## Contents

1 Introduction.................................................................................................................................................................1
1.1 Paths Mentioned in Documentation .........................................................................................................................1
2 Command-Line Arguments for rtiddsgen........................................................................................................................3
3 Generated Files .................................................................................................................................................................8
4 Customizing the Generated Code.....................................................................................................................................11
5 Boosting Performance with Server Mode.......................................................................................................................13
Using RTI Code Generator

1 Introduction

RTI Code Generator creates the code needed to define and register a user data type with 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).

1.1 Paths Mentioned in Documentation

The documentation refers to:

❏ <NDDSHOME>

This refers to the installation directory for Connext DDS.

The default installation paths are:

• Mac OS X systems:
  /Applications/rti_connext_dds-version

• UNIX-based systems, non-root user:
  /home/your user name/rti_connext_dds-version

• UNIX-based systems, root user:
  /opt/rti_connext_dds-version

• Windows systems, user without Administrator privileges:
  <your home directory>ti_connext_dds-version

1. This document is provided with RTI Connext DDS. You can also access it from the RTI Community’s Documentation page.
• Windows systems, user with Administrator privileges:
  
  C:\Program Files\rti_connext_dds-version (for 64-bits machines) or
  C:\Program Files (x86)\rti_connext_dds-version (for 32-bit machines)

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-version\bin\rtiddsgen"

or if you have defined the NDDSHOME environment variable:

"%NDDSHOME%\bin\rtiddsgen"

• RTI Workspace directory, rti_workspace

The RTI Workspace is where all configuration files for the applications and example files are located. All configuration files and examples are copied here the first time you run RTI Launcher or any script in <NDDSHOME>/bin. The default path to the RTI Workspace directory is:

• Mac OS X systems:

  /Users/your user name/rti_workspace

• UNIX-based systems:

  /home/your user name/rti_workspace

• Windows systems:

  your Windows documents folder\rti_workspace

  Note: 'your Windows documents folder' depends on your version of Windows. For example, on Windows 7, the folder is C:\Users\your user name\Documents; on Windows Server 2003, the folder is C:\Documents and Settings\your user name\Documents.

You can specify a different location for the rti_workspace directory. See the RTI Connext DDS Core Libraries Getting Started Guide for instructions.

• <path to examples>

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

By default, the examples are copied to rti_workspace/version/examples

So the paths are:

• Mac OS X systems:

  /Users/your user name/rti_workspace/version/examples

• UNIX-based systems:

  /home/your user name/rti_workspace/version/examples
2 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 ]
    [ -autoGenFiles <architecture> ]
    [ -create <typefiles| examplefiles|makefiles> ]
    [ -convertToIdl | -convertToXML ]
    [ -D <name> [=value> ] ]
    [ -d <outdir> ]
    [ -enableEscapeChar ]
    [ -example <architecture> ]
    [ -express ]
    [ -I <directory> ]
    [ [-inputId] <IDLInputFile.idl> | [-inputXml] <XMLInputFile.xml> ]
    [ -language <Ada|C|C++|C++03|C++11|C#|Java> ]
    [ -namespace ]
    [ -noCopyable ]
    [ -notypecode ]
    [ -obfuscate ]
    [ -package <packagePrefix> ]
    [ -platform <architecture> ]
    [ -ppDisable ]
    [ -ppPath <path to preprocessor> ]
    [ -ppOption <option> ]
    [ -reader ]
    [ -replace ]
    [ -sequenceSize <unbounded sequences size> ]
    [ -sharedLib ]
    [ -stringLiteral <unbounded strings size> ]
    [ -U <name> ]
    [ -unboundedSupport ]
    [ -update <typefiles| examplefiles|makefiles> ]
    [ -use42eAlignment ]
    [ -V <name> [=value> ] ]
    [ -verbosity [1-3] ]
    [ -version ]
    [ -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> ]
Command-Line Arguments for rtiddsgen

[-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 2.1 describes the options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-autoGenFiles &lt;architecture&gt;</td>
<td>Updates the auto-generated files, i.e, the typefiles and makefile/project files. Valid options for &lt;architecture&gt; are listed in the RTI Connext DDS Core Platform Notes, or you can replace &lt;architecture&gt; with the string <code>universal</code> (-autoGenFiles universal) to generate compatible publisher/subscriber code for all supported platforms. The <code>universal</code> architecture will not generate makefiles/project files. This is a shortcut for: -update typefiles -update makefiles -platform &lt;architecture&gt;</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>-corba [CORBA Client header file]</td>
<td>This option is only available when using the RTI CORBA Compatibility Kit for Connext (available for purchase as a separate product and described in the RTI Connext DDS Core Libraries User’s Manual).</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-D &lt;name&gt;[=value&gt;]</td>
<td>Defines preprocessor macros. On Windows systems, enclose the argument in quotation marks: <code>-D &quot;&lt;name&gt;[=value&gt;]&quot;</code></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>-dataReaderSuffix &lt;suffix&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-dataWriterSuffix &lt;suffix&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-enableEscapeChar</td>
<td>Enables use of the escape character '[_]' in IDL identifiers.</td>
</tr>
<tr>
<td>-example &lt;architecture&gt;</td>
<td>Generates type files, example files, and a makefile. This is a shortcut for: <code>-create typefiles -create examplefiles -create makefiles -platform &lt;architecture&gt;</code> Valid options for &lt;architecture&gt; are listed in the RTI Connext DDS Core Libraries Platform Notes or you can replace &lt;architecture&gt; with the string <code>universal</code> (-example universal) to generate compatible publisher/subscriber code for all supported platforms. The <code>universal</code> architecture will not generate makefiles/project files.</td>
</tr>
<tr>
<td>-express</td>
<td>Generates the C# project files needed to build with Microsoft Visual Studio Express. This option is only compatible with architectures i86Win32VS2008 and i86Win32VS2010. Newer versions of Microsoft Visual Studio Express Edition do not need this flag.</td>
</tr>
<tr>
<td>-I &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>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>-inputXml</td>
<td>Indicates that the input file is an XML file, regardless of the file extension.</td>
</tr>
<tr>
<td>-help</td>
<td>Prints out the command-line options for rtiddsgen.</td>
</tr>
<tr>
<td>For Connext DDS Core: -language</td>
<td>Specifies the language to use for the generated files. The default language is C++.</td>
</tr>
<tr>
<td>&lt;Ada</td>
<td>C</td>
</tr>
<tr>
<td>-micro</td>
<td>Generates code and support files for Connext DDS Micro, instead of Connext Professional. Use -micro -help to list the options supported by Code Generator when targeting RTI Connext DDS Micro.</td>
</tr>
</tbody>
</table>
Table 2.1  **Options for rtiddsgen**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-namespace</code></td>
<td>Specifies the use of C++ namespaces. (For C++ only. For C++/CLI and C#, it is implied—namespaces are always used.)</td>
</tr>
<tr>
<td><code>-noCopyable</code></td>
<td>Forces Code Generator to put ‘copy’ logic into the corresponding TypeSupport class, rather than the type itself. This option is only used for Java code generation.</td>
</tr>
<tr>
<td><code>-notypecode</code></td>
<td>Disables type-code support.</td>
</tr>
<tr>
<td></td>
<td>If you are using a large data type (more than 64 K) and type-code support, you will see a warning when type-code information is sent. Connext DDS has a type-code size limit of 64K. To avoid the warning when working with data types with type codes larger than 64K, turn off type-code support by using <code>-notypecode</code>. Type-code support must be enabled if you are going to use ContentFilteredTopics with the default SQL filter.</td>
</tr>
<tr>
<td><code>-obfuscate</code></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>
<tr>
<td><code>-orb &lt;CORBA ORB&gt;</code></td>
<td>Only applies if <code>&lt;corba [CORBA Client header file]&gt;</code> is also specified.</td>
</tr>
<tr>
<td></td>
<td>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 <code>-orb</code> option. The default is ACE_TAO1.6.</td>
</tr>
<tr>
<td><code>-package &lt;packagePrefix&gt;</code></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><code>-platform &lt;architecture&gt;</code></td>
<td>Required if <code>-create makefiles</code> or <code>-update makefiles</code> is used.</td>
</tr>
<tr>
<td></td>
<td>Valid options for <code>&lt;architecture&gt;</code> are listed in the RTI Connext DDS Core Libraries Platform Notes or you can replace <code>&lt;architecture&gt;</code> with the string <code>universal</code> (<code>-platform universal</code>) to generate compatible publisher/subscriber code for all supported platforms. The <code>universal</code> architecture will not generate makefiles/ project files.</td>
</tr>
<tr>
<td><code>-ppDisable</code></td>
<td>Disables the preprocessor.</td>
</tr>
<tr>
<td><code>-ppOption &lt;option&gt;</code></td>
<td>Specifies a preprocessor option. This option can be used multiple times to provide the command-line options for the specified preprocessor. See <code>-ppPath &lt;path to preprocessor&gt;</code>.</td>
</tr>
<tr>
<td><code>-ppPath &lt;path to preprocessor&gt;</code></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 <code>cpp</code> for non-Windows architectures and <code>cl.exe</code> for Windows architectures.</td>
</tr>
<tr>
<td></td>
<td>If you use <code>-ppPath</code> to provide the full path and filename for <code>cl.exe</code> or the <code>cpp</code> preprocessor, you must also use <code>-ppOption &lt;option&gt;</code> to set the following preprocessor options:</td>
</tr>
<tr>
<td></td>
<td>If you use a non-default path for <code>cl.exe</code>, you also need to set:</td>
</tr>
<tr>
<td></td>
<td><code>-ppOption /nologo -ppOption /C -ppOption /E -ppOption /X</code></td>
</tr>
<tr>
<td></td>
<td>If you use a non-default path for <code>cpp</code>, you also need to set:</td>
</tr>
<tr>
<td></td>
<td><code>-ppOption -C</code></td>
</tr>
<tr>
<td><code>-reader</code></td>
<td>Generates support for a DataReader (only with <code>-micro</code>).</td>
</tr>
</tbody>
</table>
Command-Line Arguments for rtiddsgen

Table 2.1 Options for rtiddsgen

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 DDS shared libraries (by default, the makefile will link with the static libraries)</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>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.</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. 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 de-allocated and memory of sufficient size to hold the deserialized data will be allocated. When initially constructed, sequences will not pre-allocate any elements, thus starting with a maximum of zero elements. To use -unboundedSupport, you must also:  
  - Specify the -language option as C++/CLI, C#, C, or C++.  
  - Set the properties dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size on the DataWriter and dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size on the DataReader.  
  - Set the QoS reader_resource_limits.dynamically_allocate_fragmented_samples on the DataReader to true. |
| -update <typefiles | examplefiles | makefiles> | Creates the files indicated if they do not exist. Is 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. |
| -use42eAlignment | Makes the generated code compatible with RTI Data Distribution Service 4.2e. This option should be used when compatibility with 4.2e is required and the topic data types contain double, long long, unsigned long long, or long double members. |
| -V <name< [=<value>] | 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. |
Generated Files

3 Generated Files

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

- C, C++, C++/CLI, C# Files Created for Example “Hello.idl” (Table 3.1)
- Java Files Created for Example “Hello.idl” (Table 3.2)
- Ada Files Created for Example “Hello.idl” (Table 3.3)

Table 3.1 C, C++, C++/CLI, C# Files Created for Example “Hello.idl”

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

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>-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>
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.

Hello_publisher.[c, cxx, cpp, cs]
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.

Hello_subscriber.[c, cxx, cpp, cs]
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.

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>
<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>
<tr>
<td>Structures/Unions</td>
<td>Hello.java, HelloSeq.java,</td>
<td>Structure/Union class</td>
</tr>
<tr>
<td></td>
<td>HelloDataReader.java,</td>
<td>Sequence class</td>
</tr>
<tr>
<td></td>
<td>HelloDataWriter.java,</td>
<td>DDS DataReader and DataWriter classes</td>
</tr>
<tr>
<td></td>
<td>HelloTypeSupport.java</td>
<td>Support (serialize, deserialize, etc.) class</td>
</tr>
<tr>
<td>Typedef of</td>
<td>Hello.java, HelloSeq.java,</td>
<td>Wrapper class</td>
</tr>
<tr>
<td>sequences or arrays</td>
<td>HelloTypeSupport.java</td>
<td>Sequence class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Support (serialize, deserialize, etc.) class</td>
</tr>
</tbody>
</table>

Table 3.1 C, C++, C++/CLI, C# Files Created for Example “Hello.idl”

Table 3.2 Java Files Created for Example “Hello.idl”
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

(Note: this is not generated if you use `-notypecode`)

Type code class associated with the IDL type, Hello

---

### Table 3.2  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></td>
<td>HelloPublisher.java</td>
<td>Example code for applications that publish or subscribe to the user data type.</td>
</tr>
<tr>
<td></td>
<td>HelloSubscriber.java</td>
<td></td>
</tr>
<tr>
<td></td>
<td>makefile_Hello_&lt;architecture&gt;</td>
<td></td>
</tr>
</tbody>
</table>

---

### Table 3.3  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_idl_file[.adb,.ads]</td>
<td></td>
</tr>
<tr>
<td>hello_idl_file-hello_datawriter[.adb,.ads]</td>
<td>DataReader and DataWriter classes and serialize/deserialize methods.</td>
</tr>
<tr>
<td>hello_idl_file-hello_datareader[.adb,.ads]</td>
<td></td>
</tr>
<tr>
<td>hello_idl_file-hello_typesupport[.adb,.ads]</td>
<td></td>
</tr>
<tr>
<td>hello_idl_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_idl_file-hello_subscriber[.adb,.ads] (in the samples/ directory)</td>
<td></td>
</tr>
<tr>
<td>hello_idl_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>
4 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:

```java
#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 that 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{
        long longMember;
    }
    struct ComplexStruct{
        int field1;
        struct MyStruct field2;
    }
}
```

One example of a template that generates such a method is the `ComplexStructFieldMemberDetails` template. This template generates a method that prints the members of a struct and checks if it is an array or sequence. The template looks like this:

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

This template uses the predefined set of variables provided with *Code Generator*. For example, the `memberFieldMapList` variable contains all the members of a struct. The `constructMapList` variable contains all the constructMaps for each type in the IDL file. The `constructKind` variable contains the kind of each type in the IDL file. The `type` variable contains a list of types in the IDL file. The `module` variable contains a list of modules in the IDL file. The `topLevelType` variable contains a list of top-level types in the IDL file. The `envMap` variable contains environmental or general variables.

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 that 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.
long arrayMember[2][100];
sequence<char,2> sequenceMember;
sequence<long,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:

```c
#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:

```c
#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 hash map. You can refer to it in the template as `$userVarList.name` or `$userVarList.name.equals(value)`.

For more information on velocity templates, see [http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html](http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html).
5 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:

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.

To invoke Code Generator in server mode, use the script rtiddsgen_server(.bat), which is in the scripts directory.

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_servetimeout.