ADAPT Programmer's Guide

April 2013

Prepared for:

Defense Advanced Research Projects Agency
Research Development and Engineering Command

Prepared By: Leidos Corporation

ISR Sensing Division
10260 Campus Point Drive
San Diego, CA 92121

Short-Title of Work:	ADAPT Programmer's Guide
Document Number:	SAIC-12/3011

Revision History

Revision	Date	Comments
0.9	10-Jan-2013	Draft. Several TBDs remain.
1.0	24-Apr-2013	Initial release
1.1	24-Apr-2014	Name change from SAIC to Leidos, re-write of NDK build, updates throughout

Table of Contents

1. Introduction

The ADAPTable Sensor System (ADAPT) consists of a core hardware platform based on the Qualcomm Snapdragon chipset (used in cell phones), a core operating system based on Android, and middleware and applications provided by other contractors on the program. The intent of the ADAPT Core (hardware and software) is to provide a basis for rapid development of systems of interest to DARPA.

The ADAPT OS is the name of the core operating system. The ADAPT OS provides extensive functionality which includes both embedded Linux and Android version 4.1.2 (Ice Cream Sandwich). The ADAPT OS supports downloading of general purpose and mission-specific applications (apps) over USB or Wi-Fi connections.

This Programmer's Guide describes several methods to develop applications to run on the ADAPT OS. This guide also describes the methods of installing the software, and recommended locations in the file system of the Core for the software. In brief, the varieties of programming methods are list in Table 1.

Table 1: Enumeration of Programming Methods

Programming Method	Description
Android app SDK-based	Customary Android app using Android SDK (all Java)
Android app NDK-based	Hybrid Android app using Android NDK (supports C and Java)
Native Linux application	C/C++ Application that runs on the host Linux independently of Android

1.1 ADAPT OS Lineage

The ADAPT OS is the end result of several stages of development spanning different organizations. Understanding this may be useful to programmers.

1 Google provides the Android Open Source Project (AOSP) (source code) for open access. This includes a customized Linux kernel.
2 The Code Aurora Forum (CAF), sponsored by Qualcomm, starts with the AOSP and applies modifications to enable operation on Qualcomm processors (as used in the ADAPT Core).
3 The Original Design Manufacturer (ODM) vendor for the ADAPT Core hardware starts with the CAF code and applies modifications to support the specific onboard peripherals and sensors on the ADAPT Core hardware. This release is named Titan by the ODM.
4 Leidos starts with the Titan source code and further customizes for the ADAPT program goals and then makes stable source and binary releases available that can be obtained from LS Research (http://lsr.smartfile.com, password required), and as stable/unstable

development source (https://trac.leidoshost.com/gerrit/platform/manifest, password required). This release is called the ADAPT OS.

5 ADAPT contractors with repository access credentials may add modifications or entire applications to the ADAPT OS source repository. These changes then become part of the ADAPT OS in subsequent releases.

1.2 Audience

This document is intended primarily for applications programmers, and secondarily for parties interested in estimating development efforts for applications to run on the ADAPT Core.

1.3 Glossary

Table 2: Glossary

Term	Meaning
ADAPT Core	The hardware platform and provided operating system. The Core hardware includes the processor, several radios, memory, and onboard sensors. The Core does not include antennas, power supply (battery), packaging, or mission-specific sensors.
API	Application Programming Interface. A specification of a software interface. A common case is that an interface is implemented in libraries by a provider, and the application developer makes calls into the library by adhering to the specification.
SDK	Software Development Kit. Android (Java) libraries and APIs for development of applications strictly using Java
NDK	Native Development Kit. C libraries and APIs for development that allow a mixture of C and Java code.
CodeSourcery	A cross-compilation toolchain offered by Mentor Graphics.
ODM	Original Design Manufacturer. The vendor supplying the ADAPT Core hardware.
AOSP	Android Open Source Project. Sponsored by Google.
CAF	Code Aurora Forum. Sponsored by Qualcomm.
Eclipse	An open source Integrated Development Environment (IDE) (editor/compiler/debugger).
OTA	Over-The-Air. In this context, load software over a wireless connection.

1.4 References

LSR8960 Software Programmers Manual, 2012-08-22, version 1.04, http://lsr.smartfile.com/ftp/login/, (password required – ask LSR), last accessed 31-Mar-2014.

ADAPT Core Software Datahandler and SAS User's Guide, July 2012, version 1.0, https://trac.leidoshost.com/trac/ADAPT/wiki/ProgrammersGuide, (password required - ask Leidos), last accessed 31-Mar-2014

https://trac.leidoshost.com/gerrit/adapt/manifest, (password required - ask Leidos), last accessed 15-Apr-2014

http://developer.android.com/about/index.html, last accessed 31-Mar-2014

http://developer.android.com/tools/help/adb.html , last accessed 31-Mar-2014

http://developer.android.com/sdk/index.html, last accessed 31-Mar-2014

http://developer.android.com/tools/sdk/ndk/index.html, last accessed 31-Mar-2014

http://developer.android.com/guide/components/index.html, last accessed 31-Mar-2014

2. ADAPT OS Programming Tools and Environment

2.1 Comparison of Methods

The ADAPT OS supports several development environments for creating new programs/apps to run on the ADAPT Core. The recommended method for new development is the Android Software Development Kit (SDK) as is used to develop Android phone/tablet apps. Another method of development is to use the Android Native Development Kit (NDK), where C/C++ code may be compiled with Java into an Android application. Finally, legacy C/C++ code may be run on the Linux kernel independent of Android. Table 3 provides a summary of the pros and cons of the different development methods.

Table 3: Overview of Development Methods

Method	Document Section	Pros	Cons	Best Use
Android SDK	2.3 (full sample in Appendix A)	Android provides consistent API to sensors and services.	Subject to Android policies (e.g. low-memory actions and garbage collection)	➢ New development. ➢ Middleware/General purpose apps. ➢ Apps without extreme time constraints
Android NDK	2.4 (full sample in Appendix B)	Android provides consistent API to sensors and services.	Subject to Android policies (e.g. low-memory actions and garbage collection)	➢ New development. ➢ Middleware/General purpose apps.
Native Linux	2.5 (full sample in section 2.5)	Allows re-use of an organization's C/C++ development environment and tools.	No opportunity to leverage Android API for sensors and services.	➢ Legacy C/C++ code within the organization. ➢ Porting open source C/C++ projects. ➢ Apps with extreme time constraints

2.2 Development setup

2.2.1 Java

Java 6 is required for development. If Java is not installed already in your development environment, go to http://www.oracle.com/technetwork/java/javase/downloads/index.html, scroll to the bottom and select 'Previous Releases', then select Java SE 6, then select the top item in the list, likely 'Java SE Development Kit 6u45', or this direct link may still be available: http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase6-

419409.html#jdk-6u45-oth-JPR. Select the appropriate Windows or Linux package (e.g.; jdk-6u45-linux-x64.bin for 64-bit Ubuntu). Registration with Oracle is required in either case.

Once downloaded, install the Java package in any location of your choosing. On Linux, see Table 4 for installation commands.

Table 4: Installing Java from the command line

```
$ chmod a+x jdk-6u45-linux-x64.bin
$ ./jdk-6u45-linux-x64.bin

# the expansion of these commands should go in ~/.bashrc
$ export JAVA_HOME=`pwd`/jdk-6u45-linux-x64
$ export PATH=$PATH:$JAVA_HOME/bin
```

2.2.2 SDK (Eclipse/ADT)

The Eclipse IDE with the Android Developers Tool (ADT) plugin is required for SDK-based development. The ADT may be added to an existing Eclipse installation; however, the Android project provides an Eclipse/ADT bundle that includes all essential Android SDK components. The current ADT bundle is available at http://developer.android.com/sdk/index.html. The full details of downloading and installing the Android development environment can be found at http://developer.android.com/sdk/installing/bundle.html.

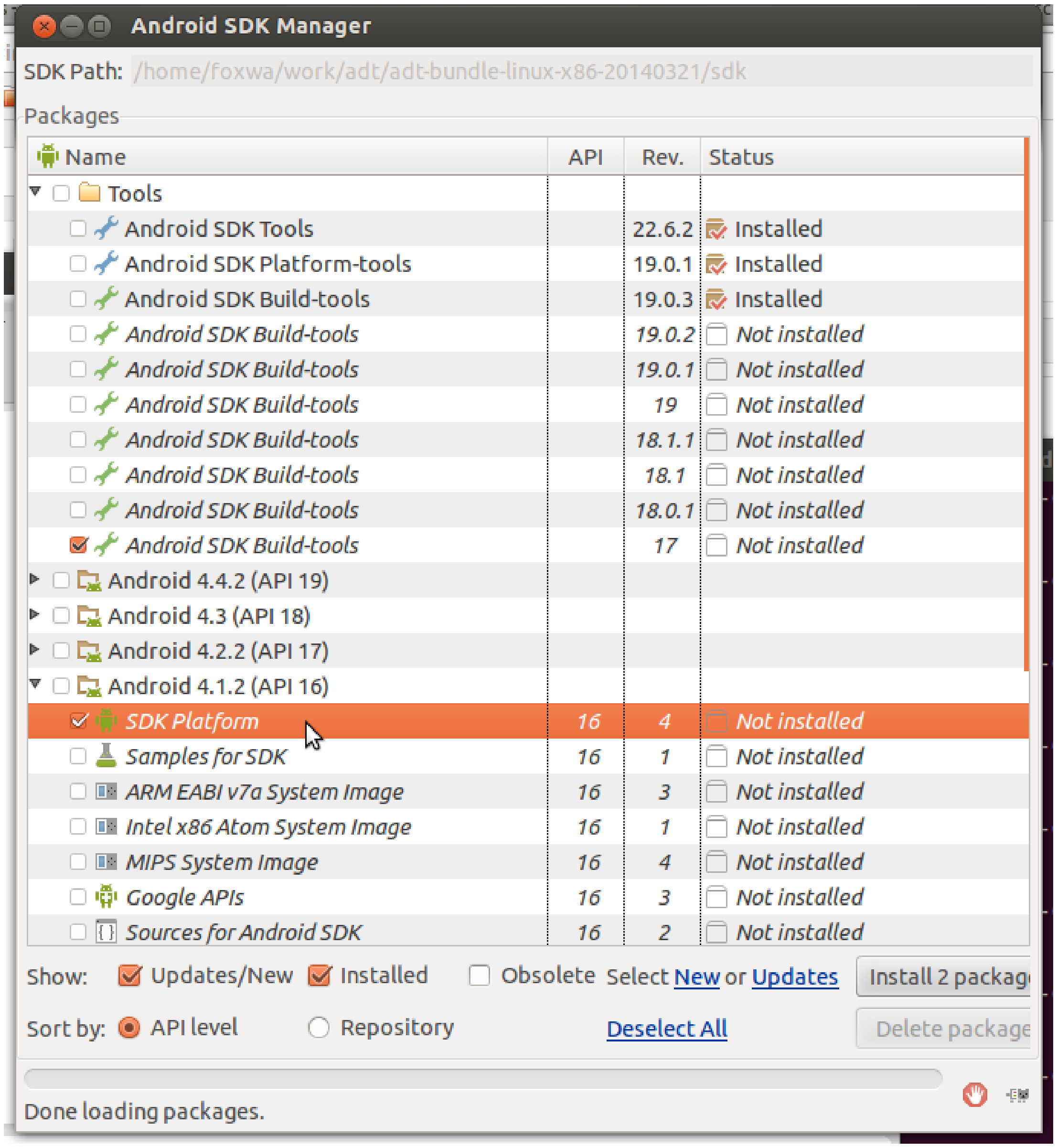
Once the Eclipse/ADT package is installed, a specific version of the ADT tools must be installed to match that of the ADAPT OS (Android 4.1.2). To install specific tool versions, run the *Android SDK Manager* launched by either of two methods:

- run Eclipse[1] (located in `*installdir*/eclipse/`)
    - accept the default workspace
    - your choice on "Contribute Usage Stats"
    - menu – *Window -> Android SDK Manager*
- on a Linux command line, `*installdir*/sdk/tools/android sdk`

[1] If Eclipse fails to launch at this point, choose the command line option or ensure Java is installed.

In the ensuing user interface (shown in Figure 1), select these two packages for installation by navigating the triangles on the left side:

- Tools -> Android SDK Build-tools, Rev. 17
- Android 4.1.2 (API 16) -> SDK Platform, API 16



Android SDK Manager

SDK Path: /home/foxwa/work/adt/adt-bundle-linux-x86-20140321/sdk

Packages

Name	API	Rev.	Status
☐ Tools			
☐ Android SDK Tools		22.6.2	Installed
☐ Android SDK Platform-tools		19.0.1	Installed
☐ Android SDK Build-tools		19.0.3	Installed
☐ Android SDK Build-tools		19.0.2	Not installed
☐ Android SDK Build-tools		19.0.1	Not installed
☐ Android SDK Build-tools		19	Not installed
☐ Android SDK Build-tools		18.1.1	Not installed
☐ Android SDK Build-tools		18.1	Not installed
☐ Android SDK Build-tools		18.0.1	Not installed
☑ Android SDK Build-tools		17	Not installed
☐ Android 4.4.2 (API 19)			
☐ Android 4.3 (API 18)			
☐ Android 4.2.2 (API 17)			
☐ Android 4.1.2 (API 16)			
☑ SDK Platform	16	4	Not installed
☐ Samples for SDK	16	1	Not installed
☐ ARM EABI v7a System Image	16	3	Not installed
☐ Intel x86 Atom System Image	16	1	Not installed
☐ MIPS System Image	16	4	Not installed
☐ Google APIs	16	3	Not installed
☐ Sources for Android SDK	16	2	Not installed

Show: ☑ Updates/New ☑ Installed ☐ Obsolete Select New or Updates

Install 2 packag

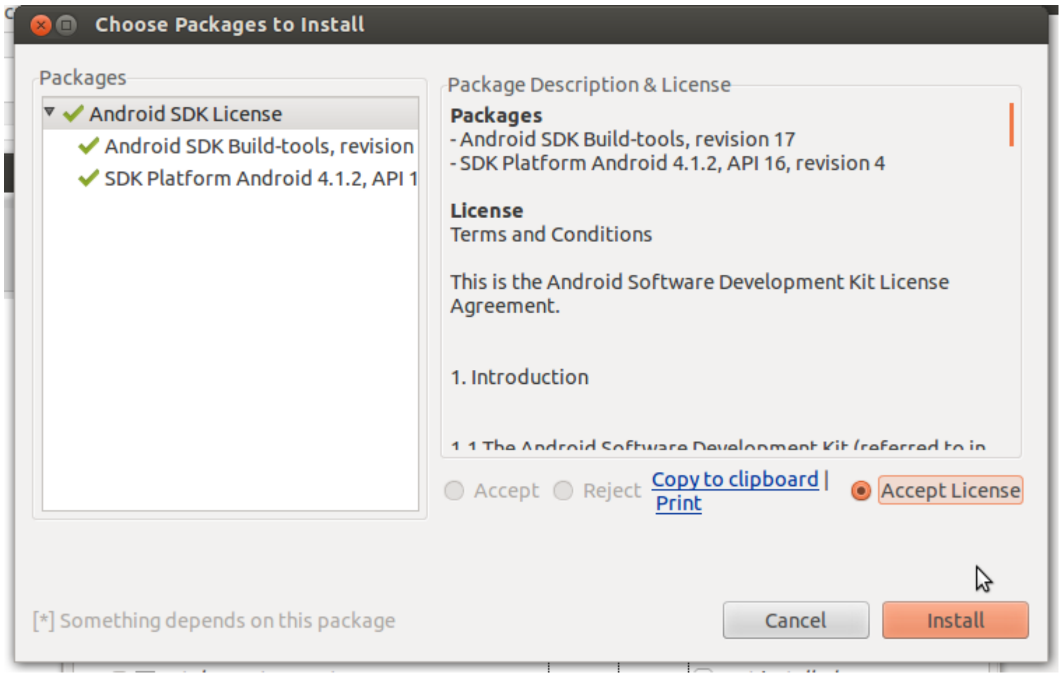
Sort by: ● API level ○ Repository Deselect All

Delete package

Done loading packages.

Figure 1: Android SDK Manager

Then select "Install packages…" towards the lower right corner. In the subsequent dialog, review the packages, accept the license, and Install (see Figure 2).


Choose Packages to Install
Packages
Android SDK License
Android SDK Build-tools, revision
SDK Platform Android 4.1.2, API 1
Package Description & License
Accept
Reject
Copy to clipboard | Print
Accept License
[*] Something depends on this package
Cancel
Install

Packages
- Android SDK Build-tools, revision 17
- SDK Platform Android 4.1.2, API 16, revision 4

License
Terms and Conditions

This is the Android Software Development Kit License Agreement.

1. Introduction

Figure 2: Install SDK packages

2.2.3 NDK

The Android NDK is required for NDK-based development. Download the NDK package from http://developer.android.com/tools/sdk/ndk/index.html. In your selected install area (suggest `/opt`), unpack the downloaded file using *bunzip2* (Linux). On Linux, this completes the installation; however, you may which to shorten the pathname with a symbolic link *ndk* to the actual directory (e.g., `ln -s android-ndk-r9d-linux-x86_64 ndk`). Note the pathname of the directory at the top of the install tree, this is called NDK_DIR in later examples (e.g., `NDK_DIR = /opt/ndk`).

Documentation on Android NDK development is located in the NDK_DIR/*docs/* directory. Start by loading the file *sidenav.html* in your browser, this is the index to the other files. Windows users should read the link listed as "Installation Guide" (in the *sidenav.html* page) to learn of additional setup for Windows.

2.2.4 GCC cross-compiler

A GCC cross-compiler is suggested for Native Linux development. An easy to install pre-compiled package is available for Ubuntu. The cross compiler may be installed with the commands shown in Table 5[2].

[2] The CodeSourcery toolchain works equally well once installed. Installation of CodeSourcery is not documented here, see the Mentor Graphics website.

Table 5: Installation of gcc cross-compiler on Ubuntu

```
$ sudo add-apt-repository ppa:linaro-maintainers/toolchain
$ sudo apt-get update
$ sudo apt-get install gcc-arm-linux-gnueabi
$ which arm-linux-gnueabi-gcc   # confirmation
$ /usr/bin/arm-linux-gnueabi-gcc
```

2.3 Android SDK application development

Android SDK development yields apps which are fully integrated into the Android system, i.e., the full Android API is available to developers, and at run-time, the app is managed by Android. This type of development uses the Java programming language.

Development for ADAPT OS has one major difference from regular Android development. Since the ADAPT OS is designed to run headless (without a screen and input device), use of the Android *Activity* class is disallowed. The Android *Service* class must be used instead to allow for headless operation of ADAPT OS applications. To catch this condition early in development, attempts to install applications that use the *Activity* class will fail with the errors; this is illustrated in 2.3.1.4 Troubleshooting.

Android SDK development requires Java and the Eclipse/ADT; installation of these utilities is described in Sections 2.2.1 Java and 2.2.2 SDK (Eclipse/ADT).

2.3.1 Hello World Example

2.3.1.1 Develop

This section presents an overview of creating a Hello World example application. Note, this example is provided in *adapt/ExampleApps/android/HelloWorld* for those that have access to the Leidos repository[3]. Appendix A presents more detailed information including screen captures of Eclipse dialog windows. The basic steps are:

1. Create an Android Application as a template in Eclipse.
2. Modify the template to add a *Service* and *Broadcast Receiver*.
3. Edit the *Service* and *Broadcast Receiver* to listen and act upon the ADAPT-specific Android intents (`ADAPT_BOOT_COMPLETED, ACTION_HELLO_APP_RUN).`
4. Edit the *AndroidManifest.xml* file to request delivery of the Android intents.
5. Build the final apk with *Run -> Run As -> 1 Android Application*. On success, you will see the *Android Device Chooser*. On failure, an error dialog box comes up. In this case, check the Problems tab for more information.
6. Upon successful installation and verification (next sections), return to this source code and begin extending the functionality to meet your needs.

[3] https://trac.leidoshost.com/gerrit/adapt/manifest

This application registers interest in the `ACTION_HELLO_APP_RUN` intent with Android. When this intent is received, the *Broadcast Receiver* launches the *Service* to handle the intent, and the *Service* prints a "Hello World" message (see Table 6) to the Android log. This model may be extended with additional unique intent names to control starting and stopping of various actions run from within the *Service*.

2.3.1.2 Install

The application is built with each file save, each build is successful if no errors appear in the Eclipse *Problems* tab (center pane near bottom). Two final steps are needed to run the application on an ADAPT core.

1. Install the newly built app to a USB-connected ADAPT core either via
    a. Eclipse (*Run -> Run As -> 1 Android Application*)
    b. a Linux command line: `adb install HelloWorld.apk` (this file is located in *HelloWorld/bin/*)
2. Send an Android broadcast via *adb* to complete the setup (permission to install)

```
adb shell am broadcast -a
android.intent.action.ADAPT_BOOT_COMPLETED --include-stopped-
packages
```

2.3.1.3 Verify

The application has now been installed and will run at boot. This may be verified by checking *logcat* output for the message printed by the app. Example command lines for these actions are shown in Table 6.

Table 6: Verification of HelloWorld application

```
# this command shows the app has been installed via ADAPT_BOOT_COMPLETED intent
$ adb logcat -d | grep HelloWorldService
…
I/HelloWorldService(1561): Hello World!!! The HelloWorldService is running.
…
# this command fires the app to run again
$ adb shell am broadcast -a android.intent.action.ACTION_HELLO_APP_RUN

# which can be seen observed by running logcat in another window
$ adb logcat | grep HelloWorldService
…
I/HelloWorldService(1561): Hello World!!! The HelloWorldService is running.
I/HelloWorldService(1561): Hello World!!! The HelloWorldService is running.
…
```

2.3.1.4 Troubleshooting

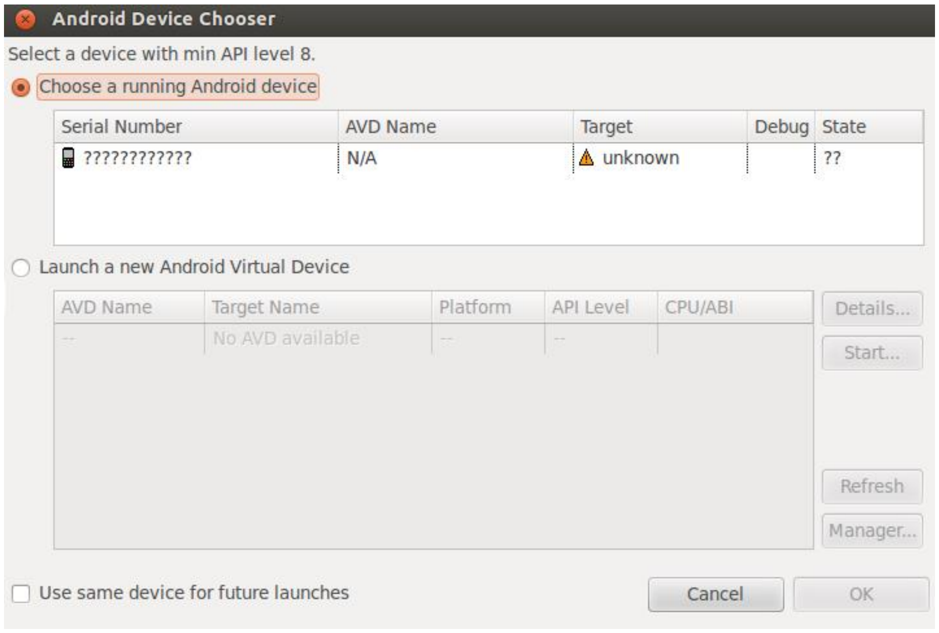
Error generating final archive: Unable to get debug signature key

This error may come up during an initial build. The likely problem is that the Default debug keystore lacks MD5 and SHA1 fingerprints, as seen in *Windows* -> *Preferences* -> *Android* -> *Build*.

The solution is to set the Custom debug keystore to a properly configured keystore (perhaps in *~/.android*). Use the Browse button to add the pathname to a compliant *debug.keystore* file.

Device Chooser unable to report Serial Numbers

If the device serial number(s) are all question marks as shown in Figure 3, then the *adb* daemon does not have the appropriate permissions. The solution is to restart *adb* using the commands as shown in Table 7.


Android Device Chooser
Select a device with min API level 8.
Choose a running Android device
Launch a new Android Virtual Device
Details...
Start...
Refresh
Manager...
Use same device for future launches
Cancel
OK

Serial Number	AVD Name	Target	Debug	State
????????????	N/A	⚠ unknown		??

AVD Name	Target Name	Platform	API Level	CPU/ABI
--	No AVD available	--	--	

Figure 3: Android Device Chooser unable to identify devices

Table 7: Start adb with root permissions on Linux

```
$ adb kill-server
$ sudo adb start-server
```

Invalid APK file

The ADAPT OS disallows installation of applications that use the *Activity* class. Installations that use the *Activity* class will fail to install with the error show in Table 8.

Table 8: Errors when installing applications that use the Activity class

```
Console Error Message:
[Timestamp – Project Name] Installation failed due to invalid APK file!
[Timestamp – Project Name] Please check logcat output for more details.
[Timestamp – Project Name] Launch canceled!
Logcat Error Message:
Package Name: contains activities. Activities are not supported in ADAPT OS
```

2.4 Android NDK application development

Android NDK development using Eclipse yields apps which are fully integrated into the Android system, i.e., the full Android API is available to developers, and at run-time, the app is managed by Android. This type of development uses a combination of C/C++ and Java programming languages.

This method allows for C code to be included within an Android app. The control layer of the app is Java, yet the bulk of the processing may be performed in C. For more documentation on this style, including the pros and cons, see the documentation in *NDK_DIR/docs/Overview.html*. An excerpt from that document is included here:

> The NDK is *not* a good way to write generic native code that runs on Android devices. In particular, your applications should still be written in the Java programming language, handle Android system events appropriately to avoid the "Application Not Responding" dialog or deal with the Android application life-cycle.
>
> Note however that it is possible to write a sophisticated application in native code with a small "application wrapper" used to start/stop it appropriately.

Android NDK development requires Java and the NDK; installation of these utilities is described in Sections 2.2.1 Java and 2.2.3 NDK.

2.4.1 Hello World C/JNI Example

2.4.1.1 Develop

This section presents an overview of creating a Hello World example application. Note, this example is provided in *adapt/ExampleApps/android/NDKSampleApp* for those that have access to the Leidos repository[3]. Appendix B presents more detailed information including screen captures of Eclipse dialog windows. The basic steps are:

1. Create an Android Application as a template in Eclipse.
2. Modify the template to add a *Service* and *Broadcast Receiver*.
3. Edit the *Service* and *Broadcast Receiver* to listen and act upon the ADAPT-specific Android intents (`ADAPT_BOOT_COMPLETED, ACTION_NDK_APP_RUN`).
4. Extend the Application to add Native Support (adds *jni/* directory).

5. Disable CDT (C code) builds in the *jni/* directory
6. Create a C header file for the public interface in *jni/* using the *javah* tool
7. Add *Android.mk* and *Application.mk* files to the *jni/* directory
8. Edit the template C file in *jni/*
9. Edit the *AndroidManifest.xml* file to request delivery of the Android intent.
10. Build this app starting with an NDK-specific build of the C/C++ source code in *jni/*. This build may be achieved in either of two ways:
    a. Manually, by running the command line *ndk-build* in either *project* or *project/jni*.
    b. Automatically, by configuring auto run of *ndk-build*, see section B.10.1 Auto build of C library.
11. Build the final apk with *Run -> Run As -> 1 Android Application*. On success, you will see the *Android Device Chooser*. On failure, an error dialog box comes up. In this case, check the Problems tab for more information.
12. Upon successful installation and verification (next sections), return to this source code and begin extending the functionality to meet your needs.

This application registers interest in the `ADAPT_BOOT_COMPLETED` and `ACTION_NDK_APP_RUN` intents with Android. The former intent is used for installation and launch at boot. The latter intent is used to launch a service dynamically (if desired). In this example, both intents lead to the *Service* printing a "Hello World" message to the Android log. This model may be extended by adding additional unique intents to control starting and stopping of various actions run the *Service*.

2.4.1.2 Install

The application is built with each file save, each build is successful if no errors appear in the Eclipse window. Two final steps are needed to run the application on an ADAPT core.

1. Install the newly built app to a USB-connected ADAPT core either via Eclipse (*Run -> Run As -> 1 Android Application*), or from a Linux command line (`adb install NDKSampleApp.apk`).
2. Send an Android broadcast via *adb* to complete the setup
    a.
    ```
    adb shell am broadcast -a
    android.intent.action.ADAPT_BOOT_COMPLETED --include-stopped-
    packages
    ```

2.4.1.3 Verify

The application has now been installed and will run at boot. This may be verified by checking *logcat* output for a message from this service and/or by checking *ps* for a running process. Example command lines for these actions are show in Table 9.

Table 9: Verification of NDKSampleApp application

```
$ adb logcat -d | grep NDKService
…
I/NDKService(12303): onCreate called
I/NDKService(12303): onStartCommand: entry
I/NDKService(12303): onStartCommand: Hello from the C World
…
$ adb shell pgrep ndksampleapp
12303
```

2.4.1.4 Troubleshooting

Troubleshooting of the NDK application includes the same issues as described in section 2.3.1.4 (SDK Troubleshooting), and with additional issues related to the *ndk-build* utility. Many problems can be fixed by running the `ndk-build clean` command as shown in Table 10.

Table 10: NDK build command

```
$ cd NDKSampleApp
$ ndk-build clean
$ ndk-build
# or use the verbose flag to see the build command lines
$ ndk-build V=1
```

2.5 Native Linux application development

Although Android runs on top of Linux, Android provides no support for user applications to run strictly within the Linux environment (sans Android). Leidos has added such support which allows ADAPT OS programmers the opportunity to quickly port legacy or open-source C/C++ applications to the ADAPT OS.

This type of development requires the GCC cross compiler, installation of these related utilities is described in Sections 2.2.4 GCC cross-compiler.

2.5.1 Hello World C Example

2.5.1.1 Develop

This section presents the complete instructions for creating a Hello World example application. The source code for *hello.c* is given in Table 11. The makefile for this project is given in Table 12. Both these files are available in *adapt/ExampleApps/c/HelloWorld* for those that have access to the Leidos repository[3].

Table 11: Source code for hello.c

```
#include <stdio.h>
int main(int argc, char **argv) {
  printf(“hello world\n”);
  return 0;
}
```

Table 12: Makefile for hello world

```
CC=arm-linux-gnueabi-gcc

CFLAGS=-march=armv7-a -mfpu=neon-vfpv4 -mfloat-abi=softfp

hello: hello.c

        # tab required

        $(CC) $(CFLAGS) -o hello hello.c
```

To build, run *make* on the Linux command line as shown in Table 13.

Table 13: Building hello world

```
$ make

# tab required

arm-linux-gnueabi-gcc -march=armv7-a -mfpu=neon-vfp4 -mfloat-abi=softfp -o hello
hello.c

$
```

To examine a binary to ensure it was built for arm, use either the Linux *file* command (a summary), or the *objdump* command (more details) as shown in Table 14.

Table 14: Confirmation of compile for ARM

```
$ file hello
hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses
shared libs), for GNU/Linux 2.6.31, not stripped

$ arm-none-linux-gnueabi-objdump -T hello_arm

hello_arm:     file format elf32-littlearm

DYNAMIC SYMBOL TABLE:
00000000  w   D  *UND*     00000000              __gmon_start__
0000830c      DF *UND*     00000000  GLIBC_2.4   puts
00008300      DF *UND*     00000000  GLIBC_2.4   printf
00008334      DF *UND*     00000000  GLIBC_2.4   abort
00008318      DF *UND*     00000000  GLIBC_2.4   __libc_start_main
$
```

2.5.1.2 Install

Linux applications are installed by way of file transfer to a writable location on the device file system, i.e., somewhere within `/data`. With a USB-connected device, the *adb* utility may be used for file transfer as shown in Table 15. In this example, the binary file on the device is given the pathname `/data/local/hello`.

Table 15: Simple installation of hello world

```
$ adb push hello /data/local
```

2.5.1.3 Verify

The binary may be run from the *adb* shell. Program output directed to stdout is reported by *adb*. See Table 16 for an example.

Table 16: Output of hello world

```
$ adb shell /data/local/hello
hello world
$
```

2.5.1.4 Troubleshooting

No execute permission

The error message shown in Table 17 indicates that execute permissions are not set. Set permissions with the *chmod* command as shown.

Table 17: Execute permission denied

```
$ adb shell
root@android: # cd /data/local
root@android:/data/local # ./hello
/system/bin/sh: ./hello: can't execute: Permission denied
126|root@android:/data/local # chmod 755 hello
root@android:/data/local # ./hello
hello world
root@android:/data/local #
```

Other run-time failures

Other failures are often due to files not found at run-time. To check for the paths being searched for files, use the *strace* command with the *-eopen* option (a filter) as shown in Table 18.

Table 18: Running strace for diagnostics

```
$ adb shell
root@android: # cd /data/local
root@android:/data/local # strace –eopoen ./hello
… [many lines deleted]
open("/lib/libc.so.6", O_RDONLY)    = 3
hello world
root@android:/data/local #
```

3. ADAPT OS Programming API

3.1 Android General API

The ADAPT OS primarily leverages the extensive Android API. The Android API includes the ability to access on-board sensors, manage processes and communicate between processes. See http://developer.android.com/guide/components/index.html for a starting point that describes this API. Please note that the ADAPT OS removes one significant component from the Android API. Due to the lack of a user interface (touchscreen) on the ADAPT Core, the Android *Activity* class (http://developer.android.com/guide/components/activities.html) is disallowed for creating applications. The workaround is to use the Android *Service* class and send *Intents* to the *Service* to control its behavior. See Appendix A of this document for an example of controlling an application built upon a *Service*.

The ADAPT development community is actively creating Android applications, based on *Services*, that expose the current readings from the onboard sensors. One example of this is the GPS location application (*GPSLocService*). This application polls the GPS sensor on occasion, and provides an open Linux socket for applications (including Linux applications) to connect and receive timing and location information. This application and others of this nature (e.g. *WifiService* and *ImageCapture*) are available from the Leidos repository[3].

3.2 Linux Sensors API

In addition to the Android API, the ADAPT OS provides direct Linux-based (*sysfs*) access to some of the single readout onboard sensors (light, proximity, temperature and pressure) and the battery condition. Table 19 lists the direct access sensors and the *sysfs* pathname of the sensor value. To read the value of the sensor, simply read the value in the file.

Table 19: Direct access sensors

Sensor	Sysfs access point	Example value
Light	`/sys/class/input/input0/get_als_data`	7869
Proximity	`/sys/class/input/input0/get_proximity_data`	3
Temperature	`/sys/class/i2c-dev/i2c-11/device/11-0077/get_temperature_data`	346
Pressure	`/sys/class/i2c-dev/i2c-12/device/12-0077/get_pressure_data`	100700
Battery Health	`/sys/class/power_supply/battery/health`	Good
Battery Status	`/sys/class/power_supply/battery/status`	Charging

Battery Voltage	`/sys/class/power_supply/battery/voltage_now`	4253853

3.3 ADAPT Datahandler API

The ADAPT OS provides an API to an embedded mesh network and distributed database. See the ADAPT Core Software Datahandler and SAS User's Guide for details on the programming interface to this set of services.

3.4 ADAPT FPGA API

The ADAPT FGPA API is described in the document ADAPT FPGA 3rd Party IP Integration Guide. This document and release notes for current and older FPGA images is available from the wiki at https://trac.leidoshost.com/trac/ADAPT/attachment/wiki/FPGAload/ (requires login credentials).

4. Installation Methods

Installation of applications onto the ADAPT Core may be accomplished via several methods. During development and test, the most convenient method is over a USB connection to the ADAPT Core. During large scale testing, the most convenient method is over a Wifi connection, which allows installation on multiple ADAPT Core boards in parallel.

4.1 Installation via adb

Installation via *adb* requires a USB connection between the host computer and the ADAPT Core. *Adb* is a utility program available from the Google Android Software Development Kit (SDK) (http://developer.android.com/sdk/index.html). Install the SDK (see 2.2.2 SDK (Eclipse/ADT)) and make appropriate modifications to your PATH variable to include *adb* on the PATH. On Linux, *adb* must be started with root permissions as shown in Table 7.

4.1.1 Install an APK

If using the Android SDK or NDK method of program development, the output of the build process is an Android application package file (APK). The process to install an APK from the command line is to issue two commands, listed here and illustrated in Table 20:

1. `adb install` *program.apk*
2. `adb shell am broadcast -a android.intent.action.ADAPT_BOOT_COMPLETED --include-stopped-packages`

Table 20: Installation of an APK

```
$ ls *.apk
HelloWorld.apk
$ adb install HelloWorld.apk
5929 KB/s (122060 bytes in 0.020s)   <- sample output
        pkg: /data/local/tmp/WifiStart.apk
Success
$ adb shell am broadcast –a
android.intent.action.ADAPT_BOOT_COMPLETED --include-stopped-
packages
Broadcasting: Intent {---}
Broadcast completed: result=0
$
```

4.1.2 Install a native Linux application

If using the native Linux application method of program development, the installation process is by way of file copy using *adb*, or *scp* (over Wifi) or similar. Ensure each component of the application (executable, libraries (if required), data (if required)) is copied. In the case of the sample *hello world* application (), only the executable needs to be installed via the *adb* push command, listed here and illustrated in Table 21:

1. `adb push /`*path-on-host*`/`*program* `/data/app/`*company*`/`*program*

Table 21: Installation of a native Linux application

```
$ adb shell mkdir /data/app/leidos
$ adb push hello /data/app/leidos
165 KB/s (6719 bytes in 0.039s)
$ adb shell /data/app/leidos/hello
hello world
$
```

4.2 Installation location

Table 22 lists the install location within the ADAPT Core for applications installed by a user.

Table 22: Install location

Install method	Application and data location within ADAPT OS
adb install (APK)	`/data/app/`*file.apk* – the app `/data/data/`*reverse-DNS.app*`/` - libraries, run-time data
adb push (Linux exe)	`/data/app/`*organization*`/`*appname* - executable `/data/data/`*organization*`/`*appname*`/` - run-time data

4.3 Launch at boot

Table 23 lists some methods to ensure that user applications get launched at boot of the ADAPT Core.

Table 23: Application launch at boot

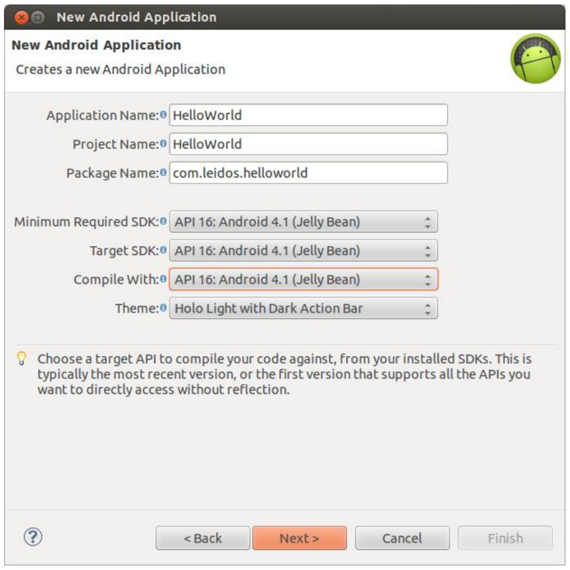
Application type	Method to ensure application launch at boot
APK	1. Add `ADAPT_BOOT_COMPLETED` as an <intent-filter> in the <receiver> section of your app's *AndroidManifest.xml* file 2. In the project, create a *BroadcastReceiver* component 3. In the *BroadcastReceiver*, start your *Service*.
Native Linux	Edit the file */data/app/init.sh* found in ADAPT OS v0.01.06-8 (or newer). Add necessary shell commands to launch your application. Edit the file */data/app/sas/bin/mission.sh* found in ADAPT OS v0.01.06-7 (or older). Add necessary shell commands to launch your application.

Appendix A. Android Hello World Application, SDK-based

This section presents the details of creating a Hello World example application. Note, this example is provided in *adapt/ExampleApps/android/HelloWorld* for those that have access to the Leidos repository.

A.1 Create an Android Application template

Launch Eclipse and select menu option *File -> New -> Android Application Project*. Provide the Application Name of *HelloWorld*, change the organization portion of the Package Name to your organization's name, and select API 16 in the next 3 boxes (if API 16 is not an option, see section 2.2.2 SDK (Eclipse/ADT) for instructions on installing API 16). Figure 4 illustrates this step. On the ensuing dialog box, uncheck the top two items as shown in Figure 5Figure 18.



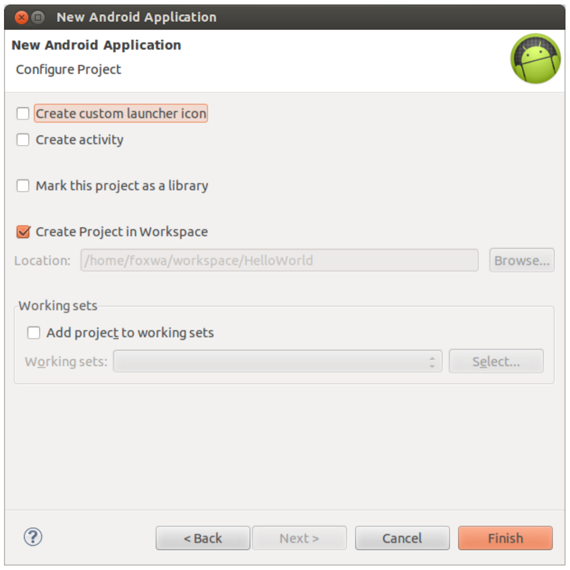
New Android Application
Creates a new Android Application

Application Name: HelloWorld
Project Name: HelloWorld
Package Name: com.leidos.helloworld

Minimum Required SDK: API 16: Android 4.1 (Jelly Bean)
Target SDK: API 16: Android 4.1 (Jelly Bean)
Compile With: API 16: Android 4.1 (Jelly Bean)
Theme: Holo Light with Dark Action Bar

Choose a target API to compile your code against, from your installed SDKs. This is typically the most recent version, or the first version that supports all the APIs you want to directly access without reflection.

Figure 4: Name the new Android Application



New Android Application
Configure Project

☐ Create custom launcher icon
☐ Create activity

☐ Mark this project as a library

☑ Create Project in Workspace
Location: /home/foxwa/workspace/HelloWorld

Working sets
☐ Add project to working sets
Working sets:

Figure 5: Configure Application without activity

Upon completion, open the Package Explorer if not already visible, and then open the *HelloWorld*. The result should look similar to Figure 6.

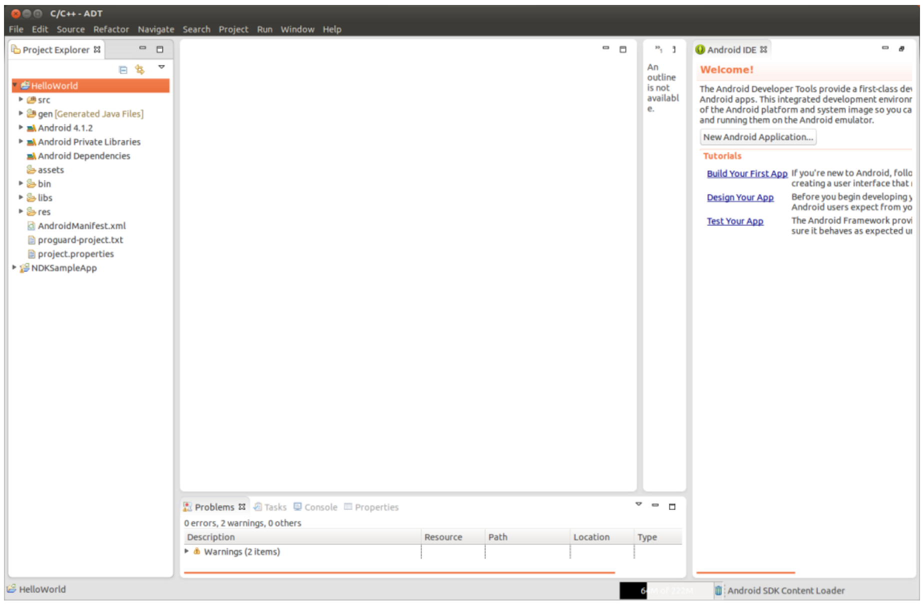


Figure 6: New project HelloWorld

A.2 Modify the template to add a BroadcastReceiver and Service.

First, add a *BroadcastReceiver* class to the project. Within Eclipse, in the Project Explorer pane, right-click on the project name, select *New* -> *Other…*, and then open the Android topic if not open. Then select *Android Object*, followed by *BroadcastReceiver*; then change the Class Name to be *HelloWorldBroadcastReceiver*. See Figure 20, Figure 8, Figure 9, and Figure 10 for this sequence.

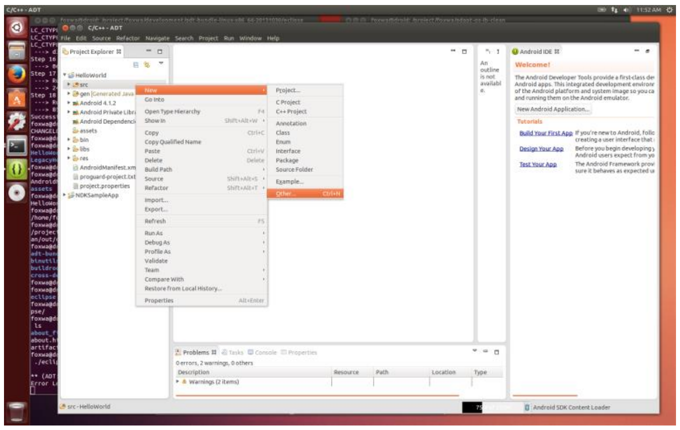
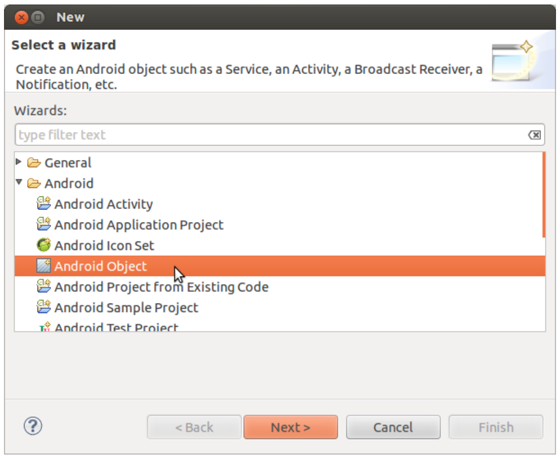


Figure 7: Adding source file to HelloWorld

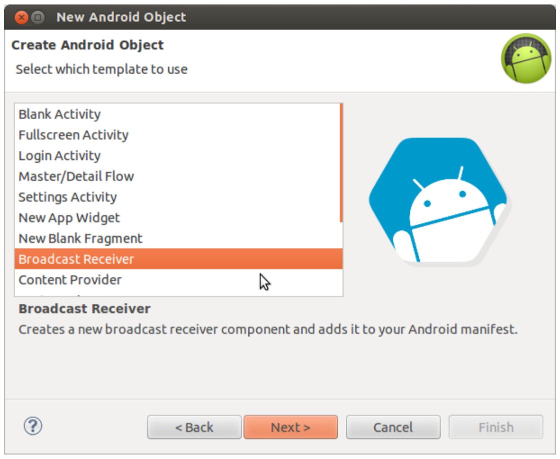

New

Select a wizard

Create an Android object such as a Service, an Activity, a Broadcast Receiver, a Notification, etc.

Wizards:

Figure 8: File is a specialized Android Object


New Android Object

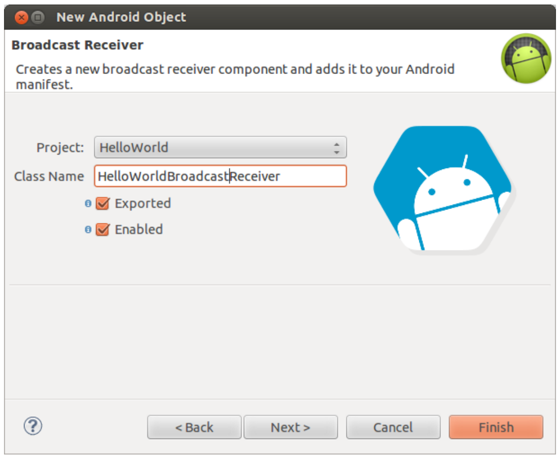
Create Android Object

Select which template to use

Broadcast Receiver

Creates a new broadcast receiver component and adds it to your Android manifest.

Figure 9: Select BroadcastReceiver


New Android Object

Broadcast Receiver

Creates a new broadcast receiver component and adds it to your Android manifest.

Project: HelloWorld

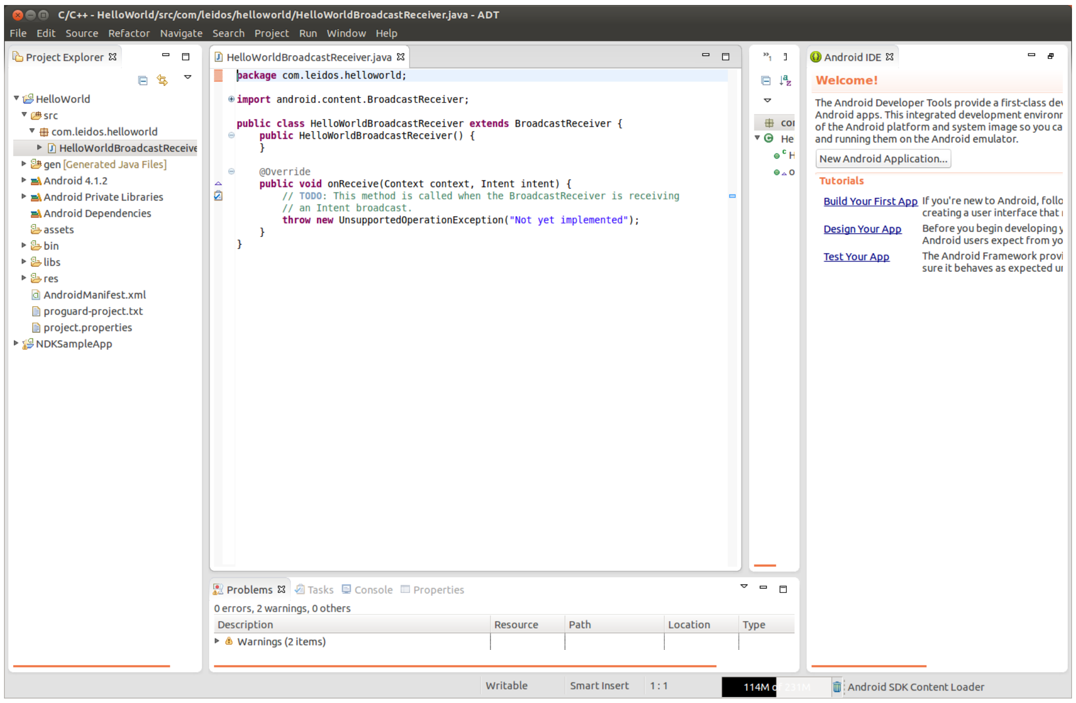
Class Name: HelloWorldBroadcastReceiver

[✓] Exported

[✓] Enabled

Figure 10: Provide name for BroadcastReceiver

After adding the Java class called *HelloWorldBroadcastReceiver*, template code will be provided as shown in Figure 11. This will be replaced by code presented below.



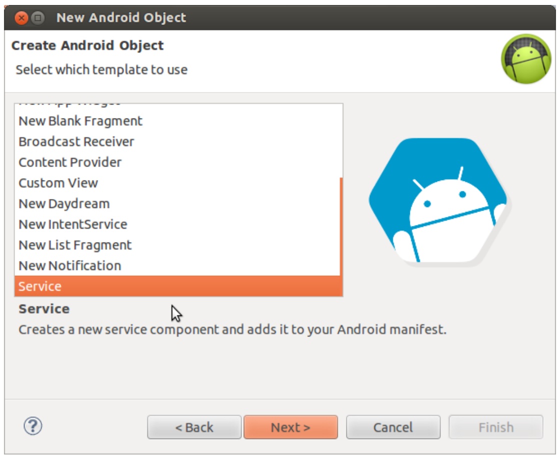
C/C++ - HelloWorld/src/com/leidos/helloworld/HelloWorldBroadcastReceiver.java - ADT

```
package com.leidos.helloworld;

import android.content.BroadcastReceiver;

public class HelloWorldBroadcastReceiver extends BroadcastReceiver {
    public HelloWorldBroadcastReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO: This method is called when the BroadcastReceiver is receiving
        // an Intent broadcast.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

Problems: 0 errors, 2 warnings, 0 others — Warnings (2 items)

Figure 11: Template for BroadcastReceiver

Next, add a *Service* class to the project. Within Eclipse, in the Project Explorer pane, right-click on the project name, select *New -> Other…*, and then open the Android topic if not open. Then select *Android Object*, followed by *Service*; then change the Class Name to be *HelloWorldService*. See Figure 7, Figure 8, Figure 12, and Figure 13 for this sequence.



New Android Object

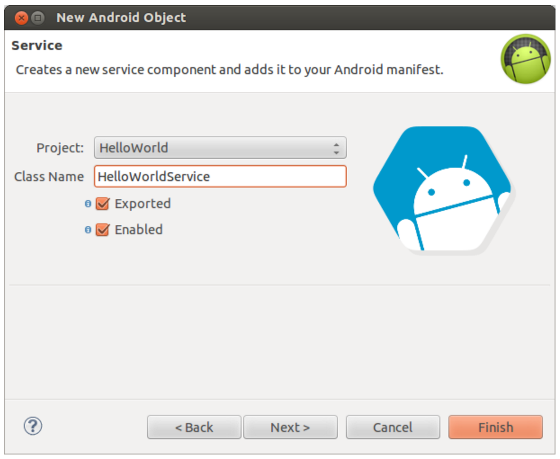
Create Android Object

Select which template to use

- New Blank Fragment
- Broadcast Receiver
- Content Provider
- Custom View
- New Daydream
- New IntentService
- New List Fragment
- New Notification
- Service

Service

Creates a new service component and adds it to your Android manifest.



New Android Object

Service

Creates a new service component and adds it to your Android manifest.

Project: HelloWorld

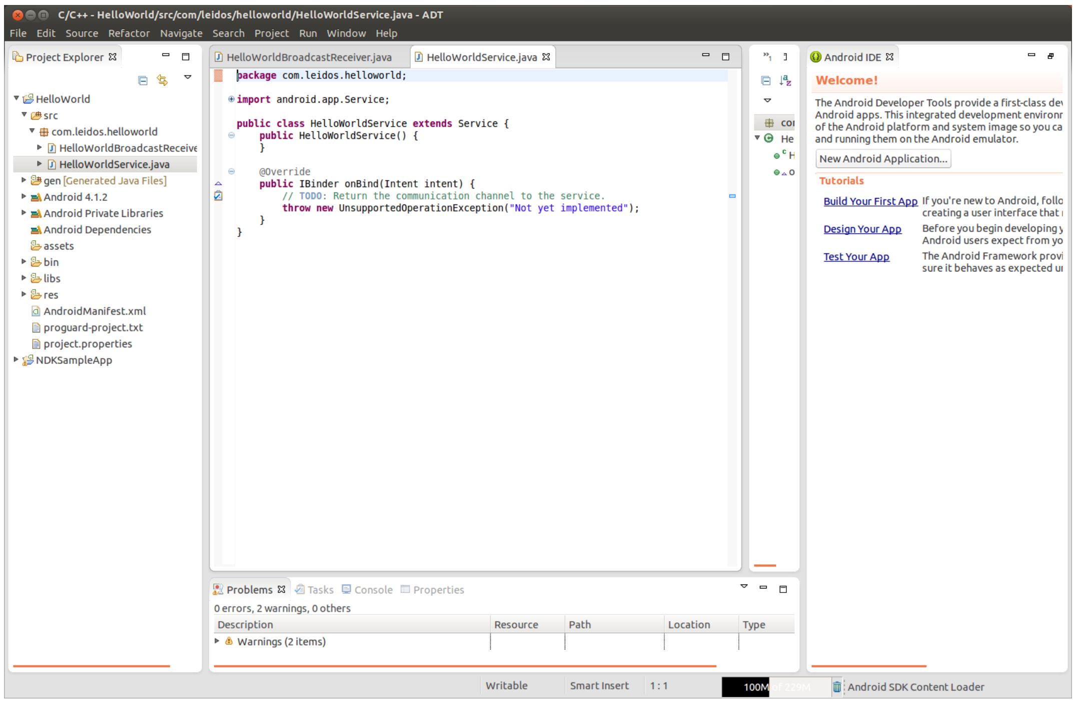
Class Name: HelloWorldService

[✓] Exported

[✓] Enabled

Figure 12: Select Service **Figure 13: Provide name for Service**

After adding the Java class called *HelloWorldService*, template code will be provided as shown in Figure 14. This will be replaced by code presented below.


C/C++ - HelloWorld/src/com/leidos/helloworld/HelloWorldService.java - ADT
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
HelloWorldBroadcastReceiver.java
HelloWorldService.java
Android IDE
Problems
Tasks
Console
Properties
0 errors, 2 warnings, 0 others
Warnings (2 items)
Writable
Smart Insert
1 : 1
Android SDK Content Loader

```
package com.leidos.helloworld;

import android.app.Service;

public class HelloWorldService extends Service {
    public HelloWorldService() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

Figure 14: Template for Service

A.3 Edit the Broadcast Receiver and Service

Now, replace the code in the *HelloWorldBroadcastReceiver.java* and *HelloWorldService.java* files with the source code provided in Table 24 and Table 25. The font has been reduced to save space; presumably readers will be using cut-and-paste to move the code from this document to Eclipse.

Table 24: HelloWorldBroadcastReceiver.java

```
package com.leidos.helloworld;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class HelloWorldBroadcastReceiver extends BroadcastReceiver {
        public HelloWorldBroadcastReceiver() {
        }

        @Override
        public void onReceive(Context context, Intent intent) {
                // This method starts the HelloWorldService
                Intent hwsIntent = new Intent(context, HelloWorldService.class);
                context.startService(hwsIntent);
        }
}
```

Table 25: HelloWorldService.java

```
package com.leidos.helloworld;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
```

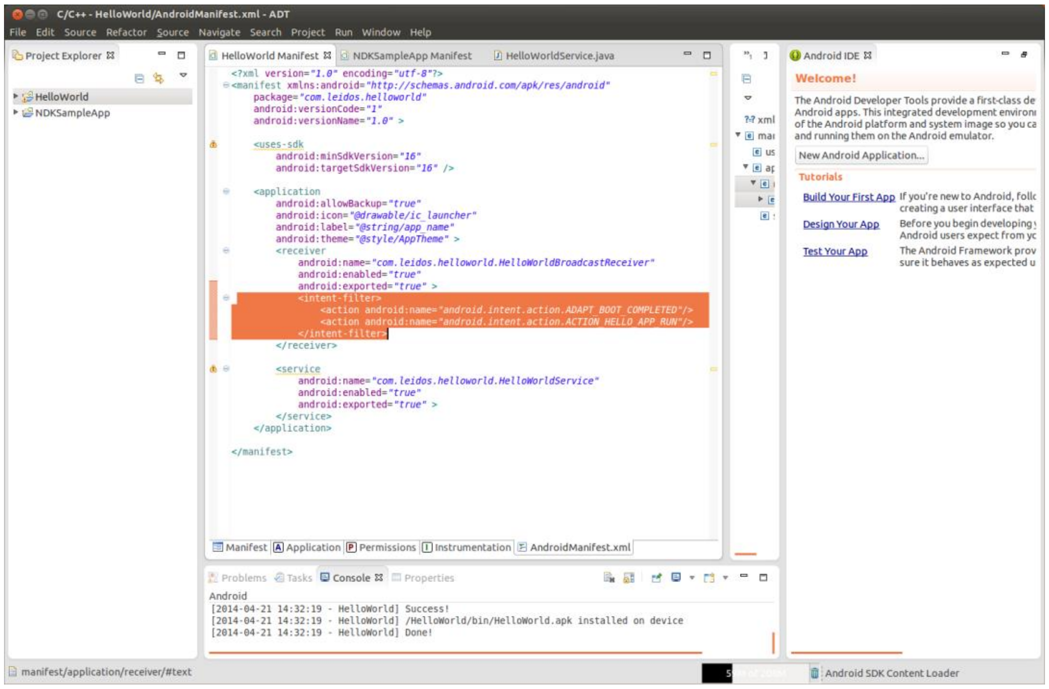
```
public class HelloWorldService extends Service {
        private static final String TAG = "HelloWorldService";
        public HelloWorldService() {
        }

        @Override
        public IBinder onBind(Intent intent) {
                // TODO: Return the communication channel to the service.
                throw new UnsupportedOperationException("Not yet implemented");
        }

        @Override
        public void onStart(Intent intent, int startid) {
                // Code to execute when the service is created
                Log.i(TAG, "Hello World!!! The HelloWorldService is running");
        }

}
```

A.4 Edit the AndroidManifest.xml file

Next, add a section to the *AndroidManifest.xml* (the file at the project level) to request delivery of the Android intent to the *HelloWorldBroadcastReceiver*. The lines to add are provided in Table 26. The final product is shown in Figure 16.

Table 26: Intent filter specification

```
<intent-filter>
   <action android:name="android.intent.action.ADAPT_BOOT_COMPLETED"/>
   <action android:name="android.intent.action.ACTION_NDK_APP_RUN"/>

</intent-filter>
```



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.leidos.helloworld"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="16"
        android:targetSdkVersion="16" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <receiver
            android:name="com.leidos.helloworld.HelloWorldBroadcastReceiver"
            android:enabled="true"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.ADAPT_BOOT_COMPLETED"/>
                <action android:name="android.intent.action.ACTION_HELLO_APP_RUN"/>
            </intent-filter>
        </receiver>

        <service
            android:name="com.leidos.helloworld.HelloWorldService"
            android:enabled="true"
            android:exported="true" >
        </service>
    </application>

</manifest>
```

The Android Developer Tools provide a first-class de
Android apps. This integrated development environ
of the Android platform and system image so you ca
and running them on the Android emulator.

Build Your First App — If you're new to Android, follc creating a user interface that
Design Your App — Before you begin developing Android users expect from yc
Test Your App — The Android Framework prov sure it behaves as expected u

```
Android
[2014-04-21 14:32:19 - HelloWorld] Success!
[2014-04-21 14:32:19 - HelloWorld] /HelloWorld/bin/HelloWorld.apk installed on device
[2014-04-21 14:32:19 - HelloWorld] Done!
```

Figure 16: AndroidManifest.xml with intent filter

A.5 Build the final apk

The final application may be rebuilt at any time within Eclipse by right-click on the Project, *Run -> Run As -> 1 Android Application*. On success, you will see the **Android Device Chooser**, select Cancel. On failure, an error dialog box comes up. In this case, check the **Problems** tab for more information.

A.6 Install and Verify

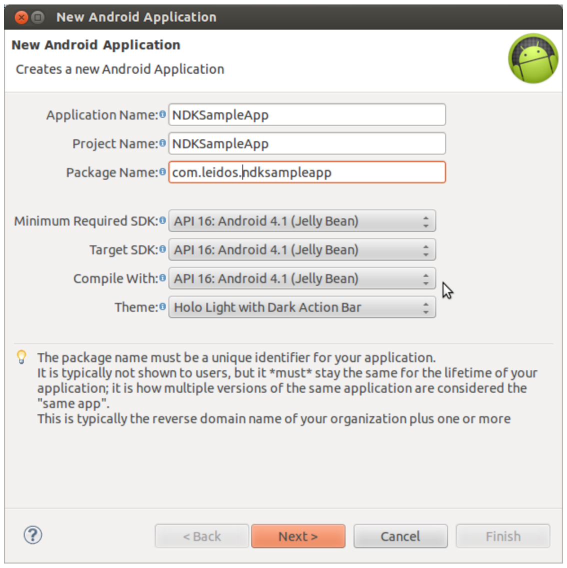
For installation and verification instructions, please refer to sections 2.3.1.2 Install and 2.3.1.3 Verify.

Appendix B. Android Hello World Application, NDK-based

This section presents the details of creating a Hello World example application. Note, this example is provided in *adapt/ExampleApps/android/NDKSampleApp* for those that have access to the Leidos repository.

B.1 Create an Android Application template

Launch Eclipse and select menu option *File -> New -> Android Application Project*. Provide the Application Name of *NDKSampleApp*, change the organization portion of the Package Name to your organization's name, and select API 16 in the next 3 boxes (if API 16 is not an option, see section 2.2.2 SDK (Eclipse/ADT) for instructions on installing API 16). Figure 17 illustrates this step. On the ensuing dialog box, uncheck the top two items as shown in Figure 18.


New Android Application

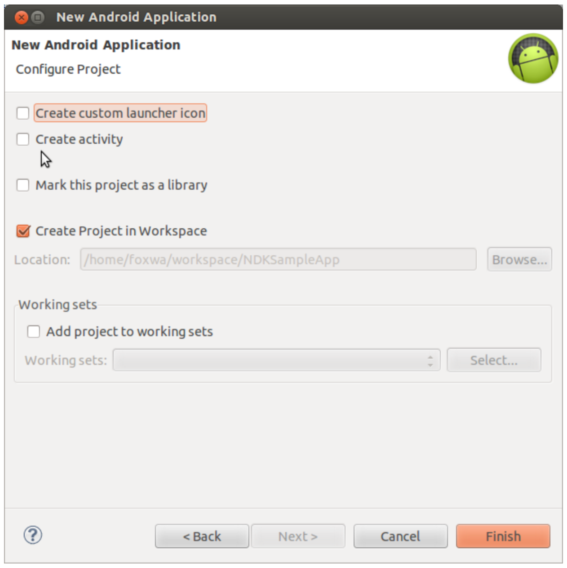
New Android Application
Creates a new Android Application

Application Name: NDKSampleApp
Project Name: NDKSampleApp
Package Name: com.leidos.ndksampleapp

Minimum Required SDK: API 16: Android 4.1 (Jelly Bean)
Target SDK: API 16: Android 4.1 (Jelly Bean)
Compile With: API 16: Android 4.1 (Jelly Bean)
Theme: Holo Light with Dark Action Bar

The package name must be a unique identifier for your application.
It is typically not shown to users, but it *must* stay the same for the lifetime of your application; it is how multiple versions of the same application are considered the "same app".
This is typically the reverse domain name of your organization plus one or more

Figure 17: Name the new Android Application


New Android Application

New Android Application
Configure Project

☐ Create custom launcher icon
☐ Create activity
☐ Mark this project as a library
☑ Create Project in Workspace
Location: /home/foxwa/workspace/NDKSampleApp

Working sets
☐ Add project to working sets
Working sets:

Figure 18: Configure Application without activity

Upon completion, open the Package Explorer if not already visible, and then open the *NDKSampleApp*. The result should look similar to Figure 19.

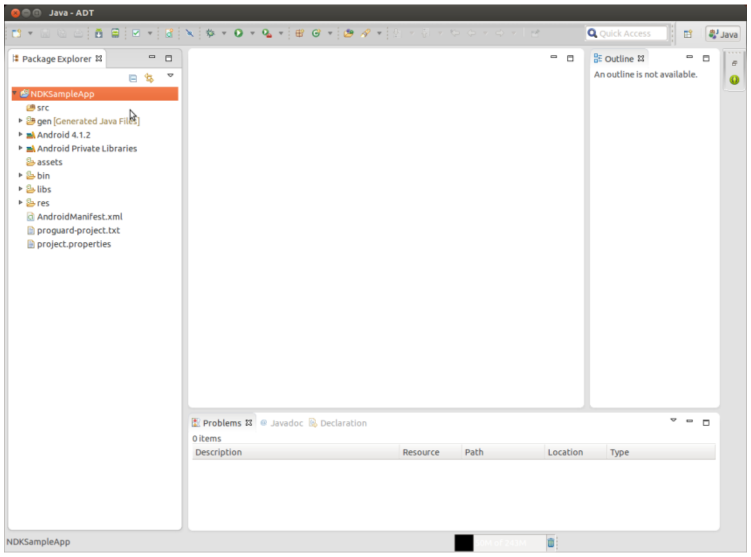


Figure 19: New project NDKSampleApp

B.2 Modify the template to add a BroadcastReceiver and Service.

First, add a *BroadcastReceiver* class to the project. Within Eclipse, in the Project Explorer pane, right-click on the project name, select *New* -> *Other…*, and then open the Android topic if not open. Then select *Android Object*, followed by *BroadcastReceiver*; then change the Class Name to be *NDKBroadcastReceiver*. See Figure 20, Figure 21, Figure 22, and Figure 23 for this sequence.

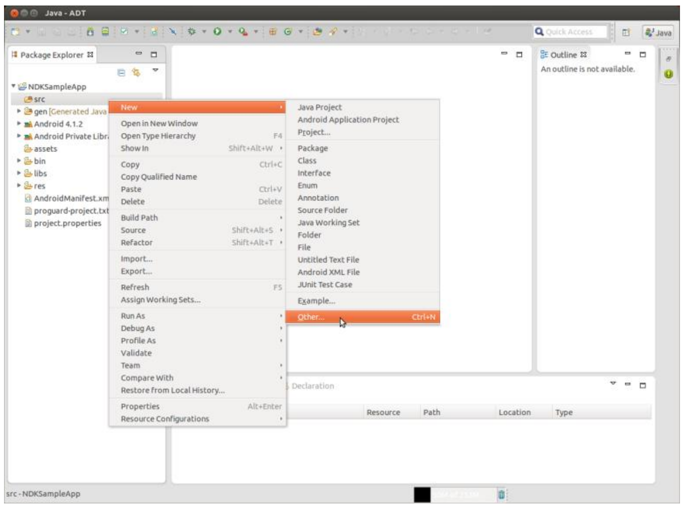
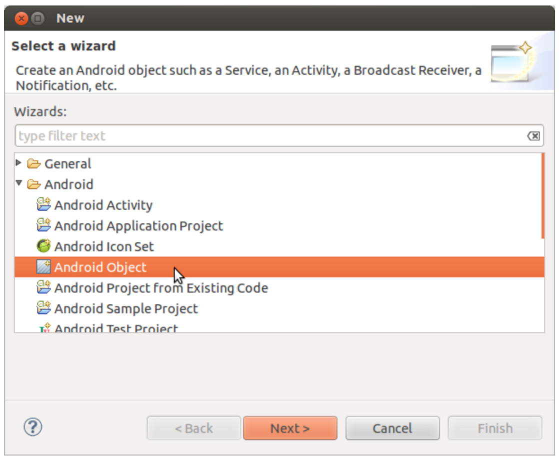
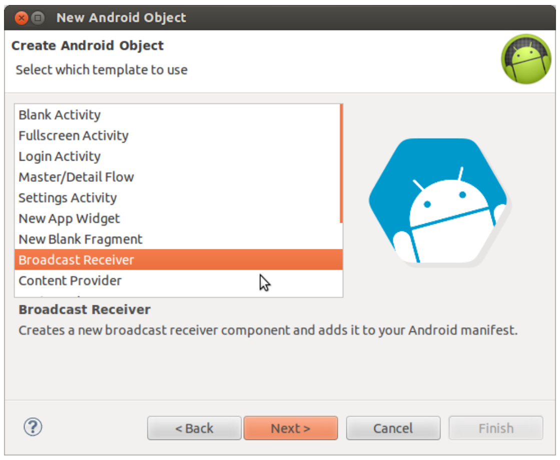


Figure 20: Adding source file to NDKSampleApp



Create an Android object such as a Service, an Activity, a Broadcast Receiver, a Notification, etc.

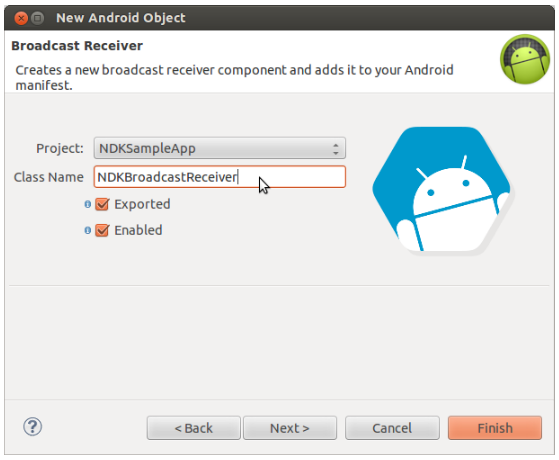
Figure 21: File is a specialized Android Object



Broadcast Receiver

Creates a new broadcast receiver component and adds it to your Android manifest.

Figure 22: Select BroadcastReceiver



Creates a new broadcast receiver component and adds it to your Android manifest.

Project: NDKSampleApp

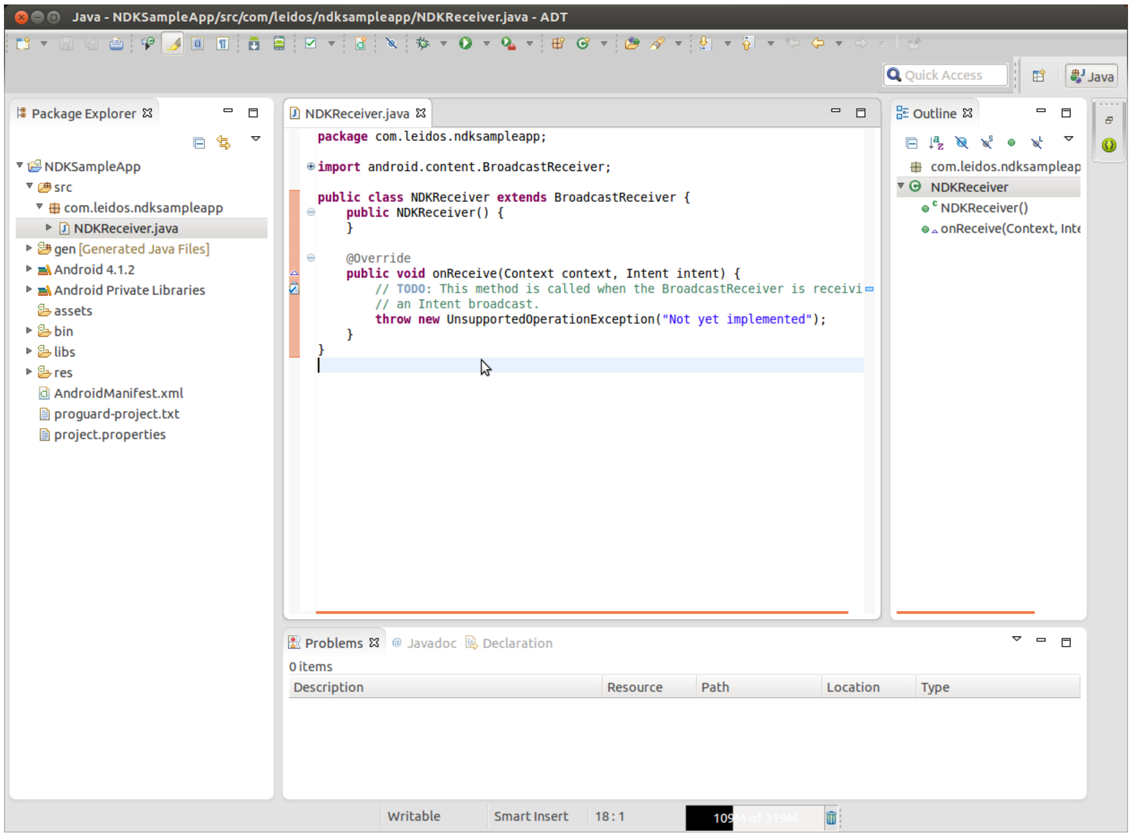
Class Name NDKBroadcastReceiver

Exported ✓

Enabled ✓

Figure 23: Provide name for BroadcastReceiver

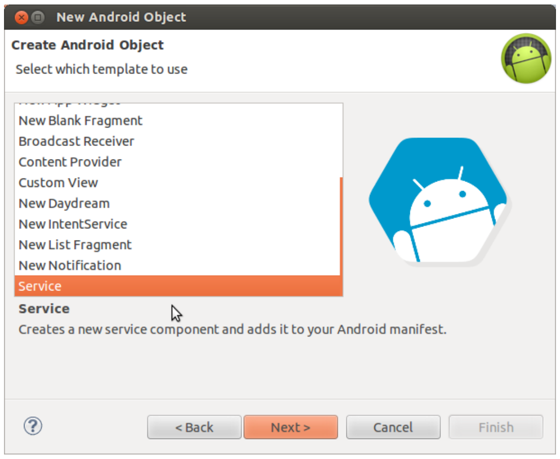
After adding the Java class called *NDKBroadcastReceiver*, template code will be provided as shown in Figure 24. This will be replaced by code presented below.



```
package com.leidos.ndksampleapp;

import android.content.BroadcastReceiver;

public class NDKReceiver extends BroadcastReceiver {
    public NDKReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO: This method is called when the BroadcastReceiver is receivi
        // an Intent broadcast.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

Figure 24: Template for BroadcastReceiver

Next, add a *Service* class to the project. Within Eclipse, in the Project Explorer pane, right-click on the project name, select *New -> Other…*, and then open the Android topic if not open. Then select *Android Object*, followed by *Service*; then change the Class Name to be *NDKService*. See Figure 20, Figure 21, Figure 25, and Figure 26 for this sequence.

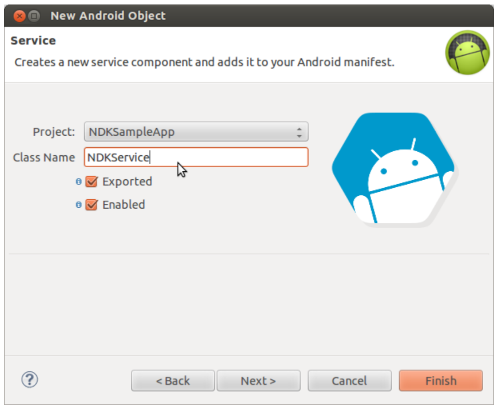


Select which template to use

New Blank Fragment
Broadcast Receiver
Content Provider
Custom View
New Daydream
New IntentService
New List Fragment
New Notification
Service

Service
Creates a new service component and adds it to your Android manifest.

Figure 25: Select Service

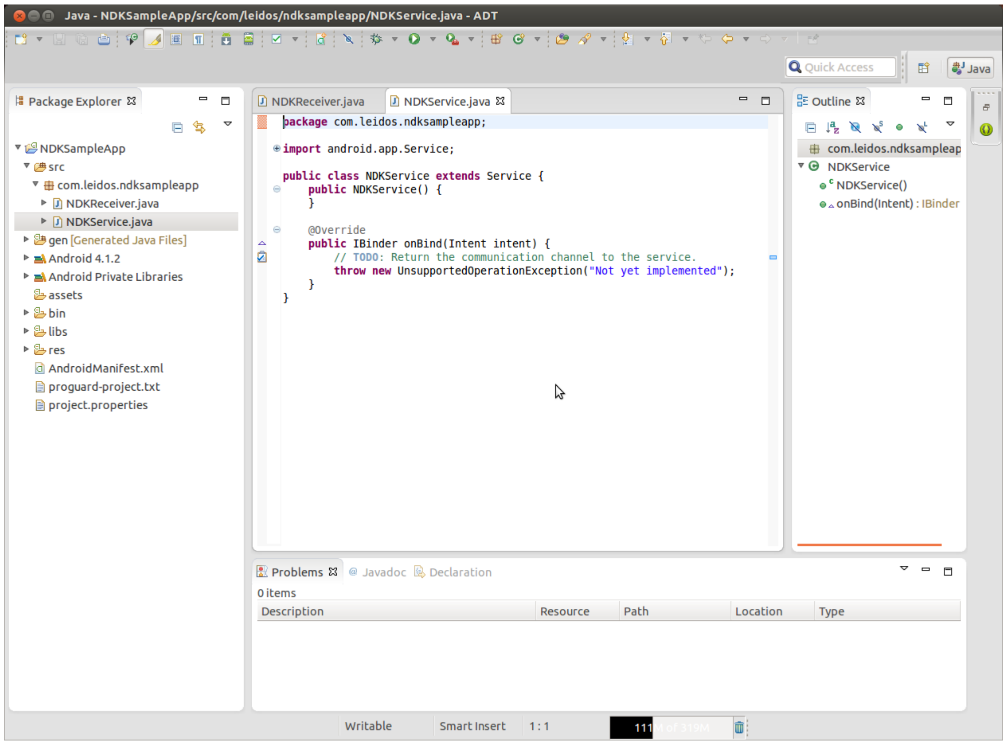


Service
Creates a new service component and adds it to your Android manifest.

Project: NDKSampleApp
Class Name NDKService
✓ Exported
✓ Enabled

Figure 26: Provide name for Service

After adding the Java class called *NDKService*, template code will be provided as shown in Figure 27. This will be replaced by code presented below.



```
NDKSampleApp
  src
    com.leidos.ndksampleapp
      NDKReceiver.java
      NDKService.java
  gen [Generated Java Files]
  Android 4.1.2
  Android Private Libraries
  assets
  bin
  libs
  res
  AndroidManifest.xml
  proguard-project.txt
  project.properties
```

```java
package com.leidos.ndksampleapp;

import android.app.Service;

public class NDKService extends Service {
    public NDKService() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

```
com.leidos.ndksampleap
NDKService
  NDKService()
  onBind(Intent) : IBinder
```

0 items

Description	Resource	Path	Location	Type

Writable   Smart Insert   1 : 1

Figure 27: Template for Service

B.3 Edit the Broadcast Receiver and Service

Now, replace the code in the *NDKBroadcastReceiver.java* and *NDKService.java* files with the source code provided in Table 27 and Table 28. The font has been reduced to save space; presumably readers will be using cut-and-paste to move the code from this document to Eclipse.

Table 27: NDKBroadcastReceiver.java

```
package com.leidos.ndksampleapp;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class NDKBroadcastReceiver extends BroadcastReceiver {
        private static final String TAG = "NDKBroadcastReceiver";

        @Override
        public void onReceive(Context context, Intent intent) {
                Log.i(TAG, "onReceive called");

                context.startService(new Intent(context, NDKService.class));
        }

}
```

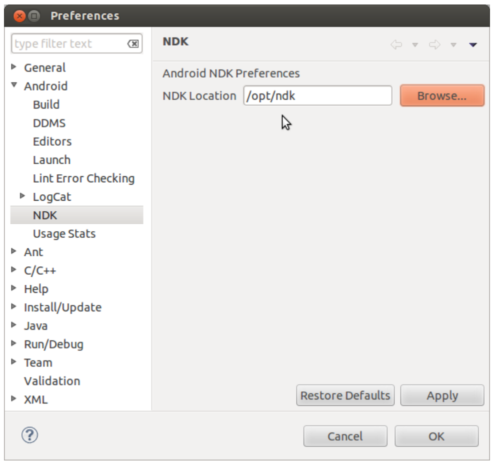
Table 28: NDKService.java

```
package com.leidos.ndksampleapp;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class NDKService extends Service {
        private static final String TAG = "NDKService";
        private  static boolean running = false;

        // these native (C) functions are located in jni/<project_name>.c
        // the C header for the prototypes is jni/<package_name>_NDKService.h
        public native String getHelloMessage();
       public native double calculatePower(double x, double y);
       public native int cMain();
       static {
          // load the library built from jni/ source and installed on the device in
          // /data/data/<package-name>/lib/ (in this case, lib<projectname>.so)
         System.loadLibrary("NDKSampleApp");
        }

        @Override
        public IBinder onBind(Intent intent) {
                return null;
        }

        @Override
        public void onCreate() {
                // Code to execute when the service is created
                Log.i(TAG, "onCreate called");
        }

        @Override
        public int onStartCommand(Intent intent, int startid, int whatup) {
                // Code to execute when the service is created
                Log.i(TAG, "onStartCommand: entry");

                // this illustrates calling a C function that returns a string
                Log.i(TAG, "onStartCommand: " + getHelloMessage());

                // this illustrates calling a C function that takes arguments and returns a value
                Log.i(TAG, "onStartCommand: " + "calculatePower(3.0, 4.0) yields " +
calculatePower(3.0, 4.0));

                // this illustrates calling a C function in the library using a new thread,
                // the function need not return (essentially calling a main()).
                if (!running) {
                        running = true;
                        new Thread(
                                        new Runnable() {
                                                @Override
                                                public void run() {
                                                        // call the C code "main"
                                                        int elapsed = cMain();
                                                        running = false;
```

```
                                                                Log.i(TAG, "onStartCommand: " + "return
from cMain, value " + elapsed);
                                                       }
                                              }
                          ).start();
                 }
                 Log.i(TAG, "onStartCommand: exit");
                 return 0;
        } // onStartCommand

}
```

B.4 Extend the Application to add Native Support

At this point, adjust some Eclipse configuration values to support NDK builds. In Eclipse, select menu *Window -> Preferences*, open Android, and select NDK. Provide the install path of your NDK installation. See Figure 29. Then right-click on the project, *Android Tools -> Add Native Support*; this process is illustrated in Figure 30.



NDK

Android NDK Preferences

NDK Location /opt/ndk

Figure 29: Adding NDK location

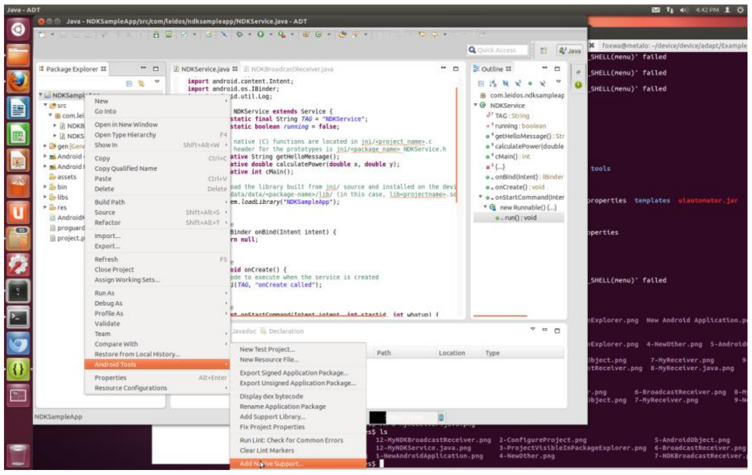
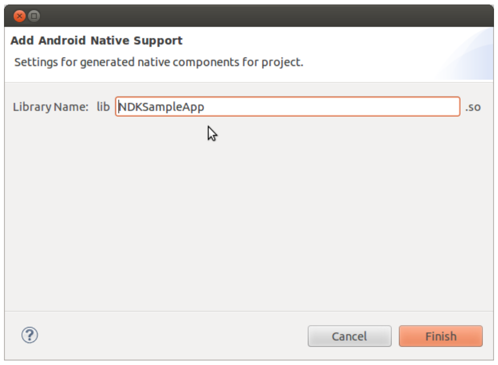


Figure 30: Add Native Support

Then, accept the default library name (Figure 31), and select Finish. The project now has a *jni* directory with a template file as shown in Figure 32.

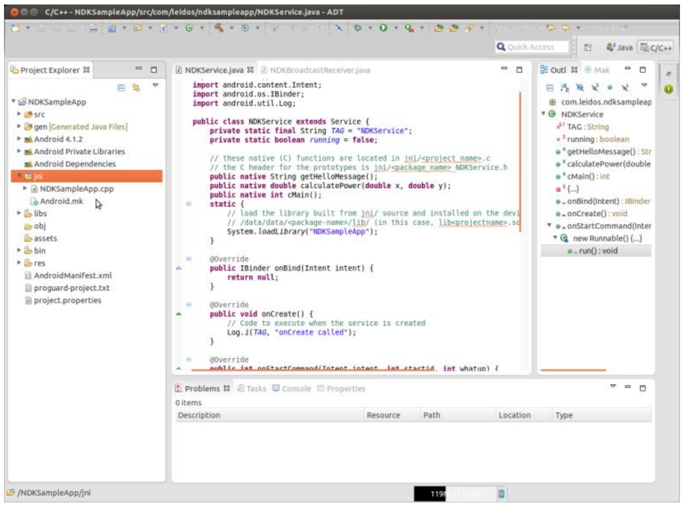


Add Android Native Support

Settings for generated native components for project.

Library Name: lib NDKSampleApp .so

Figure 31: Library Name



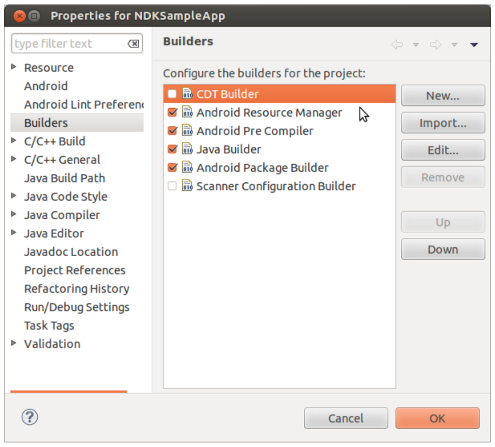
C/C++ - NDKSampleApp/src/com/leidos/ndksampleapp/NDKService.java - ADT

```
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class NDKService extends Service {
    private static final String TAG = "NDKService";
    private static boolean running = false;

    // these native (C) functions are located in jni/<project_name>.c
    // the C header for the prototypes is jni/<package_name>_NDKService.h
    public native String getHelloMessage();
    public native double calculatePower(double x, double y);
    public native int cMain();
    static {
        // load the library built from jni/ source and installed on the devi
        // /data/data/<package-name>/lib/ (in this case, lib<projectname>.so
        System.loadLibrary("NDKSampleApp");
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        // Code to execute when the service is created
        Log.i(TAG, "onCreate called");
    }

    @Override
    public int onStartCommand(Intent intent, int startId, int whatup) {
```

Figure 32: New jni directory and template file

B.5 Disable CDT (C code) builds

Adding Native Support enables the CDT (C code) Builder, which needs to be disabled to allow the *ndk-build* tool to build the *jni* directory. In addition, adding the path to the Android include files will eliminate static analysis incorrect error messages in the Problems tab. To eliminate the CDT Builder, right-click on the project, select Properties, then select Builders. Uncheck the CDT Builder and select OK. This dialog is shown in Figure 33. Then open the C/C++ General tab and select Paths and Symbols as illustrated in Figure 34.

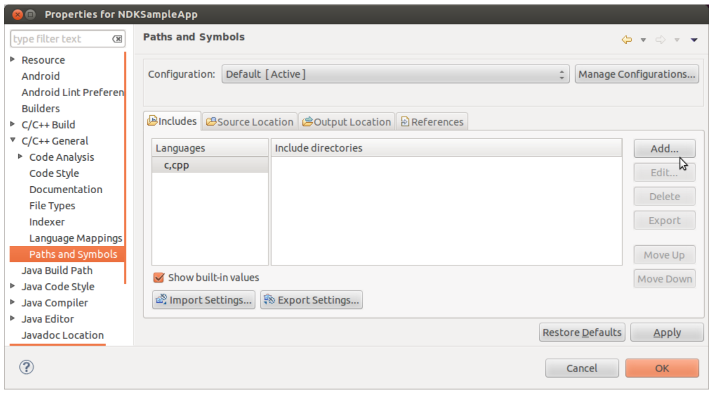


Properties for NDKSampleApp

Builders

Configure the builders for the project:

- [ ] CDT Builder
- [x] Android Resource Manager
- [x] Android Pre Compiler
- [x] Java Builder
- [x] Android Package Builder
- [ ] Scanner Configuration Builder



Properties for NDKSampleApp

Paths and Symbols

Configuration: Default [Active]

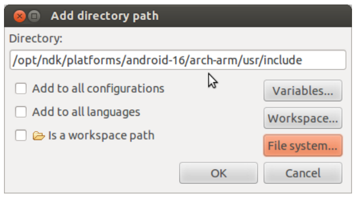
Languages	Include directories
c,cpp	

- [x] Show built-in values

Figure 34: Open the Paths and Symbols dialog

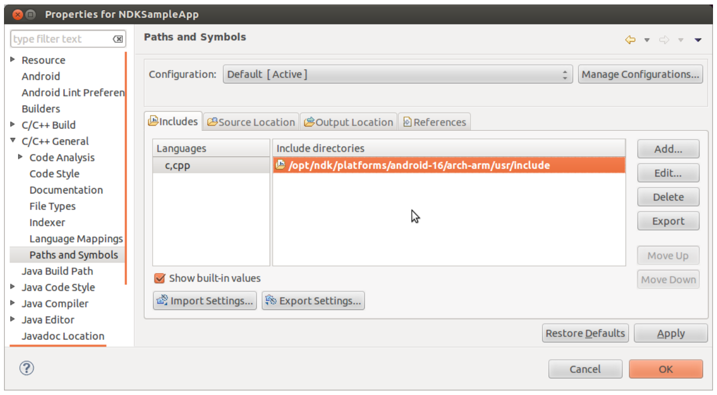
Figure 33: Disable CDT Builder

In the dialog, select *Add…*, and in the following dialog provide the path to your NDK install directory (`NDK_DIR`) followed by *platforms/android-16/arch-arm/usr/include*. Then select OK. This dialog is illustrated in Figure 35, and the resulting path is added to the Paths and Symbols dialog (Figure 36).


Add directory path

Directory: /opt/ndk/platforms/android-16/arch-arm/usr/include

Figure 35: Dialog to allow include path specification


Properties for NDKSampleApp

Paths and Symbols

Configuration: Default [Active]

Languages	Include directories
c,cpp	/opt/ndk/platforms/android-16/arch-arm/usr/include

Figure 36: Result of adding include path

B.6 Create a C header file for the public interface

Java provides a command line tool to create C header files for the JNI interface. This process is error prone if performed manually. The tool searches through specified Java class files, and creates a C header file for the *public native …* method declarations. Run the command from the top-level of the project directory. The *classpath* argument directs the search for Java class files to the *bin* directory, and the *d* argument directs the output file to the *jni* directory. See Table 29 for an example run of the *javah* command.

Table 29: Running javah to generate a header file

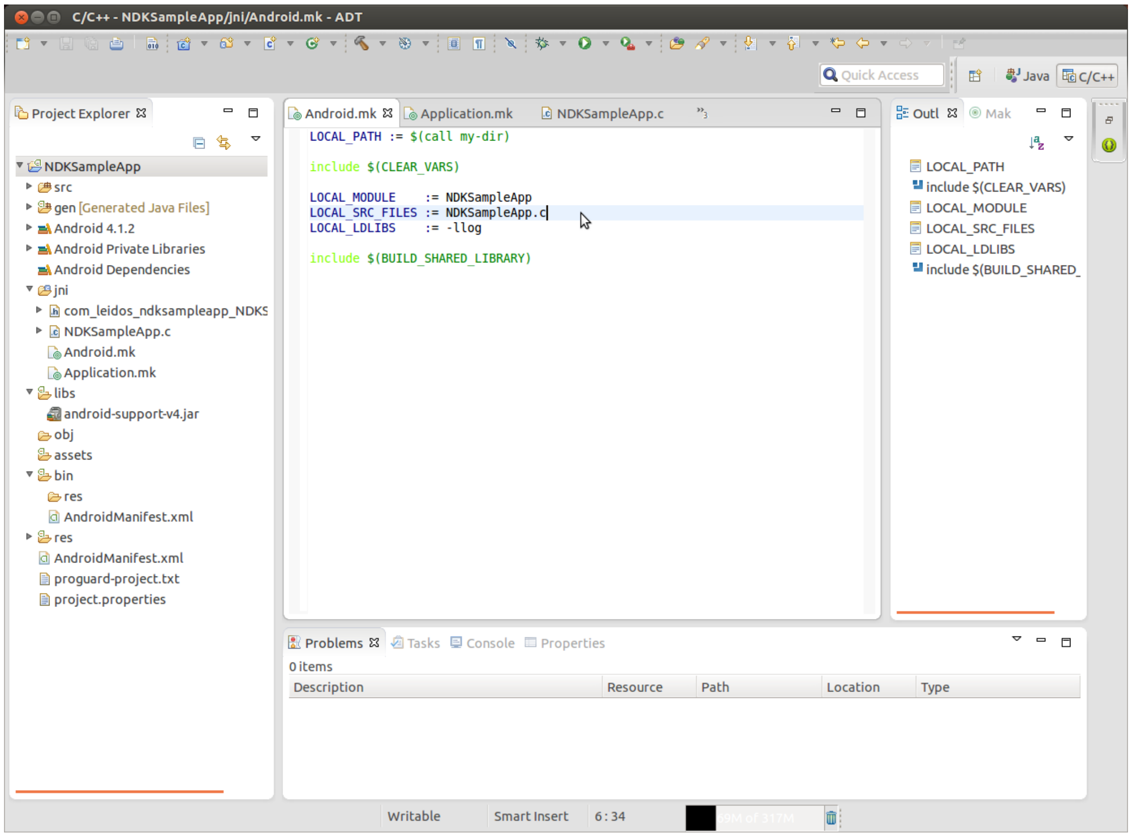
```
~/workspace/NDKSampleApp$ javah -classpath bin/classes -d jni -jni
com.leidos.ndksampleapp.NDKService

~/workspace/NDKSampleApp$ ls jni/*.h
jni/com_leidos_ndksampleapp_NDKService.h

~/workspace/NDKSampleApp$ head -10 jni/com*.h
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_leidos_ndksampleapp_NDKService */

#ifndef _Included_com_leidos_ndksampleapp_NDKService
#define _Included_com_leidos_ndksampleapp_NDKService
#ifdef __cplusplus
extern "C" {
#endif
/*
```

B.7 Add Android.mk and Application.mk files

The files in the *jni* directory require several additional changes.

1. Rename *NDKSampleApp.cpp* to *NDKSampleApp.c* (change the file extension)
2. Edit *Android.mk* to capture the filename change above, and add the logging library (all three changes can be seen in Figure 37)

3. Add a new file *Application.mk* to specify several build options. The contents of this file are shown in Table 30.



```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := NDKSampleApp
LOCAL_SRC_FILES := NDKSampleApp.c
LOCAL_LDLIBS    := -llog

include $(BUILD_SHARED_LIBRARY)
```

Figure 37: jni/Android.mk file contents

Table 30: jni/Application.mk file contents

```
# Build ARMv7-A machine code.
APP_ABI := armeabi-v7a
APP_PLATFORM := android-16

APP_CFLAGS   += -mfpu=neon-vfpv4
```

B.8 Edit the template C file

Replace the contents of *NDKSampleApp.c* with the source code provided in Table 31.

Table 31: jni/NDKSampleApp.c file contents

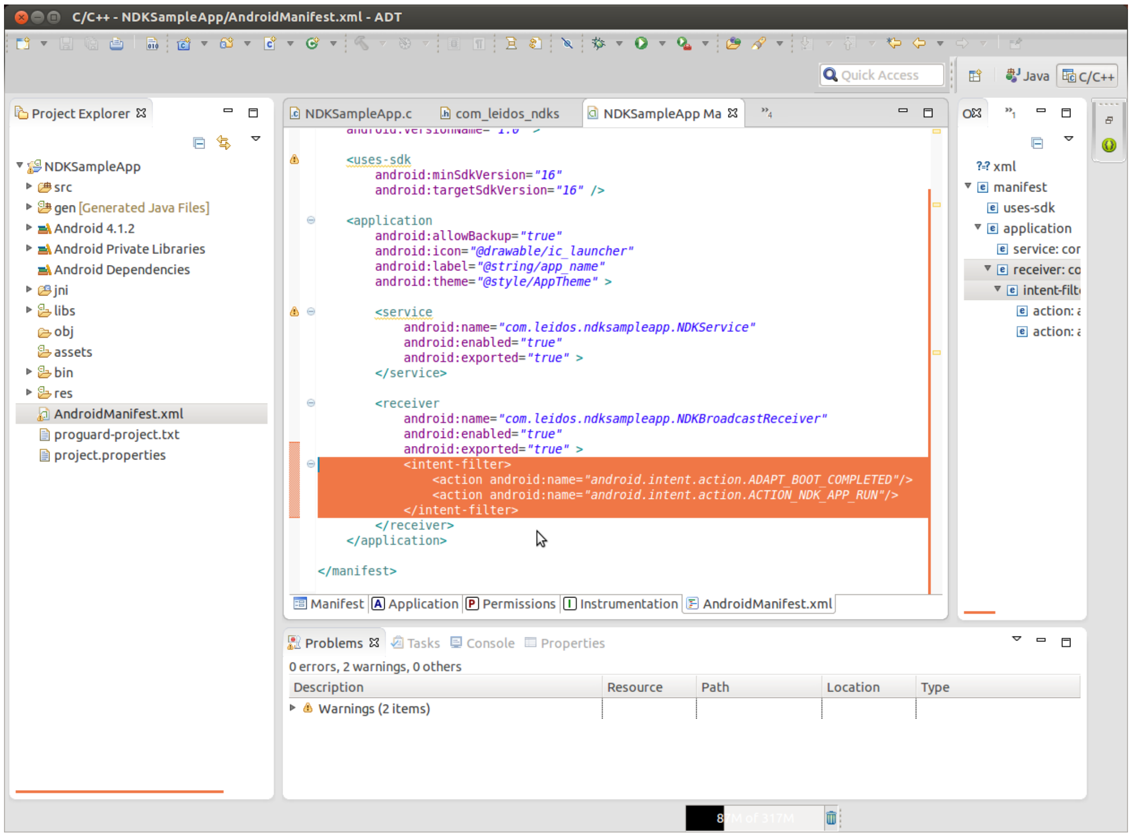
```
#include "com_leidos_ndksampleapp_NDKService.h"
#include <math.h>
#include <android/log.h>


// macro to log strings to the Android Logger
#define  LOGINFO(x...)  __android_log_print(ANDROID_LOG_INFO, "SampleJNI", x)

// demonstrate return of a string
char * getHelloMessage() {
        static char message[] = "Hello from the C World";
    LOGINFO("getHelloMessage called");
        return (message);
}

// demonstrate use of the Android math library
double calculatePower(double  x, double  y) {
    LOGINFO("calculatePower called");
    return pow(x, y);
}

// a sample main entry point.
// if invoked in a separate Java thread then this call need not return.
int cMain(const int SleepTime) {

    int sleepElapsed = 0;
    LOGINFO("cMain called");

    while(sleepElapsed < SleepTime) {
        sleep(1);
        sleepElapsed++;
    }

    return sleepElapsed;
}

// signature of the JNI method as generated in header file
JNIEXPORT jstring JNICALL Java_com_leidos_ndksampleapp_NDKService_getHelloMessage
  (JNIEnv *env, jobject obj) {
        return (*env)->NewStringUTF(env, getHelloMessage());
}

// signature of the JNI method as generated in header file
JNIEXPORT jdouble JNICALL Java_com_leidos_ndksampleapp_NDKService_calculatePower
  (JNIEnv *env, jobject obj, jdouble x, jdouble y) {
    return calculatePower(x, y);
}

// signature of the JNI method as generated in header file
JNIEXPORT jint JNICALL Java_com_leidos_ndksampleapp_NDKService_cMain
  (JNIEnv *env, jobject obj) {
    return cMain(50);

}
```

B.9 Edit the AndroidManifest.xml file

Next, add a section to the *AndroidManifest.xml* (the file at the project level) to request delivery of the Android intent to the *NDKBroadcastReceiver*. The lines to add are provided in Table 32. The final product is shown in Figure 38.

Table 32: Intent filter specification

```
<intent-filter>
   <action android:name="android.intent.action.ADAPT_BOOT_COMPLETED"/>
   <action android:name="android.intent.action.ACTION_NDK_APP_RUN"/>

</intent-filter>
```



```
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="16"
        android:targetSdkVersion="16" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        <service
            android:name="com.leidos.ndksampleapp.NDKService"
            android:enabled="true"
            android:exported="true" >
        </service>

        <receiver
            android:name="com.leidos.ndksampleapp.NDKBroadcastReceiver"
            android:enabled="true"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.ADAPT_BOOT_COMPLETED"/>
                <action android:name="android.intent.action.ACTION_NDK_APP_RUN"/>
            </intent-filter>
        </receiver>
    </application>

</manifest>
```

```
0 errors, 2 warnings, 0 others
Description            Resource   Path   Location   Type
▸ Warnings (2 items)
```

Figure 38: AndroidManifest.xml with intent filter

B.10 Build this app starting with an NDK-specific build of the C/C++ source

The application is now ready to build. First, build the library produced by the C code in the *jni* directory. This build may be achieved in either of two ways:

1. Manually, by running the command line *ndk-build* in either the project or project/jni directory (see Figure 39)
2. Automatically, by configuring auto run of *ndk-build* (instructions below)

In either case, the output is a set of library files in *obj* (not stripped) and *libs* (stripped).

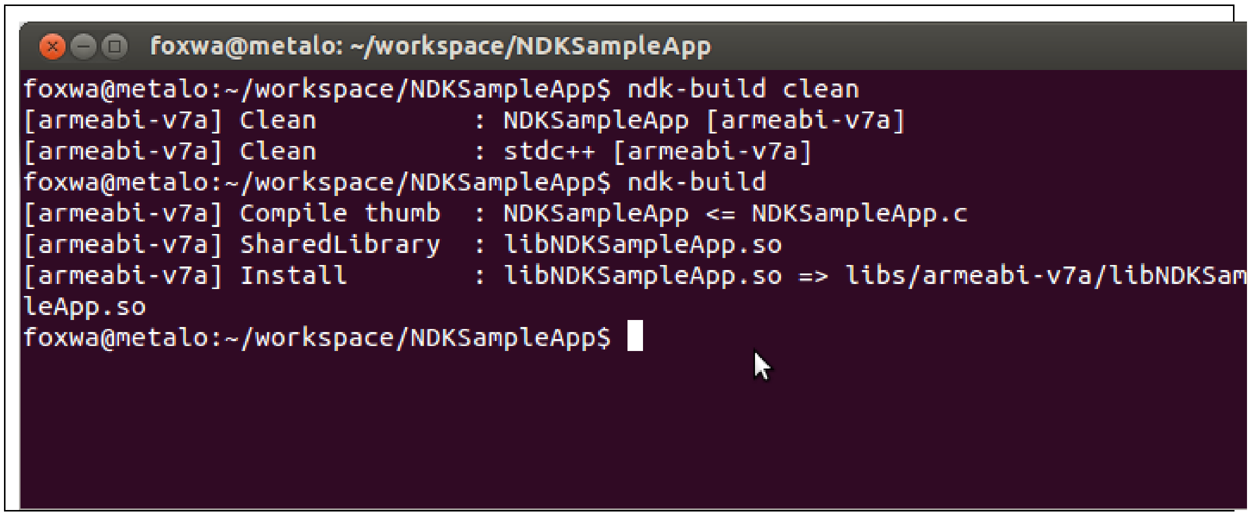
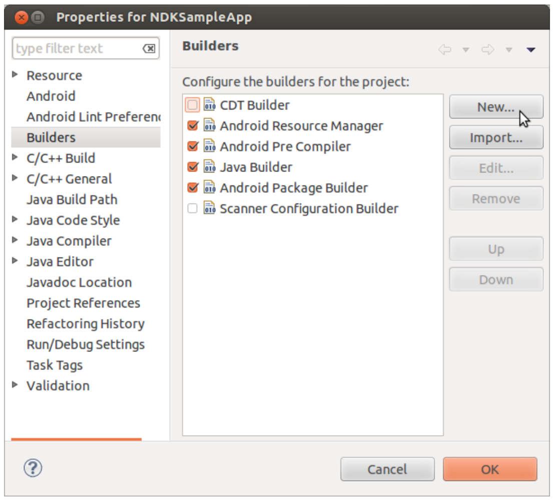

```
foxwa@metalo: ~/workspace/NDKSampleApp
foxwa@metalo:~/workspace/NDKSampleApp$ ndk-build clean
[armeabi-v7a] Clean          : NDKSampleApp [armeabi-v7a]
[armeabi-v7a] Clean          : stdc++ [armeabi-v7a]
foxwa@metalo:~/workspace/NDKSampleApp$ ndk-build
[armeabi-v7a] Compile thumb  : NDKSampleApp <= NDKSampleApp.c
[armeabi-v7a] SharedLibrary  : libNDKSampleApp.so
[armeabi-v7a] Install        : libNDKSampleApp.so => libs/armeabi-v7a/libNDKSam
leApp.so
foxwa@metalo:~/workspace/NDKSampleApp$ 
```

Figure 39: ndk-build command line example

B.10.1 Auto build of C library

To build the C library in the *jni* directory automatically upon source code change, follow these instructions. In Eclipse, right-click on the project, select Properties, Builders, New… This is shown in Figure 40. Select *Program* then *OK*, this is shown in Figure 41. Then in the *Main* tab, provide for *Name* the value *NDK Builder*, provide for *Location* the full path of *ndk-build* entry, and provide for *Working Directory* the value *${workspace_loc:/NDKSampleApp}* which is most easily entered by selecting *Browse Workspace…* and then selecting the project name. This is shown in Figure 42. Then in the Refresh tab, select *Refresh resources upon completion*, and *Specific resources*. This is shown in Figure 43.

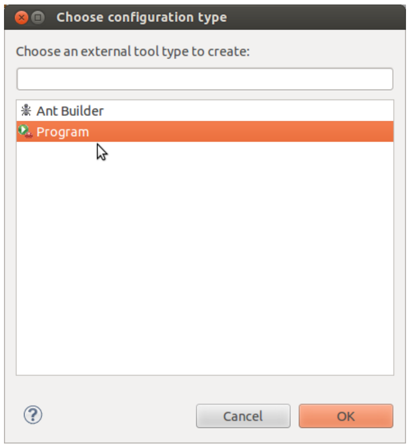

Properties for NDKSampleApp

Builders

Configure the builders for the project:

- [ ] CDT Builder
- [x] Android Resource Manager
- [x] Android Pre Compiler
- [x] Java Builder
- [x] Android Package Builder
- [ ] Scanner Configuration Builder

Figure 40: Creating new Builder

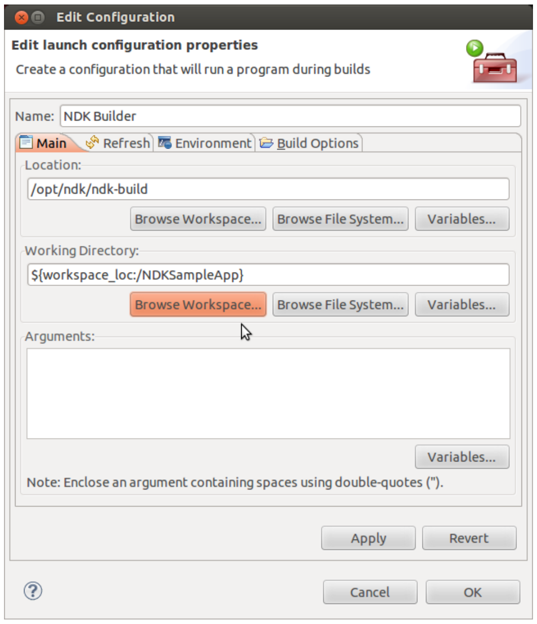

Choose configuration type

Choose an external tool type to create:

Ant Builder

Program (selected)

Figure 41: Builder is of type Program


Edit Configuration

Edit launch configuration properties

Create a configuration that will run a program during builds

Name: NDK Builder

Main | Refresh | Environment | Build Options

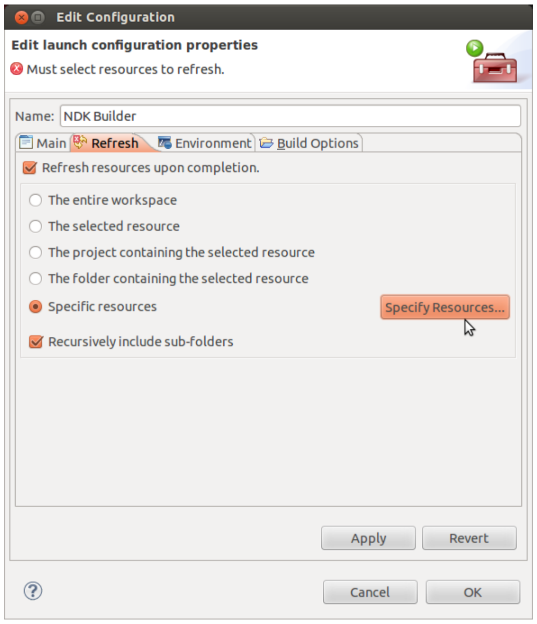
Location: /opt/ndk/ndk-build

Working Directory: ${workspace_loc:/NDKSampleApp}

Arguments:

Note: Enclose an argument containing spaces using double-quotes (").

Figure 42: NDK Builder Main tab


Edit Configuration

Edit launch configuration properties

Must select resources to refresh.

Name: NDK Builder

Main | Refresh | Environment | Build Options

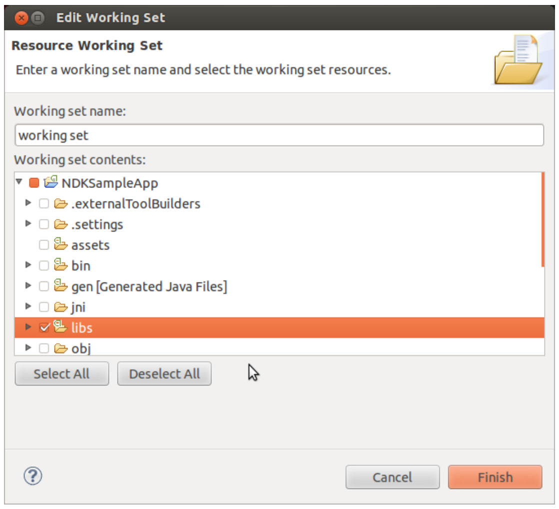
- [x] Refresh resources upon completion.

- ○ The entire workspace
- ○ The selected resource
- ○ The project containing the selected resource
- ○ The folder containing the selected resource
- ● Specific resources

- [x] Recursively include sub-folders

Figure 43: NDK Builder Refresh tab

From the *Refresh* tab, select *Specify Resources…* In the ensuing dialog, open the Project and select *libs* and Finish. This is shown in Figure 44. Now select the *Build Options* tab, and in the ensuing dialog, add *During auto builds* and *Specify working set of relevant resources*. This is shown in Figure 45. Select *Specify Resources…* and in the following dialog, select *jni*. See Figure 46. Select Finish and OK, and you should see the new *NDK Builder* as shown in Figure 47.



Edit Working Set

Resource Working Set

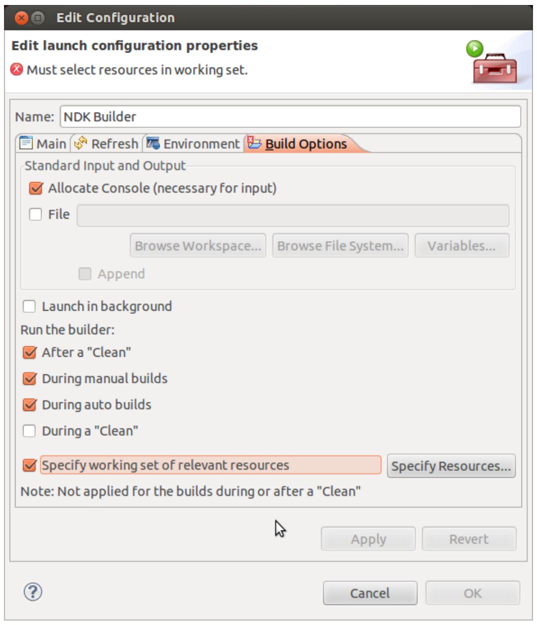
Enter a working set name and select the working set resources.

Working set name: working set

Working set contents:

- NDKSampleApp
  - ☐ .externalToolBuilders
  - ☐ .settings
  - ☐ assets
  - ☐ bin
  - ☐ gen [Generated Java Files]
  - ☐ jni
  - ✓ libs
  - ☐ obj

Figure 44: Resources for Refresh tab



Edit Configuration

Edit launch configuration properties

Must select resources in working set.

Name: NDK Builder

Standard Input and Output

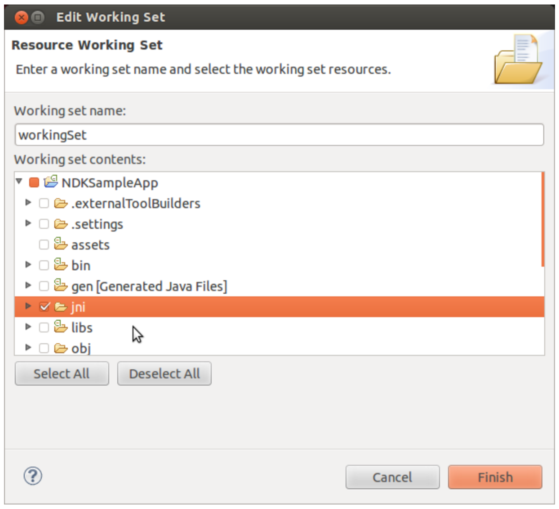
- ✓ Allocate Console (necessary for input)
- ☐ File
- ☐ Append
- ☐ Launch in background

Run the builder:

- ✓ After a "Clean"
- ✓ During manual builds
- ✓ During auto builds
- ☐ During a "Clean"
- ✓ Specify working set of relevant resources

Note: Not applied for the builds during or after a "Clean"

Figure 45: NDK Builder Build Options tab



Edit Working Set

Resource Working Set

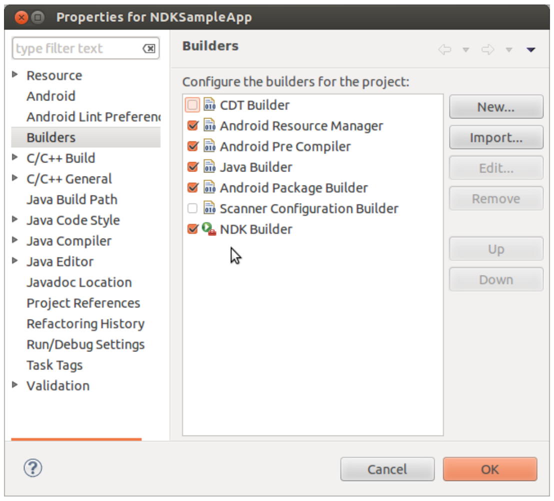
Enter a working set name and select the working set resources.

Working set name: workingSet

Working set contents:

- NDKSampleApp
  - ☐ .externalToolBuilders
  - ☐ .settings
  - ☐ assets
  - ☐ bin
  - ☐ gen [Generated Java Files]
  - ✓ jni
  - ☐ libs
  - ☐ obj

Figure 46: Resources for Build Options tab



Properties for NDKSampleApp

Builders

Configure the builders for the project:

- ☐ CDT Builder
- ✓ Android Resource Manager
- ✓ Android Pre Compiler
- ✓ Java Builder
- ✓ Android Package Builder
- ☐ Scanner Configuration Builder
- ✓ NDK Builder

Figure 47: Completed NDK Builder

B.11 Build the final apk

The final application may be rebuilt at any time within Eclipse by right-click on the Project, *Run -> Run As -> 1 Android Application*. On success, you will see the *Android Device Chooser*, select Cancel. On failure, an error dialog box comes up. In this case, check the *Problems* tab for more information.

B.12 Install and Verify

For installation and verification instructions, please refer to sections 2.4.1.2 Install and 2.4.1.3 Verify.