

RTI Connex DDS

Core Libraries

Getting Started Guide

Addendum for Android Systems

Version 5.3.0



© 2017 Real-Time Innovations, Inc.

All rights reserved.

Printed in U.S.A. First printing.

June 2017.

Trademarks

Real-Time Innovations, RTI, NDDS, RTI Data Distribution Service, DataBus, Connex, Micro DDS, 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.

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 Android Systems	1
Chapter 2 Installing and Configuring for Android Development	
2.1 Prerequisites	3
2.2 Installing Connex DDS for Android Development	4
2.3 Setting Up a Virtual Device or Using Hardware Devices for Connection	4
Chapter 3 Android Development Model	5
Chapter 4 Using Connex DDS Utilities on an Android Platform	
4.1 RTI DDS Spy App	7
4.2 RTI DDS Ping App	8
4.3 RTI DDS Prototyper App	9
4.4 Generating Example Code and a Makefile from IDL	11
Chapter 5 Using the Connex DDS Java Libraries with Eclipse	
5.1 Install the Android Plug-In	15
5.2 Create an Android Project	16
5.3 Edit the Project Source Code	19
5.4 Add the Connex DDS Libraries	21
5.5 Update the Android Manifest	24
5.6 Set Up Before Running Example	26
5.7 Run the Example	27
Chapter 6 Using the Connex DDS C and C++ Libraries with Eclipse	
6.1 Install the Android Plug-In	29
6.2 Create an Android Project	30
6.3 Edit the Project Source Code	33
6.4 Update the Android Manifest	37
6.5 Configure the Build	38
6.6 Set Up Before Running Example	41

6.7 Run the Example	41
Chapter 7 Building and Running a Hello World Example with C/C++	
7.1 Building and Running a C/C++ Application as an Executable	44
7.2 Building and Running a C/C++ Application as an Android App	45
Chapter 8 Building and Running a Hello World Example with Java	
8.1 Building and Running a Java Application as a Dalvik Command	51
8.2 Building and Running a Java Application as an Android App	53
Chapter 9 Additional Considerations	
9.1 Output when Running Connexrt DDS Applications on Android Devices and Emulators	58
9.1.1 DDS Log Messages from Core Libraries	58
9.1.2 Output from rtiddsgen-Generated Examples or User Applications	58
9.1.3 Output from Connexrt DDS Utilities	59
9.2 Communication between Connexrt DDS Applications on Emulator and Host	60
9.3 Loading XML-Specified QoS Settings	62
9.4 Using DDS C++ Libraries for Android Applications	63
9.5 Android Threads	64
9.6 Setting Permission in App Manifest	64

Chapter 1 Android Systems

This document provides instructions to install and configure *RTI® Connex® DDS* for Android™ development and run Connex DDS applications on Android systems. Please refer to the documentation provided by Google® for more information on the Android operating system and Android platform: <https://developer.android.com/develop/index.html>.

This section will guide you through the process of generating, compiling, and running a Hello World application on an Android system by expanding on "Generating Code with *rtiddsgen*" in the [RTI Connex DDS Core Libraries Getting Started Guide](#).

[Installing and Configuring for Android Development \(Chapter 2 on page 3\)](#) describes how to install and configure Connex DDS for Android development:

[Android Development Model \(Chapter 3 on page 5\)](#) gives an overview of the Android development model:

If you want to use the Connex DDS Ping, Spy, or Prototyper utilities, please read [Using Connex DDS Utilities on an Android Platform \(Chapter 4 on page 7\)](#)

The next sections guide you through the steps to generate, modify, build, and run an example HelloWorld application on an Android target. You can do so build the App two ways: using Eclipse or using a makefile created by *RTI Code Generator (rtiddsgen)*. In both cases, you will use *rtiddsgen* to generate example source code.

To build with Eclipse, see either:

- [Using the Connex DDS Java Libraries with Eclipse \(Chapter 5 on page 15\)](#)
- [Using the Connex DDS C and C++ Libraries with Eclipse \(Chapter 6 on page 29\)](#)

To build manually with a makefile, see:

- [Generating Example Code and a Makefile from IDL \(Section 4.4 on page 11\)](#)
- [Building and Running a Hello World Example with Java \(Chapter 8 on page 51\)](#)
- [Building and Running a Hello World Example with C/C++ \(Chapter 7 on page 43\)](#)

The last section describes some additional considerations for Android development:

- [Additional Considerations \(Chapter 9 on page 58\)](#)

Chapter 2 Installing and Configuring for Android Development

2.1 Prerequisites

1. Install an Android SDK (Software Development Kit), available from <https://developer.android.com/sdk/index.html>. The released Connex DDS libraries and applications were built with r22. Later SDKs should also work. Use an ADT bundle, which includes the SDK and a plug-in for Eclipse. Use of the Eclipse plug-in is optional. This document describes development with and without Eclipse.
2. If you plan to develop an Android App in C or C++, install an Android NDK (Native Development Kit), available from <https://developer.android.com/tools/sdk/ndk/index.html>. The released Connex DDS libraries were built with r9. Later NDKs should also work. If you are using a Windows machine as the host, Cygwin needs to be installed (available from <http://www.cygwin.com/>) and <CYGWINHOME>/bin should be in your **Path**.
3. If you plan to build an Android App (**.apk**) using the SDK or you plan to use the Connex DDS Java API, Java is required. Java JDK 1.6 was used to build this Connex DDS release. Note that the Dalvik VM (used by the Android operating system) is partially Java 1.6 compliant. Using Java 1.7 or 1.8 tools to compile class files may lead to problems if the Java code uses language features that are not supported by the Dalvik VM. (See Android documentation for further and up-to-date information).
4. This document assumes you will use either Eclipse or SDK command-line tools for building and running your Android App. Both Eclipse and the command-line builds with the SDK require Ant. This Connex DDS release was built using Ant 1.8.2. (Eclipse includes Ant; you can check the Ant version by selecting **About Eclipse, Installation Details, Plug-ins.**)
5. If you are using a Windows host and you plan to build an Android App (**.apk**) using a makefile on the command line, you need to have Cygwin installed (available from <http://www.cygwin.com/>).

2.2 Installing Connex DDS for Android Development

You will need to install a host archive and a target archive. The host archive contains documentation, header files, and other files you need for development; the target archive contains the libraries and other files you need for deployment. Unpack them as described in the [RTI Connex DDS Core Libraries Getting Started Guide](#), Installing on a UNIX-Based System.

2.3 Setting Up a Virtual Device or Using Hardware Devices for Connection

You can run the Android operating system either on an actual Android device or on an Android Virtual Device (AVD). See Android documentation for details:

- <http://developer.android.com/tools/devices/index.html>
- <http://developer.android.com/tools/device.html>

To see if your Android device or emulator is attached to your host machine, run the following command:

```
$(ANDROID_SDK_HOME)/sdk/platform-tools/adb devices
```

where `$(ANDROID_SDK_HOME)` is the location of your Android SDK installation.

Chapter 3 Android Development Model

The Android operating system (OS) was developed by Google and is available as open source. It is based on the Linux OS. There are some omissions from common Linux distributions since the Android OS is designed for mobile devices.

Users may build an Android OS for any suitable hardware. Connex DDS will run on such a "bare Android OS" in the same way it runs on other Linux distributions—as static and shared libraries and executable utilities. Executable utilities and user executables linked with Connex DDS libraries can be passed arguments from the command-line and they will be effected by environment variable settings.

The most common Android platforms are "consumer devices" such as smart phones and tablets. For these devices, Android is more than an operating system. It provides a framework for Android Apps. These Apps run in a container called the Dalvik VM (Virtual Machine), which in turn runs as an operating system process. A Dalvik VM process starts whenever an Android App is launched (by touching its icon on a consumer device). Importantly, the App has no control over the environment variables defined when the Dalvik VM starts, nor can the App specify or receive command-line arguments.

The Android SDK comes with an emulator for running Android virtual devices. These virtual devices can be configured to mimic many Android consumer devices. An emulator has relaxed security requirements and so the remote shell ("adb shell") for an emulator does have root access. An emulator can run Android Apps and also run executables.

The Android SDK "adb shell" command provides a connection from a host terminal to the emulator and logs on as root. The emulator can be used for both modes of development: as a consumer device for Apps and as a customer self-built bare OS.

A consumer device can be "rooted" to provide a shell with root access. Once a device has direct access to the Android OS, it can run executables in the same way as a bare Android OS. A "rooted device" can still run Android Apps, since the application framework is still available.

On Linux and other UNIX-like OSs, the root user can gain access to the entire operating system and can do anything. By default, you do not have root access to your Android consumer device. It is possible to "root" an Android consumer device. Due to the many differences in configuration, such a device is not officially supported. However, any problems that can be reproduced on the emulator can be addressed as a support issue.

Some customers may have built their own Android OS from the Android open source. In this case, Android will behave like a Linux variant and provide the usual access to a command-line shell with environment variables.

Connex DDS has been tested on a consumer device (Nexus 7) and on an emulator. These are the officially supported platforms. It is expected that it will also work for a rooted device and bare Android platforms. The remainder of this document describes running Android Apps on consumer devices, and running Android Apps and executables on an emulator.

Chapter 4 Using Connex DDS Utilities on an Android Platform

You can use the Connex DDS *rtiddsspy*, *rtiddsping* and *rtiddsprototyper* utilities either as Android Apps or as command-line programs. When running as Android Apps on a real Android device, there is no command prompt for entering parameters and no shell for outputting data. So to facilitate their use, the utilities come with a simple GUI.

For information on using *rtiddsspy* and *rtiddsping* from the command-line, see the Connex DDS API Reference HTML documentation (under **Modules**, select **Programming Tools**). For information on *Prototyper*, see the *Prototyper with Lua Getting Started Guide*.

Before installing these utilities: The **adb install** command requires that you connect the device to a host machine and enable USB debugging in the device. For details on how to do this, see [these instructions \(http://developer.android.com/tools/device.html#setting-up\)](http://developer.android.com/tools/device.html#setting-up).

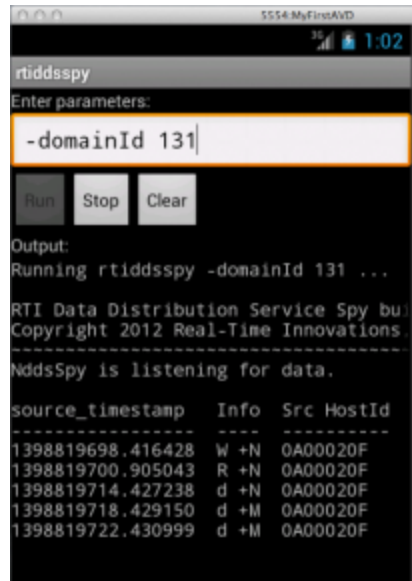
4.1 RTI DDS Spy App

To install and use *rtiddsspy* on an Android device:

1. Install *rtiddsspy* with this command (replace *<arch>* with your architecture name):

```
adb install $NDDSHOME/resource/app/bin/<arch>/rtiddsspy.apk
```

2. Touch or click the *rtiddsspy* icon.
3. On the displayed screen, enter parameters exactly as they would be entered on the command line when using *rtiddsspy* on other platforms.



- Click or touch **Run**. Data is displayed in the output panel.

You may need to scroll the panel to the right to see the full output.

Note that when running *rtiddsspy* as an App on Android, the **-printSample** option is *not* supported.

- Note that the **Run** button will be disabled until the Spy process is finished (i.e., if you have specified a **-timeout** value in the parameters). To stop a running Spy process gracefully, click or touch the **Stop** button. Once the Spy process is finished, the **Run** button will be enabled again.
- (Optional) Click or touch the **Clear** button to clear the current output.

To uninstall *rtiddspy* from the Android device through the **adb** shell, use this command:

```
adb uninstall com.rti.dds.rtiddsspy
```

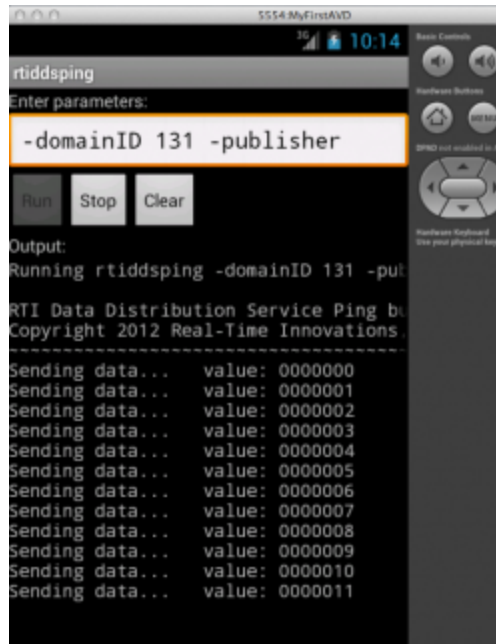
4.2 RTI DDS Ping App

To install and use *rtiddsping* on an Android device:

- Install *rtiddsping* with this command (replace *<arch>* with your architecture name)

```
adb install $NDDSHOME/resource/app/bin/<arch>/rtiddsping.apk
```

- Touch or click the *rtiddsping* icon.
- On the displayed screen, enter parameters exactly as they would be entered on the command line when using *rtiddsping* on other platforms.



4. Note that the **Run** button will be disabled until the Ping process is finished (i.e., if you have specified (in the parameters) a **-numSamples** value with the publisher, or a **-timeout** value with the subscriber). To stop a running Ping process gracefully, click or touch the **Stop** button. Once the ping process is finished, the **Run** button will be enabled again.
5. (Optional) Click or touch the **Clear** button to clear the current output.

To uninstall *rtiddsping* from the Android device through the **adb** shell, use this command:

```
adb uninstall com.rti.dds.rtiddsping
```

4.3 RTI DDS Prototyper App

To install and use *rtiddsprototyper* on an Android device:

1. Install *rtiddsprototyper* with this command (replace *<arch>* with your architecture name):

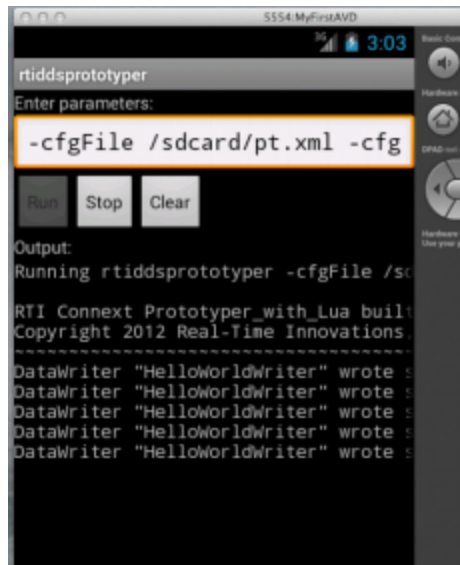
```
adb install $NDDSHOME/resource/app/bin/<arch>/rtiddsprototyper.apk
```

2. Enter the following (or you may specify your own configuration file). For example:

```
adb push <path to examples>/prototyper/hello_world/USER_QOS_PROFILES.xml \
  /sdcard/pt.xml
```

Note that the configuration file should be put in the Android device's external storage (see [Loading XML-Specified QoS Settings \(Section 9.3 on page 62\)](#)).

3. Touch or click the *rtiddsprototyper* icon.
4. On the displayed screen, enter parameters exactly as they would be entered on the command line when using *rtiddsprototyper* on other platforms. Note that **-cfgFile** must indicate a file pushed to the device with **adb**, and **-cfgName** must indicate a configuration within that file. If **-cfgName** is not given and there is more than one configuration in the file, all available configurations will be displayed to the output. (Please note that the interactive manner for selecting the configuration is not supported when used as an Android App.)



For example, enter the following as the parameters:

```
-cfgFile /sdcard/pt.xml
-cfgName MyParticipantLibrary::PublicationParticipant
```

5. Touch or click **Run**. Data is displayed in the output panel.
6. Note that the **Run** button will be disabled until the Prototyper process is finished (i.e., if you have specified the **-runDuration** value in the parameters). To stop a running Prototyper process gracefully, click or touch the **Stop** button. Once the Prototyper process is finished, the **Run** button will be enabled again.
7. Touch or click the **Clear** button to clear the current output.

When running *rtiddsprototyper* as an App on Android, the value of the sample that is received by the reader will *not* be printed (not in the UI and not in **logcat**).

To uninstall *rtiddsprototyper* from the Android device through the **adb** shell, use this command:

```
adb uninstall com.rti.dds.rtiddsprototyper
```

4.4 Generating Example Code and a Makefile from IDL

This section describes how to generate example code and a makefile from IDL using *RTI Code Generator* (*rtiddsgen*). The generated code will run without modification. The generated makefiles contain recipes for performing various Android SDK and NDK tasks to simplify the process of building and running a HelloWorld example. They also contain default settings for certain variables. These defaults are chosen to get the Hello World examples to run out of the box for as wide a range of SDK and NDK configurations as possible. However for real development, these default values should be reviewed.

To create the example applications:

1. Set up the environment on your development machine: set the NDDSHOME environment variable and update your PATH as described in the [RTI Connex DDS Core Libraries Getting Started Guide](#).
 - The PATH environment variable should also include the SDK's tools, build-tools and platform-tools directories. For example:

```
setenv ANDROID_SDK_HOME <path where Android SDK is installed>
setenv PATH $(ANDROID_SDK_HOME)/tools:$(ANDROID_SDK_HOME)/
    build-tools/<build tool version>:$(ANDROID_SDK_HOME)/
    platform-tools:$(PATH)
```

(Replace *<build tool version>* to match the one in your installation.)

- When building Android Apps (*.apk), the PATH should include the **bin** directory of Ant 1.8.2. For example:

```
setenv ANT_HOME <path where Ant is installed>
setenv PATH <ANT_HOME>/bin:$(PATH)
```

- Check to make sure that you have executable permission to **ant** in **<ANT_HOME>/bin/ant** or **<ANT_HOME>/bin/ant.bat**
- When building Java examples or Android Apps, the **PATH** should include JDK 1.6. For example:

```
setenv JAVA_HOME <path where Java is installed>
setenv PATH $(JAVA_HOME)/bin:$(PATH)
```

- When building C and C++ examples, both as executables and as Android Apps, the PATH should include the Android NDK. For example:

```
setenv ANDROID_NDK_PATH <path where Android NDK is installed>
setenv PATH $(ANDROID_NDK_PATH)/toolchains/<tool chain name>/
prebuilt/<host name>/bin:${PATH}
```

(Replace *<tool chain name>* and *<host name>* to match those in your installation.)

- Note that the Dalvik VM (used by the Android operating system) is partially Java 1.6 compliant. Using Java 1.7 or 1.8 tools to compile class files may lead to problems if the Java code uses language features that are not supported by the Dalvik VM. (See Android documentation for further and up-to-date information).
2. Create a directory to work in. In this example, we use a directory called **myhello**.
 3. In the **myhello** directory, create (or copy, see below) a file called **HelloWorld.idl** that contains a user-defined data type. For example:

```
const long HELLODDS_MAX_PAYLOAD_SIZE = 8192;
const long HELLODDS_MAX_STRING_SIZE = 64;
struct HelloWorld {
    string<HELLODDS_MAX_STRING_SIZE> prefix;
    long sampleId;
    sequence<octet, HELLODDS_MAX_PAYLOAD_SIZE> payload;
};
```

The above example IDL file can be copied from *<path to example>/connext_dds/java/hello_idl/src/HelloWorld.idl*, which can be used to test against a matching publisher and subscriber for other platforms, if desired.

4. Use *RTI Code Generator* to generate sample code and a makefile as described in the [RTI Connext DDS Core Libraries Getting Started Guide](#). Choose Java, C, or C++. For example:

```
$(NDDSHOME)/bin/rtiddsgen -language C -example <arch> HelloWorld.idl
$(NDDSHOME)/bin/rtiddsgen -language C++ -example <arch> HelloWorld.idl
$(NDDSHOME)/bin/rtiddsgen -language Java -example <arch>
    -package com.rti.hello.idl HelloWorld.idl
```

Replace *<arch>* (following **-example**) with a supported Android architecture name. For example, *<arch>* may be *armv7aAndroid2.3gcc4.8*. See the *RTI Connext DDS Core Libraries Platform Notes* for the supported Android architectures for each language.

Note that Android Apps (*.apk) require the Java code generated by *rtiddsgen* to be in a package. You can achieve this by using either the **modules** keyword in IDL or the **-package** command-line option with *rtiddsgen*. The IDL used here does not have modules, so the **-package** option is used instead.

5. See if you need to change any of the default settings in the generated makefile:

- **DEVICE**

The generated makefile assumes that computer on which you building and running these examples only has a single Android device or emulator attached. If this is not the case, you will need to set the **DEVICE** variable in the makefile. By default, this variable is not set. To see what devices are attached and what value to set **DEVICE** to, execute "**adb devices**" from the command line.

- **TARGET_ID**

If you are building an Android App (*.apk), **TARGET_ID** should be set to the "build target" for your application. It corresponds to an Android platform library that you would like to build your project against. Execute "**android list targets**" to print a list of available Android platforms that you have downloaded for your SDK and the corresponding ID. By default, this is set to 1.

- **ANDROID_PACKAGE_PREFIX**

If you are building an Android App (*.apk), this should be set to the Android package name prefix for the application. By default, this is set to **com.rti.example<language>.<IDL filename>**, which will eventually be used to generate two Android applications: one with package name **com.rti.<IDL filename>example.publisher** and another one with the package name **com.rti.<IDL filename>example.subscriber** (where <language> is Java, C, or C++; <IDL filename> is the name of the IDL file that you used for your example).

If you change the **ANDROID_PACKAGE_PREFIX** value, you will need to update the package name in these generated files:

- **objs/<arch>/publisher/src/com/rti/example<language>/<IDL filename>/publisher/<IDL filename>PublisherActivity.java**
 - **objs/<arch>/subscriber/src/com/rti/example<language>/<IDL filename>/subscriber/<IDL filename>SubscriberActivity.java**
 - **objs/<arch>/publisher/AndroidManifest.xml**
 - **objs/<arch>/subscriber/AndroidManifest.xml**
- **ANDROID_PUSH_DEST**

If you want to run the example directly on the emulator or device shell, this should be set to the location where files are copied to. By default, this is set to **/data/data/-com.rti.example<language>.<IDL filename>**, where <IDL filename> is the name of the IDL file that you are using.

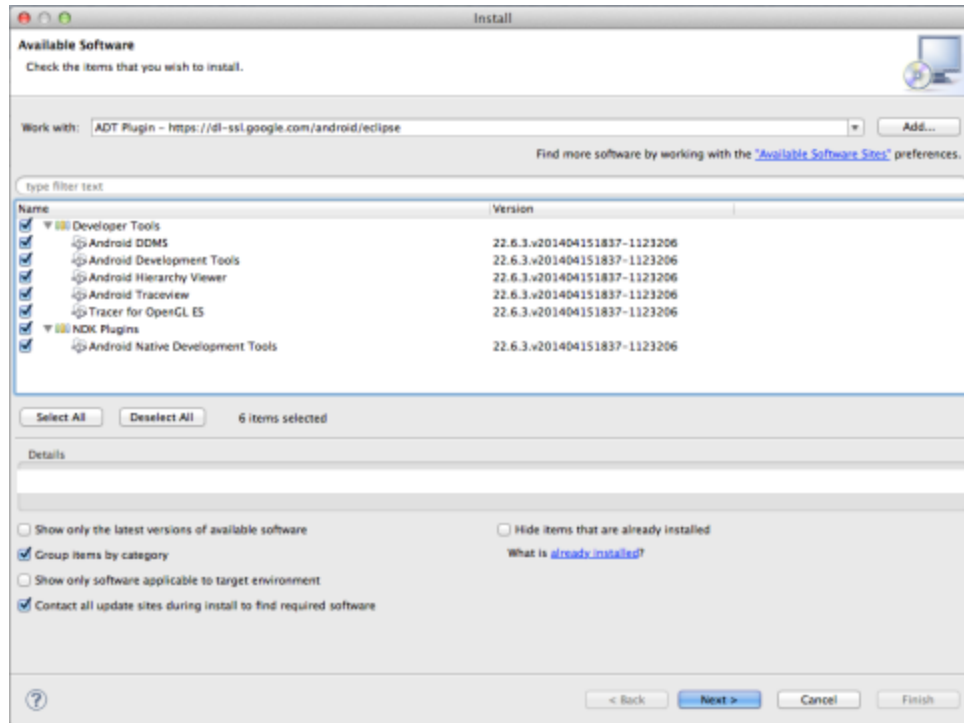
2. If you modify the publisher or subscriber code, the makefile, or QoS file, be careful if you re-run *rtiddsgen*: do not use **–replace** in conjunction with **–example**, as that would overwrite those modifications.

Chapter 5 Using the Connexx DDS Java Libraries with Eclipse

The Android Developer Tools bundle can be downloaded from Google. It contains the SDK and an Eclipse-based Android IDE. It is also possible to install an Android plug-in to an existing Eclipse installation. This section will describe how to use Android within an existing Eclipse installation to build a Connexx DDS application.

5.1 Install the Android Plug-In

1. Make sure the Android SDK is installed and accessible from the machine running Eclipse.
2. From the Eclipse menu bar, click **Help**, then **Install New Software ...**. In the resulting dialog, select the **site to work with** to **ADT Plugin** or add this URL: **<https://dl-ssl-google.com/android/eclipse>**. Check the items to install and click **Next >**.



3. Complete the install procedure. It may be necessary to "OK" the installation of unsigned software. It may also be necessary to point Eclipse to the Android SDK installation. At the end, restart Eclipse. The Eclipse menus should now have the Android Development Tools items.

5.2 Create an Android Project

This example uses the same HelloWorld IDL file used in [Generating Example Code and a Makefile from IDL \(Section 4.4 on page 11\)](#). The Android App built in this section can interoperate with those built in other sections of the document. If a different IDL file is used, then note that the following steps will need to be altered slightly as the IDL file name, "HelloWorld" is used in completing fields and naming other files in this example. This example shows only the creation of the Publisher App.

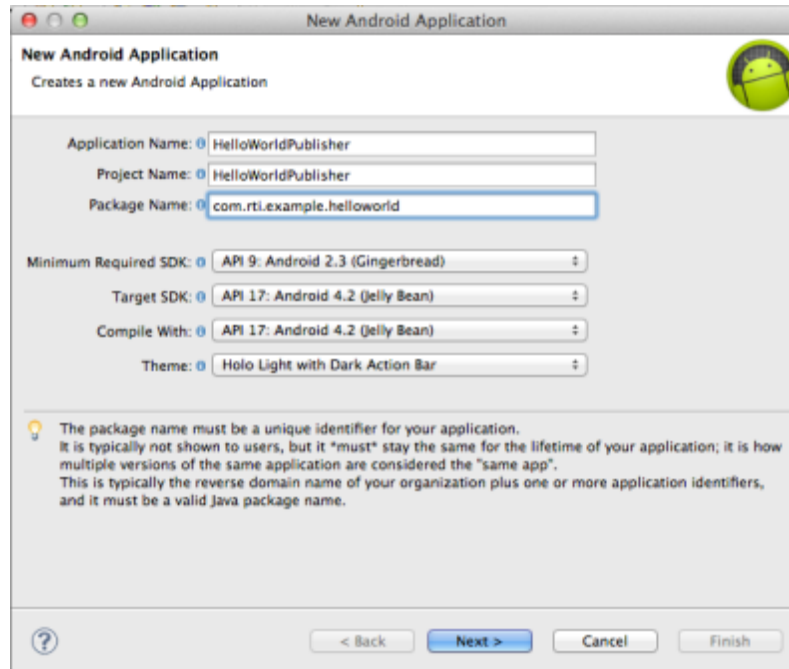
These steps can be repeated to create a HelloWorldSubscriber App. If the subscriber and the publisher will be installed on the same device, make sure the subscriber's Application, Project, and Package names are different than the publisher's.

1. From the Eclipse menu bar, select **File, New, Android Application Project**.

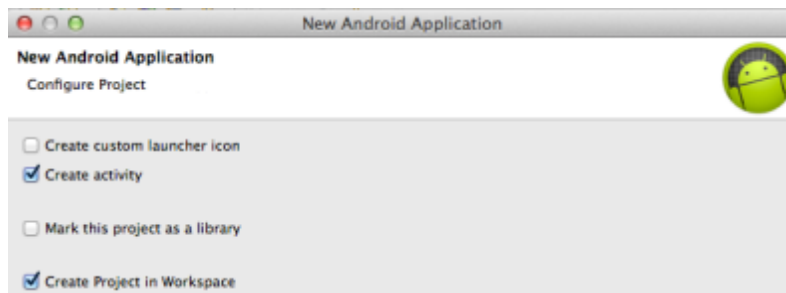


If the menu option is not present, review [Install the Android Plug-In \(Section 5.1 on page 15\)](#).

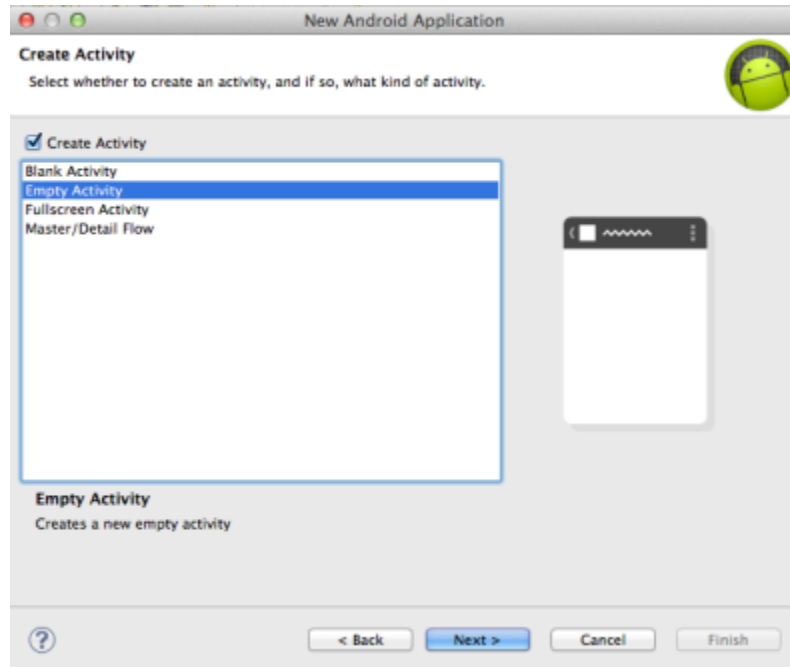
2. In the displayed dialog, fill in the fields as below and select the SDK API level. Other choices for API level are possible but for the first attempt use those below. When the fields are complete, click **Next**.



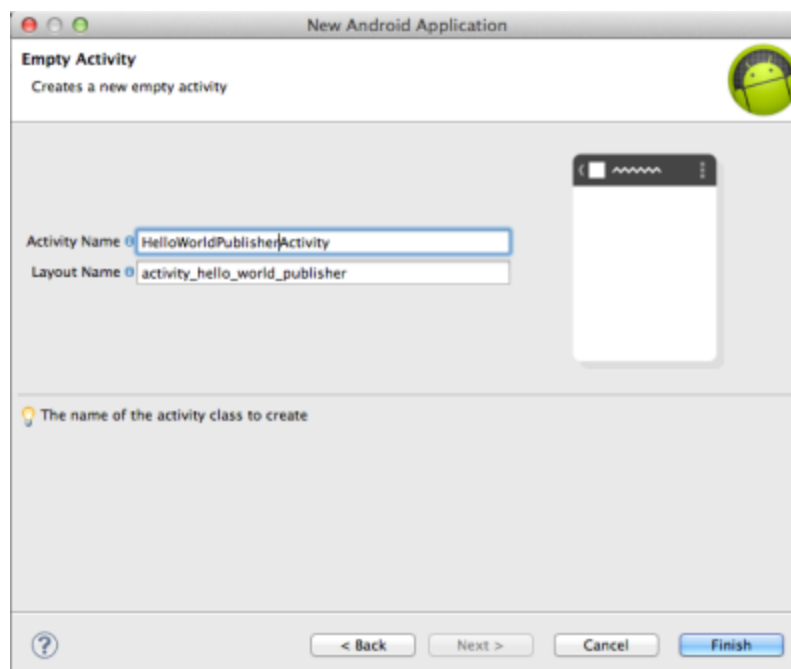
3. Uncheck the **Create custom launcher icon** box, check the **Create in Project Workspace** box, then click **Next**.



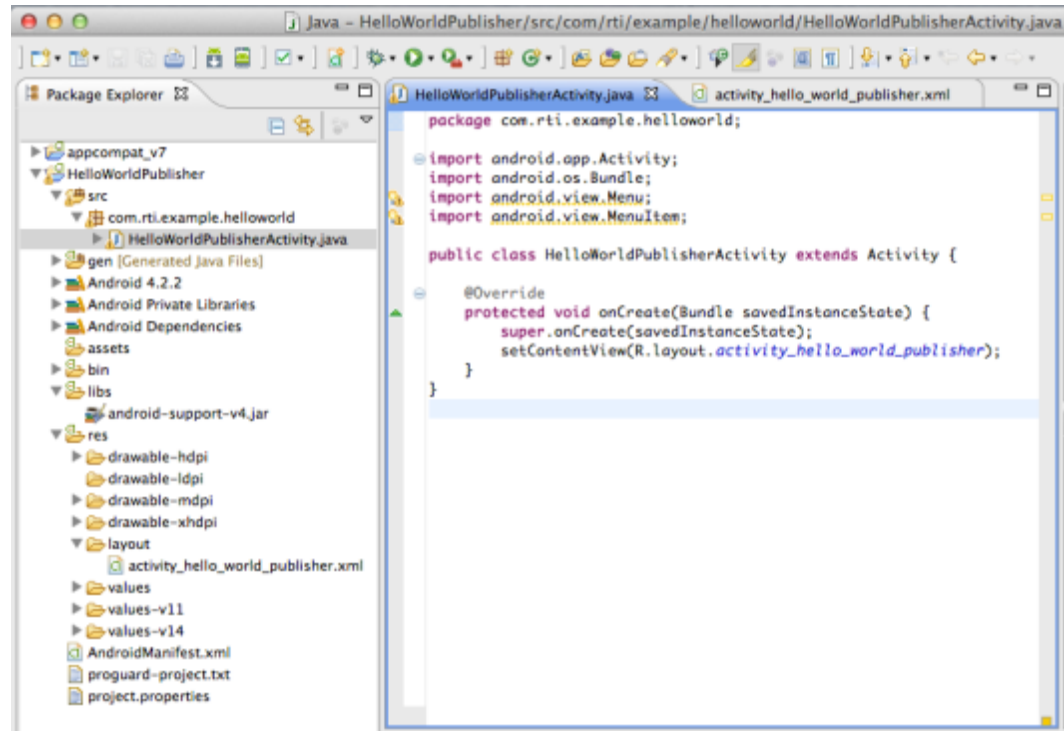
4. Choose **Empty Activity** since this example does not include any GUI elements, then click **Next**.



5. Edit the **Activity Name** as shown below. There is no need to edit the **Layout Name** since this is driven by the Activity Name. The layout will not be touched in this example. Click **Finish**.

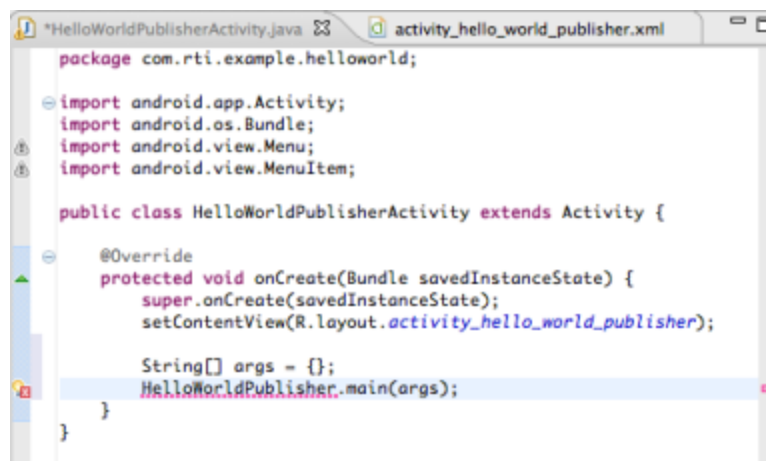


6. After the project has been created, Eclipse will display the Package Explorer as shown below. At this point, **HelloWorldPublisherActivity.java** is the only source file in the project. Examine it to see the generated default code.



5.3 Edit the Project Source Code

1. Edit the generated Java class by adding the two lines shown below. This will create the simplest Android App with Connext DDS. More sophisticated Android code can and probably would be used for a real App. Note that Eclipse shows **HelloWorldPublisher** with underlined red text because that class does not yet exist in the project. Save the changes to **HelloWorldPublisherActivity.java**.



2. Create or copy the **HelloWorld.idl** file in the project's **src** folder. Right-click and "Refresh" the **src** folder in the Eclipse Package Explorer to see the newly added **HelloWorld.idl**. For example, you may create a **HelloWorld.idl** file with these contents:

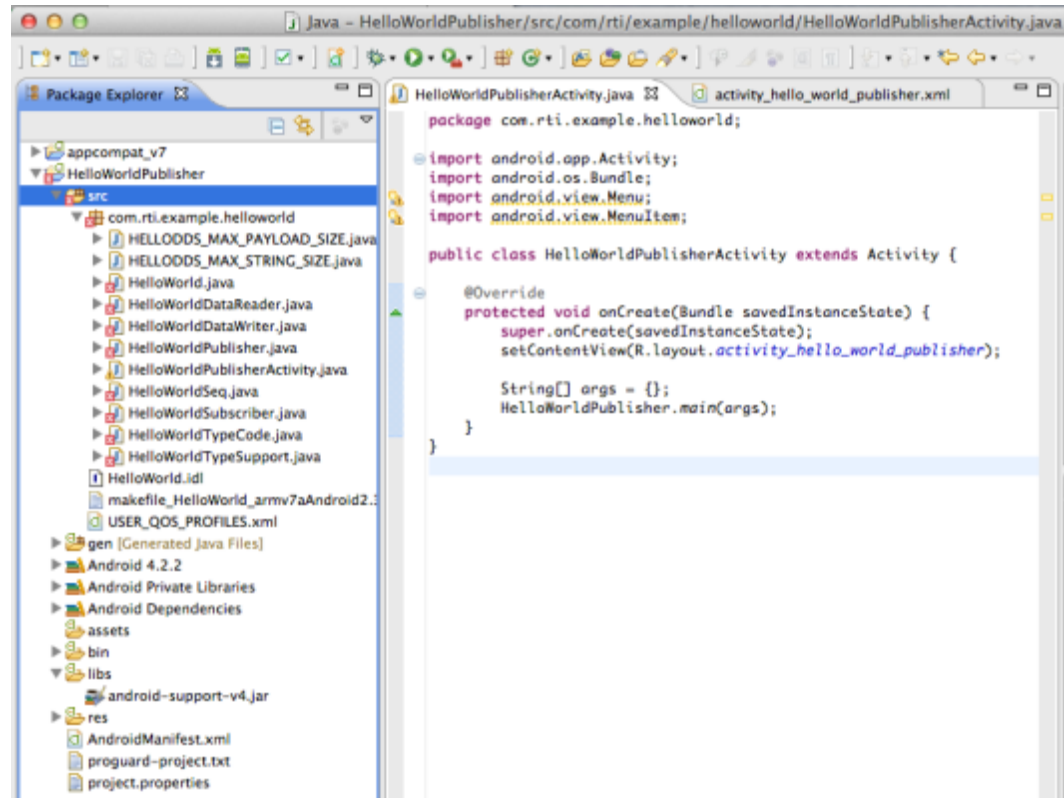
```
struct HelloWorld {  
    string name;  
    long count;  
}
```

3. From a terminal window/command prompt, ensure that the **NDDSHOME** environment variable is set correctly to the Connex DDS installation directory and that the **PATH** environment variable includes **\$NDDSHOME/bin**. Change directory (cd) to the **src** folder of the Eclipse Android project and execute this command (all on one line):

```
rtiddsgen -language Java -package com.rti.example.helloworld -example  
armv7aAndroid2.3gcc4.8 HelloWorld.idl
```

Note that the **-package** argument is the same value as used when creating the project. If the *rtidds-gen*-generated java classes are in a different package to that of the project then "import" statements will need to be added to some java files and the Eclipse Package Explorer would look slightly different from the following screen shots.

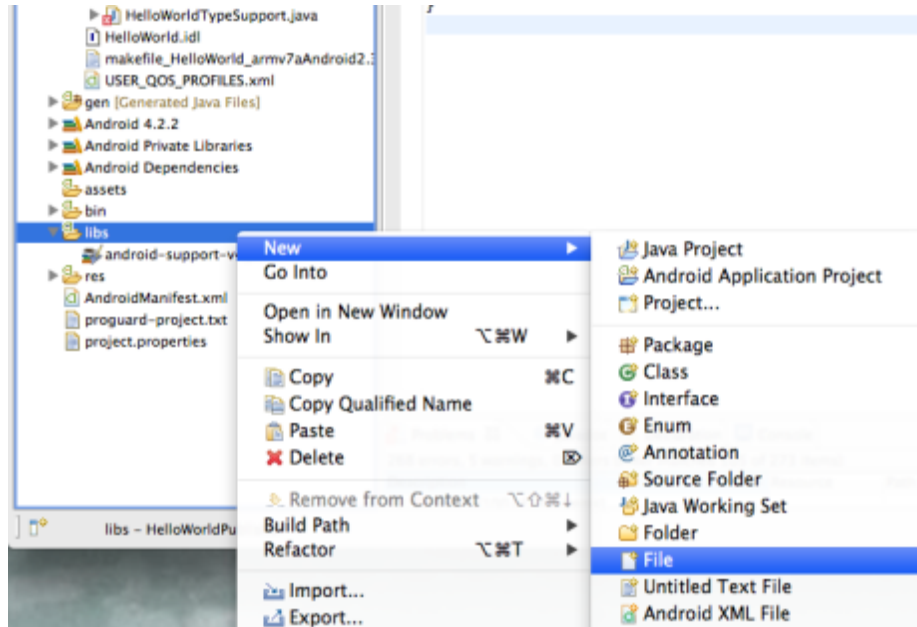
4. Right-click and "Refresh" the **src** folder again.



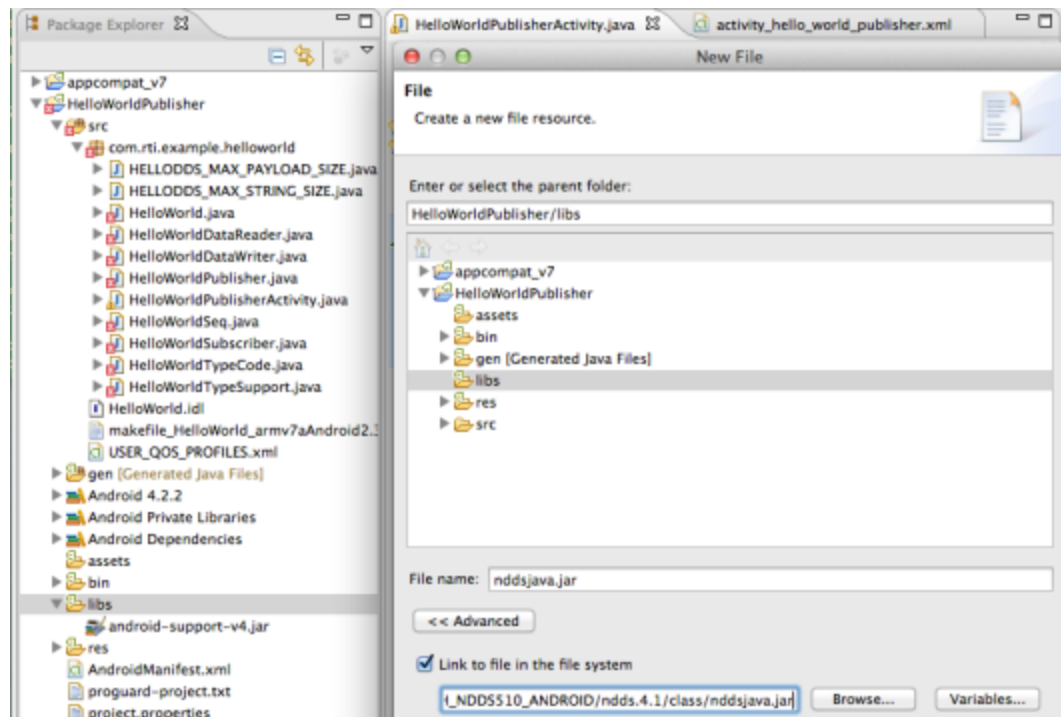
Note that the error in **HelloWorldPublisherActivity.java** is now resolved but that many of the generated files contain errors. These are symbols from Connex DDS that are not being resolved because the Eclipse project has no knowledge of Connex DDS. The **HelloWorldSubscriber.java** file is not needed for this project and may optionally be deleted. The **USER_QOS_PROFILES.xml** file is not used by this project; it may be deleted or kept for future use. The **makefile** is not used by this project; it may be deleted or kept for reference.

5.4 Add the Connex DDS Libraries

1. Resolve the Connex DDS symbols in the generated java files by adding the Connex DDS libraries to the project.
2. Right-click the **libs** folder. Select **New** and **File**.



3. Select the "**Advanced >>**" options, check the **Link to file in the file system** box, and **Browse** for the **nddsjava.jar** file within the Connex DDS installation at **\$NDDSHOME/class/nddsjava.jar** (where **NDDSHOME** is the path to the Connex DDS installation), not the text in highlighted field below.



4. Click **Finish**.

Eclipse may try to display the added file. If it is unable to display a Jar file, you may see a notification that it "can't launch" it. It is safe to click **OK** for this notification if it occurs. The errors previously flagged against the generated Java files should now be cleared.

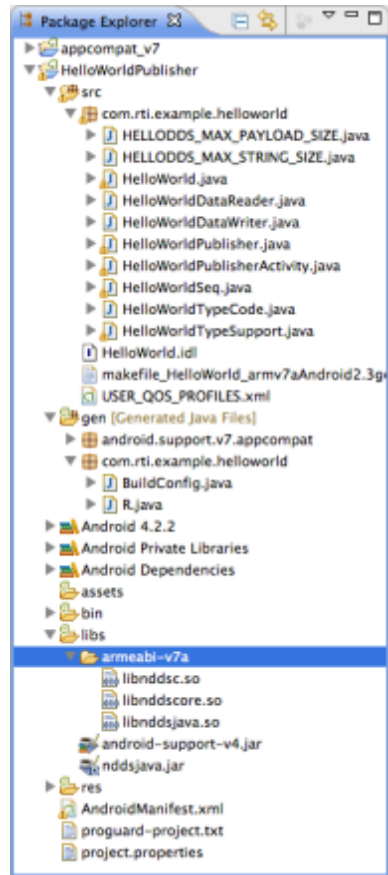
Connex DDS is not pure Java, it also consists of some native libraries. When building the project, you will not see any complaints if these libraries are missing, but when you try to run the App, the LogCat panel will show errors such as:

```
05-13 15:46:49.309: E/AndroidRuntime(764): FATAL EXCEPTION: main
05-13 15:46:49.309: E/AndroidRuntime(764):
java.lang.ExceptionInInitializerError
05-13 15:46:49.309: E/AndroidRuntime(764): at com.rti.example.helloworld.
    HelloWorldPublisher.publisherMain(HelloWorldPublisher.java:127)
```

5. To add the necessary native libraries to the project, the files must be copied into the project tree. Creating links to files in the file system, as was done for **nddsjava.jar**, does not work. The native libraries do not appear in the APK file.

Right-click on the **libs** folder and select **New** and **Folder**. Name the new folder **armeabi-v7a**.

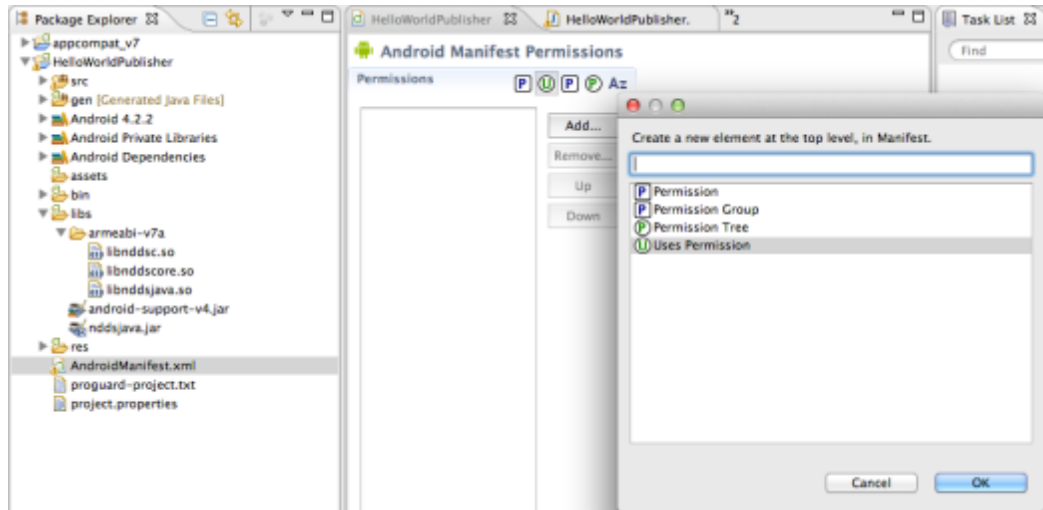
6. From a terminal/command shell, copy **libnddsjava.so** from **\$NDDSHOME/lib/arm-v7aAndroid2.3gcc4.8** to the newly created folder.
7. Copy **libnddsc.so** from **\$NDDSHOME/lib/armv7aAndroid2.3gcc4.8** to the newly created folder.
8. Copy **libnddscore.so** from **\$NDDSHOME/lib/armv7aAndroid2.3gcc4.8** to the newly created folder.
9. Right-click and "Refresh" the **libs** folder to see the added libraries.



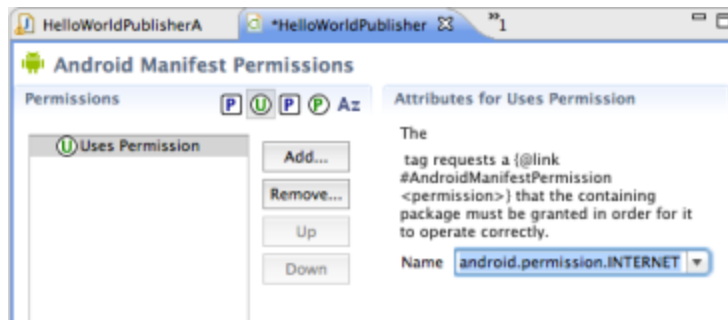
5.5 Update the Android Manifest

Connex DDS expects to use the Internet and Wi-Fi to access the Internet. It needs to be able to change some of the Wi-Fi settings. Connex DDS also needs to access external storage if it is to read a **USER_QOS_PROFILES.xml** file (although the current example doesn't make use of that).

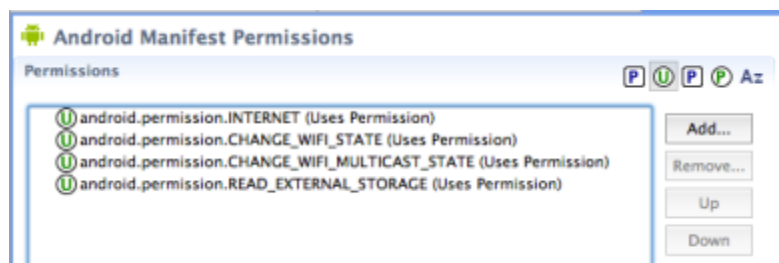
1. In the Package Explorer, double-click the **AndroidManifest.xml** file to edit it. Edit the copy at the outer-most level of the project, the third from bottom in the figure above.
2. Choose the **Permissions** tab at the bottom of the edit panel.
3. Click **Add...**
4. Select **Uses Permission** and click **OK**.



5. From the **Name** pull-down list, select **android.permission.INTERNET**.



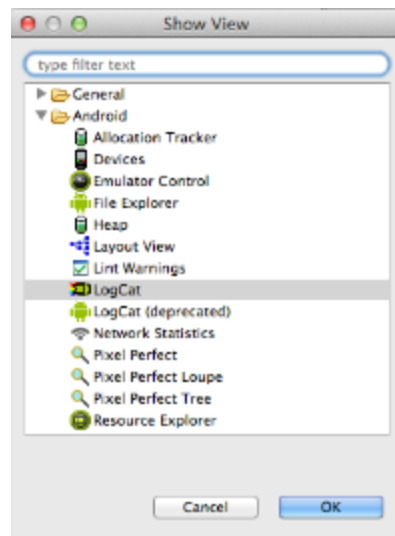
6. Click the left pane (under "Uses Permission" in the figure above) to complete the selection. (The dialog doesn't have an "OK" button.)
7. Repeat to add **android.permission.CHANGE_WIFI_STATE**.
8. Repeat to add **android.permission.CHANGE_WIFI_MULTICAST_STATE**.
9. Repeat to add **android.permission.READ_EXTERNAL_STORAGE**.



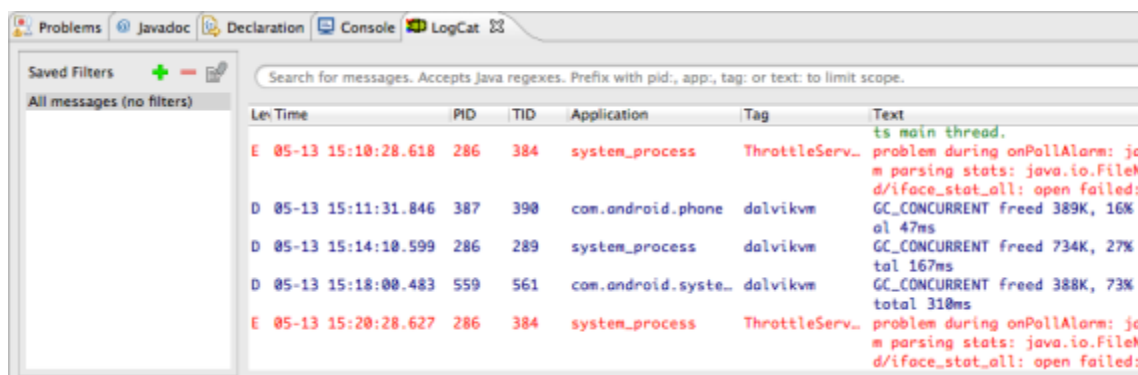
10. Save the changes to the Android Manifest.

5.6 Set Up Before Running Example

1. Ensure that either an Android device (e.g., Nexus 7) is connected via USB to the computer running Eclipse or that an Android Virtual Device (AVD) has been defined. See Android documentation for details.
2. Connex DDS uses Android's **logcat** facility to output certain messages. To make sure these are captured before running the example the Eclipse LogCat panel needs to be set up:
 - a. From the Eclipse menu, select **Window, Show View >, Other**
 - b. From the resulting **Show View** dialog, select **Android >** and **LogCat**, then click **OK**.

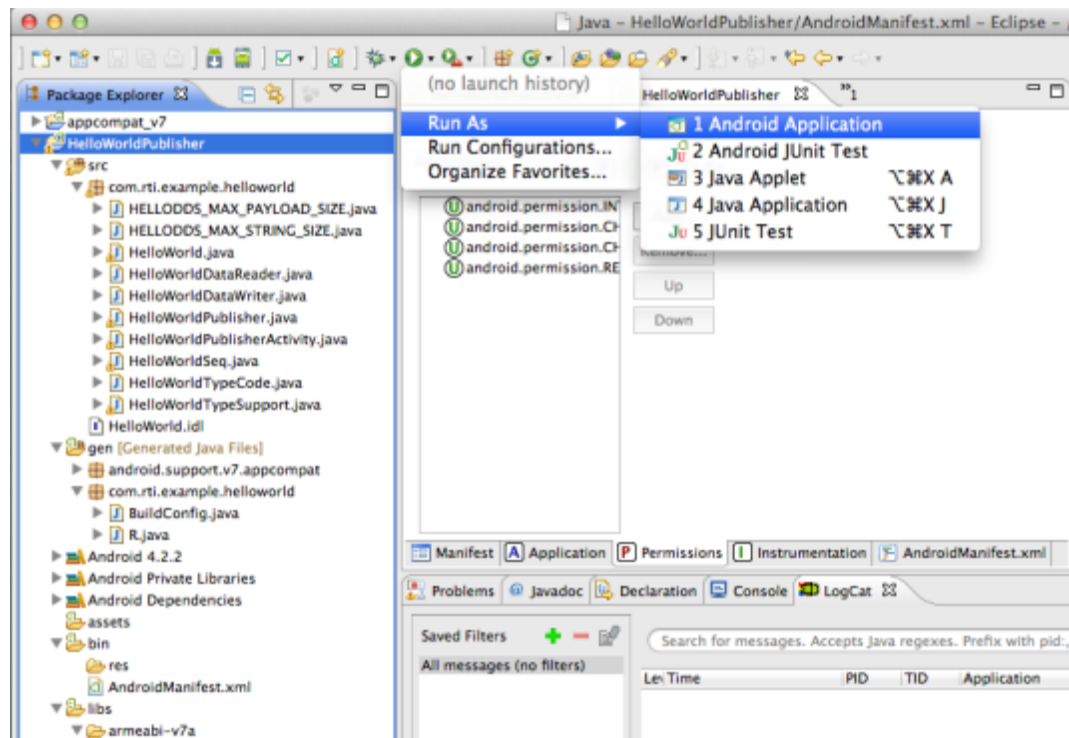


Eclipse will open a panel as shown below. The messages contained here are from the various processes running on a connected Android device or an AVD running on an emulator. The machine running Eclipse will run an **adb** process to communicate with the emulator or device. (See Android documentation for more detail on the Android Debug Bridge.) Eclipse should start this process implicitly if it isn't already running.



5.7 Run the Example

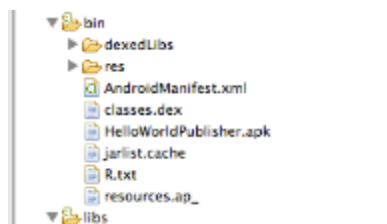
1. Before building the example, which happens implicitly as part of running it, note the contents of the **bin** folder.
2. From the Eclipse toolbar, select **Run As ...** and **Android Application** (or if run previously, select **HelloWorldPublisher** from the launch history).



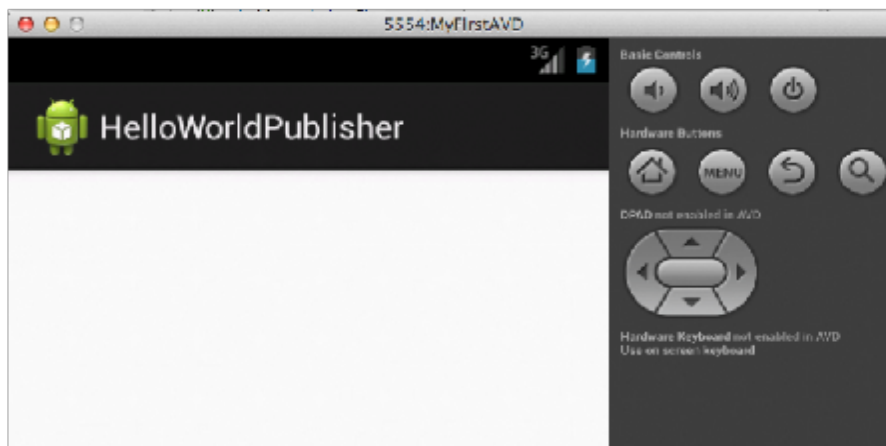
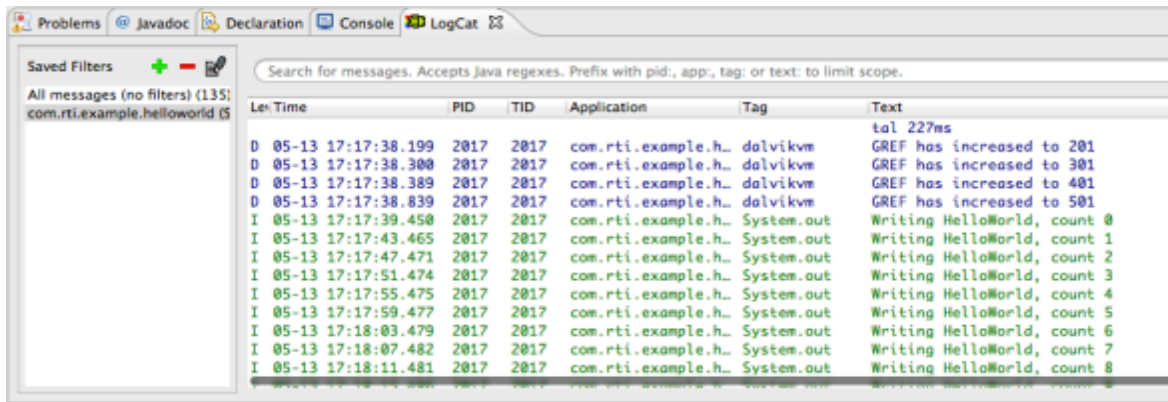
3. It can take quite a while for Eclipse to build the Android App (*.apk file), start the emulator, let the AVD go through its start up, install the App on the device and run it. You can check the progress at the bottom right corner of the Eclipse GUI.



4. After the build completes, the **bin** folder in the Package Explorer will be updated.



5. When the emulator displays the AVD, unlock it by dragging the padlock to the right. This will allow the App to be visible on the emulator.
6. A successful **HelloWorldPublisher** App will display output to **LogCat** and put a minimal interface up on the emulator.



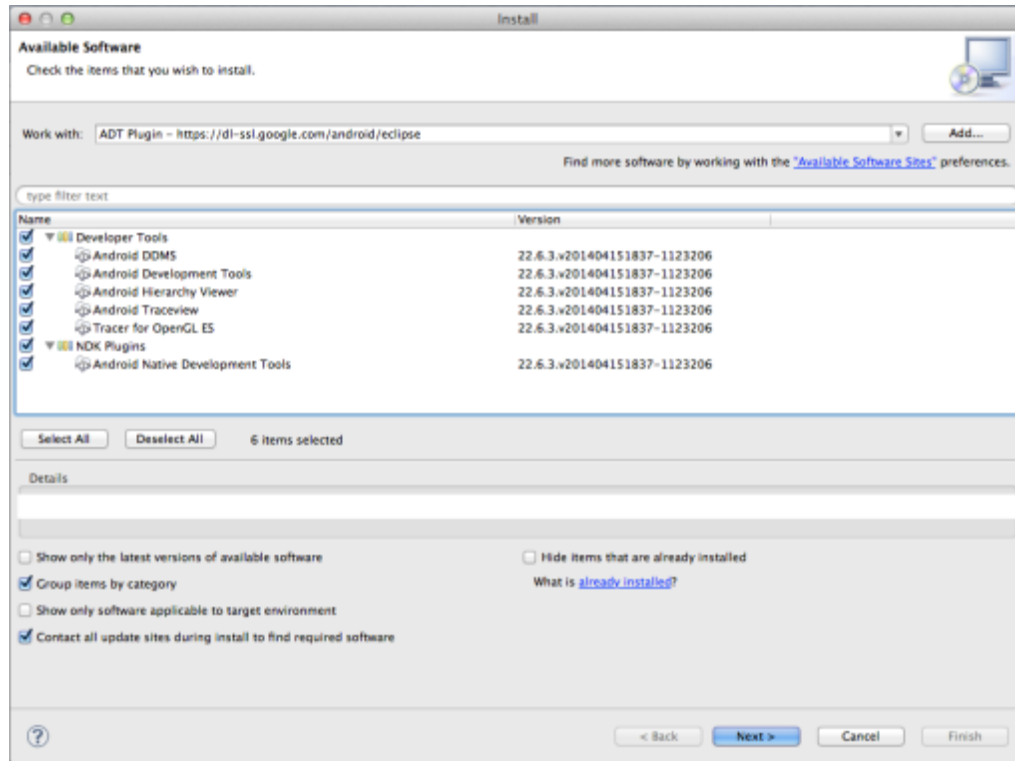
Chapter 6 Using the Connex DDS C and C++ Libraries with Eclipse

This section describes how to use Android within an existing Eclipse installation to build a Connex DDS application with C++. The process for building a Connex DDS application with C is similar to that for C++. The following instructions and screenshots are for C++; any differences in the instructions for C will be called out in the affected steps. Also note that when developing with C, instructions that include the text “Cpp” should be replaced with “C”, such as in filenames, App names, etc. For example, HelloCppSubscriber for C++ would be HelloCSubscriber for C.

The Android Developer Tools bundle can be downloaded from Google. It contains the SDK and an Eclipse-based Android IDE. It is also possible to install an Android plug-in to an existing Eclipse installation.

6.1 Install the Android Plug-In

1. Make sure both the Android SDK and Android NDK are installed and accessible from the machine running Eclipse. Eclipse also needs to have the C++ Development Tools installed; use version 8.0.2.
2. From the Eclipse menu, click **Help**, then **Install New Software** In the resulting dialog, select the site to work with to ADT Plugin or add this URL: **[https://dl-ssl-google.com/android/eclipse](https://dl-ssl.google.com/android/eclipse)**.
3. Check the items to install and click **Next >**. Note that the NDK Plugin and Developer tools are required.



4. Complete the install procedure. It may be necessary to "OK" the installation of unsigned software. It may also be necessary to point Eclipse to the Android SDK installation. At the end, restart Eclipse. The Eclipse menu should now have the Android Development Tools items.
5. From the Eclipse menu bar, select **Eclipse** and **Preferences** (or depending on version of Eclipse, select **Window** and **Preferences**). Within the Android preferences, set the SDK Location to point to the Android SDK installation (this may already be set correctly). Within the Android preferences, select NDK and set the NDK Location to the Android NDK installation from [Step](#) . Click **OK**.

6.2 Create an Android Project

This example uses the same HelloWorld IDL file used in [Generating Example Code and a Makefile from IDL \(Section 4.4 on page 11\)](#). The Android App built in this section can interoperate with those built in other sections of the document. If a different IDL file is used, then note that the following steps will need to be altered slightly as the IDL file name, **HelloWorld**, is used in completing fields and naming other files in this example. This example shows only the creation of the publisher App. The steps from here can be repeated to create a HelloWorldSubscriber App.

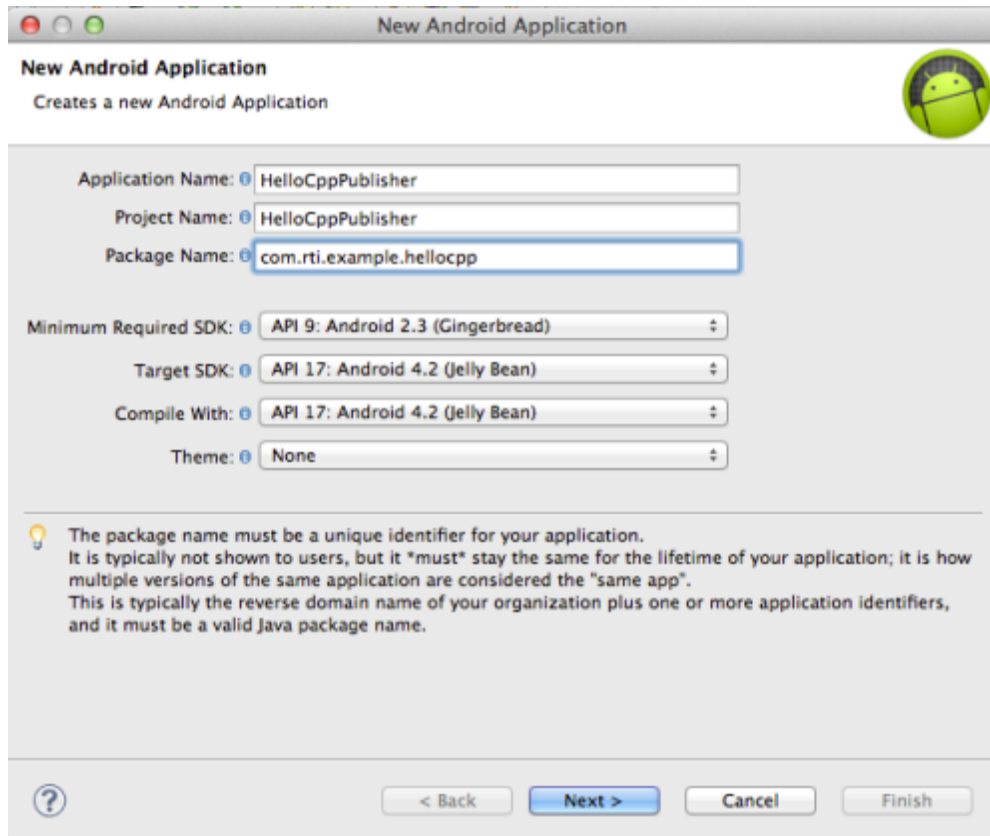
If the subscriber and publisher will be installed on the same device, make sure the subscriber's Application, Project, and Package names are different than the publisher's.

1. Make sure Eclipse is in the Java perspective and not in C++. From the Eclipse menu, select **File**, **New**, **Android Application Project**.

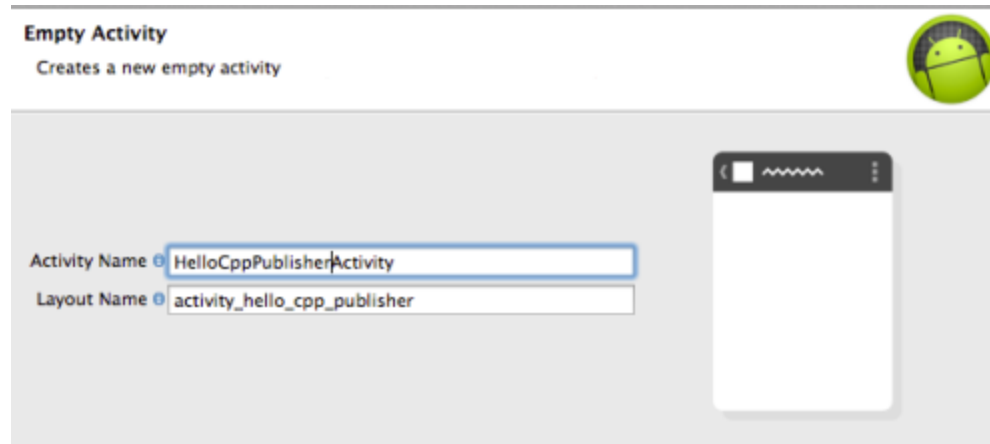


If the menu option is not present, review [Install the Android Plug-In \(Section 6.1 on page 29\)](#).

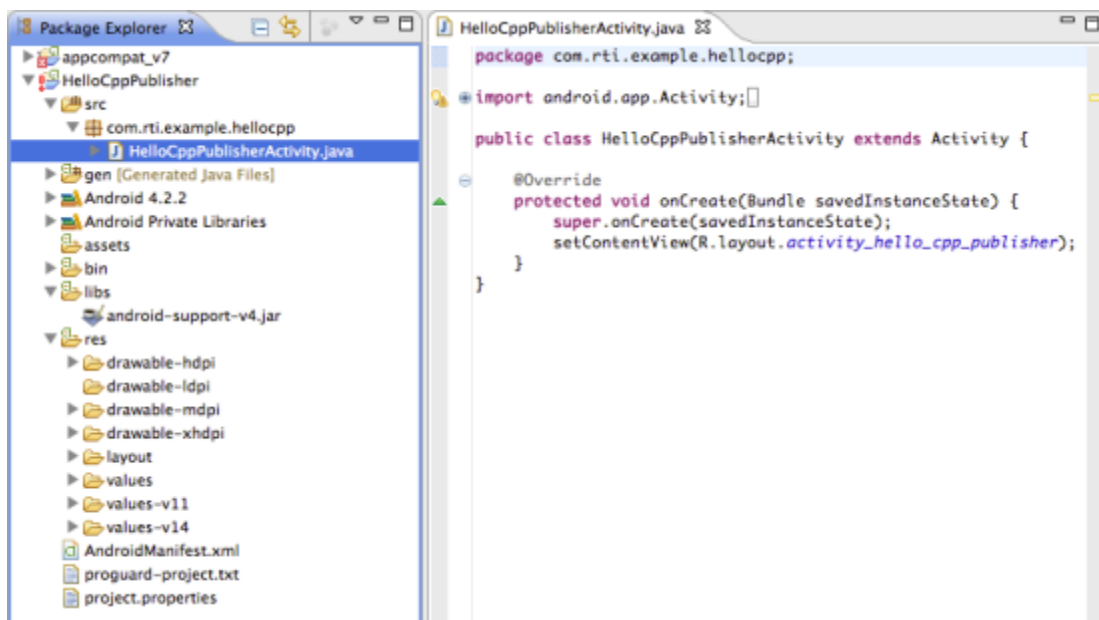
2. In the displayed dialog, fill in the fields as seen here and select the SDK API level. Other choices for API level are possible, but for the first attempt use those seen below. When the fields are complete, click **Next**.



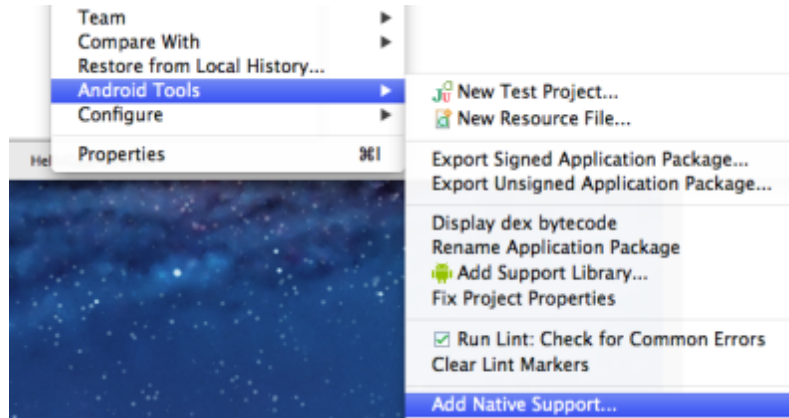
3. Uncheck the **Create custom launcher icon** box, check the **Create in Project Workspace** box, then click **Next**.
4. Choose **Empty Activity** since this example does not include any GUI elements, then click **Next**.
5. Edit the **Activity Name** as seen below. There is no need to edit the **Layout Name** since this is driven by the **Activity Name**. The layout will not be touched in this example. Click **Finish**.



6. After the project has been created, Eclipse will display the Package Explorer as seen below. At this point, **HelloCppPublisherActivity.java** is the only source file in the project. Examine it to see the generated default code.

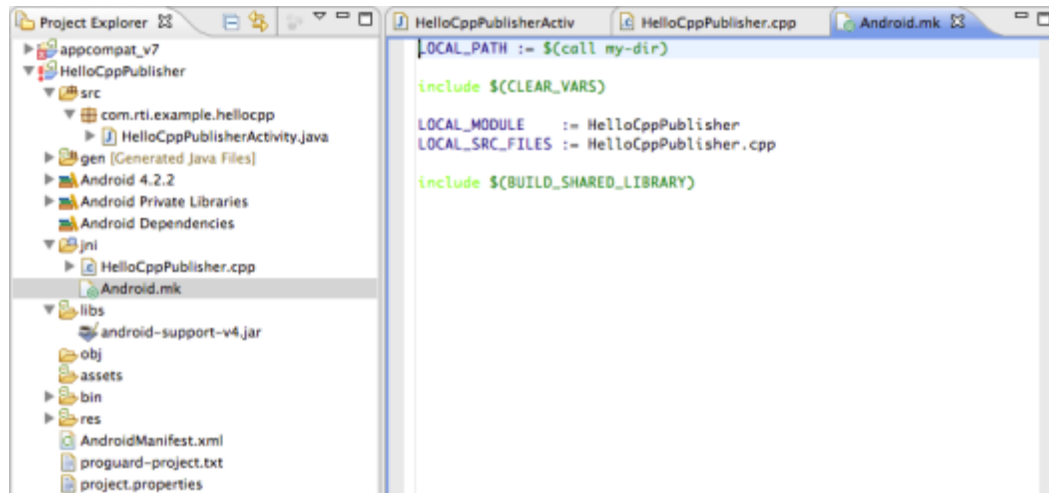


7. Right-click on the Android project in the Package Explorer and select **Android Tools** and **Add Native Support**.



It is not possible to do this step if the Eclipse project already has a C++ nature (this is why [Step](#) puts Eclipse into the Java perspective before creating the project). Note that the project now has a **jni** folder that contains **HelloCppPublisher.cpp** and **Android.mk**.

8. Accept the offered **Library Name**, **HelloCppPublisher**, and click **Finish**. Note the addition of the **jni** folder with two files in it.



Note for C Development:

The library name will be shown as **HelloCPublisher**. The resulting file in the **jni** folder will have a **.cpp** suffix. This can be left as C++ if desired, as the code placed here will not call Connexx DDS. The code in this file is easy to convert to C. Note that Eclipse shows a C++ perspective, even though this is a C project.

6.3 Edit the Project Source Code

1. Edit the generated Java class by adding the code shown below. The **onCreate()** method will make the JNI call to the C++ code. The static block loads the library containing all the C++ code. Save the changes to **HelloCppPublisherActivity.java**.

```

package com.rti.example.hellocpp;
import android.app.Activity;
import android.os.Bundle;
public class HelloCppPublisherActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_cpp_publisher);
        //added
        final String parameters = null;
        nativeCallMain(parameters);
    }
    //added
    public static native int nativeCallMain(String parameters);
    //added
    static {
        System.loadLibrary("HelloCppPublisher");
    }
}

```

2. Edit the **HelloCppPublisher.cpp** file, replacing its contents with the code below. Note that in extern "C" form, the method name, **nativeCallMain()**, is prepended by the Java package and class name. If a different package or class is used, this name will need to be changed to be consistent.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <jni.h>
#define ARGV_MAX_SIZE 32
#define JNI_VERSION JNI_VERSION_1_2
/* the following must be implemented in native code */
extern "C" int main(int argc, char *argv[]);
extern "C" JNIEXPORT jint JNICALL
    Java_com_rti_example_hellocpp_HelloCppPublisherActivity_
    nativeCallMain
    (JNIEnv *env, jclass self_class, jstring javaParamString) {
    int returnVal = -1;
    int argc = 0;
    char *argv[ARGV_MAX_SIZE];
    returnVal = main(argc, argv);
    return returnVal;
}

```

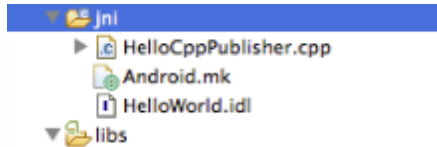
Note for C Development:

Omit extern "C" from the two lines above, so they appear as:

```
int main(int argc, char *argv[]);
```

```
JNIEXPORT jint JNICALL ...
```

3. Create or copy the **HelloWorld.idl** file in the project's **jni** folder. Right-click and "Refresh" the **jni** folder in the Eclipse Package Explorer to see the newly added **HelloWorld.idl**.



4. Open a terminal window/command prompt on a machine with architecture compatible to the host bundle for Connex DDS that was installed. If this is done on a different machine, the generated files can be copied to the project's **jni** folder. Ensure that the **NDDSHOME** environment variable is set correctly to the Connex DDS installation directory and that the **PATH** environment variable includes **\$NDDSHOME/bin**. Change directory (**cd**) to the **jni** folder of the Eclipse Android project and execute this command (all on one line):

```
rtiddsgen -language C++ -example armv7aAndroid2.3gcc4.8 HelloWorld.idl
```

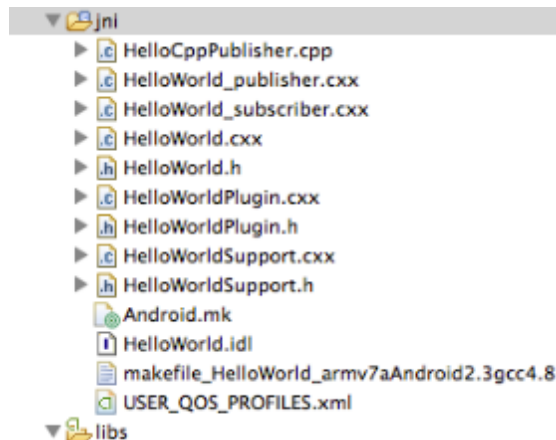
Note for C Development: Change the **-language** option to C:

```
rtiddsgen -language C -example armv7aAndroid2.3gcc4.8 HelloWorld.idl
```

The generated files names will have ".c" suffixes instead of ".cxx" suffixes.

5. Right-click and **Refresh** the **jni** folder again.

The **HelloWorld_subscriber.cxx** file is not needed for this project and may optionally be deleted. The **USER_QOS_PROFILES.xml** file is not used by this project; it may be deleted or kept for future use. The makefile is not used by this project; it may be deleted or kept for reference.

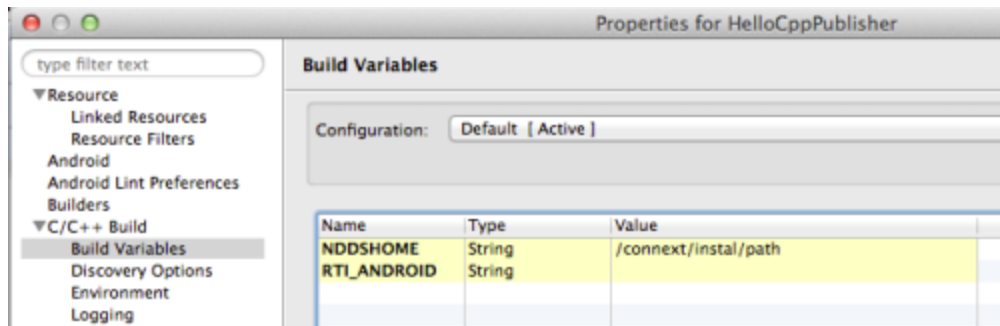


- Set the include paths for the project. Right-click on the project in the Package Explorer and select **Properties**. Navigate to the **C/C++ General** section, then to the **Paths and Symbols** subsection. In the **Includes** tab, add the following two entries:

```
{NDDSHOME}/include
{NDDSHOME}/include/ndds
```

Click **OK**.

- If the actual path is used in place of the **NDDSHOME** variable, this step can be omitted. In project **Properties**, navigate to **C/C++ Build** and **Build Variables**. Add variables for **NDDSHOME** and **RTI_ANDROID** as seen below, but use your installation location for Connex DDS.



- Right-click on the project and select **Index** and **Rebuild**. This should resolve any errors in the generated source files. If a handful of errors remain, this may be due to a bug in Eclipse. If these "phantom" errors cause the build to fail when you [Configure the Build \(Section 6.5 on page 38\)](#), they can be suppressed (see [Run the Example \(Section 6.7 on page 41\)](#)).
- Edit the generated file, **HelloWorld_publisher.cxx**, to replace the **#define** with **printf**, as seen below:

Change this:

```
#define printf Android_printf
```

To this:

```
#include <android/log.h>
#define printf(x...) __android_log_print(ANDROID_LOG_INFO,
"HelloCppPublisher", x)
```

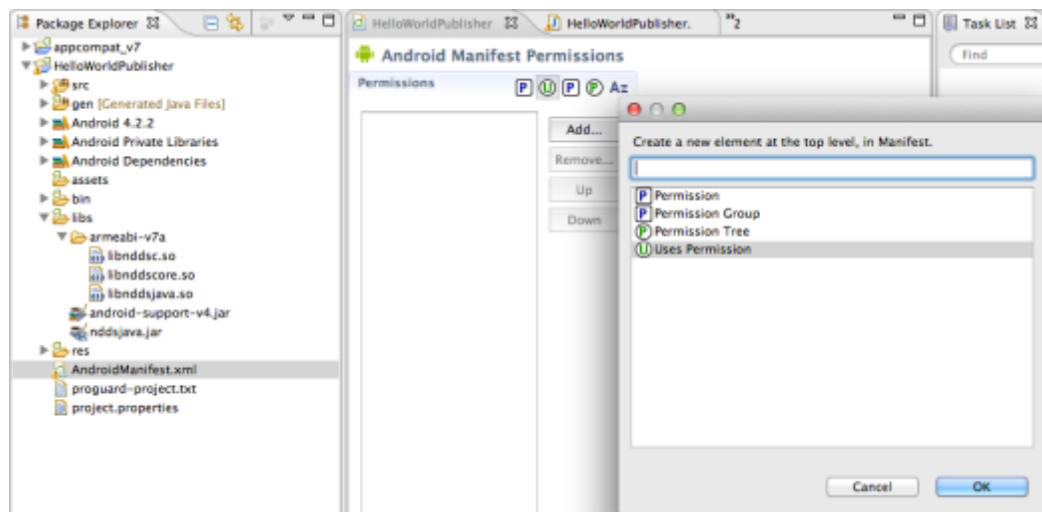
This is the quickest way to redirect stdout to logcat.

Other ways are possible but they require more JNI code in **HelloCppPublisher.cpp**. (See the command-line example for C++ for the full source code for this approach).

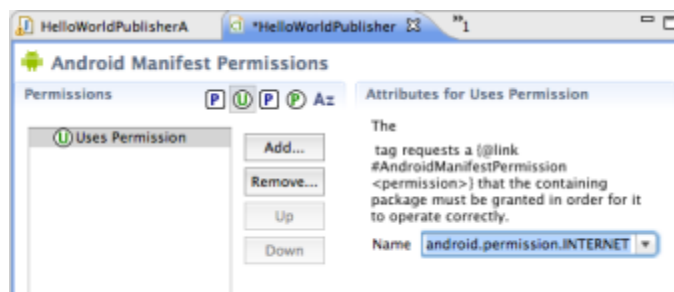
6.4 Update the Android Manifest

Connex DDS expects to use the Internet and Wi-Fi to access the Internet. It needs to be able to change some of the Wi-Fi settings. *Connex DDS* also needs to access external storage if it is to read a **USER_QOS_PROFILES.xml** file (although the current example doesn't make use of that).

1. In the Package Explorer, double-click the **AndroidManifest.xml** file to edit it. Edit the copy at the outer-most level of the project, the third from bottom in the figure below.
2. Choose the **Permissions** tab at the bottom of the edit panel.
3. Click **Add...**
4. Select **Uses Permission** and click **OK**.

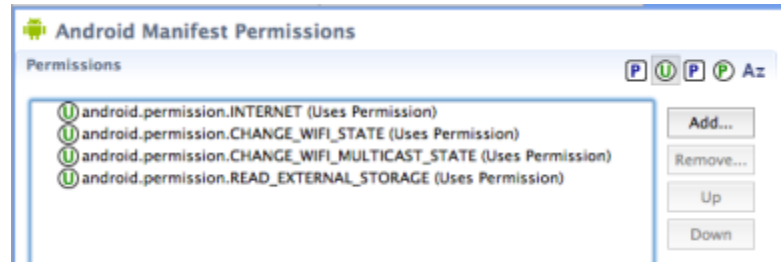


5. From the **Name** pull-down list, select **android.permission.INTERNET**.



6. Click the left pane (under "**Uses Permission**" in the figure above) to complete the selection. (The dialog doesn't have an "OK" button.)

7. Repeat to add **android.permission.CHANGE_WIFI_STATE**.
8. Repeat to add **android.permission.CHANGE_WIFI_MULTICAST_STATE**.
9. Repeat to add **android.permission.READ_EXTERNAL_STORAGE**.



10. Save the changes to the Android Manifest.

6.5 Configure the Build

For the Java Connex DDS App for Android, the Connex DDS libraries have to be placed in the project in the correct place and they then get added automatically to the App (the *.apk file). For native development all native code and libraries are collected into a single shared library (see Android ndk-build documentation for further information).

To configure the build properly follow the steps below.

1. Edit the **Android.mk** file to have the contents shown below. The three local modules contain the Connex DDS static libraries. **NDDSHOME** should be set to the Connex DDS installation (i.e., the same as **\$NDDSHOME** earlier). Note that the last **LOCAL_SRC_FILES** does not fit on one line, so we use a continuation character "\".

```
LOCAL_PATH := $(call my-dir)
#added
NDDSHOME := /path to rti_connex_dds-5.3.0/
RTI_SYSROOT := $(ANDROID_NDK_PATH)/platforms/$(APP_PLATFORM)/arch-arm
include $(CLEAR_VARS)
LOCAL_MODULE := nddscore
LOCAL_SRC_FILES := $(NDDSHOME)/lib/armv7aAndroid2.3gcc4.8/libnddscorez.a
include $(PREBUILT_STATIC_LIBRARY)
include $(CLEAR_VARS)
LOCAL_MODULE := nddsc
LOCAL_SRC_FILES := $(NDDSHOME)/lib/armv7aAndroid2.3gcc4.8/libnddscz.a
include $(PREBUILT_STATIC_LIBRARY)
include $(CLEAR_VARS)
LOCAL_MODULE := nddscpp
LOCAL_SRC_FILES := $(NDDSHOME)/lib/armv7aAndroid2.3gcc4.8/libnddscppz.a
include $(PREBUILT_STATIC_LIBRARY)
```

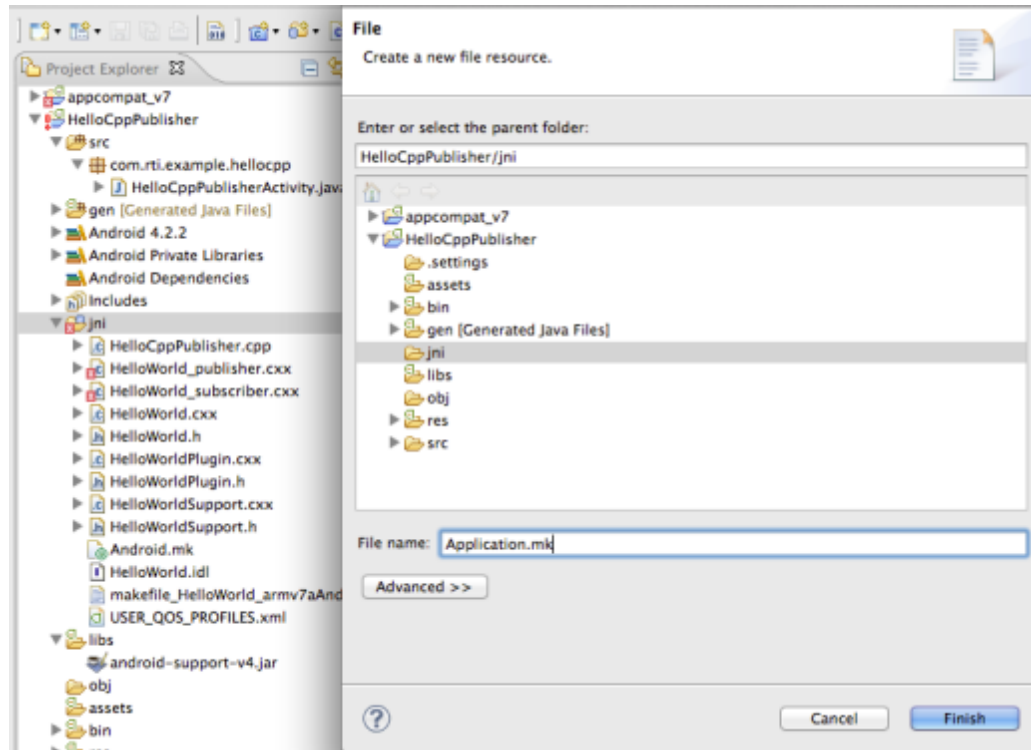
```
include $(CLEAR_VARS)
LOCAL_MODULE      := HelloCppPublisher
#was just HelloCppPublisher.cpp
LOCAL_SRC_FILES   := HelloCppPublisher.cpp HelloWorld_publisher.cxx \
    HelloWorld.cxx HelloWorldPlugin.cxx HelloWorldSupport.cxx
#added
LOCAL_C_INCLUDES  := $(NDDSHOME)/include/ $(NDDSHOME)/include/ndds/
LOCAL_CFLAGS      := -DRTI_UNIX -DRTI_ANDROID -mlong-calls
LOCAL_STATIC_LIBRARIES := nddscpp nddsc nddscore
LOCAL_LDLIBS      := -L$(RTI_SYSROOT)/usr/lib -llog -landroid -lEGL -lGLESv1_
CM
include $(BUILD_SHARED_LIBRARY)
```

Notes for C Development:

Omit the four lines for the **nddscpp** module. The last section in the file should be replaced with:

```
LOCAL_MODULE      := HelloCPublisher
#was just HelloCPublisher.c
LOCAL_SRC_FILES   := HelloCPublisher.c HelloWorld_publisher.c \
    HelloWorld.c HelloWorldPlugin.c HelloWorldSupport.c
#added
LOCAL_C_INCLUDES  := $(NDDSHOME)/include/ $(NDDSHOME)/include/ndds/
LOCAL_CFLAGS      := -DRTI_UNIX -DRTI_ANDROID -mlong-calls
LOCAL_STATIC_LIBRARIES := nddsc nddscore
LOCAL_LDLIBS      := -L$(RTI_SYSROOT)/usr/lib -llog -landroid -lEGL -lGLESv1_
CM
include $(BUILD_SHARED_LIBRARY)
```

2. Add an **Application.mk** file to the **jni** folder.

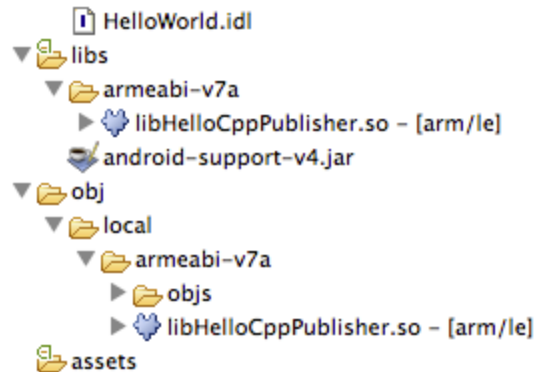


The contents of **Application.mk** should be as shown below. Note the use of static GNU STL library. This is picked up from the NDK, it is not redistributed as part of Connex DDS.

```
APP_ABI := armeabi-v7a
APP_PLATFORM := android-9
APP_STL := gnuSTL_static
NDK_TOOLCHAIN_VERSION := 4.8
```

Note for C Development: Omit the line for **APP_STL**

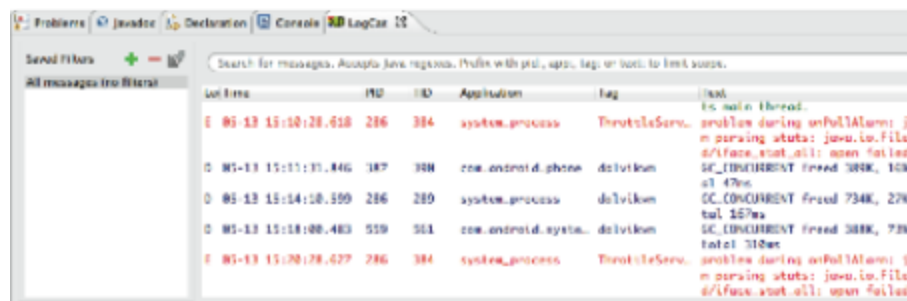
3. Right-click the project in Package Explorer and select **Build Project**. This will create the **libHelloCppPublisher.so** file in the **obj** folder and copy it to the **libs** folder.



6.6 Set Up Before Running Example

1. Ensure that either an Android device (e.g., Nexus 7) is connected via USB to the computer running Eclipse or that an Android Virtual Device (AVD) has been defined. See Android documentation for details.
2. Connex DDS uses Android's logcat facility to output certain messages. To make sure these are captured before running the example the Eclipse LogCat panel needs to be set up:
 - a. From the Eclipse menu, select **Window, Show View >, Other ...**
 - b. From the resulting Show View dialog, select **Android >** and **LogCat**, then click **OK**.

Eclipse will open a panel as shown below. The messages contained here are from the various processes running on a connected Android device or an AVD running on an emulator. The machine running Eclipse will run an **adb** process to communicate with the emulator or device. (See Android documentation for more detail on the Android Debug Bridge.) Eclipse should start this process implicitly if it isn't already running.



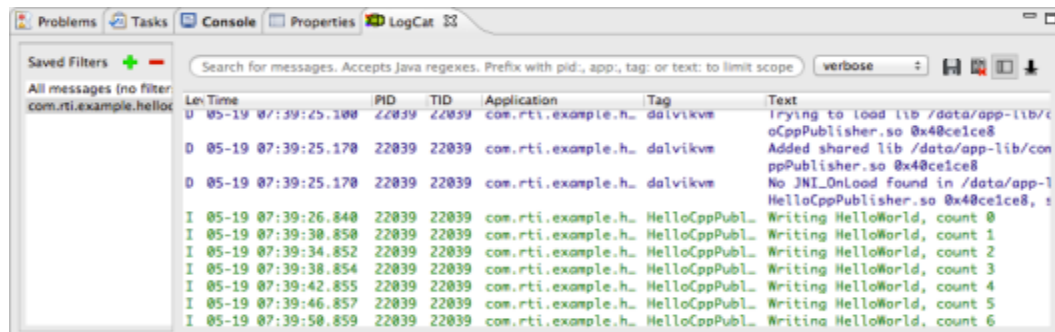
6.7 Run the Example

1. From the Eclipse toolbar, select **Run As ...** and **Android Application** (or if run previously, select **HelloCppPublisher** from the launch history).

If Eclipse complains about needing to resolve problems before the App can launch and these problems have to do with the `appcompat_v7` library project and/or other build path issues, try saving all files, closing Eclipse, and re-opening Eclipse. At the time of writing, there is an Eclipse bug in this area. Restarting Eclipse after attempting a build also appears to populate the Include directories in the C/C++ General, Paths and Symbols properties.

2. It can take quite a while for Eclipse to build the Android App (*.apk file), start the emulator, let the AVD go through its start-up process, install the App on the device, and run it. You can check the progress at the bottom-right corner of the Eclipse GUI.
3. After the build completes, the **bin** folder in the Package Explorer will be updated.
4. When the emulator displays the AVD, unlock it by dragging the padlock to the right. This will allow the App to be visible on the emulator.

A successful HelloCppPublisher App will display output to LogCat and put a minimal interface up on the emulator.



Chapter 7 Building and Running a Hello World Example with C/C++

This section will guide you through the steps required to successfully run an *rtiddsgen*-generated example application written in C or C++ on an Android target, as an Android App (*.apk) or as an executable.

Note that the C and C++ Android App examples use a Java wrapper class with JNI. It is also possible to write pure C and C++ native code to build Android Apps; see the Android documentation for details (<https://developer.android.com/tools/sdk/ndk/index.html>).

1. Compile the generated C or C++ code and link it into the executables **HelloWorld_publisher** and **HelloWorld_subscriber** in the **objs/<arch>/** directory. For example:

```
make -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8
```

Note that these executables are statically linked with the Connex DDS libraries:

For C: **libnddscorz.a** and **libnddscz.a**.

For C++: **libnddscorz.a**, **libnddscz.a** and **libnddscppz.a**, and with the GNU C++ runtime static library, **libgnustl_staticz.a**.

2. Depending on what form the application is to take, follow the steps in one of these sub-sections:
 - [Building and Running a C/C++ Application as an Executable \(Section 7.1 on the next page\)](#)
 - [Building and Running a C/C++ Application as an Android App \(Section 7.2 on page 45\)](#)
3. If you want to see DDS log messages or values of the samples being printed on the sub-

scriber, run **adb logcat** with a filter in a host machine shell. For example:

```
adb logcat -v tag RTIConnnextLog:I *:S
```

7.1 Building and Running a C/C++ Application as an Executable

This procedure will not work for a non-rooted Android consumer device, see [Building and Running a C/C++ Application as an Android App \(Section 7.2 on the facing page\)](#) instead.

1. Push (via **adb**) the executables to the emulator, in a user-specified directory. For example:

```
make -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8 HelloWorld.pushexes
```

This will push all the required files to the location specified by `ANDROID_PUSH_DEST`.

If you want to use **USER_QOS_PROFILES.xml** for your example, that should also be pushed to the emulator. To do so, see the generated makefile and uncomment the command that pushes **USER_QOS_PROFILES.xml** to the device in the HelloWorld.pushexes target rule before running the above command.

2. To run the example on the emulator, open two **adb** shells. Run the publisher executable in one shell and the subscriber executable in the other. For example:

To run the publisher:

```
adb shell
cd /data/data/com.rti.helloworldexample
./HelloWorld_publisher <args>
```

To run the subscriber:

```
adb shell
cd /data/data/com.rti.helloworldexample
./HelloWorld_subscriber <args>
```

Note: The `<args>` are optional. See the generated source code for usage.

3. Examine the **adb** shell output printed by the examples and the optional **logcat** output that shows the value of the received samples.

The output in the **adb** shell should look like this:

[illegible]

The **logcat** output should look like this:

```
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933): prefix:
I/RTIConnextLog( 3933): "count is ..."
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933): sampleId:
I/RTIConnextLog( 3933): 8
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933): payload:
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933): payload[0]:
I/RTIConnextLog( 3933): <29>
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933): payload[1]:
I/RTIConnextLog( 3933): <2a>
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933): payload[2]:
I/RTIConnextLog( 3933): <2b>
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933):
I/RTIConnextLog( 3933): payload[3]:
```

7.2 Building and Running a C/C++ Application as an Android App

1. See [Step](#) in [Generating Example Code and a Makefile from IDL \(Section 4.4 on page 11\)](#) regarding the `ANDROID_PACKAGE_PREFIX` and `TARGET_ID` variables; if you want to change these variables in the makefile, do so now.
2. Create separate Android projects for the publisher and subscriber. For example:

```
make -f makefile HelloWorld armv7aAndroid2.3gcc4.8 HelloWorld.projects
```

The project will be placed in the `./objs/<arch>` and `./objs/<arch>/subscriber` directories (where `<arch>` is the Android architecture that you are working with). Each project directory will have a directory structure for an Android project, it will contain files that have been created from provided Connex DDS templates. These files include the Java Activity class source file, where the publisher/subscriber example native code is called through JNI. The “**android update project**” command is run by the makefile to set the directory up as a correct Android project.

3. Create shared libraries (*.so) and build the Android projects with the resulting shared libraries. For example:

```
gmake -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8 HelloWorld.apks
```

The above command generates separate shared libraries for the publisher and subscriber examples. This also generates separate JNI source code files for both the publisher and subscriber; it also compiles and links the JNI code into separate jni shared libraries, one for the publisher and one for the subscriber. These shared libraries are packaged into Android Apps along with a **classes.dex** file for the Java Activity class.

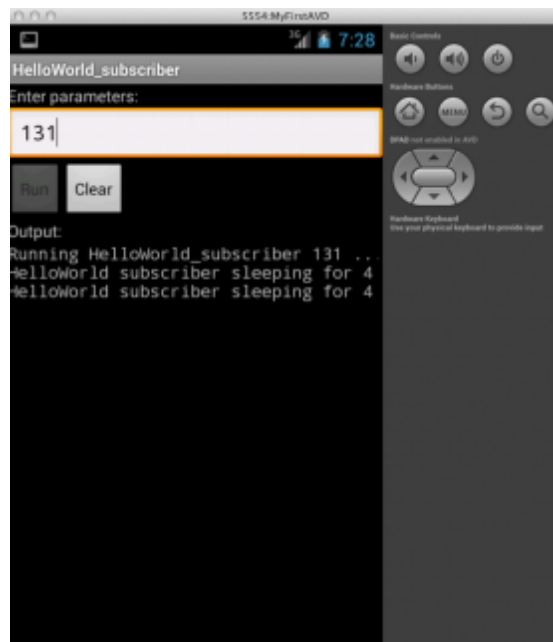
4. Install (via **adb**) the two Apps on a connected consumer device or an emulator. For example:

```
gmake -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8 HelloWorld.installapks
```

5. Run the example on an emulator with a GUI (i.e., not started with the **-no-window** flag) or on a consumer device:
 - a. Navigate to the Apps screen displaying icons, it should show **HelloWorld_publisher** and **HelloWorld_subscriber**.



- b. Touch (on a device) or click (on an emulator) the **HelloWorld_subscriber** icon.
- c. Optionally, enter a single number for the DDS Domain ID as a parameter.
- d. Touch or click **Run**.

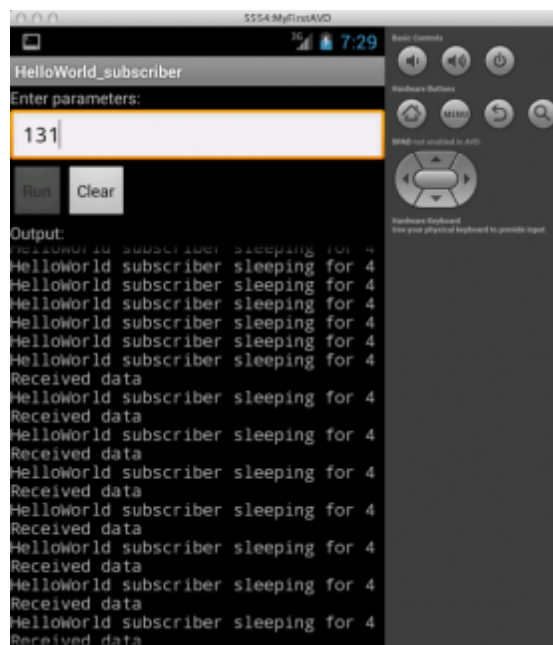


- e. Navigate to the Apps icons (via the **Home** button) and launch **HelloWorld_publisher**

- f. Optionally, for parameters, enter the DDS Domain ID (use the same ID entered for the subscriber in [Step](#)) as the first number. Optionally, you can enter the number of samples to be published as the second parameter.
- g. Touch or click **Run**.



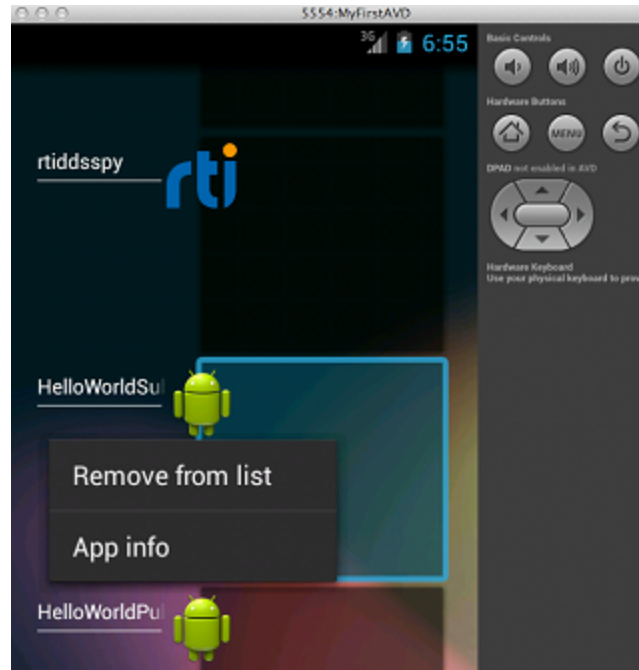
- h. Navigate to the Apps icons and return to **HelloWorld_subscriber**. Note the samples received.



6. An Android App will continue to run in the background when the focus is given to another App. How to stop a running Android App without uninstalling it depends on the device or the virtual device within the emulator. For example, on the emulator hold the **Home** button until a list of applications is shown. Stop an App by swiping it left or right. On a Nexus 7 tablet, touch the **Recent Apps** button (a soft key with an over-lapping rectangle) and swipe the App you want to stop. Alternatively, open **Settings**, choose **Apps**, in the subheadings swipe to **Running**, select **Show cached processes**, and lastly touch and **Stop** the App.



On an emulator, press the Home button to see a list of running apps.



Touch the desired app, then select Remove from list.

Chapter 8 Building and Running a Hello World Example with Java

This section will guide you through the steps required to successfully run an *rtiddsgen*-generated example application written in Java on an Android target as a Dalvik command or as an Android App (*.apk).

1. Compile the generated Java code and archive it into **HelloWorld.jar**. For example:

```
make -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8
```

2. Depending on what form the application is to take, follow the steps in one of these sub-sections:
 - [Building and Running a Java Application as a Dalvik Command \(Section 8.1 below\)](#)
 - [Building and Running a Java Application as an Android App \(Section 8.2 on page 53\)](#)
3. If you want to see DDS log messages, run "adb logcat" with a filter in a host machine shell. For example:

```
adb logcat -v tag RTIConnnextLog:I *:S
```

8.1 Building and Running a Java Application as a Dalvik Command

This procedure not work for a non-rooted Android consumer device, see [Building and Running a Java Application as an Android App \(Section 8.2 on page 53\)](#) instead.

1. Compile the Java class files into Dalvik executable format; archive that in **HelloWorld.dex**. For example:

```
make -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8 HelloWorld.dex
```

2. Push (via **adb**) the native libraries and **HelloWorld.dex** to the emulator or device to a specified user directory. For example:

```
make -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8 HelloWorld.pushexe
```

This will push all the required files to the location specified by **ANDROID_PUSH_DEST**.

If you want to use **USER_QOS_PROFILES.xml** for your example, that file should also be pushed to the emulator or device. To do so, open the generated makefile and uncomment the command that pushes **USER_QOS_PROFILES.xml** to the device in the **HelloWorld.pushdex** target rule before running the above command.

3. To run the example on the emulator: Open two **adb** shells. In each **adb** shell, set up the **LD_LIBRARY_PATH** variable to include the location of the native libraries that are copied. Run the publisher in one shell and the subscriber in the other. For example:

To run the publisher:

```
adb shell
cd /data/data/com.rti.helloworldexample
export LD_LIBRARY_PATH=/data/data/com.rti.helloworldexample:$LD_LIBRARY_PATH
dalvikvm -cp HelloWorld.dex com.rti.example.HelloWorldPublisher <args>
```

To run the subscriber:

```
adb shell
cd /data/data/com.rti.helloworldexample
export LD_LIBRARY_PATH=/data/data/com.rti.helloworldexample:$LD_LIBRARY_PATH
dalvikvm -cp HelloWorld.dex com.rti.example.HelloWorldSubscriber <args>
```

Note: The optional *<args>* can be replaced with the arguments that you want to pass to your example. See the generated source code for usage.

4. Examine the **adb** shell output printed by the examples.


```

adb telnet adb ...
HelloWorld subscriber sleeping for 4 sec...
HelloWorld subscriber sleeping for 4 sec...
HelloWorld subscriber sleeping for 4 sec...
Received:
  prefix: count is ...
  sampleId: 0
  payload: 41

HelloWorld subscriber sleeping for 4 sec...
Received:
  prefix: count is ...
  sampleId: 1
  payload: 41, 42

HelloWorld subscriber sleeping for 4 sec...
Received:
  prefix: count is ...
  sampleId: 2
  payload: 41, 42, 43

HelloWorld subscriber sleeping for 4 sec...
Received:
  prefix: count is ...
  sampleId: 3

```

8.2 Building and Running a Java Application as an Android App

1. See Step 5 in [Generating Example Code and a Makefile from IDL \(Section 4.4 on page 11\)](#) regarding the **ANDROID_PACKAGE_PREFIX** and **TARGET_ID** variables; if you want to change these variables in the makefile, do so now.
2. Create separate Android projects for the publisher and subscriber. For example:

```
make -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8 HelloWorld.projects
```

The project will be placed in the **./objs/<arch>/publisher** and **./objs/<arch>/subscriber** directories. Each project directory will contain files that have been created from provided Connexst DDS templates. These files include the Java Activity class source file, where the Java publisher/subscriber example code is called. The “**android update project**” command is run by the makefile to set the directory up as a correct Android project.

3. Using the created projects, build two Android Apps: one for the publisher and one for the subscriber. For example:

```
gmake -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8 HelloWorld.apks
```

4. Install (via **adb**) the two Apps on a connected consumer device or an emulator. For example:

```
gmake -f makefile_HelloWorld_armv7aAndroid2.3gcc4.8 HelloWorld.installapks
```

5. To run the example on an emulator with a GUI (i.e., not started with the **-no-window** flag) or on a consumer device:

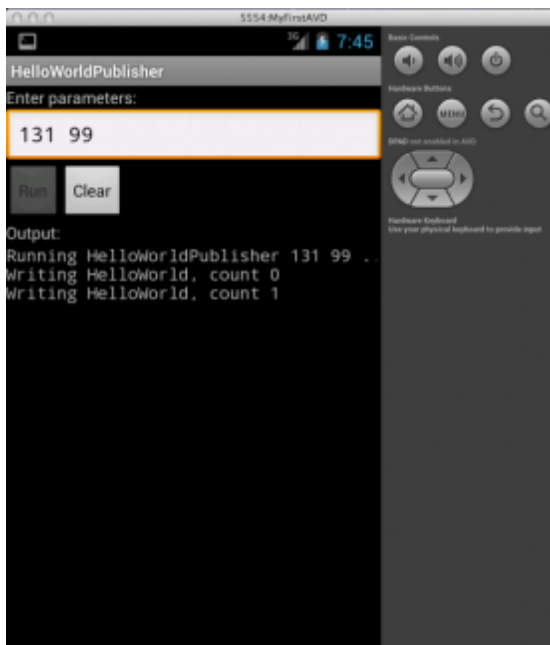
- a. Navigate to the Apps screen displaying icons, it should show **HelloWorldPublisher** and **HelloWorldSubscriber**.



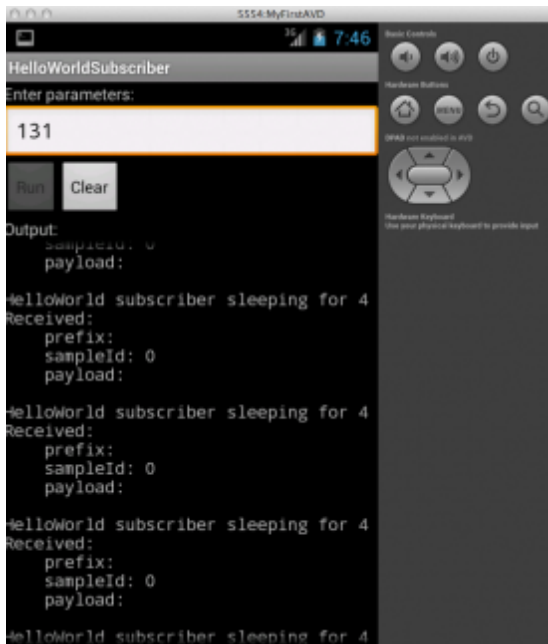
- b. Touch (on a device) or click (on an emulator) the **HelloWorldSubscriber** icon.
- c. Optionally, enter a single number for the DDS Domain ID as a parameter.
- d. Touch or click **Run**.



- e. Navigate to the Apps icons (via the **Home** button) and launch **HelloWorldPublisher**.
- f. Optionally, for parameters, enter the DDS Domain ID (use the same ID entered for the subscriber in [Step](#)) as the first number. Optionally, you can enter the number of samples to be published as the second parameter.
- g. Touch or click **Run**.



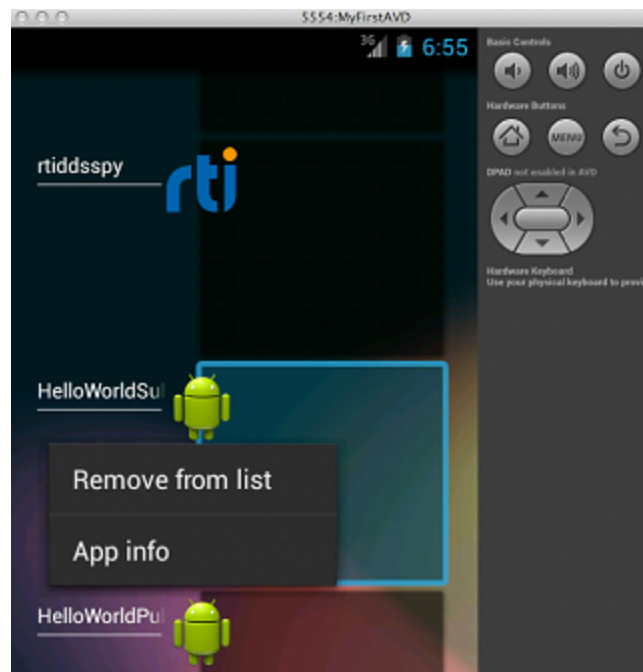
- h. Navigate to the Apps icons and return to **HelloWorldSubscriber**. Note the samples received.



6. An Android App will continue to run in the background when the focus is given to another App. How to stop a running Android App without uninstalling it depends on the device or the virtual device within the emulator. For example, on the emulator hold the **Home** button until a list of applications is shown. Stop an App by swiping it left or right. On a Nexus 7 tablet, touch the **Recent Apps** button (a soft key with an over-lapping rectangle) and swipe the App you want to stop. Alternatively, open **Settings**, choose **Apps**, in the subheadings swipe to **Running**, select **Show cached processes**, and lastly touch and **Stop** the App.



On an emulator, press the Home button to see a list of running apps.



Touch the desired app, then select Remove from list.

Chapter 9 Additional Considerations

9.1 Output when Running Connex DDS Applications on Android Devices and Emulators

9.1.1 DDS Log Messages from Core Libraries

All DDS log messages from the provided core C and C++ libraries are printed using the Android logging system as "information" using the **RTIConnexLog** tag. Log messages from the Connex DDS Java files are printed to **System.out** and **System.err**. The **adb logcat** command without the **-v** option will print all logged messages. To view any of the DDS log messages in real time, use the Android **logcat** tool with the appropriate filter through **adb**. For example:

```
adb logcat -v tag RTIConnexLog:I System.out:I System.err:I *:S
```

Note that if you are calling the **TypeSupport_print_data()** method of any of the *rtiddsgen*-generated data types in C or C++, the value of the samples will also be logged using the Android logging system as stated above.

Refer to the Android documentation for details: <http://developer.android.com/tools/debugging/debugging-log.html>.

9.1.2 Output from *rtiddsgen*-Generated Examples or User Applications

For the *rtiddsgen*-generated Java publisher/subscriber examples, **System.out.println()** is used for printing output.

For the *rtiddsgen*-generated C/C++ examples, **printf()** is used for printing output.

By default, Android systems send stdout and stderr output to **/dev/null**. If you are running the generated publisher/subscriber with the **adb** shell (typically on a device on which you have root access), you should be able to see the example output directly in the shell.

If you are using the Android project files generated by the *rtiddsgen*-generated makefile to compile the publisher and subscriber examples as Android Apps, the generated source code for the application is doing something special to redirect the output to allow them to show up in the Android App.

For the Java example scenario, all the **System.out** output is redirected to the TextView in the GUI of the Android App. Refer to the generated Java Activity class source code under the project **src** directory for details on how this is done.

For the C/C++ example scenario, a special **RTIAndroid_registerOnPrintf()** method can be found in the generated publisher and subscriber example source code to redirect all **printf()** calls to a function callback. The **RTIAndroid_registerOnPrintf()** method is called when the example is run for the first time. Each time **printf()** is called from the generated publisher and subscriber example source code, the registered function callback, implemented in JNI, will be called. In the JNI method, we first attach the current thread to the Dalvik VM if it has not been done, then we call a method in the Java Activity class to display the message in the TextView UI. The native thread will be detached from the Dalvik VM, if necessary, before we return to the native publisher/subscriber example code. Refer to the generated Java Activity class source code under the project **src** directory, JNI code, and publisher/subscriber C/C++ code in the example's main directory for details on how this is done.

Note that when running the *rtiddsgen*-generated examples as an Android App, DDS logging messages and the printing of the Samples in the C or C++ subscriber will still need to be viewed using **logcat**—they will not show up in the GUI of the Android App.

Alternatively, you can change the generated example code to cause all the output to go to **logcat**. For C and C++, replace all calls to **printf()** with **__android_log_print()** with your desired priority and tag. For Java, use methods in the **android.util.Log** class instead of the **System.out** class.

Refer to Android SDK documentation for details: <http://developer.android.com/tools/debugging/debugging-log.html>.

9.1.3 Output from Connex DDS Utilities

The Connex DDS *rtiddsspy*, *rtiddsping* and *rtiddsprototyper* utilities are implemented with Connex DDS core libraries. As a result, DDS log messages must be viewed with **logcat**.

If you run *rtiddsping*, *rtiddsspy*, and *rtiddsprototyper* in a shell, all the output from the utility will show up in the shell (even though DDS log messages still need to be viewed with **logcat**).

If you run *rtiddsping*, *rtiddsspy*, and *rtiddsprototyper* as Android Apps, most of the utility output will show up on the App UI, with the following exceptions. When running *rtiddsspy* with the **-printSample** option as an Android App, and when running *rtiddsprototyper* as an Android App, the value of the received sample is printed to stdout and therefore will not show up in either the UI or in **logcat**. The rest of the output from the utility itself will show up in the App UI as expected. DDS log messages still need to be viewed with **logcat**.

9.2 Communication between Connex DDS Applications on Emulator and Host

The emulator has its own virtual LAN: 10.0.2.*. In particular, 10.0.2.15 is the emulator's address for the host's NIC and 10.0.2.2 is the emulator's address for the host's loopback interface (localhost). To reach addresses outside of its virtual LAN, the emulator will route packets through its virtual network interface, 10.0.2.15 (the host's NIC). For communication from host to emulator, packets must be sent to the host's loopback interface, 127.0.0.1. The emulator can be directed to open ports on the host 127.0.0.1.

The emulator does not (for the version used at time of writing) support multicast. Therefore, discovery will be over unicast UDP. The number of participants within the host + emulator is limited due to the need to allocate ports for each participant. By default, that limit is five participants. In normal Connex DDS usage, the participants would acquire ports automatically. (The assignment would be implicitly based on the order of start up). The emulator that encapsulates one or more participants needs to explicitly acquire the ports that its contained participants will use. Therefore, it is necessary to explicitly manage the ports used.

Connex DDS determines ports for unicast communication based on Domain and Participant IDs. (For the algorithm used to calculate port numbers, see the *RTI Connex DDS Core Libraries User Manual*, Section 14.5).

Consider the following example: A **Hello_simple** subscriber on the host is communicating with a **Hello_simple** publisher (Android) on an emulator.

Assuming Domain ID 0, the subscriber is given a Participant ID of 3. As there will be no contention, the publisher can assume the Participant ID is 0. The publisher will use ports 7410 for meta-traffic and 7411 for user-traffic. The subscriber will use ports 7416 for meta-traffic and 7417 for user-traffic.

To cause the emulator to open ports 7410 and 7411 on behalf of the publisher, it is necessary to set up some redirections. Use the “**adb devices**” command on the host to get the name of the emulator (e.g., “emulator-5554”) and determine its control port (5554).

If the control port is 5554, enter this at the host to determine that no redirections are currently in place:

```
telnet localhost 5554
redir list
```

Then enter the following to set up redirections:

```
redir add udp:7410:7410
redir add udp:7411:7411
```

These redirections apply for 127.0.0.1 on the host, redirected to 10.0.2.2 in the emulator. The **redir** command causes the emulator to open the host port (the first of the two ports specified in the command) on the 127.0.0.1 interface. If another process already has that port, **redir** will report an error.

The publisher, running in the emulator, requires the following settings, shown below in a snippet of a QoS file's `participant_qos` section:

```
<discovery>
  <initial_peers>
    <element>builtin.udpv4://10.0.2.2</element>
  </initial_peers>
</discovery>
<property>
  <value>
    <element>
      <name>dds.transport.UD Pv4.builtin.public_address</name>
      <value>127.0.0.1</value>
    </element>
  </value>
</property>
```

The above sets the locators provided by the publisher; these locators instruct the subscriber to direct traffic to the host's loopback interface. The virtual LAN address that would otherwise be used is not accessible to the subscriber running on the host.

Set the environment for the publisher (on the emulator) so that `NDDS_QOS_PROFILES` points to the QoS file. Note that the **initial_peers** can alternatively be set via the `NDDS_DISCOVERY_PEERS` environment variable.

Note that for an Android App (*.apk), these settings can be made programmatically.

The subscriber, running on the host, requires the following settings, shown below in a snippet of a QoS file's **participant_qos** section:

```
<discovery>
  <initial_peers>
    <element>builtin.udpv4://127.0.0.1</element>
  </initial_peers>
</discovery>
<transport_builtin>
  <mask>UD Pv4</mask>
</transport_builtin>
<wire_protocol>
  <participant_id>3</participant_id>
</wire_protocol>
```

Explicitly attempting to communicate the participant ID prevents the subscriber from implicitly using participant ID 0 (which would cause a later attempt to set redirections for the emulator to fail). Explicitly setting the builtin transport to be UDPv4 only will prevent the host from communicating with the emulator via shared memory.

Set the environment for the subscriber (on the host) so that `NDDS_QOS_PROFILES` points to the QoS file. Note that you can also set **initial_peers** via the `NDDS_DISCOVERY_PEERS` environment variable. All these settings can also be made programmatically.

Finally, if you need to run *rtiddsspy* on the host to communicate with DDS applications running on the emulator, be careful when specifying its transports, discovery addresses, and participant ID. For example:

```
$ (NDDSHOME)/bin/rtiddsspy -peer 127.0.0.1 -transport 1 -index 1
```

There are many variation on this theme to achieve communication between host and emulator. This example illustrates the issues that need to be addressed.

9.3 Loading XML-Specified QoS Settings

If you have root access to the device or you are running on an emulator, you can run your Connex DDS application in the Android shell and set up environment variables in the shell just as you would for any other Linux system. In this scenario, XML QoS profiles can be loaded in the usual ways.

If you want to run *Connex DDS* through an Android App, there is no support for environment variables, which limits the ways that can be used to load XML-specified QoS settings. However, it can still be done by specifying one of the following fields in the `ProfileQosPolicy` in `DomainParticipantFactoryQos`:

- **url_profile**
- **string_profile**

See the *RTI Connex DDS Core Libraries User's Manual* for details.

When running an Android App on a non-rooted consumer device, if you want to use **url_profile** to specify the location of the file that contains the XML-specified QoS settings, it will work only if the file is in Android external storage (for example, under **/sdcard**). You will also need to make sure the **Android-Manifest.xml** file of the project has specified the correct permission to allow reading from external storage.

For example, use the following command to put the file in external storage:

```
adb push <host-qos-file> /sdcard/<target-qos-file>
```

In the Connex DDS source code, specify the location of the file in the external storage in the **url_profile** field of **ProfileQosPolicy** in **DomainParticipantFactoryQos**. For example, in Java, you can do the following before creating DDS entities:

```
DomainParticipantFactoryQos factoryQos = new DomainParticipantFactoryQos();
```

```
DomainParticipantFactory factory = DomainParticipantFactory.get_instance();
factory.get_qos(factoryQos);
factoryQos.profile.url_profile.add("file:///sdcard/USER_QOS_PROFILES.xml");
factory.set_qos(factoryQos);
```

In the **AndroidManifest.xml** file of the project, add the following permission:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Refer to Android documentation for details regarding Android external storage.

9.4 Using DDS C++ Libraries for Android Applications

The DDS C++ libraries are built and linked against the GNU STL C++ shared library.

The *rtiddsgen*-generated makefile for C++ examples builds a shared library (**.so**) that is used by the Android App. When building this shared library, we are linking against the GNU STL C++ static library.

The GNU STL C++ shared library and GNU STL C++ static library are both *not* included as part of the Connex DDS bundle. You can obtain these libraries as part of the Android NDK, available from <https://developer.android.com/tools/sdk/ndk/index.html>.

The GNU STL C++ static library can be used only if it is linked into a single binary. For example, our *rtiddsgen*-generated C++ example only has one shared library that is linked with the GNU STL C++ static library. For this use case, it is fine to use the GNU STL C++ static library.

If your project contains multiple shared-library modules that are linked with the GNU STL C++ runtime, you should use the GNU STL C++ shared library instead.

If you use the GNU STL C++ shared library in the project for your Android App, you will not be able to write pure native code by just using the Android **NativeActivity** class in your App's manifest file. Instead, you will need to write Java code and call **System.loadLibrary()** to explicitly load the GNU STL C++ runtime library first, before loading the shared library of your project. For example:

```
static {
    System.loadLibrary("gnustl_shared");
    System.loadLibrary("HelloWorld_publisher");
}
```

Please also pay attention to the license for GNU STL C++ libraries before using that in your project. Full details can be found here: <http://gcc.gnu.org/onlinedocs/libstdc++/manual/license.html>.

Refer to the Android NDK documentation under "C++ support with the Android NDK" for more details.

9.5 Android Threads

When an Android application is launched, the system creates a thread of execution for the application, called "main." This main thread is also sometimes called the UI thread. Due to Android's single thread model, one should never block the UI thread and one should not access the Android UI toolkit from outside the UI thread.

The project files generated by the *rtiddsgen*-generated makefile has taken care of this for you. If you click the **Run** button to run the publisher/subscriber example, a new thread is created to handle the execution of the example, so that the UI thread will not be blocked. Similarly, if the Android UI needs to be updated (for example, to display output of the publisher/subscriber example in the TextView UI), **runOnUiThread()** is called to make sure the display is always updated in the UI thread.

See the generated Java Activity class source code under the project **src** directory for details. If you are building from the command line, this source code was created in [Step](#) in [Building and Running a Java Application as an Android App \(Section 8.2 on page 53\)](#). If you are building with Eclipse, you will not have this file.

9.6 Setting Permission in App Manifest

When using Connex DDS on an Android platform as an App, you need to grant the proper permissions for networking in **AndroidManifest.xml** for your project:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.CHANGE_WIFI_MULTICAST_STATE"
/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
```

If you want to use Android external storage to load XML-specified QoS settings for your Connex DDS Android App, you also need to grant the proper permissions to read the external storage:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Refer to Android documentation for more details: <http://developer.android.com/reference/android/Manifest.permission.html>.