RTI Code Generator

User's Manual

Version 3.0.1

© 2020 Real-Time Innovations, Inc.
All rights reserved.
Printed in U.S.A. First printing.
March 2020.

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 under and subject to the RTI software license agreement. The software may be used or copied only under the terms of the license agreement.

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/).

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

RTI® Code Generator creates the code needed to define and register a user data type with *RTI Connext® DDS*.

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 DDS Core Libraries User's Manual).
- You are using one of the built-in types (see Built-in Data Types (Section 3.2) in the RTI Connext DDS Core Libraries User's 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 RTI Connext DDS Core Libraries User's Manual (Sections 3.3 and 3.4).

Chapter 2 Paths Mentioned in Documentation

The documentation refers to:

- **<NDDSHOME>**

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

  - macOS® systems:
    /Applications/rti_connext_dds-6.0.1
  - Linux systems, non-*root* user:
    /home/<*your user name*>/rti_connext_dds-6.0.1
  - Linux systems, *root* user:
    /opt/rti_connext_dds-6.0.1
  - Windows® systems, user without Administrator privileges:
    <*your home directory*>\rti_connext_dds-6.0.1
  - Windows systems, user with Administrator privileges:
    C:\Program Files\rti_connext_dds-6.0.1

  You may also see **$NDDSHOME** 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-6.0.1\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/6.0.1/examples**
- Linux systems: **/home/<*your user name*>/rti_workspace/6.0.1/examples**
- Windows systems: ***<your Windows documents folder>*\rti_workspace\6.0.1\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 DDS Installation Guide*.

Chapter 3 Command-Line Arguments for rtiddsgen

On Windows systems: Before running *rtiddsgen*, run **VCVARS32.BAT** in the same command prompt that you will use to run *rtiddsgen*. The **VCVARS32.BAT** file is usually located in **<*Visual Studio Installation Directory*>/VC/bin**. Alternatively, run *rtiddsgen* from the Visual Studio Command Prompt under the Visual Studio Tools folder.

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

```
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>]
                 [-express]
                 [-I <directory>]
                 [[-inputIdl] <IDLInputFile.idl> | [-inputXml] <XMLInputFile.xml>
                  |[-inputXsd <IDLInputFile.idl>]]
                 [-language  <Ada|C|C++|C++03|C++11|C++/CLI|C#|Java>]
                 [-legacyPlugin]
                 [-namespace]
                 [-obfuscate]
                 [-optimization <level>]
                 [-package <packagePrefix>]
                 [-platform <architecture>]
                 [-ppDisable]
                 [-ppPath <path to preprocessor>]
                 [-ppOption <option>]
```

```
                [-qualifiedEnumerator]
                [-reader]
                [-replace]
                [-sequenceSize <unbounded sequences size>]
                [-sharedLib]
                [-stringSize  <unbounded strings size>]
                [-U <name>]
                [-unboundedSupport]
                [-update <typefiles|   examplefiles|makefiles>]
                [-use52Keyhash]
                [-use526Keyhash]
                [-useStdString]
                [-V <name< [=<value>]]
                [-verbosity [1-3]]
                [-version]
                [-virtualDestructor]
                [-writer]
```

If you have *RTI CORBA Compatibility Kit*, you can use the above options, plus these:

```
                [-corba [CORBA Client header file]]
                [-dataReaderSuffix <suffix> ]
                [-dataWriterSuffix <suffix> ]
                [-orb <CORBA ORB>]
                [-typeSequenceSuffix <suffix> ]
```

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

```
rtiddsgen       [-help]
                [-create <typefiles|   examplefiles|makefiles>]
                [-convertToIdl | -convertToXML]
                [-D <name>[=<value>]]
                [-d <outdir>]
                [-enableEscapeChar]
                [-I <directory>]
                [[-inputIdl] <IDLInputFile.idl> | [-inputXml] <XMLInputFile.xml>]
                [-language <C|C++>]
                [-micro]
                [-namespace]
                [-ppDisable]
                [-ppPath <path to preprocessor>]
                [-ppOption <option>]
                [-reader]
                [-replace]
                [-sequenceSize <unbounded sequences size>]
                [-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.

Table 3.1 Options for rtiddsgen

Option	Description
-allocateWithMalloc	Use this flag to obtain backward-compatibility when allocating optional members with DDS_Heap_malloc in C++.
-alwaysUseStdVector	Only applies if -language <C|C++> is specified and -legacyPlugin is not 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.
-autoGenFiles <*architecture*>	Updates the auto-generated files, i.e, the typefiles and makefile/project files. To see the valid options for <*architecture*> per language, run *rtiddsgen* with the **-help** option, or use the string "universal" (**-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 <architecture>`
-constructor	Only applies if **-language C++** 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.
-create <typefiles| examplefiles|makefiles>	Creates the files indicated (typefiles, examplefiles, or makefiles) if they do not exist. If the files already exist, the files are not modified and a warning is printed. There can be multiple **-create** options. If both **-create** and **-update** are specified for the same file type, only **-update** will be applied.
-convertToIdl	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.
-convertToXML	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.
-convertToXsd	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.
-corba [*CORBA Client header file*]	This option is only available when using the *RTI CORBA Compatibility Kit for Connext DDS* (available for purchase as a separate product and described in the *RTI Connext DDS Core Libraries User's Manual*).
-D <*name*>[=<*value*>]	Defines preprocessor macros. On Windows systems, enclose the argument in quotation marks: **-D "<*name*>[=<*value*>]"**
-d <*outdir*>	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.

Table 3.1 Options for rtiddsgen

Option	Description
-dataReaderSuffix <*suffix*>	Only applies if -corba [CORBA Client header file] is also specified. Assigns a suffix to the name of a *DataReader* interface. By default, the suffix is **DataReader**. Therefore, given the type **Foo**, the name of the *DataReader* interface will be **FooDataReader**.
-dataWriterSuffix <*suffix*>	Only applies if -corba [CORBA Client header file] is also specified. Assigns a suffix to the name of a DataWriter interface. By default, the suffix is **DataWriter**. Therefore, given the type **Foo**, the name of the DataWriter interface will be **FooDataWriter**.
-disableXSDValidation	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.
-dllExportMacroSuffix <suffix>	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_<Suffix>.
-enableEscapeChar	Enables use of the escape character '_' in IDL identifiers.
-example <*architecture*>	Generates type files, example files, and a makefile. This is a shortcut for: `-create typefiles -create examplefiles -create makefiles -platform <architecture>` To see the valid options for <*architecture*> per language, run *rtiddsgen* with the **-help** option, or use the string "universal" (**-example universal**) to generate compatible publisher/subscriber code for all supported platforms. The universal architecture will not generate makefiles/project files.
-express	Generates the C# project files needed to build with Microsoft Visual Studio Express. This option is only compatible with architecture i86Win32VS2010. Newer versions of Microsoft Visual Studio Express Edition do not need this flag.
-help	Prints out the command-line options for *rtiddsgen*.
-I <*directory*>	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.
-inputIdl	Indicates that the input file is an IDL file, regardless of the file extension.
-inputXml	Indicates that the input file is an XML file, regardless of the file extension.
-inputXsd	Indicates that the input file is an XSD file, regardless of the file extension.
IDLInputFile.idl	A file containing IDL descriptions of your data types. If -inputIdl is not used, the file must have a '**.idl**' extension.

Table 3.1 Options for rtiddsgen

Option	Description
For *Connext DDS Core*: -language <Ada|C|C++|C++03|C++11|C++/CLI|C#|Java> For *Connext DDS Micro*: -language <C|C++>	Specifies the language to use for the generated files. The default language is C++.
-legacyPlugin	Only applies if **-language C++03** or **-language C++11** is also specified. Generates code that uses *Connext DDS*-specific types in the type definition and type plugin. The default behavior when this option is *not* specified is to map to STL (Standard Template Library) types, as follows: • Unbounded sequences map to **std::vector** (also requires -unboundedSupport) • Bounded sequences map to **rti::core::bounded_sequence** (or **std::vector** if -alwaysUseStdVector is specified) • Strings map to **std::string**, wide strings map to **std::wstring**, wide characters map to **wchar_t**. • Members annotated with @external and pointers map to **dds::core::external**.
-micro	Generates code and support files for *Connext DDS Micro*, instead of *Code Generator*. Use **-micro -help** to list the options supported by *Code Generator* when targeting *RTI Connext DDS Micro*.
-namespace	Specifies the use of C++ namespaces. (For C++ only. For C++/CLI and C#, it is implied–namespaces are always used.)
-obfuscate	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.
-orb <CORBA ORB>	Only applies if -corba [CORBA Client header file] is also specified. Specifies the CORBA ORB. The majority of generated code is independent of the ORB. However, for some IDL features, the generated code depends on the ORB. *Code Generator* generates code compatible with ACE-TAO or JacORB. To select an ACE_TAO version, use the **-orb** option. The default is ACE_TAO1.6.

Table 3.1 Options for rtiddsgen

Option	Description
-optimization <level>	Level of optimization of the code: • 0: No optimization. • 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, C#, and C++/CLI languages. • 2: (Default) This optimization level applies only to C, C++, C++03, 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). See Chapter 6 Optimizing the Code Generation Process on page 18 for more information. 2 is the default for C, C++, C++03, C++11, microC, microC++, and Ada languages (but you can change it to 0 or 1). 1 is always used for Java, C#, and C++/CLI languages, and you cannot change it.
-package <*packagePrefix*>	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.
-platform <*architecture*>	Required if **-create makefiles** or **-update makefiles** is used. To see the valid options for <*architecture*> per language, run *rtiddsgen* with the **-help** option, or use the string "universal" (**-platform universal**) to generate compatible publisher/subscriber code for all supported platforms. The universal architecture will not generate makefiles/project files.
-ppDisable	Disables the preprocessor.
-ppOption <*option*>	Specifies a preprocessor option. This option can be used multiple times to provide the command-line options for the specified preprocessor. See -ppPath <path to preprocessor>.

-ppPath <*path to preprocessor*>	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.
	If you use **-ppPath** to provide the full path and filename for **cl.exe** or the **cpp** preprocessor, you must also use -ppOption <option> to set the following preprocessor options:
	If you use a non-default path for **cl.exe**, you also need to set:
	`-ppOption /nologo -ppOption /C -ppOption /E -ppOption /X`
	If you use a non-default path for cpp, you also need to set:
	`-ppOption -C`

-qualifiedEnumerator	Uses the fully qualified name for enumerator values including the enum value.
-reader	Generates support for a *DataReader* (only with -micro).

Table 3.1 Options for rtiddsgen

Option	Description
-replace	Deprecated option. Instead, use -update <typefiles| examplefiles|makefiles> for the proper files (typefiles, examplefiles, makefiles). This option is maintained for backwards compatibility. It allows *Code Generator* to overwrite any existing generated files. If it is not present and existing files are found, *Code Generator* will print a warning but will not overwrite them.
-sequenceSize <*unbounded sequences size*>	Sets the size assigned to unbounded sequences. The default value is 100 elements.
-sharedLib	Generates makefiles that compile with the *Connext DDS* shared libraries (by default, the makefile will link with the static libraries)
-stringSize <*unbounded strings size*>	Sets the size assigned to unbounded strings, not counting a terminating NULL character. The default value is 255 bytes.
-typeSequenceSuffix <*suffix*>	This option can only be used with the -corba [CORBA Client header file] option. Assigns a suffix to the names of the implicit sequences defined for IDL types. By default, the suffix is **Seq**. Therefore, given the type 'Foo' the name of the implicit sequence will be **FooSeq**.
-U <*name*>	Cancels any previous definition of <*name*>.
-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 DDS Core Libraries User's Manual*: • "Sequences" • "Strings and Wide Strings" • "DDS Sample-Data and Instance-Data Memory Management"
-update <typefiles| examplefiles|makefiles>	Creates the files indicated if they do not exist. If the files already exist, this overwrites the files without printing a warning. There can be multiple **-update** options. If both **-create** and **-update** are specified for the same file type, only the **-update** will be applied.
-use52Keyhash	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).
-use526Keyhash	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).

Table 3.1 Options for rtiddsgen

Option	Description
-useStdString	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.
-V <*name*< [=<*value*>]	Defines a user variable that can be used in the templates as **$userVarList.name** or **$user-VarList.name.equals(value)**. The variables defined with this option are case sensitive.
-verbosity [1-3]	Sets the *Code Generator* verbosity: 1: Exceptions 2: Exceptions and warnings 3: Exceptions, warnings and information (Default)
-version	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).
-virtualDestructor	Only applies if **-language C++** 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.
-writer	Generates support for a *DataWriter* (only with -micro).
XMLInputFile.xml	A file containing XML descriptions of your data types. If -inputXml is not used, the file must have an **.xml** extension.

Note: Before using a makefile created by *Code Generator* to compile an application, make sure the **${NDDSHOME}** environment variable is set as described in Step 1, Set up the Environment, in the RTI Connext DDS Core Libraries Getting Started Guide.

Chapter 4 Generated Files

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

- Table 4.1 C, C++, C++/CLI, C# Files Created for Example "Hello.idl"
- Table 4.2 Java Files Created for Example "Hello.idl"
- Table 4.3 Ada Files Created for Example "Hello.idl"

Table 4.1 C, C++, C++/CLI, C# Files Created for Example "Hello.idl"

<table>
<tr><th>Generated Files</th><th>Description</th></tr>
<tr><td colspan="2">The following files are required for the user data type. The source files should be compiled and linked with your application. The header files are required to use the data type in source.
You should not modify these files unless you intend to customize the generated code supporting your type.</td></tr>
<tr><td>Hello.[c,cxx, cpp]
HelloSupport.[c, cxx, cpp]
HelloPlugin.[c,cxx, cpp]</td><td>Generated code for the data types. These files contain the implementation for your data types.</td></tr>
<tr><td>Hello.h
HelloSupport.h
HelloPlugin.h</td><td>Header files that contain declarations used in the implementation of your data types.</td></tr>
<tr><td colspan="2">The following optional files are generated when you use the -example <<i>architecture</i>> 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.</td></tr>
<tr><td>Hello_publisher.[c, cxx, cpp, 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>Hello_subscriber.[c, cxx, cpp,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 using a "listener" function.
No modification of this file is required. It is ready for you to compile and run.</td></tr>
</table>

Table 4.1 C, C++, C++/CLI, C# Files Created for Example “Hello.idl”

Generated Files	Description
Hello.sln, Hello_publisher.v[c, cs, cx]proj, Hello_subscriber.v[c, cs, cx]proj	Microsoft Visual Studio solution and project files, generated only for Visual Studio-based architectures. To compile the generated source code, open the workspace file and build the two projects.
makefile_Hello_<*architecture*>	Makefile for non-Visual-Studios-based architectures. An example <*architecture*> is i86Linux2.6gcc4.4.5.

Table 4.2 Java Files Created for Example “Hello.idl”

Data Type	Generated Files	Description
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.		
Constants	Hello.java	Class associated with the constant
Enums	Hello.java	Class associated with enum type
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
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.		
Structures/Unions	HelloPublisher.java HelloSubscriber.java	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.
	makefile_Hello_<*architecture*>	Makefile for non-Windows-based architectures. An example <*architecture*> is i86Linux2.6gcc4.4.5jdk.
Structures/Unions/ Typedefs/Enums	HelloTypeCode.java	Type code class associated with the IDL type, Hello

Table 4.3 Ada Files Created for Example “Hello.idl”

Generated Files	Description
Hello[.h,.c]	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.
HelloSupport[.h,.c]	
HelloPlugin[.h,.c]	
hello_idl_file[.adb, .ads]	
hello_idl_file-hello_datawriter.ads	DataReader and DataWriter classes and serialize/deserialize methods.
hello_idl_file-hello_datareader.ads	
hello_idl_file-hello_typesupport[.adb,.ads]	
hello_idl_file-hello_metptypesupport[.adb,.ads]	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).
hello_idl_file-hello_publisher[.adb,.ads] (in the **samples**/ directory)	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.
hello_idl_file-hello_subscriber[.adb,.ads] (in the **samples**/ directory)	
hello_idl_file-hello_subscriberlistener[.adb,.ads] (in the **samples**/ directory)	
hello.gpr	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.)
hello_c.gpr	
hello-samples.gpr (in the **samples** directory)	

Chapter 5 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 relative to the **<NDDSHOME>/resource/app/app_support/rtiddsgen/templates** folder:

```
#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_rtiddsgen_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:

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

We want to generate this:

```
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:

```
## 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 an 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.

Chapter 6 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.

6.1 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, C#, and C++/CLI languages.

For example:

```
typedef long Latitude;
typedef long 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:

```
LatitudePlugin_serialize(...) {
    serialize_long(...)
}
```

```
LongitudePlugin_serialize(...) {
    serialize_long(...)
}

Position_serialize(...) {
    LatitudePlugin_serialize(...)
    LongitudePlugin_serialize(...)
}
```

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++03, 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).

6.2 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.

6.2.1 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 {
    long x;
    long y;
};

struct Dimension {
    long height;
    long width;
};
```

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

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

```
struct Rectangle {
    long leftTop_x;
    long leftTop_y;
    long size_height;
    long size_width;
};
```

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

6.2.2 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:

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

After optimization:

```
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.

6.2.3 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: 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.

```
struct Dimension {
    long height;
    long width;
}; // Inlinable

struct Dimension {
    string label; // Inlinable structures cannot contain strings
    long height;
    long 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 6.1 Alignments and Sizes for Primitive Types

Primitive Type	Alignment (bytes)	Size (bytes)
int16	2	2
uint16	2	2
int32	4	4
uint32	4	4
int64	8	8
uint64	8	8
float	4	4
double	8	8
octet	1	1
char	1	1

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

struct Dimension {
    short height;
    long 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.

```
struct Dimension {
    long height;
    short 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 {
    long height;
    short width;
    short 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.

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

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

6.2.4 Guidelines

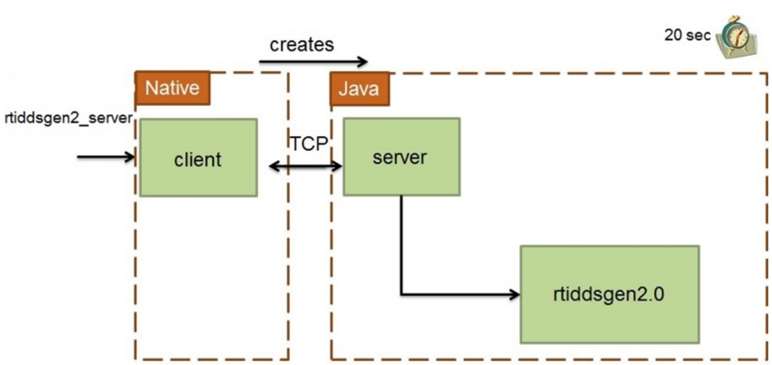
As a rule of thumb, to take advantage of optimization level 2 for types containing only primitive types:

- 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 7 Boosting Performance with Server Mode

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 JVM and compiling the velocity templates.

To help with the above scenario, you can 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.

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 with the parameter **-n_connectiontimeout**.

The *Code Generator* server will automatically stop if it is not used 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 parameter **-n_servertimeout**.