentioned in Documentation

The documentation refers to:

- `<NDDSHOME>`

This refers to the installation directory for RTI® Connext®. The default installation paths are:

- macOS® systems:
  `/Applications/rti_connext.dds-7.0.0`

- Linux systems, non-root user:
  `/home/<your user name>/rti_connext.dds-7.0.0`

- Linux systems, root user:
  `/opt/rti_connext.dds-7.0.0`

- Windows® systems, user without Administrator privileges:
  `<your home directory>\rti_connext.dds-7.0.0`

- Windows systems, user with Administrator privileges:
  `C:\Program Files\rti_connext.dds-7.0.0`

You may also see `SNDDSHOME` or `%NDDSHOME%`, which refers to an environment variable set to the installation path.

Wherever you see `<NDDSHOME>` used in a path, replace it with your installation path.
Note for Windows Users: When using a command prompt to enter a command that includes the path `C:\Program Files` (or any directory name that has a space), enclose the path in quotation marks. For example:

```
"C:\Program Files\rti_connext_dds-7.0.0\bin\rtiddsgen"
```

Or if you have defined the `NDDSHOME` environment variable:

```
"%NDDSHOME%\bin\rtiddsgen"
```

- `<path to examples>`

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>`.

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

Default path to the examples:

- macOS systems: `/Users/<your user name>/rti_workspace/7.0.0/examples`
- Linux systems: `/home/<your user name>/rti_workspace/7.0.0/examples`
- Windows systems: `<your Windows documents folder>\rti_workspace\7.0.0\examples`

Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 10, the folder is `C:\Users\<your user name>\Documents`.

Note: You can specify a different location for `rti_workspace`. You can also specify that you do not want the examples copied to the workspace. For details, see *Controlling Location for RTI Workspace and Copying of Examples* in the *RTI Connext Installation Guide*. 
Chapter 3 Command-Line Arguments for rtiddsgen

On Windows systems: Before running rtiddsgen, run the "vcvars" batch file that is appropriate for your architecture in the same command prompt that you will use to run rtiddsgen. For example: vcvarsall.bat x86. The location of the "vcvars" batch file varies depending on your version of Visual Studio. Consult the Microsoft Visual Studio documentation for the exact location and parameters for your installation. Alternatively, run rtiddsgen from the Visual Studio Command Prompt under the Visual Studio Tools folder. Or, use the command-line argument -ppDisable. See -ppDisable in Table 3.1 Options for rtiddsgen on page 6.

If you are generating code for Connext, the options are:

```bash
rtiddsgen [-help]
        [-allocateWithMalloc]
        [-alwaysUseStdVector]
        [-autoGenFiles <architecture>]
        [-constructor]
        [-create <typefiles| examplefiles|makefiles>]
        [-convertToIdl | -convertToXML | -convertToXsd]
        [-D <name>={=<value}>]
        [-d <outdir>]
        [-disableXSDValidation]
        [-dllExportMacroSuffix <suffix>]
        [-enableEscapeChar]
        [-example <architecture>]
        [-exampleTemplate]
        [-I <directory>]
        [[-inputIdl] <IDLInputFile.idl> | [-inputXml] <XMLInputFile.xml>
        |[-inputXsd <IDLInputFile.idl>]]
        [-language <Ada|C|C++|98|11|C#|Java|Python>]
        [-namespace]
        [-obfuscate]
        [-optimization <level>]
        [-package <packagePrefix>]
        [-platform <architecture>]
        [-ppDisable]
```
Chapter 3 Command-Line Arguments for rtiddsgen

[-ppPath <path to preprocessor>]  
[-ppOption <option>]  
[-qualifiedEnumerator]  
[-reader]  
[-replace]  
[-sequenceSize <unbounded sequences size>]  
[-sharedLib]  
[-showTemplates]  
[-strict]  
[-stringSize <unbounded strings size>]  
[-typeSequenceSuffix <suffix>]  
[-U <name>]  
[-unboundedSupport]  
[-update <typefiles | examplefiles | makefiles>]  
[-use52Keyhash]  
[-use526Keyhash]  
[-useStdString]  
[-V <name> [=<value>]]  
[-verbosity [1-3]]  
[-version]  
[-virtualDestructor]  
[-writer]

If you are generating code for RTI Connext Micro, the options are:

rtiddsgen  [-help]  
[-create <typefiles | examplefiles | makefiles>]  
[-convertToIdl | -convertToXML]  
[-D <name> [=<value>]]  
[-d <outdir>]  
[-enableEscapeChar]  
[-example]  
[-exampleTemplate]  
[-I <directory>]  
[-I <inputIdl] <IDLInputFile.idl> | [-I<inputXml] <XMLInputFile.xml>]  
[-language <C|C++>]  
[-micro]  
[-namespace]  
[-ppDisable]  
[-ppPath <path to preprocessor>]  
[-ppOption <option>]  
[-reader]  
[-replace]  
[-showTemplates]  
[-sequenceSize <unbounded sequences size>]  
[-strict]  
[-stringSize <unbounded strings size>]  
[-U <name>]  
[-update <typefiles | examplefiles | makefiles>]  
[-V <name> [=<value>]]  
[-verbosity [1-3]]  
[-version]  
[-writer]
Table 3.1 Options for rtiddsgen describes the options.

**Note:** Before using a makefile created by Code Generator to compile an application, make sure the
$\{NDDSHOME\}$ environment variable is set as described in Set Up Environment Variables (rtisetenv),
in "Hands-On 1” of Introduction to Publish/Subscribe, in the RTI Connext Getting Started Guide.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-allocateWithMalloc</td>
<td>Use this flag to obtain backward-compatibility when allocating optional members with DDS_Heap_malloc in C++.</td>
</tr>
<tr>
<td>-alwaysUseStdVector</td>
<td>Only applies if -language C++11 is specified. Generates code that maps all sequences to std::vector, even bounded sequences that would otherwise map to rti::core::bounded_sequence. Alternatively, the @use_vector annotation can be applied to each sequence member.</td>
</tr>
<tr>
<td>-autoGenFiles &lt;architecture&gt;</td>
<td>Updates the auto-generated files, i.e., the typefiles and makefile/project files. To see the valid options for &lt;architecture&gt; per language, run rtiddsgen with the -help option, or use the string &quot;universal&quot; (-autoGenFiles universal) to generate compatible publisher/subscriber code for all supported platforms. The universal architecture will not generate makefiles/project files. This is a shortcut for: -update typefiles -update makefiles -platform &lt;architecture&gt;</td>
</tr>
<tr>
<td>-constructor</td>
<td>Only applies if -language C++98 is also specified. Generates the types default constructor, copy constructor, copy assignment operator, and destructor. Using this option will also disable the generation of the following TypeSupport methods: create_data(ex), delete_data(ex), initialize_data(ex), finalize_data(ex), copy_data.</td>
</tr>
<tr>
<td>-create &lt;typefiles</td>
<td>examplefiles</td>
</tr>
<tr>
<td>-convertToIdl</td>
<td>Converts the input type description file into IDL format. This option creates a new file with the same name as the input file and a .idl extension.</td>
</tr>
<tr>
<td>-convertToXML</td>
<td>Converts the input type description file into XML format. This option creates a new file with the same name as the input file and a .xml extension.</td>
</tr>
<tr>
<td>-convertToXsd</td>
<td>Converts the input type description file into XSD format. This option creates a new file with the same name as the input file and a .xsd extension.</td>
</tr>
<tr>
<td>-D &lt;name&gt;[=&lt;value&gt;]</td>
<td>Defines preprocessor macros. On Windows systems, enclose the argument in quotation marks: -D &quot;&lt;name&gt;[=&lt;value&gt;]&quot;</td>
</tr>
<tr>
<td>-d &lt;outdir&gt;</td>
<td>Generates the output in the specified directory. By default, Code Generator will generate files in the directory where the input type-definition file is found.</td>
</tr>
<tr>
<td>-disableXSDValidation</td>
<td>Causes Code Generator not to check that the input XSD file is well-formed. The use of this option is not recommended in general, as Code Generator may receive invalid inputs.</td>
</tr>
</tbody>
</table>
**Table 3.1 Options for rtiddsgen**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-dllExportMacroSuffix &lt;suffix&gt;</td>
<td>Defines the suffix of the macro that is used to export symbols when building Windows DLLs. The default macro is NDDS_USER_DLL_EXPORT. When this option is specified, the name of the macro is NDDS_USER_DLL_EXPORT_&lt;Suffix&gt;.</td>
</tr>
<tr>
<td>-enableEscapeChar</td>
<td>Enables use of the escape character _ in IDL identifiers.</td>
</tr>
<tr>
<td></td>
<td>For Connext Professional:</td>
</tr>
<tr>
<td></td>
<td>-example &lt;architecture&gt;</td>
</tr>
<tr>
<td></td>
<td>For Connext Micro:</td>
</tr>
<tr>
<td></td>
<td>-example</td>
</tr>
<tr>
<td></td>
<td>Generates type files, example files, and a makefile.</td>
</tr>
<tr>
<td></td>
<td>This is a shortcut for:</td>
</tr>
<tr>
<td></td>
<td>-create typefiles -create examplefiles -create makefiles -platform &lt;architecture&gt;</td>
</tr>
<tr>
<td></td>
<td>To see the valid options for &lt;architecture&gt; per language, run rtiddsgen with the -help option, or use the string &quot;universal&quot; (-example universal) to generate compatible publisher/subscriber code for all supported platforms. The universal architecture will not generate makefiles/project files.</td>
</tr>
<tr>
<td></td>
<td>For Python, when generating examples, the only available platform is &quot;universal&quot;. For example:</td>
</tr>
<tr>
<td></td>
<td>rtiddsgen -language Python -example -platform universal Hello.idl</td>
</tr>
<tr>
<td></td>
<td>For Connext Micro, you do not need to specify an &lt;architecture&gt;. rtiddsgen for Connext Micro generates CMakeLists, which can be used to compile for any architecture.</td>
</tr>
</tbody>
</table>
Table 3.1 Options for rtiddsgen

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| -exampleTemplate  | For Connext Professional: Generates a custom example application using the specified custom publisher and subscriber templates, instead of the default example files. Place your custom templates in $NDDSHOME/resource/app/app_support/rtiddsgen/templates/example/<language>/<exampleTemplateDirectoryName>. Name them publisher.vm and subscriber.vm; you can also add a QoS template (qosProfile.vm) file to your example template directory (if you don’t, Code Generator will use the default one, $NDDSHOME/resource/app/app_support/rtiddsgen/templates/example/qosProfile.vm). You can base your templates on those that Connext provides in $NDDSHOME/resource/app/app_support/rtiddsgen/templates/example/<language>. The -exampleTemplate option must be used in combination with one of the following command-line options:  
  - -create examplefiles
  - -update examplefiles
  - -example <architecture>

For example:

```
rtiddsgen -language C++11 -example x64Darwin17clang9.0 -exampleTemplate myCustomTemplate foo.idl
```

"myCustomTemplate" is your <exampleTemplateDirectoryName> in $NDDSHOME/resource/app/app_support/rtiddsgen/templates/example/<language>/<exampleTemplateDirectoryName>.

To get a full list of the available custom templates in your Connext installation, use the -showTemplates option.

For C++, C++11, C#, and Java, you can also use an advanced -exampleTemplate option included with Connext. See 4.3 Advanced Example on page 14.

The -exampleTemplate option is not supported in Ada.
Table 3.1 Options for rtiddsgen

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-exampleTemplate</td>
<td>For Connext Micro, <code>&lt;exampleTemplateName&gt;</code> behaves the same as it does for Connext Professional, except for the following differences: Place your custom templates in <code>$RTMEHOME/rtiddsgen/resource/rtiddsgen/templates/example/&lt;language&gt;/&lt;exampleTemplateDirectoryName&gt;</code>. Name them publisher.vm and subscriber.vm. The following files are also in the <code>$RTMEHOME/rtiddsgen/resource/rtiddsgen/templates/example/&lt;language&gt;</code> directory: application.vm, applicationHeader.vm, and readme.vm. Template files mag.vm and mag_qoa.vm are optional. If present in the directory, they can be used to generate RTI Micro Application Generator (MAG) applications. See Connext Micro Application Generation in the RTI Connext XML-Based Application Creation Getting Started Guide. For example: rtiddsgen -micro -language C -example &lt;exampleTemplateName&gt; foo.idl qoaProfile.vm is not supported by Connext Micro.</td>
</tr>
<tr>
<td>-help</td>
<td>Prints out the command-line options for rtiddsgen.</td>
</tr>
<tr>
<td>-l &lt;directory&gt;</td>
<td>Adds to the list of directories to be searched for type-definition files (IDL or XML files). Note: A type-definition file in one format cannot include a file in another format.</td>
</tr>
<tr>
<td>-inputIdl</td>
<td>Indicates that the input file is an IDL file, regardless of the file extension.</td>
</tr>
<tr>
<td>-inputXml</td>
<td>Indicates that the input file is an XML file, regardless of the file extension.</td>
</tr>
<tr>
<td>-inputXsd</td>
<td>Indicates that the input file is an XSD file, regardless of the file extension.</td>
</tr>
<tr>
<td>IDLInputFile.idl</td>
<td>A file containing IDL descriptions of your data types. If -inputIdl is not used, the file must have a `'idl' extension.</td>
</tr>
<tr>
<td></td>
<td>For Connext Professional:</td>
</tr>
<tr>
<td>-language &lt;Ada</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Notes:</td>
</tr>
<tr>
<td></td>
<td>• For Connext Professional, use <code>-language C++98</code> or <code>-language C++11</code> to select the Traditional C++ or Modern C++ API respectively. Note that C++98 and C++11 are the minimum C++ versions required by each API. Applications can use newer C++ standards.</td>
</tr>
<tr>
<td></td>
<td>• For Python, when generating examples, the only available platform is &quot;universal&quot;. For example: rtiddsgen -language Python -example -platform universal Hello.idl</td>
</tr>
<tr>
<td></td>
<td>• For Connext Micro, use <code>-language C++</code> to select the Traditional C++ API, which supports C++98 (or newer).</td>
</tr>
<tr>
<td>-micro</td>
<td>Generates code and support files for Connext Micro, instead of for Connext Professional. Use <code>-micro -help</code> to list the options supported by Code Generator when targeting RTI Connext Micro.</td>
</tr>
<tr>
<td>-namespace</td>
<td>Specifies the use of C++ namespaces, for traditional C++ only. For modern C++ and C#, it is implied—namespaces are always used.</td>
</tr>
<tr>
<td>-obfuscate</td>
<td>Generates an obfuscated IDL file from the input file. Note that even if the input type is XML, this option generates an obfuscated IDL file.</td>
</tr>
</tbody>
</table>
### Table 3.1 Options for rtiddsgen

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-optimization &lt;level&gt;</td>
<td>Level of optimization of the code:</td>
</tr>
<tr>
<td></td>
<td>- 0: No optimization.</td>
</tr>
<tr>
<td></td>
<td>- 1: The compiler generates extra code for typedefs but optimizes its use. If a type that is used is a typedef that can be resolved to a primitive, enum, or aggregated type (struct, union, or value type), the generated code will invoke the code of the most basic type to which the typedef can be resolved. This level can be used if the generated code is not expected to be modified. This is the only optimization level supported for Java and C# languages.</td>
</tr>
<tr>
<td></td>
<td>- 2: (Default) This optimization level applies only to C, C++, C++11 and higher, microC, microC++, and Ada languages. With this optimization level, rtiddsgen optimizes the serialization/deserialization of structures and valuetypes by using more aggressive techniques. These techniques include inline expansion of nested types and serialization/deserialization of a set of consecutive members with a single copy function invocation (memcpy) when the memory layout (C, C++ structure layout) is the same as the wire layout (XCDR). See Chapter 7 Optimizing the Code Generation Process on page 27 for more information.</td>
</tr>
<tr>
<td>-package &lt;packagePrefix&gt;</td>
<td>Specifies the root package into which generated classes will be placed. It applies to Java only. If the type-definition file contains module declarations, those modules will be considered subpackages of the package specified here.</td>
</tr>
<tr>
<td>-platform &lt;architecture&gt;</td>
<td>Required if -create makefiles or -update makefiles is used.</td>
</tr>
<tr>
<td></td>
<td>To see the valid options for &lt;architecture&gt; per language, run rtiddsgen with the -help option, or use the string &quot;universal&quot; (-platform universal) to generate compatible publisher/subscriber code for all supported platforms. The universal architecture will generate makefiles/project files.</td>
</tr>
<tr>
<td></td>
<td>For Python, when generating examples, the only available platform is &quot;universal&quot;. For example: rtiddsgen -language Python -example -platform universal Hello.idl</td>
</tr>
<tr>
<td>-ppDisable</td>
<td>Disables the preprocessor.</td>
</tr>
<tr>
<td></td>
<td>Code Generator supports the standard preprocessor directives defined by the IDL specification, such as #if, #endif, #include, and #define.</td>
</tr>
<tr>
<td></td>
<td>To support these directives, Code Generator calls an external C preprocessor before parsing the IDL file. On Windows systems, the preprocessor is 'cl.exe'. On other architectures, the preprocessor is 'cpp'. You can change the default preprocessor with the -ppPreprocessor option. If you do not want to run the preprocessor, use the -ppDisable option.</td>
</tr>
<tr>
<td></td>
<td>The argument -ppDisable tells the code generator not to attempt to invoke the C preprocessor on the IDL file prior to generating code. In some cases, the IDL file contains no preprocessor directives, so no precompilation is necessary. Therefore, you may want to use the -ppDisable option to tell Code Generator to skip calling the preprocessor. However, if the preprocessor executable is already on your system path (on Windows systems, it will be on your path if you've run the Visual Studio &quot;vcvars&quot; batch file that is appropriate for your architecture), you don't need to use -ppDisable.</td>
</tr>
<tr>
<td></td>
<td>In summary, -ppDisable is a useful, quick remedy for simple examples (including HelloWorld examples) for which the IDL is simple and does not need Code Generator to invoke the preprocessor. The longterm recommendation, however, is to have the preprocessor in your path (or use the -ppOption). Linux architectures often already have the preprocessor in the path; Windows architectures may require you to run a &quot;vcvars&quot; batch file as described at the beginning of this chapter to put the preprocessor in your path.</td>
</tr>
<tr>
<td>-ppOption &lt;option&gt;</td>
<td>Specifies a preprocessor option. This option can be used multiple times to provide the command-line options for the specified preprocessor. See -ppPath &lt;path to preprocessor&gt;.</td>
</tr>
<tr>
<td>-ppPath &lt;path to preprocessor&gt;</td>
<td>Specifies the preprocessor. If you only specify the name of an executable (not a complete path to that executable), the executable must be found in your Path. The default value is cpp for non-Windows architectures and cl.exe for Windows architectures.</td>
</tr>
<tr>
<td></td>
<td>If you use -ppPath to provide the full path and filename for cl.exe or the cpp preprocessor, you must also use -ppOption &lt;option&gt; to set the following preprocessor options:</td>
</tr>
<tr>
<td></td>
<td>If you use a non-default path for cl.exe, you also need to set: -ppOption /nologo -ppOption /C -ppOption /E -ppOption /X</td>
</tr>
<tr>
<td></td>
<td>If you use a non-default path for cpp, you also need to set: -ppOption /C</td>
</tr>
</tbody>
</table>

2 is the default for C, C++, C++11 and higher, microC, microC++, and Ada languages (but you can change it to 0 or 1). 1 is always used for Java and C# languages, and you cannot change it.
# Table 3.1 Options for rtiddsgen

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-qualifiedEnumerator</td>
<td>Uses the fully qualified name for enumerator values including the enum value.</td>
</tr>
<tr>
<td>-reader</td>
<td>Generates support for a DataReader (only with -micro).</td>
</tr>
<tr>
<td>-replace</td>
<td>Deprecated option. Instead, use -update &lt;typefiles</td>
</tr>
<tr>
<td>-sequenceSize &lt;unbounded sequences size&gt;</td>
<td>Sets the size assigned to unbounded sequences. The default value is 100 elements.</td>
</tr>
<tr>
<td>-sharedLib</td>
<td>Generates makefiles that compile with the Connext shared libraries (by default, the makefile will link with the static libraries)</td>
</tr>
<tr>
<td>-showTemplates</td>
<td>Prints and generates an XML file containing a list of the available example templates in your Connext installation, organized per language. When you use the -exampleTemplate option, you can specify one of these example templates. Code Generator will then generate the example you specified instead of the default one.</td>
</tr>
<tr>
<td>-strict</td>
<td>Enforces compliance with the OMG Extensible and Dynamic Topic Types for DDS specification, version 1.3 by turning informational (INFO) messages into errors. (Informational messages are described in -verbosity [1-9].) Currently in this release, by default, Code Generator will report an informational (INFO) message for key derived structures when running rtiddsgen, but still generate code: INFO com.rti.ndds.nddsgen.antlr.annotation.AnnotationApplier struct/valueType derived from a struct/valueType should not contain @key fields. This is not compliant with the XTypes specification. You can make this message an error by using the -strict flag If you want Code Generator to enforce the specification, use the -strict option. This option will report an error when there are key derived structures, and not generate code. See the RTI Code Generator Release Notes or the Migration Guide on the RTI Community Portal (<a href="https://community.rti.com/documentation">https://community.rti.com/documentation</a>) for enforcement details that might occur in future releases.</td>
</tr>
<tr>
<td>-stringSize &lt;unbounded strings size&gt;</td>
<td>Sets the size assigned to unbounded strings, not counting a terminating NULL character. The default value is 255 bytes.</td>
</tr>
<tr>
<td>-typeSequenceSuffix &lt;suffix&gt;</td>
<td>Assigns a suffix to the name of the implicit sequence defined for IDL types. The option is compatible with languages C, C++, C#, and Java. By default, the suffix is 'Seq'. For example, given the type 'Foo', the name of the sequence would be 'FooSeq'.</td>
</tr>
<tr>
<td>-U &lt;name&gt;</td>
<td>Cancels any previous definition of &lt;name&gt;.</td>
</tr>
</tbody>
</table>
| -unboundedSupport             | Generates code that supports unbounded sequences and strings. This option is not supported in Ada. When the option is used, the command-line options -sequenceSize and -stringSize are ignored. This option also affects the way unbounded sequences are deserialized. When a sequence is being received into a sample from the DataReader's cache, the old memory for the sequence will be deallocated and memory of sufficient size to hold the deserialized data will be allocated. When initially constructed, sequences will not preallocate any elements having a maximum of zero elements. For more information on using the -unboundedSupport option, including some required QoS settings, see these sections in the RTI Connext Core Libraries User's Manual:  
  - Sequences (Section 3.1.1)  
  - Strings and Wide Strings (Section 3.1.2)  
  - Sample and Instance Memory Management chapter in the RTI Connext Core Libraries User's Manual |
Table 3.1 Options for rtiddsgen

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-update &lt;typefiles</td>
<td>examplefiles</td>
</tr>
<tr>
<td>-use52Keyhash</td>
<td>This option should be used when compatibility with 5.2.3 and earlier General Access Releases (GARs) is required when using keyed mutable types (related to RTI Issue IDs CODEGENII-501 and CODEGENII-693).</td>
</tr>
<tr>
<td>-use526Keyhash</td>
<td>This option should be used when compatibility with 5.2.6 is required when using keyed mutable types (related to RTI Issue ID CODEGENII-693).</td>
</tr>
<tr>
<td>-useStdString</td>
<td>Use 'std::string' instead of 'char*' when generating code for IDL strings when the language option is C++. Using this option will automatically enable constructor generation. Therefore you can use this option with or without -constructor and achieve the same result.</td>
</tr>
<tr>
<td>-V &lt;name&gt; [&lt;=value&gt;]</td>
<td>Defines a user variable that can be used in the templates as $userVarList.name or $userVarList.name.equals(value). The variables defined with this option are case sensitive.</td>
</tr>
<tr>
<td>-verbosity [1-3]</td>
<td>Sets the Code Generator verbosity: 1: Exceptions 2: Exceptions and warnings 3: Exceptions, warnings and information (Default)</td>
</tr>
<tr>
<td>-version</td>
<td>Displays the version of Code Generator being used, such as 2.x.y, as well as the version of the templates being used (xxxx-xxxx-xxxx).</td>
</tr>
<tr>
<td>-virtualDestructor</td>
<td>Only applies if -language C++98 is also specified. Generates a virtual destructor for the generated types in C++. Using this option will automatically enable the -constructor option. Note that using this option will affect filtering performance when using ContentFilteredTopics or QueryConditions.</td>
</tr>
<tr>
<td>-writer</td>
<td>Generates support for a DataWriter (only with -micro).</td>
</tr>
<tr>
<td>XMLInputFile.xml</td>
<td>A file containing XML descriptions of your data types. If -inputXml is not used, the file must have an .xml extension.</td>
</tr>
</tbody>
</table>
Chapter 4 Generating Example Code

Most Connext Getting Started Guides (such as the RTI Connext Getting Started Guide or the RTI Connext Core Libraries Getting Started Guide Addendum for Embedded Systems) have you create a simple "Hello World" application to get started. For example, on macOS:

```bash
$ rtiddsgen -language c++11 -example x64Darwin17clang9.0 hello_world.idl
```

Note: Before using a makefile created by Code Generator to compile an application, make sure the ${NDDSHOME} environment variable is set as described in Set Up Environment Variables (rtisetenv), in "Hands-On 1" of Introduction to Publish/Subscribe, in the RTI Connext Getting Started Guide.

4.1 Input Files (IDL, XML, XSD)

For detailed information about IDL, XML, or XSD support in Connext, including the mapping between these formats, see the "Data Types and DDS Data Samples" chapter in the RTI Connext Core Libraries User's Manual.

4.1.1 IDL Language

In the IDL language, data types are described in a fashion almost identical to structures in "C."

The complete description of the language can be found at the OMG website.

4.1.2 XML Language

Connext provides DTD and XSD files that describe the XML format.

The DTD definition of the XML elements can be found in rti.dds_topic_types.dtd under <NDDSHOME>/resource/app/app_support/rtiddsgen/schema.

The XSD definition of the XML elements can be found in rti.dds_topic_types.xsd under <NDDSHOME>/resource/app/app_support/rtiddsgen/schema.
The XML validation performed by rtiddsgen always uses the DTD definition. If the <!DOCTYPE> tag is not present in the XML file, rtiddsgen will look for the DTD document under <NDDSHOME>/resource/app/app_support/rtiddsgen/schema. Otherwise, it will use the location specified in <!DOCTYPE>.

4.1.3 XSD Language

For detailed information about XSD support in Connext, see the "Data Types and DDS Data Samples" chapter in the RTI Connext Core Libraries User's Manual.

4.2 C++ Example

The following is an example that generates the Connext type myDataType:

**IDL notation**

```idl
struct myDataType {
    long value;
};
```

**XML notation**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="rti.dds_topic_types.xsd">
    <struct name="myDataType">
        <member name="value" type="long"/>
    </struct>
</types>
```

**XSD notation**

```xsd
<?xml version="1.0" encoding="UTF-8"?>
    <xsd:import namespace="http://www.omg.org/dds" schemaLocation="rti.dds_topic_types_common.xsd"/>
    <xsd:complexType name = "myDataType">
        <xsd:sequence>
            <xsd:element name="value" minOccurs="1" maxOccurs="1" type="xsd:int"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

4.3 Advanced Example

After you get familiar with the simple Hello World example, you can create a more advanced example by specifying the 'advanced' template option, for traditional C++, modern C++, C#, and Java languages.
The advanced template is already included with *Code Generator* (see `-exampleTemplate` and `-showTemplates`). For example, on macOS:

```
rtiddsgen -language c++11 -example x64Darwin17clang9.0 -exampleTemplate advanced hello_world.idl
```

There are a couple of main differences between the simple and advanced examples that *Code Generator* generates, described below: **is_default_qos** and **Listeners**.

**Note:** The advanced generated example is not supported on Android™ or INTEGRITY® platforms.

### 4.3.1 is_default_qos (true vs. false)

*Connext* loads QoS profiles from a file named USER_QOS_PROFILES.xml in your current working directory. (*Connext* may also look for this file in other locations; see *How to Load XML-Specified QoS Settings* in the [RTI Connext Core Libraries User's Manual](https://www.rti.com/products/rti-connext-core-libraries-users-manual)). The simple example sets **is_default_qos**=`true` in the USER_QOS_PROFILES.xml file. It creates the DDS entities without specifying a profile, so it uses the default from USER_QOS_PROFILES.xml.

The advanced example also loads QoS from the USER_QOS_PROFILES.xml file. However, the advanced example omits the **is_default_qos** setting, which means that *Connext* assumes the default setting of false. Therefore, there is no default QoS provided by the XML. The example explicitly specifies which QoS profile to use from the XML file when creating DDS entities.

Setting **is_default_qos=true** is a convenient way to get you started quickly, but in production applications you should explicitly specify which QoS profile to use, instead of relying on a default. See the Basic QoS chapter, in the [RTI Connext Getting Started Guide](https://www.rti.com/products/rti-connext-getting-started-guide).

**Note:** The **RTI Connext Getting Started Guide** example is more complex than the advanced example generated by *Code Generator*; however, it does not use **Listeners** like the *Code Generator* example does.

### 4.3.2 Listeners vs. WaitSets

Both the simple and advanced examples use *WaitSets* to block a thread until data is available. This is the safest way to get data, because it does not affect any middleware threads (it is blocking the application’s main thread until data is available). In the advanced C# example, `TakeAsync()` is used instead of a *WaitSet*, which allows iterating over an asynchronous stream of data samples as they are received.

In addition, the advanced example installs **Listeners** on both the *DataReader* and *DataWriter* with callbacks that you can implement to accomplish a desired behavior. These **Listener** callbacks are triggered for various events, such as discovering a matched *DataWriter* or *DataReader*. **Listener** callbacks are called back from a middleware thread, which you should not block. There are benefits to using **Listeners** for non-data callbacks, because you will not miss events. However, if you block or do slow processing in a **Listener**, it can cause undesired behavior such as data loss. See the **Listeners** chapter, in the [RTI Connext Core Libraries User's Manual](https://www.rti.com/products/rti-connext-core-libraries-users-manual).
4.4 Using Generated Types Without Connext (Standalone)

You can use the generated type-specific source and header files without linking the Connext libraries or even including the Connext header files. That is, the generated files for your data types can be used standalone.

The directory `<NDDSHOME>resource/app/app_support/rtiddsgen/standalone/include` contains the helper files required to work in standalone mode:

- include: header and templates files for C/C++
- src: source files for C/C++
- class: Java jar file

4.4.1 Using Standalone Types in C

The generated files that can be used standalone are:

- `<idl file name>.c`: Types source file
- `<idl file name>.h`: Types header file

You cannot use the type plug-in (`<idl file>Plugin.c `<idl file>Plugin.h`) or the type support (`<idl file>Support.c `<idl file>Support.h`) code standalone.

To use the `rtiddsgen`-generated types in a standalone manner:

- Include the directory `<NDDSHOME>resource/app/app_support/rtiddsgen/standalone/include` in the list of directories to be searched for header files.
- Add the source files `ndds_standalone_type.c` and `<idl file name>.c` to your project.
- Include the file `<idl file name>.h` in the source files that will use the generated types in a standalone way.
- Compile the project using the two following preprocessor definitions:
  - `NDDS_STANDALONE_TYPE`
  - The definition for your platform: `RTI_VXWORKS`, `RTI_QNX`, `RTI_WIN32`, `RTI_INTY`, `RTI_LYNX` or `RTI_UNIX`

4.4.2 Using Standalone Types in C++

The generated files that can be used standalone are:
4.4.3 Using Standalone Types in Java

- `<idl file name>.cxx` : Types source file
- `<idl file name>.h` : Types header file

You cannot use the type plugin (`<idl file>Plugin.cxx `<idl file>Plugin.h`) or the type support (`<idl file>Support.cxx `<idl file>Support.h`) code standalone.

To use the generated types in a standalone manner:

- Include the directory `<NDDSHOME>resource/app/app_support/rtiddsgen/standalone/include` in the list of directories to be searched for header files.
- Add the source files `ndds_standalone_type.cxx` and `<idl file name>.cxx` to your project.
- Include the file `<idl file name>.h` in the source files that will use the generated types in a standalone way.
- Compile the project using the two following preprocessor definitions:
  - `NDDS_STANDALONE_TYPE`
  - The definition for your platform: `RTI_VXWORKS`, `RTI_QNX`, `RTI_WIN32`, `RTI_INTY`, `RTI_LYNX` or `RTI_UNIX`

### 4.4.3 Using Standalone Types in Java

The generated files that can be used standalone are:

- `<idl type>.java`
- `<idl type>Seq.java`

You cannot use the type code (`<idl file>TypeCode.java`), the type support (`<idl type>TypeSupport.java`), the data reader (`<idl file>DataReader.java`) or the data writer code (`<idl file>DataWriter.java`) standalone.

To use the generated types in a standalone manner:

- Include the file `ndds_standalone_type.jar` in the classpath of your project.
- Compile the project using the standalone types files (`<idl type>.java `<idl type>Seq.java`).
Chapter 5 Generated Files

The following tables show the files that Code Generator creates for an example IDL file called Hello.idl.

- Table 5.1 C and Traditional C++ Files Created for Example “Hello.idl”
- Table 5.2 Modern C++ Files Created for Example “Hello.idl”
- Table 5.3 C# Files Created for Example “Hello.idl”
- Table 5.4 Python Files Created for Example “Hello.idl”
- Table 5.5 Java Files Created for Example “Hello.idl”
- Table 5.6 Ada Files Created for Example “Hello.idl”

By default, Code Generator will not overwrite these files. You must use the -replace argument to do that.

Table 5.1 C and Traditional C++ Files Created for Example “Hello.idl”

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The following files are required for the user data type. The source files</td>
</tr>
<tr>
<td></td>
<td>should be compiled and linked with your application. The header files are</td>
</tr>
<tr>
<td></td>
<td>required to use the data type in source. You should not modify these files</td>
</tr>
<tr>
<td></td>
<td>unless you intend to customize the generated code supporting your type.</td>
</tr>
<tr>
<td>Hello.[c, cxx]</td>
<td>Generated code for the data types. These files contain the implementation</td>
</tr>
<tr>
<td>HelloSupport.[c, cxx]</td>
<td>for your data types.</td>
</tr>
<tr>
<td>HelloPlugin.[c, cxx]</td>
<td></td>
</tr>
<tr>
<td>Hello.h</td>
<td>Header files that contain declarations used in the implementation of your</td>
</tr>
<tr>
<td>HelloSupport.h</td>
<td>data types.</td>
</tr>
<tr>
<td>HelloPlugin.h</td>
<td></td>
</tr>
</tbody>
</table>
### Table 5.1 C and Traditional C++ Files Created for Example "Hello.idl"

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>application.h (traditional C++ only)</td>
<td>Helper code for the Hello_publisher.cxx and Hello_subscriber.cxx applications</td>
</tr>
<tr>
<td>Hello_publisher.[c, cxx]</td>
<td>Example code for an application that publishes the user data type. This example shows the basic steps to create all of the DDS objects needed to send data. You will need to modify the code to set and change the values being sent in the data structure. Otherwise, just compile and run.</td>
</tr>
<tr>
<td>Hello_subscriber.[c, cxx]</td>
<td>Example code for an application that subscribes to the user data type. This example shows the basic steps to create all of the DDS objects needed to receive data. No modification of this file is required. It is ready for you to compile and run.</td>
</tr>
<tr>
<td>Hello-&lt;architecture&gt;.sln</td>
<td>Microsoft Visual Studio solution and project files, generated only for Visual Studio-based architectures. (An example &lt;architecture&gt; is x64Win64VS2017.) To compile the generated source code, open the workspace file and build the two projects.</td>
</tr>
<tr>
<td>Hello_publisher-&lt;architecture&gt;.v[c, cx]proj</td>
<td>Makefile for non-Visual-Studio-based architectures. An example &lt;architecture&gt; is arm-v8Linux4gcc7.3.0.</td>
</tr>
<tr>
<td>Hello_subscriber-&lt;architecture&gt;.v[c, cx]proj</td>
<td>Makefile for non-Visual-Studio-based architectures. An example &lt;architecture&gt; is arm-v8Linux4gcc7.3.0.</td>
</tr>
<tr>
<td>makefile_Hello-&lt;architecture&gt;</td>
<td>Makefile for non-Visual-Studio-based architectures. An example &lt;architecture&gt; is arm-v8Linux4gcc7.3.0.</td>
</tr>
<tr>
<td>USER_QOS_PROFILES.xml</td>
<td>The Quality of Service (QoS) configuration of the DDS entities in the generated example is loaded from this file.</td>
</tr>
</tbody>
</table>

### Table 5.2 Modern C++ Files Created for Example "Hello.idl"

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello.cxx</td>
<td>Generated code for the data types. These files contain the implementation for your data types.</td>
</tr>
<tr>
<td>HelloPlugin.cxx</td>
<td>Generated code for the data types. These files contain the implementation for your data types.</td>
</tr>
<tr>
<td>Hello_hpp</td>
<td>Header files that contain declarations used in the implementation of your data types.</td>
</tr>
<tr>
<td>HelloPlugin_hpp</td>
<td>Header files that contain declarations used in the implementation of your data types.</td>
</tr>
<tr>
<td>Hello_publisher.cxx</td>
<td>Example code for an application that publishes the user data type. This example shows the basic steps to create all of the DDS objects needed to send data. You will need to modify the code to set and change the values being sent in the data structure. Otherwise, just compile and run.</td>
</tr>
<tr>
<td>application_hpp</td>
<td>Helper code for the Hello_publisher.cxx and Hello_subscriber.cxx applications</td>
</tr>
</tbody>
</table>

The following optional files are generated when you use the `-example <architecture>` command-line option. You may modify and use these files as a way to create simple applications that publish or subscribe to the user data type.
### Table 5.2 Modern C++ Files Created for Example “Hello.idl”

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello_subscriber.cxx</td>
<td>Example code for an application that subscribes to the user data type. This example shows the basic steps to create all of the DDS objects needed to receive data. No modification of this file is required. It is ready for you to compile and run.</td>
</tr>
<tr>
<td>Hello-&lt;architecture&gt;.sln</td>
<td>Microsoft Visual Studio solution and project files, generated only for Visual Studio-based architectures. (An example <code>&lt;architecture&gt;</code> is x64Win64VS2017.) To compile the generated source code, open the workspace file and build the two projects.</td>
</tr>
<tr>
<td>Hello_publisher-&lt;architecture&gt;.vcxproj</td>
<td>Makefile for non-Visual-Studios-based architectures. An example <code>&lt;architecture&gt;</code> is arm-v8Linuxxgcc7.3.0.</td>
</tr>
<tr>
<td>Hello_subscriber-&lt;architecture&gt;.vcxproj</td>
<td>The Quality of Service (QoS) configuration of the DDS entities in the generated example is loaded from this file.</td>
</tr>
</tbody>
</table>

### Table 5.3 C# Files Created for Example “Hello.idl”

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello.cs</td>
<td>The following files are required for the user data type. The source files should be compiled with your .NET project. You should not modify these files.</td>
</tr>
<tr>
<td>HelloPlugin.cs</td>
<td>Generated code for the data types. These files contain the implementation for your data types.</td>
</tr>
<tr>
<td>HelloProgram.cs</td>
<td>Code for running the HelloPublisher.cs and HelloSubscriber.cs applications</td>
</tr>
<tr>
<td>HelloPublisher.cs</td>
<td>Example code for an application that publishes the user data type. This example shows the basic steps to create all of the DDS objects needed to send data. You will need to modify the code to set and change the values being sent in the data structure. Otherwise, just compile and run.</td>
</tr>
<tr>
<td>HelloSubscriber.cs</td>
<td>Example code for an application that subscribes to the user data type. This example shows the basic steps to create all of the DDS objects needed to receive data. No modification of this file is required. It is ready for you to compile and run.</td>
</tr>
<tr>
<td>Hello.csproj</td>
<td>Used to build and run your application. NuGet.Config tells the .NET compiler where to find the Rti.ConnexxDds NuGet package.</td>
</tr>
<tr>
<td>NuGet.Config</td>
<td>The Quality of Service (QoS) configuration of the DDS entities in the generated example is loaded from this file.</td>
</tr>
<tr>
<td>USER_QOS_PROFILES.xml</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.4 Python Files Created for Example “Hello.idl”

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello.py</td>
<td>Generated code for the data types. These files contain the definition of your data types.</td>
</tr>
</tbody>
</table>

The following files are required for the user data type. You should not modify these files.

The following optional files are generated when you use the `--example <platform>` command-line option (for Python the only valid value of `<platform>` is `universal`). You may modify and use these files as a way to create simple applications that publish or subscribe to the user data type.

- **Hello_publisher.py**
  - Example code for an application that publishes the user data type. This example shows the basic steps to create all of the DDS objects needed to send data.
  - You can modify the code to set and change the values being sent in the data structure.

- **Hello_subscriber.py**
  - Example code for an application that subscribes to the user data type. This example shows the basic steps to create all of the DDS objects needed to receive data.
  - No modification of this file is required. It is ready for you to run.

- **Hello_program.py**
  - Code for running the `Hello_publisher.py` and `Hello_subscriber.py` applications. Run ”python Hello_program.py <options>”.

- **USER_QOS_PROFILES.xml**
  - The Quality of Service (QoS) configuration of the DDS entities in the generated example is loaded from this file.

Table 5.5 Java Files Created for Example “Hello.idl”

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>Hello.java</td>
<td>Class associated with the constant</td>
</tr>
<tr>
<td>Enums</td>
<td>Hello.java</td>
<td>Class associated with enum type</td>
</tr>
</tbody>
</table>
| Structures/Unions | Hello.java, HelloSeq.java, HelloDataReader.java, HelloDataWriter.java, HelloTypeSupport.java | Structure/Union class  
Sequence class  
DDS `DataReader` and `DataWriter` classes  
Support (serialize, deserialize, etc.) class |
| Typedef of sequences or arrays | Hello.java, HelloSeq.java, HelloTypeSupport.java | Wrapper class  
Sequence class  
Support (serialize, deserialize, etc.) class |

Since the Java language requires individual files to be created for each class, *Code Generator* will generate a source file for every IDL construct that translates into a class in Java.
### Table 5.5 Java Files Created for Example “Hello.idl”

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last structure/union</td>
<td>HelloPublisher.java HelloSubscriber.java</td>
<td>Example code for applications that publish or subscribe to the user data type. You should modify the code in the publisher application to set and change the value of the published data. Otherwise, both files should be ready to compile and run.</td>
</tr>
<tr>
<td></td>
<td>makefile_Hello_.&lt;architecture&gt;</td>
<td>Makefile for non-Windows-based architectures. An example &lt;architecture&gt; is armv8Linux4gcc7.3.0.</td>
</tr>
<tr>
<td></td>
<td>USER_QOS_PROFILES.xml</td>
<td>The Quality of Service (QoS) configuration of the DDS entities in the generated example is loaded from this file.</td>
</tr>
<tr>
<td>Structures/Unions/Typedefs/Enums</td>
<td>HelloTypeCode.java</td>
<td>Type code class associated with the IDL type, Hello</td>
</tr>
</tbody>
</table>

### Table 5.6 Ada Files Created for Example “Hello.idl”

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello[.h,.c]</td>
<td>Generated code for the data types, which contain the implementation for the data types, and header files that contain declarations used in the implementation of the data types.</td>
</tr>
<tr>
<td>HelloSupport[.h,.c]</td>
<td></td>
</tr>
<tr>
<td>HelloPlugin[.h,.c]</td>
<td></td>
</tr>
<tr>
<td>hello_idi_file[.adb,.ads]</td>
<td></td>
</tr>
<tr>
<td>hello_idi_file-hello_datawriter.ads</td>
<td>DataReader and DataWriter classes and serialize/deserialize methods.</td>
</tr>
<tr>
<td>hello_idi_file-hello_datareader.ads</td>
<td></td>
</tr>
<tr>
<td>hello_idi_file-hello_typesupport[.adb,.ads]</td>
<td></td>
</tr>
<tr>
<td>hello_idi_file-hello_metatypesupport[.adb,.ads]</td>
<td>These files are generated only for types that support Zero Copy transfer over shared memory (that is, are annotated with @transfer_mode(SHMEM_REF) in the IDL file).</td>
</tr>
<tr>
<td>hello_idi_file-hello_publisher[.adb,.ads] (in the samples/ directory)</td>
<td>Example code for an application that publishes the user data type. You will need to modify the code to set and change the values being sent in the data structure. Otherwise, just compile and run. The subscriberlistener file implements the on_data_available() callback.</td>
</tr>
<tr>
<td>hello_idi_file-hello_subscriber[.adb,.ads] (in the samples/ directory)</td>
<td></td>
</tr>
<tr>
<td>hello_idi_file-hello_subscriberlistener[.adb,.ads] (in the samples/ directory)</td>
<td></td>
</tr>
<tr>
<td>hello.gpr</td>
<td>Project files using Ada-like syntax. These files define the build-related characteristics of the application. These characteristics include the list of sources, the location of those sources, the location of the generated object files, the name of the main program, and the options for the various tools involved in the build process. Each of them is for a different set of files (hello-samples is for the examples, hello_c is for the c files and hello is for rest of the ada files.)</td>
</tr>
<tr>
<td>hello_c.gpr</td>
<td></td>
</tr>
<tr>
<td>hello-samples.gpr (in the samples directory)</td>
<td></td>
</tr>
</tbody>
</table>
The following optional file is generated when you use the `-example <architecture>` command-line option. You may modify and use this file as a way to create simple applications that publish or subscribe to the user data type.

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_QOS_PROFILES.xml</td>
<td>The Quality of Service (QoS) configuration of the DDS entities in the generated example is loaded from this file.</td>
</tr>
</tbody>
</table>
Chapter 6 Customizing the Generated Code

*Code Generator* allows you to customize the generated code for different languages by changing the provided templates. This version does not allow you to create new output files.

You can load new templates using the following command in an existing template, where `<pathToTemplate>` is the absolute path to the folder with the customized template:

```
#parse("<pathToTemplate>/template.vm")
```

If that *template.vm* file contains macros, you can use it within the original template. If *template.vm* contains just plain text without macros, that text will be included directly in the original file.

You can customize the behavior of a template by using the predefined set of variables provided with *Code Generator*. For more information, see the tables in *RTI_rtidsgen_template_variables.xlsx*.

This file contains two different sheets: Language-Templates and Template variables. The Language-Template sheet shows the correspondence between the Velocity Templates used and the generated files for each language. If, for example, we want to add a method in C in the *Hello.c* file, we would need to modify the template *typeBody.vm* under the *templates/c* directory.

The scope of a template can be:

- **type**: If we generate a file with that template for each type in the IDL file. For example in Java, where we generate a TypeSupport file for each type in the IDL.
- **file**: If we generate a file with that template for each IDL file. For example in C, we generate a single plugin file containing all the types Plugin information.
- **lastTopLevelType**: If we generate a file with that template for the last top-level type in the IDL file. This is commonly used for the publisher/subscriber examples.
• **module**: If we generate a file with that template for each module in the IDL file. This is used in Ada, where there are files that contain all the types of a module.

• **topLevelType**: if we generate a file with that template for each type in the idl file. This is used in ADA where the publisher/subscriber files are only generated for top level types.

The table also shows the top_level variables that can be used for that templates. These variables are explained in the sheet Template variables. For example in Java, the main unit of variables are the constructMap which is a hashMap of variables that represent a type. In C, we will have as the main unit the constructMapList, which is a List of constructMap. In the Template variables sheet, we can see which variables are contained in each constructMap, the constructKind or type that it is applicable to and the value that it contains depending on the language we use.

One important variable that contains the constructMap for a type is the memberFieldMapList. This list represent the members contained within the type. Each member is also represented as a hashMap whose variables are also described in the Template variables sheet.

Apart from that there are environmental or general variables that are not related with the types that are defined within a hashMap called envMap.

Let’s see how to use these variables with an example. Suppose we want to generate a method in C that prints the members for a structure and, if it is an array or sequence, its corresponding size. For this IDL:

```c
module Mymodule{
    struct MyStruct{
        int32 longMember;
        int32 arrayMember [2][100];
        sequence<char,2> sequenceMember;
        sequence <int32, 5> arrayOfSequenceMember[28];
    };
}
```

We want to generate this:

```c
void MyModule_MyStruct_specialPrint(){
    printf(" longMember \n");
    printf(" arrayMember is an array [2, 100] \n ");
    printf(" sequenceMember is a sequence <2> \n");
    printf(" arrayOfSequenceMember is an array [28] is a sequence <5> ");
}
```

The code in the template would look like this:

```c
## We go through all the list of types
#foreach ($node in $constructMapList)
## We only want the method for structs
#*-*#*if ($node.constructKind.equals("struct"))
void $node.nativeFQName}_specialPrint(){
## We go through all the members and call to the macros that check if they are array or sequences
#*-*-*#foreach($member in $node.memberFieldMapList)
```
print("$member.name #isAnArray($member) #isASeq($member) \n");
*/----*##end
}  
#*-*##end
#end

The **isAnArray** macro checks if the member is an array (i.e, has the variable **dimensionList**) and in that case, prints it:

```
#macro (isAnArray $member)
#if($member.dimensionList) is an array $member.dimensionList #end
#end
```

The **isASeq** macro checks if the member is a sequence (i.e, has the variable **seqSize**) and in that case, prints it:

```
#macro (isASeq $member)
#if($member.seqSize) is a sequence <$member.seqSize> #end
#end
```

You can add new variables to the templates using the `-V <name>[=<<value>>]` command-line option when starting Code Generator. This variable will be added to the userVarList hashMap. You can refer to it in the template as **$userVarList.name** or **$userVarList.name.equals(value)**.

For more information on velocity templates, see [https://velocity.apache.org/engine/1.5/user-guide.html](https://velocity.apache.org/engine/1.5/user-guide.html).
Chapter 7 Optimizing the Code Generation Process

The cost of serialization and deserialization operations increases with type complexity and sample size. It can become a significant contributor to the latency required to send and receive a sample. Code Generator provides the command-line option -optimization, which can be used to indicate the level of optimization of the serialize/deserialize operations. This command-line option allows selecting one of three different levels.

7.1 Optimization Levels

0: No optimization

1: rtiddsgen generates extra code for typedefs but optimizes its use. If a type that is used is a typedef that can be resolved to a primitive, enum, or aggregated type (struct, union, or value type), the generated code will invoke the code of the most basic type to which the typedef can be resolved. This level can be used if the generated code for typedef is not expected to be modified. This is the only optimization level supported for Java and C# languages.

For example:

```c
typedef int32 Latitude;
typedef int32 Latitude;

struct Position {
    Latitude x;
    Longitude y;
};
```

With optimization 0, the serialization of a sample with type Position will require calling the serialize methods for Latitude and Longitude. For example:

```c
LatitudePlugin_serialize(...) {
    serialize_long(...)
}
```
With optimization 1, *rtiddsgen* resolves Latitude and Longitude to their most primitive types for serialization purposes, resulting in a more efficient serialization. In this case, *rtiddsgen* will save two function/method calls.

```
Position Serialize(...) {
  serialize_long(...)
  serialize_long(...)
}
```

2: This optimization level is the default if not specified. (You can also explicitly specify it.) This optimization level applies only to C, C++, C++11, microC, microC++, and Ada languages. With this optimization level, *rtiddsgen* optimizes the serialization/deserialization of structures and valuetypes by using more aggressive techniques. These techniques include inline expansion of nested types and serialization/deserialization of a set of consecutive members with a single copy function invocation (memcpy) when the memory layout (C, C++ structure layout) is the same as the wire layout (XCDR).

### 7.2 How the Optimizations are Applied

In *Code Generator*, the optimizations (*inline expansion of nested types and serialization of consecutive members with a single copy*) are related. Inline expansion of a nested structure is only done when the C/C++ memory layout with standard packing of the structure matches the XCDR layout. (In this case, the structure’s members can be serialized with a single memcpy.) If the C/C++ memory layout with standard packing of the structure matches the XCDR layout, then *rtiddsgen* tries first to do the inline expansion, then the serialization of consecutive members with a single copy.

#### 7.2.1 Inline expansion of nested types

Inline expansion is an optimization in which *Code Generator* replaces a type definition with another one in which nested types are flattened out. This is done to remove extra function calls during serialization/deserialization. For example:

```
struct Point {
  int32 x;
  int32 y;
};

struct Dimension {
  int32 height;
  int32 width;
};
```
7.2.2 Serialization of consecutive members with a single copy

```c
struct Rectangle {
    Point leftTop;
    Dimension size;
};
```

With optimization level 2, `Code Generator` replaces the definition of Rectangle with the following equivalent definition:

```c
struct Rectangle {
    int32 leftTop_x;
    int32 leftTop_y;
    int32 size_height;
    int32 size_width;
};
```

This optimization is only done for serialization/deserialization. The generated type in C/C++ continues using Point and Dimension.

### 7.2.2 Serialization of consecutive members with a single copy

In the previous Rectangle example, `Code Generator`, using optimization level 2, further optimizes the serialization and deserialization by serializing a Rectangle sample with a single copy operation (memcpy) instead of four.

Before optimization:

```c
Rectangle_serialize(...) {
    memcpy(..., 4) // leftTop_x
    memcpy(..., 4) // leftTop_y
    memcpy(..., 4) // size_height
    memcpy(..., 4) // size_width
}
```

After optimization:

```c
Rectangle_serialize(...) {
    memcpy(..., 16) // leftTop_x
}
```

This optimization is only applicable when the memory layout of the C/C++ structure is equivalent to the serialization layout, which uses the XCDR version 1 or version 2 format.

### 7.2.3 Rules for Inline Expansion

To be inlinable, a structure 'MyStruct' has to meet the following two requirements:

- It has to have a C/C++-friendly XCDR layout.
- No members of 'MyStruct' should be marked with the @min, @max, or @range annotations.
A struct/valuetype 'MyStruct' has a C/C++-friendly XCDR layout when all of the following conditions apply:

- MyStruct is marked as @final or @appendable when the data representation is XCDR version 1. Mutable structures are not inlinable.
- MyStruct does not have a base type.
- MyStruct contains only primitive members, or complex members composed only of primitive members. A primitive member is a member with any of the following types: int8\(^1\), uint8\(^2\), int16, int32, int64, uint16, uint32, uint64, float, double, octet, and char. The following primitive types are not supported for inlining purposes: long double, wchar, boolean, enum.

```c
struct Dimension {
    int32 height;
    int32 width;
}; // Inlinable
```

```c
struct Dimension {
    string label; // Inlinable structures cannot contain strings
    int32 height;
    int32 width;
}; // Not Inlinable
```

- With any initial alignment (1, 2, 4, 8) greater than the alignment of the first member of the struct, there is no padding between the members that are part of MyStruct. To apply this rule, consider these alignments and sizes for primitive types:

### Table 7.1 Alignments and Sizes for Primitive Types

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Alignment (bytes)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8(^3)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>uint8(^4)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>uint16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int32</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

\(^1\)This type is supported only at the API level. It is still considered an octet for type matching purposes.

\(^2\)This type is supported only at the API level. It is still considered an octet for type matching purposes.

\(^3\)This type is supported only at the API level. It is still considered an octet for type matching purposes.

\(^4\)This type is supported only at the API level. It is still considered an octet for type matching purposes.
### 7.2.3 Rules for Inline Expansion

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Alignment (bytes)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint32</code></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><code>int64</code></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><code>uint64</code></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><code>float</code></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><code>double</code></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><code>octet</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>char</code></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
struct Dimension {
    int32 height;
    int16 width;
}; // Inlinable. Independently of the alignment of the starting memory address (4 or 8), there is no padding between height and width

struct Dimension {
    int16 height;
    int32 width;
}; // Not Inlinable. Starting in a memory address aligned to 4 will require adding a padding of two bytes between height and width
```

- With any initial alignment (1, 2, 4, 8) greater than the alignment of the first member of the struct, there is no padding between the elements of an array of MyStruct.

```c
struct Dimension {
    int32 height;
    int16 width;
}; // Not inlinable. Let's assume an array of two dimensions Dimension[2]. If the array starts in a memory address aligned to 4, there would be padding between the first and the second element of the array

struct Dimension {
    int32 height;
    int16 width;
    int16 padding;
}; // Inlinable
```

For serialization and deserialization purposes, *Code Generator* will consider an inlinable structure (according to the previous rules) as a primitive array where the alignment of the primitive type corresponds to the alignment of the first member of the structure. A member with type ‘MyStruct’ will be serialized with a single copy (memcpy) invocation.

When *Code Generator* serializes the members of a data structure, it will also try to coalesce the serialization of consecutive primitive members into a single copy operation if possible. *Code Generator*
only applies this optimization when the alignment of the next member is equal to or smaller than the alignment of the current member.

```c
struct Dimension {
    int16 height;
    int32 width;
}; // Coalescing not possible because the alignment of width 4 is greater than the alignment of height 2

struct Dimension {
    int32 width;
    int16 height;
}; // Coalescing is possible because the alignment of width 4 is greater than the alignment of height 2
```

### 7.2.4 Guidelines

As a rule of thumb, to take advantage of optimization level 2 for types containing only primitive types:

- Order the members in descending alignment order (this will help with copy coalescing).
- For XCDR version 2 encapsulation, use `@final` extensibility if your types will not evolve. For XCDR version 1 encapsulation, use `@final` or `@appendable` if possible (this will help with inline expansion).
- If you use ContentFilteredTopics, it is recommended that fields that appear in the filter expression are placed at the beginning of the type.
Chapter 8 Boosting Performance

If you need to invoke Code Generator multiple times with different parameters and/or type files, there will be a performance penalty derived from loading the Java Virtual Machine (JVM) and compiling the velocity templates every time you invoke Code Generator. If this is your scenario, you can run Code Generator in server mode to avoid doing this process multiple times. Or, if you would like to reduce the JVM startup and execution time, use the JVM optimization provided with Code Generator, for a reduced time in code generation. These two options (server mode and JVM optimization) cannot be used together.

8.1 Using Server Mode

One way to boost performance is to run Code Generator in server mode. Server mode runs a native executable that opens a TCP connection to a server instance of the code generator that is spawned the first time the executable is run, as depicted below:
To invoke Code Generator in server mode, use the script `rtiddsgen_server.bat`, which is in the `scripts` directory.

The default port Code Generator server attaches to is 14662. If you want to modify the port Code Generator server is attached to, use the `-n_serverport <port>` argument. Note that the Code Generator server can use up to three ports; make sure you have two free ports after the specified one.

The Code Generator server offers a log-to-file option that you can enable by using the argument `-n_logfilepath <log directory>`. In the specified log directory, the Code Generator server will create a file named `CodegenServerLog<portNumber>.txt` containing all logging messages from the server side.

Code Generator server comes with built-in timeouts, some of which you can change:

- When Code Generator is used in server mode, JVM is loaded a single time when the server is started; the velocity templates are also compiled a single time. The server will wait up to 5 seconds for Code Generator to initialize. You can change this value by specifying the number of milliseconds using the argument `-n_connectiontimeout <time in milliseconds>`.

- The Code Generator server will automatically stop if it is not used (that is, if it does not receive any calls) for a certain amount of time. The default value is 20 seconds; you can change this by editing the `rtiddsgen_server` script and adjusting the value of the argument `-n_servlettimeout <time in milliseconds>`.

- Code Generator server sends a handshake message to the client after accepting the connection. In the client, there is a timeout when waiting for the handshake message. If this message is not
received in the client in a short amount of time (10 seconds, an internal value that cannot be changed), the Code Generator client will time out. This timeout might mean that there was another application running in the port.

Notes:

• Mixing different versions of Code Generator server is not supported. See Limitations in the RTI Code Generator Release Notes.

• Code Generator server cannot be parallelized. Each execution of Code Generator server is attached to a port where it receives requests, and it can only generate code for one request at a time. Therefore, if you try to send multiple requests simultaneously, Code Generator server will process them sequentially.

8.2 Using JVM Optimization

Java Virtual Machine (JVM) offers different options that help improve its performance. RTI applies some of these options when creating the Code Generator JVM, resulting in an improvement in the Code Generator execution time. Because these options are non-standard, they are disabled by default. If you are using the JRE shipped with Connext, you can enable these options; otherwise, enabling these options is at your own risk.

The options applied to Code Generator JVM are as follows:

• -XX:+TieredCompilation
• -XX:TieredStopAtLevel=1
• -XX:ClCompilerCount=4
• -Xverify:none
• -XX:+UseParallelGC
• -XX:+OptimizeStringConcat
• -XX:CompileThreshold=5000

For more information on these options, see the JDK documentation: https://docs.oracle.com/en/java/javase/11/. (Currently, you can find most of the options described in the Tools Reference, in "Main Tools to Create and Build Applications" > "java").

To enable the performance improvement associated with the Code Generator JVM options, set the RTIDDGEN_JVM_Optimization environment variable to true. To disable the improvement, unset the environment variable. (By default, it is not set.)