

5-2012

Creating, Writing, and Reading NETCDF Files on a Mobile Device

Brittany Allesandro
University of New Orleans

Follow this and additional works at: https://scholarworks.uno.edu/honors_theses

Recommended Citation

Allesandro, Brittany, "Creating, Writing, and Reading NETCDF Files on a Mobile Device" (2012). *Senior Honors Theses*. 6.

https://scholarworks.uno.edu/honors_theses/6

This Honors Thesis-Unrestricted is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Honors Thesis-Unrestricted in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Honors Thesis-Unrestricted has been accepted for inclusion in Senior Honors Theses by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.

CREATING, WRITING, AND READING NETCDF FILES ON A MOBILE DEVICE

An Honors Thesis

Presented to

the Department of Computer Science

of the University of New Orleans

In Partial Fulfillment

of the Requirements for the Degree of

Bachelor of Science, with University Honors

and Honors in Computer Science

by

Brittany Allesandro

May 2012

Contents

List of Tables	iii
List of Figures	iv
Abstract	v
Chapter 1: Introduction	1
1.1 Intentions of DroidCDF	1
1.2 History and Usage of NetCDF	2
1.3 History and Usage of Android	6
1.4 DroidCDF's netCDF version choice	14
1.5 DroidCDF's Android target	14
1.6 Additional Libraries	16
Chapter 2: Application Overview	17
2.1 The Home Screen	17
2.2 Creating Files	22
2.2.1 CreateFileActivity	22
2.2.2 FileAttributesActivity	24
2.2.3 DimensionsActivity	28
2.2.4 VariablesActivity	30
2.2.5 ConfirmWriteActivity	32
2.3 Reading from Files	35
2.4 Writing Files	39
2.5 Permissions	43
Chapter 3: DroidCDF Analysis	44
3.1 File Creation	44
3.2 Data Write	45
3.3 Data Read	47
3.4 Limitations	47
Chapter 4: Future of DroidCDF	48
Licensing	48
References	50

List of Tables

Table 1: Android Releases [11]	9
Table 2: Numerical Distribution of Android Versions [11]	15
Table 3: Typical ViewGroups [11]	20
Table 4: File Creation Time with Varying Number of Dimensions	45
Table 5: Time to Complete Change of File's Header.....	46

List of Figures

Figure 1: Classic Data Model [7].....	5
Figure 2: Clock Widget.....	8
Figure 3: Pull-Down Notification Window	8
Figure 4: Pie Chart of Android Version Distribution [11].....	15
Figure 5: Trend-Over-Time Histogram of Android Versions [11].....	16
Figure 6: DroidCDF Workflow	17
Figure 7: DroidCDF Home Screen	18
Figure 8: Creating a New NetCDF File with DroidCDF.....	22
Figure 9: CreateFileActivity	23
Figure 10: FileAttributesActivity.....	25
Figure 11: Spinner for Value Type	27
Figure 12: DimensionsActivity.....	28
Figure 13: VariablesActivity.....	30
Figure 14: Dimensions Dialog	31
Figure 15: ConfirmWriteActivity	33
Figure 16: Creation Successful Dialog	35
Figure 17: ExistingActivity.....	36
Figure 18: Read-Write Interface	38
Figure 19: Image Selector	39
Figure 20: Two Separate ContextMenus Defined on the Same ListActivity	40
Figure 21: Add Attribute AlertDialog.....	42
Figure 22: ViewVariablesActivity	43
Figure 23: Graph of Creation Times with Varying Number of Dimensions	45

Abstract

The purpose of this research was to master several unfamiliar concepts and bring them together in one cohesive project. DroidCDF is an application for the *Android* operating system to create, write data to, and read data from netCDF format files. DroidCDF uses Unidata's NetCDF Java Library and can write files in netCDF-3 format but read from any netCDF format files. As mobile devices become more powerful and commonplace, DroidCDF provides a convenient tool for researchers. An incremental methodology was applied; the application was built from a rough workflow to eventually a robust and fully functional program. The produced files are fully portable and can be used as the input for other applications. The application has been tested with several large netCDF files with varying conventions and has handled each one remarkably well. Upon submission of this thesis, DroidCDF will be released onto the open market.

Keywords: Android, Unidata, NetCDF, mobile application, Java, common data formats

Chapter 1: Introduction

1.1 Intentions of DroidCDF

DroidCDF is an Android application that allows users to create files and read and write data in Unidata's netCDF format. NetCDF is a self-describing, portable, and scalable data format used for sharing and transferring scientific data. All Android applications are written in Java, so DroidCDF uses NetCDF Java, a library that is maintained separately from the C, Fortran, and C++ interfaces. There are several versions of the netCDF file format, but currently NetCDF Java and thus DroidCDF only support writing with the netCDF-3 classic data format. In keeping with Unidata's goal of being available to as many users as possible, DroidCDF is targeted to run on Android 2.1 and above, so that an estimated 99% of Android users are capable of using the application.

As devices such as smartphones and tablets become increasingly popular, it makes sense to have a mobile application for recording scientific data. If a researcher would like to record data while on the go, a simple to use and entirely functional application on a mobile device would be convenient and pragmatic. DroidCDF can display the contents of any netCDF file in an interactive format, optimized to fit on a mobile device. The application does not attempt to understand the data, but it would be easy to create a companion application to interpret the files in meaningful ways for specific research. It is important to note that since DroidCDF does not interpret data, it is up to the user to enforce conventions and standards that their teams or research require.

DroidCDF has three significant goals:

- **Accessible:** to be a useful, lightweight application for as many researchers as possible. This is achieved by developing the application to work on the most popular Android platforms and to support all netCDF formats that the NetCDF Java library allows.
- **Usable:** if a researcher is familiar with the netCDF format, the application should require little or no training to use. Even without familiarity, the application attempts to be self-explanatory. This means that the interface is intuitive and easy to navigate.
- **Open:** DroidCDF is an open project that any researcher may modify or extend to suit their needs.

1.2 History and Usage of NetCDF

As described by its developers, NetCDF - short for Network Common Data Form - is a set of interfaces that was created in order to facilitate the sharing of electronic scientific data, regardless of machine or operating system. It was originally developed by the Unidata Program Center, part of the University Corporation for Atmospheric Research, in order for the program's researchers to share their meteorological data and to create a reusable, cross-disciplined piece of software. [1]

In 1987, Treinish and Gough, researchers at the National Space and Science Data Center (NSSDC) of the National Aeronautics and Space Administration (NASA) described the NASA Common Data Format (CDF) which eventually laid the groundwork for Unidata's netCDF. The two explain the CDF as being a cross-platform, readily accessible software package:

This structure, called the Common Data Format (CDF), provides true data independence for applications software that has been developed at NSSDC. Scientific software systems at NSSDC use this construct so that they do not need specific knowledge of the data with which they are working. This permits users of such systems to apply the same functions

to different sets of data...The users of such data-independent NSSDC systems...rely on their own knowledge of different sets of data to interpret the results, a critical feature for the multidisciplinary studies inherent in the earth and space sciences. [2]

The idea that it is the user's responsibility to interpret the data is an important concept to CDF, netCDF, and DroidCDF. The software's primary concern is with putting the data into a unified format, not with reaching meaningful understandings; in this way, the software becomes much more stream-lined and efficient, which is of the utmost importance in a mobile environment where memory and processing power can already be limited.

After Treinish and Gough laid their foundations, Dave Raymond – a researcher at the New Mexico Institute of Mining and Technology – developed a “system written in the C language for the *analysis* and *display* of gridded numerical data (Candis),” which he described in his 1987 paper. In his system, access to data files was sequential so that applications could be constructed with pipes, a mechanism wherein the output of one module becomes the input of another module. [3]

Unidata developer Glenn Davis combined the ideas of Treinish and Gough with Raymond's, including named dimensions and variables of various shapes, into the first prototype of netCDF. The prototype was written in C, like Candis, and also layered on an External Data Representation (XDR) format. After Joe Fahle of SeaSpace, Inc., released his version of CDF software, Unidata produced and refined its own interface and documentation with additional developers Russ Rew, Ed Hartnett, John Caron, Steve Emmerson, and Harvey Davies. [1]

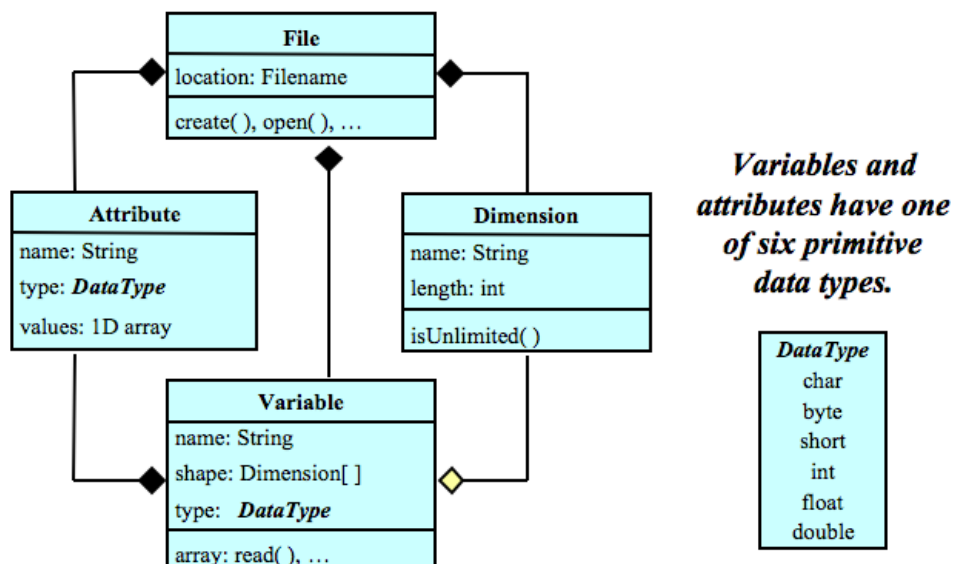
NetCDF files have a fundamental set of qualities. The first is that netCDF files are self-describing; it includes information about the data it contains in a header, such as where particular sections of data are written in the file. Files are portable and can be read by machines with

differing architectures. NetCDF files are also scalable and appendable, so data can be efficiently read and written. Finally, netCDF files will always be archivable, or backwards-compatible. [4]

Since its inception, netCDF has seen several new versions. Version 2.0 was released in October of 1991 and included changes such as storing dimension lengths as longs rather than integers. In June of 2008, version 4.0 was released. [1] This new release included a more powerful model of data representation with new primitive data types, the ability to have multiple dimensions with unlimited length, and support for parallel I/O. [5]

While netCDF files can be written in several different programming languages, Unidata maintains a single distribution of the C, Fortran, and C++ interfaces and a separate distribution of the Java interface. The latest versions of the NetCDF Java Library implement a Common Data Model (CDM), which is a generalization of the NetCDF, OpenDAP, and HDF5 data models. The library is still a prototype to netCDF-4; it completely supports reading the netCDF-4 format, but can only support writing to netCDF-3 format. [6] Since Android applications are written in Java, all files created by DroidCDF are in netCDF-3 format and extend the classic data model. While any netCDF can be read by DroidCDF, only files in netCDF-3 format can be modified.

In the classic netCDF data model, data are stored in array data structures, called variables, which are shaped by dimensions and described by attributes. **Figure 1** shows a visual representation of these relationships:



A file has named variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One dimension may be of unlimited length.

Figure 1: Classic Data Model [7]

From the netCDF User Guide, dimensions have a name and positive length, although a netCDF-3 file can have one dimension with an unlimited length. This unlimited dimension, also known as the record dimension, must always be listed first when defining variables because it is the slowest changing. These dimensions can be defined to represent real dimensions, like time or temperature, and they are used to shape variables. [1]

Variables are where the data in a netCDF file are stored. A variable has a name, type, and shape. The type of the variable indicates the type of the array of values it stores. The shape of the variable is indicated by supplying a list of dimensions. For example, if a variable has no dimensions it is a scalar value. If a variable has one dimension, its values are contained in a 1-dimensional array; if it has two dimensions, its values are contained in a 2-dimensional array, and so on. Since the netCDF Java library writes files in the netCDF-3 format, the only types allowed are char, byte, short, int, float, and double. [1]

Both the file and individual variables can be assigned attributes which provide supplementary information. Attributes have names, types, and values; variable attributes must also specify their associated variable's name. While variables and dimensions must be declared before writing data, attributes can be continuously added. [1] Examples of attributes include file version, file authors, variable units, etc.

Together, the dimensions, variables, and attributes comprise the netCDF file header. The header includes names, types, lengths, and the offset to the beginning of each variable's data. In order to minimize the size of netCDF files, the header takes up as little space as possible; generally, the data begin at the next available disk block. However, this means that if additional dimensions, variables, or attributes are added to the file, all data must be copied and moved. The data that are stored after the header is divided into two parts. The first part is for fixed-size data; here, the data are stored contiguously. If applicable, the last part of the file is for data with an unlimited record dimension. Here, according to users' guide, the data "consists of a variable number of fixed-size records, each of which contains data for all the record variables. The record data for each variable are stored contiguously in each record." [1] A detailed breakdown of the classic netCDF format can be found at <http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/Classic-Format-Spec.html>.

1.3 History and Usage of Android

"Android is a complete mobile phone software stack. It includes everything a manufacturer or operator needs to build a mobile phone." [4] This robust platform began when Andy Rubin's startup – Android, Inc. – was purchased by Google in July of 2005. [8] Contrary to popular belief, though, the Android platform is developed and maintained by more than just Google. The Open Handset Alliance (OHA) is "a group of 84 technology and mobile companies

who have come together to accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience.” Members are companies who are devoted to “making serious and ongoing contributions to openness in the mobile world.” [4] The group includes telecommunications companies, such as Sprint Nextel, T-Mobile, and Vodafone; handset manufacturers, such as HTC, Motorola, and LG Electronics; semiconductor companies, such as Intel Corporation, NVIDIA Corporation, and Synaptics; and many more software and commercialization companies. [9]

The OHA has designed Android as an open platform, meaning the group publishes and updates a comprehensive application programming interface (API). This greatly benefits developers for many reasons, including having access to easy-to-use developer tools, deeply integrated and optimized applications, and less expensive distribution and commercialization of applications. [4] All of these reasons factored into the decision to develop a mobile netCDF application for the Android operating system.

On October 22, 2008, the Open Handset Alliance released its first Android-powered smartphone: the T-Mobile G1. Android 1.0 introduced widgets, a pull-down notification window, and the Android Market. [10] Widgets are “miniature application views that can be embedded in other applications (such as the home screen) and receive periodic updates. ... An application component that is able to hold other App Widgets is called an App Widget host.” [11] For example, a screenshot from my HTC Evo demonstrates the clock widget in **Figure 2:**



Figure 2: Clock Widget

This widget updates time and weather information in real time. Other examples include an interactive calendar, a Google search bar, and music controller. All of these allow the user to utilize the application's functionality without actually launching an entire application. DroidCDF is not a widget and displays over the entire screen when in use. The pull-down notification window displays information for users in one convenient location. It could list incoming text messages, system updates, pending voicemails, or application notifications. Another screenshot demonstrates this in **Figure 3**:

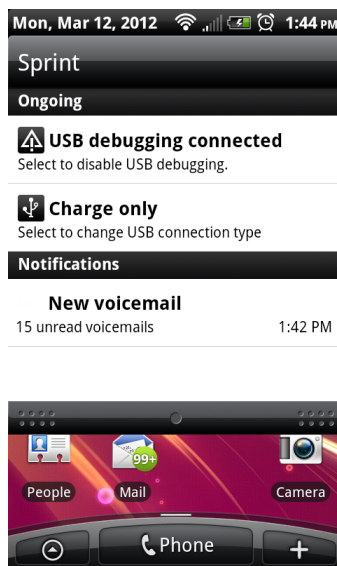


Figure 3: Pull-Down Notification Window

Here the window is in the process of being pulled down. It is displaying voicemail information as well as desktop-interactivity preferences while the phone is plugged in with a USB cable. The Android Market was, until very recently, the central hub where all applications could be downloaded and instantly synchronized to a user's Google account on the web. Google has now combined this service and offerings of eBooks, television and movie streaming, and music sales into Google Play.

Three months after Android 1.0 was released, Android 1.1 was deployed. This quick turnaround time has been continuous, and the platform has seen several major releases to date.

Table 1 lists these releases:

Platform	Codename	API Level	Release Date
Android 1.5	Cupcake	3	April 2009
Android 1.6	Donut	4	September 2009
Android 2.1	Éclair	7	January 2010
Android 2.2	Froyo	8	May 2010
Android 2.3 – Android 2.3.2	Gingerbread	9	December 2010
Android 2.3.3 – Android 2.3.7		10	February 2011
Android 3.0	Honeycomb	11	February 2011
Android 3.1		12	May 2011
Android 3.2		13	July 2011
Android 4.0 – Android 4.0.2	Ice Cream Sandwich	14	October 2011
Android 4.0.3		15	December 2011

Table 1: Android Releases [11]

With each new release came exciting new features. Cupcake brought the “soft keyboard” so that a user could type entirely on the touch screen, eliminating the need for a physical keyboard. Éclair allowed users to have multiple accounts synchronized to the phone at one time

and integrated Google Maps Navigation into several apps. [10] Global positioning systems (GPS) and geographic information systems (GIS) like Google Maps can be integrated into applications such as trackers or customized searches; DroidCDF could even extend such capabilities in the future, as netCDF is a popular input choice for GIS programs. Honeycomb has made great strides into completely eliminating the need for physical inputs; a new action bar widget allowed for developers to programmatically set all necessary virtual buttons, such as home, back, and menu, and display them on the touchscreen. Ice Cream Sandwich has added many improvements to the user interface, like a clearer font for higher resolution devices, and is incorporating new technologies, like the facial recognition unlocking and the Android Beam. The Android Beam allows two phones to transfer data by simply touching the devices together, while the new unlocking mechanism uses the front-facing camera to determine who is unlocking the phone. [10]

To create an Android application, a basic series of steps is followed. First, the development environment must be installed. Android has been fine-tuned to work easily in the Eclipse integrated development environment (IDE), but applications can also be built through other IDEs or through a text editor and terminal. The Android Development Tools (ADT) must be downloaded through the Eclipse marketplace or from the command line. [11] When creating DroidCDF, I choose to use the Eclipse IDE for ease of development, but this has no bearing on the application itself. To run the application, either Android Virtual Devices (AVDs) can be installed or a physical device can be connected. Next, the programmer actually develops the application. This phase is followed by debugging and testing the application using the Android testing and instrumentation framework. Lastly, the application is built and tested in release mode before being sold or distributed. [11]

An Android application consists of several components written in Java, Extensible Markup Language (XML) layouts, the Android Manifest file, and other resources like images and icons. The code is compiled into an Android package, or `.apk`, which is then installed on a mobile device. “The Android operating system is a multi-user Linux system in which each application is a different user.” [11] This means that, for security, android applications do not communicate by default. Each application has a unique ID and runs in its own virtual machine in isolation from other applications. However, applications can be configured to share the same Linux ID or share the same virtual machine. With these precautions in place, the Android system implements the principle of least privilege so that “each application, by default, has access only to the component that it requires to do its work and no more.” [11]

There are four Android components that define the framework of the application: activities, services, content providers, and broadcast receivers. “A unique aspect of the Android system design is that any application can start another application’s component.” [11] For example, when selecting a directory to create a file in, DroidCDF actually starts another application to provide the file chooser. It appears as if the file chooser is part of the DroidCDF application, but the two are separate and DroidCDF is actually waiting for the resulting directory path to be returned.

An activity is “a single screen with a user interface” that extends from the `Activity` class. [11] DroidCDF is built almost entirely from activities; for instance, when launching the application, the initial screen is the `HomeScreenActivity`. From there, the user could go to either an activity to create a new file or view existing files. All activities are independent, so applications can start any of them; however, one activity can be designated as the “main” activity and will start when the application is launched. Only one activity can run at a time, but any

stopped activities are preserved in the operating system's last in, first out (LIFO) "back stack".

[11]

An activity can run in the foreground with focus (resumed), run in the foreground without focus (paused), or run in the background (stopped). In the latter two cases, the operating system can kill the activity if needed to reclaim memory. There are callback methods, also called hooks, for the transitions between each of these states. A stubbed example of the activity lifecycle from the Developer's Guide is below:

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus (this activity is about to be
        "paused").
    }
    @Override
    protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }
}
```

The method `onCreate()` is especially important, as that is where the XML layout for this activity should be defined; however, layouts can also be created programmatically rather than

through XML. `onDestroy()` is also important; it can be used to release resources, such as threads or input/output, when the activity is finished. To use an activity, it must be declared in the `AndroidManifest` with the `<activity android:name="ActivityName" />` tag.

[11]

A service is “a component that runs in the background to perform long-running operations or to perform work for remote processes” and extends from the `Service` class. [11] A good example of a service is a music player. This service can play music in the background while a user runs other applications in the foreground.

There are two forms of services: started and bound. A started service is called by another component and usually performs one operation, such as downloading a file, before destroying itself. A bound service allows one or more components to interact with it indefinitely, but the service is destroyed if there are no components bound to it. Services should not be confused with threads, because the former can run even when the application is in the background, but the latter runs only when the user is interacting with the application. If a music player was run in a thread rather than a service, the music would stop when the application is sent to the background. [11]

Like activities, services are declared in the `AndroidManifest` with the `<service android:name="ServiceName" />` tag. DroidCDF’s file chooser, written by Hai Bison, declares one service, the `LocalFileProvider` service, but the remaining components are activities.

A content provider “manages a shared set of application data.” This allows data from one application to be used directly by another. A broadcast receiver is “a component that responds to system-wide broadcast announcements.” DroidCDF currently uses neither of these components.

1.4 DroidCDF's netCDF version choice

As noted in Section 1.1, netCDF has undergone a number of version changes. The most significant versions include different formats, including netCDF-3 classic, netCDF-3 64-bit offset, and netCDF-4/HDF5. As DroidCDF was to be developed for the Android operating system, the NetCDF Java library was required. This library allows the application to create and write files in the netCDF-3 class format, but files of any version can be read. As the NetCDF Java library is modified to support writing in other formats, DroidCDF could also be modified to support these implementations. Currently, DroidCDF is dependent upon the NetCDF Java version 4.3, the most recent release.

1.5 DroidCDF's Android target

A major goal of the DroidCDF application is to be useful to as many researchers as possible. Thus the ability to write only to netCDF-3 files, the most portable version, is not a real restriction in the context of this goal. Similarly, the application should be able to run on as many platforms as possible. Android applications are forward compatible [11], so the application should run on the lowest versioned platform, with a significant population of users, as possible. In general, a significant population will have at least 5% of the distribution.

Google tracks the estimated number of users of each Android version by recording the number of devices with that platform that access Google Play. The most recent distribution, as of April 2, 2012, is seen in the pie chart shown in **Figure 4** with corresponding numerical data in **Table 2**:

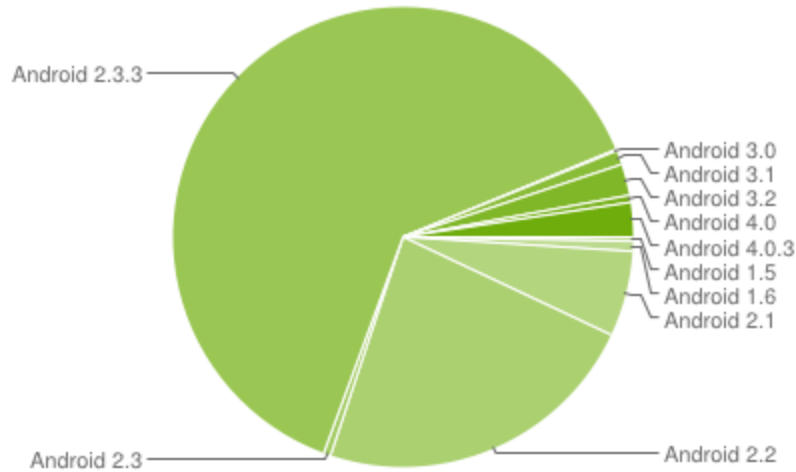


Figure 4: Pie Chart of Android Version Distribution [11]

Platform	Distribution
Android 1.5	0.3%
Android 1.6	0.7%
Android 2.1	6.0%
Android 2.2	23.1%
Android 2.3 – Android 2.3.2	0.5%
Android 2.3.3 – Android 2.3.7	63.2%
Android 3.0	0.1%
Android 3.1	1.0%
Android 3.2	2.2%
Android 4.0 – Android 4.0.2	0.5%
Android 4.0.3	2.4%

Table 2: Numerical Distribution of Android Versions [11]

This indicates that at least Android 2.1 should be supported. Supporting this version would mean that approximately 99% of all Android devices could support the application, due to an application's forward capability. Unfortunately, if DroidCDF's target platform is Android 2.1, the application cannot utilize any of the significant user interface (UI) changes introduced in later versions, such as the ActionBar. However, a sleek UI is secondary to DroidCDF's main goal of being functional to as many researchers as possible. An aesthetically pleasing UI can still be

achieved with earlier versions of Android, so the official target platform for DroidCDF is Android 2.1. As the market changes and later versions of the Android operating system become the standard, future releases of DroidCDF can include an updated interface.

The version distribution in **Figure 4**, which was used to decide DroidCDF's target platform, has been consistent in recent months, as shown in the histogram in **Figure 5**:

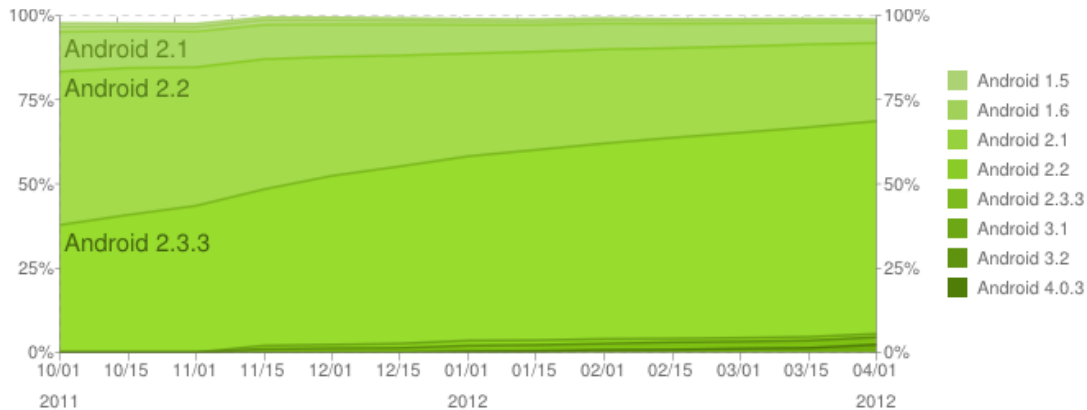


Figure 5: Trend-Over-Time Histogram of Android Versions [11]

This indicates the trend will likely continue for the next several months. A new version of DroidCDF will be released when older versions, such as 2.1, are replaced with Android 3.0 and higher.

1.6 Additional Libraries

DroidCDF integrates Hai Bison's android-filechooser version 3.3 project into the application. This filechooser is licensed under the Apache License, version 2.0. None of Bison's files have been modified in any way and all attributions have remained intact. This particular filechooser was selected to be a part of the application because of its simple to use interface, general look and feel, and licensing. One of the main goals of DroidCDF is to be an open application that anyone can modify, so a dependent library that can be redistributed is necessary.

Chapter 2: Application Overview

DroidCDF has three main phases: creation of a file, writing data to a file, and reading data from a file. A graphical representation of the application's workflow is seen in **Figure 6**:

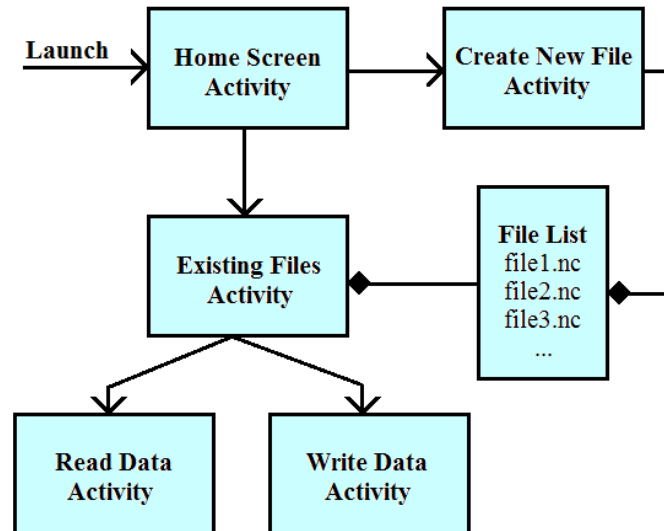


Figure 6: DroidCDF Workflow

Once the application is launched, the user can choose to either create a new file or browse the existing files. If the former action is chosen, then the user will define the file's meta-data and the file will be written to the device's file system. If the user chooses to browse the device's existing netCDF files, the user selects a file and can then either write to or read from the file.

2.1 The Home Screen

When the DroidCDF application is launched, the `HomeScreenActivity` is launched, which can be seen in **Figure 7**:



Figure 7: DroidCDF Home Screen

This entry point is specified in the AndroidManifest file in the following manner:

```
<activity android:name="edu.uno.droidcdf.HomeScreenActivity"
android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Together, the tags `<action />` and `<category />` specify that `HomeScreenActivity` is the main entry point. All other activities are specified with simply the `<activity />` tag. The label `android:name` specifies which package the desired element resides in, while the tag `android:label` refers to the title to be displayed for the activity. It is best practice to not

hard-code strings in the XML files; instead, there is a resource file (strings.xml) that contains all strings. A specific string is referenced by @string/string_name.

Every activity is assigned a layout. In DroidCDF, these layouts are XML files that are supplied in the activity's onCreate () method. For example, the XML layout for HomeScreenActivity is defined in the file main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/RLayout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <ImageView
        android:id="@+id/logoImage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:src="@drawable/image1" />

    <TextView
        android:id="@+id/lblWelcome"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/logoImage"
        android:layout_centerHorizontal="true"
        android:text="@string/welcome" />

    <Button
        android:id="@+id/btnCreateNew"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/lblWelcome"
        android:layout_centerHorizontal="true"
        android:text="@string/create"
        android:onClick="switchToCreateFileActivity"/>

    <Button
        android:id="@+id/btnExisting"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/btnCreateNew"
        android:layout_centerHorizontal="true"
        android:text="@string/existing"
        android:onClick="switchToExistingActivity"/>

</RelativeLayout>
```

and is loaded with the method setContentView():

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}

```

where `R.layout.main` is the layout argument, `main` being the name of the file less the extension. Every element in the layout is called a `View` or a `ViewGroup`, which is “a special view that can contain other views (called children).” [11] The layout file must have one root `ViewGroup`. Typically, this root is one of several subclasses, which are described in **Table 3:**

Class	Description
<code>FrameLayout</code>	Layout that acts as a view frame to display a single object.
<code>Gallery</code>	A horizontal scrolling display of images, from a bound list.
<code>GridView</code>	Displays a scrolling grid of m columns and n rows.
<code>LinearLayout</code>	A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.
<code>ListView</code>	Displays a scrolling single column list.
<code>RelativeLayout</code>	Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).
<code>ScrollView</code>	A vertically scrolling column of elements.
<code>Spinner</code>	Displays a single item at a time from a bound list, inside a one-row textbox. Rather like a one-row listbox that can scroll either horizontally or vertically.
<code>SurfaceView</code>	Provides direct access to a dedicated drawing surface. It can hold child views layered on top of the surface, but is intended for applications that need to draw pixels, rather than using widgets.
<code>TabHost</code>	Provides a tab selection list that monitors clicks and enables the application to change the screen whenever a tab is clicked.
<code>TableLayout</code>	A tabular layout with an arbitrary number of rows and columns, each cell holding the widget of your choice. The rows resize to fit the largest column. The cell borders are not visible.
<code>ViewFlipper</code>	A list that displays one item at a time, inside a one-row textbox. It can be set to swap items at timed intervals, like a slide show.
<code>ViewSwitcher</code>	Same as <code>ViewFlipper</code> .

Table 3: Typical ViewGroups [11]

The `HomeScreenActivity` layout uses a `RelativeLayout` as the root `ViewGroup`. Each child `View` is given an `android:id` that the other children can reference

in order to specify their relative positions. On the home screen, a `Button` to switch to the existing files activity is below the `Button` to switch to the create a new file activity, which is below the welcome text `TextView`, which is below an `ImageView` of the application's logo. The labels `android:layout_width` and `android:layout_height` should be set for every `View` and `ViewGroup`. The value `fill_parent` declares that the child should take up either the entire width or entire height of its parent, while `wrap_content` declares that the child should only take up as much width or height as it needs.

Android `Buttons` should have an associated `onClickListener()`. This listener can be set in the XML file to call a specific activity when the `Button` is click with the tag `android:onClick="methodName"`, or the listener can be set programmatically. In the `HomeScreenActivity`, clicking on the create button will call the method `switchToCreateFileActivity`. In the code, this method looks like the following:

```
public void switchToCreateFileActivity(View v) {
    startActivity(new Intent(getApplicationContext(), CreateFileActivity.class));
}
```

The parameter `View v` is the `View` that called the method, in this case the `Button` `btnCreateNew`. When this method is called, `startActivity()` is called with `CreateFileActivity.class` as an argument, which is the activity that will be started. The `Button` for existing files is symmetrical to this process.

An image resource, such as a logo or an icon, can have several resolutions. In the application's directory, there are four basic folders: `drawable-ldpi`, `drawable-mdpi`, `drawable-hdpi`, and `drawable-xhdpi`. The image is placed into each of the folders saved with an appropriate resolution – respectively low, medium, high, and extra-high resolution. The Android operating system will automatically supply the best image for the device. Similarly, several layouts can be

defined; for example, an application could have one layout for smartphones and another layout for extra-large devices, such as tablets. A different layout can even be specified for landscape versus portrait orientation. The application would automatically run the most appropriate layout for the device.

A `TextView` displays text to the user. If this text should be editable, the subclass `EditText` should be used, although a `TextView` can have its text modified. [11]

2.2 Creating Files

Creating a netCDF-3 file with DroidCDF is an easy task. The general workflow of creating a new netCDF file with DroidCDF is shown in **Figure 8**:

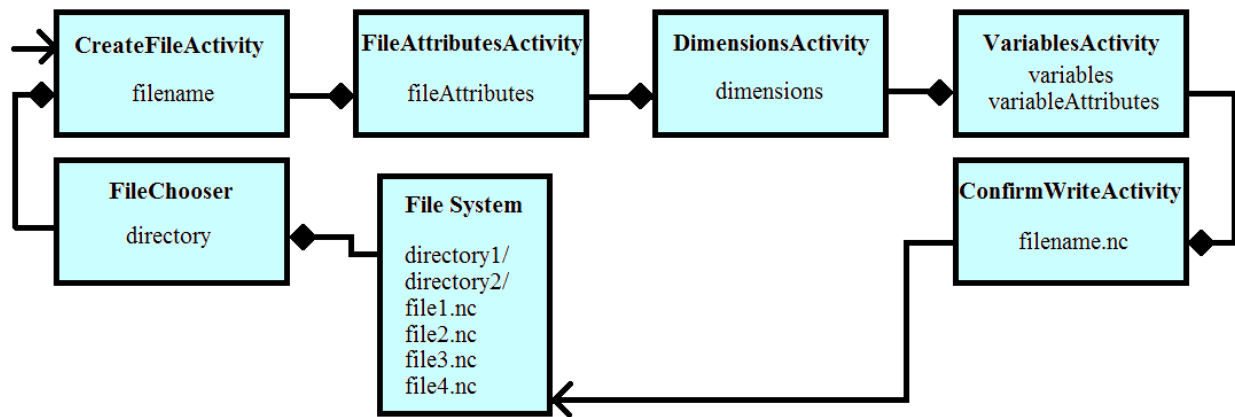


Figure 8: Creating a New NetCDF File with DroidCDF

2.2.1 CreateFileActivity

The `CreateFileActivity` is called by the clicking on the create file Button on the application's home screen. The `CreateFileActivity` allows the user to supply a filename and choose the directory to place the new netCDF file. It can be seen in **Figure 9**:

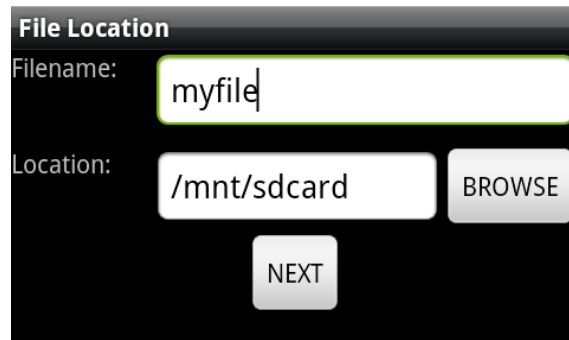


Figure 9: CreateFileActivity

The user must type in a filename but can either type in the directory or use the filechooser to select the directory by clicking on the browse Button.

The file chooser is started with the method `startActivityForResult()` rather than simply `startActivity()`. This allows the filechooser to return the directory path to the activity with the method `onActivityResult()`. The return value is then displayed in an `EditText` with the filename displayed in an additional `EditText`. When the user is satisfied with the filename and directory he clicks the next Button. This button's `onClickListener()` calls the method `goToFileAttributesActivity()`.

This method first checks that the filename and directory the user supplied are valid, meaning:

- The filename is a valid name for the Android file system. If the user wishes to transfer this file to a different operating system at a later time, it is the user's responsibility to ensure the filename will also be valid for that operating system.
- The directory path ends with a forward slash. If the directory path does not end with a forward slash, the parser will place one at the end of the directory path.
- The filename begins with a forward slash. If the filename begins with a forward slash, it will be removed.

- The filename ends with the extension “.nc”. If it does not, the parser will place this extension at the end of the filename. While the file extension is arbitrary, this is the general convention for a netCDF classic file.
- The directory is a valid, writeable directory. As a condition of using the application, the user must give permission for DroidCDF to write to external storage. This means that files can be written to any read-write directory, including on the SD card.
- The filename is not already in use by another file. It is important to note that this condition is checked here, because the application will not block I/O. If the user leaves the application for a while, creates a file with the given name, and then resumes creation of the netCDF file, the netCDF file will overwrite the existing file.

If any of the above conditions are not met and cannot be corrected by the parser, the method will throw a new `IOException` which will be displayed in an `AlertDialog` to the user. If the parser does not throw any errors, the `FileAttributesActivity` is started. If an error is caught, though, the user will not be able to proceed through the workflow until the error is corrected. If the user chooses to press the back button, then this `CreateFileActivity` will be popped from the backstack.

To start the `FileAttributesActivity`, a new `Intent` is defined. The filename is added to the `Intent` with the method `putExtra()`. This extra can then be retrieved by the class started by the `Intent`.

2.2.2 FileAttributesActivity

The `FileAttributesActivity` allows the user to add attributes to describe the entire file. These global attributes could include version number, authors, creation date, etc. Each attribute has a name, value, and value type. This screen can be seen in **Figure 10:**

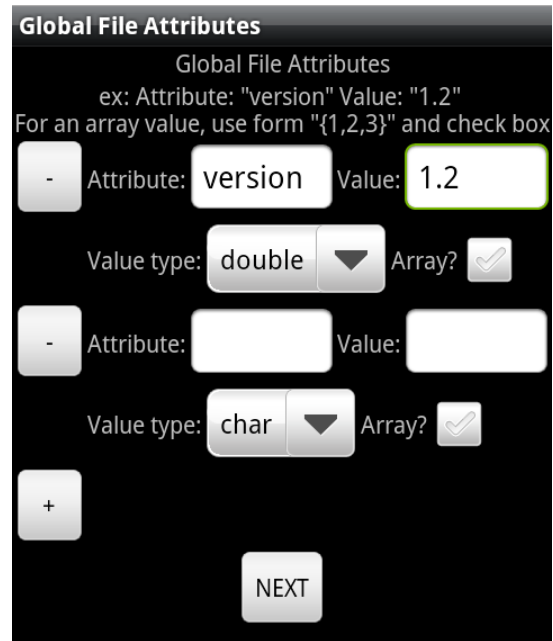


Figure 10: FileAttributesActivity

The `FileAttributesActivity` uses a `LinearLayout` as the root `ViewGroup`. When using a `LinearLayout`, the layout's value `android:orientation` should be set to either `horizontal` or `vertical`. This specifies in which direction child Views should be placed.

An important child View of the root `ViewGroup` is an additional `LinearLayout` labeled `fileAttributesList`. Child Views can be added to this layout programmatically.

In the code, this layout is defined in the `onCreate()` method:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.fileattributes);
    list = (LinearLayout) findViewById(R.id.fileAttributesList)
    ...
}
```

First the main layout, `fileattributes.xml`, is loaded with `setContentView()`. From then, any element in `fileattributes.xml` can be referenced with the method `findViewById()`. When the user clicks on the “add file attribute” Button, a child View is added to `list` (`DroidCdf`

refers to this `View` as a row) with the helper method `addFileAttr()`. In turn, `addAttributeRow()` is called.

```
public void addAttributeRow(final Context context, final LinearLayout
    attributesLayout) {

    LayoutInflater inflater = (LayoutInflater)
        context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
    View attributeRow = inflater.inflate(R.layout.fileattributesrow, null);

    Spinner spinner = (Spinner)attributeRow.findViewById(R.id.spinType);
    ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(
        this, R.array.types, android.R.layout.simple_spinner_item);

    adapter.setDropDownViewResource(
        android.R.layout.simple_spinner_dropdown_item);
    spinner.setAdapter(adapter);
    Button delete = (Button)attributeRow.findViewById(R.id.btnDelete);
    delete.setTag(attributeRow);
    delete.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            View attributeRow = (View)v.getTag();
            attributesLayout.removeView(attributeRow);
        }
    });

    attributesLayout.addView(attributeRow);
}
```

This method uses a `LayoutInflater` to instantiate the row's layout, `fileattributesrow.xml`.

Each file attribute is associated with a unique row `View`. Every row has a delete `Button`, which when clicked will remove the row from the `LinearLayout` list. The appropriate row is deleted by setting the `Button`'s tag to be the attribute row. When the `OnClickListener()` method `onClick()` is called, the `Button`'s tag can be retrieved, cast, and deleted. Each row also has a `Spinner`, which is a drop-down menu widget that uses radio buttons to select items. [11] This `Spinner` is opened by clicking on it; it can be seen in use in Figure 11:

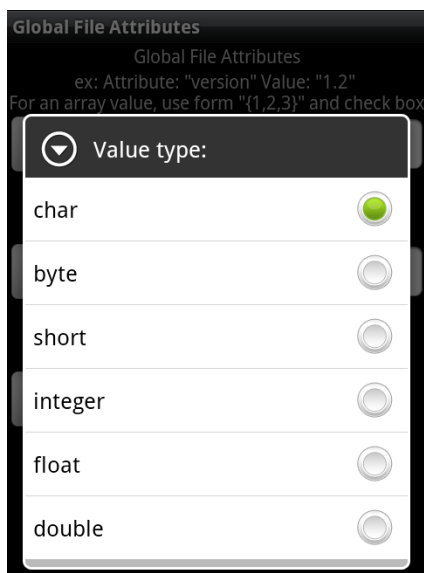


Figure 11: Spinner for Value Type

This spinner is used to select the type for the attribute's value. The selected value can be retrieved with the method `Spinner.getSelectedItem()`. Additionally, each row has a `CheckBox`; if this box is selected, the value is saved as an array of the same type.

When `FileAttributesActivity` is started, `HashMap<String, Object>` `attributes` and `String filename` are instantiated in addition to `LinearLayout list`. `filename` is the extra that was sent by `HomeScreenActivity` and `attributes` is a mapping of attribute names with their value. They are both set in the activity's `onCreate()` method:

```
public void onCreate(Bundle savedInstanceState) {
    ...
    filename = getIntent().getStringExtra("filename");
    attributes = new HashMap<String, Object>();
    ...
}
```

When the next `Button` is clicked, the method `goToDimensionsActivity()` is called.

This method first clears the `attributes` map in case the user arrives at the screen from pressing the back button; this could cause the method to attempt to place double keys in the map.

Next, `parseAttributes()` is called. This method iterates through each row in `list` and

attempts to add the data to `attributes`. To be a valid attribute, the row must meet the following conditions:

- The name cannot be an empty string.
- The name cannot be in use by a file attribute already.
- The value cannot be an empty string.
- The value must agree with the selected value type. For example, the value 1.3 cannot be saved as an integer, but it can be saved as a double.

If any of these conditions are not met, a new `ParsingException` will be thrown. When all attributes have been parsed, `filename` and `attributes` are added as extras to a new `Intent`, which then starts `DimensionsActivity`.

2.2.3 DimensionsActivity

The `DimensionsActivity` allows users to add dimensions to the file. All dimensions for every file should be defined here, though each variable's dimensions will be specified in the next activity. Each Dimension can be used multiple times so it only needs to be defined once. The `DimensionsActivity` can be seen in **Figure 12:**

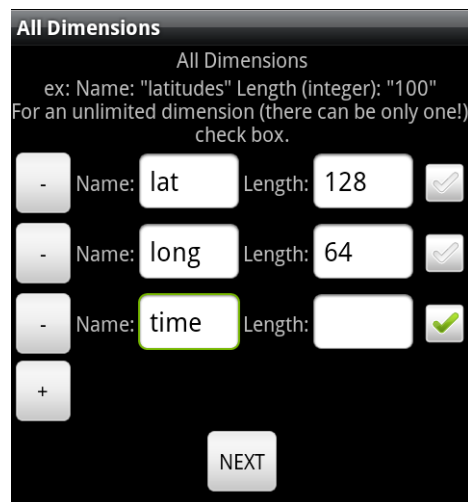


Figure 12: DimensionsActivity

Like the file attributes, there is a `LinearLayout` that holds dimension rows. Each row is inflated with an instance of `dimensionsrow.xml`. If the `add Button` is selected, a new dimension row is added to the `LinearLayout`. Similar to the attribute rows, each dimension row has a `delete Button` that will remove the row when clicked.

Every netCDF dimension must have a name and length; one dimension can have an unlimited length. To specify an unlimited dimension, the row's `CheckBox` is checked. Otherwise, the length must be a positive integer. A user does not have to declare any dimensions; in this case, every variable would be a scalar variable.

When the user clicks the `next Button`, the method `goToVariablesActivity()` is called. This method first clears the `HashMap<String, Integer> dimensions` to prevent duplicate dimensions if the activity was started by clicking the back button. The dimension rows are then parsed; each row must meet the following conditions:

- The dimension name cannot be an empty string.
- The dimension name cannot be in use by another dimension already.
- The dimension length must be a positive integer, unless the dimension is unlimited. In this case, the length is set to -1.
- The dimension cannot be unlimited if another dimension is already unlimited.

In the future, if NetCDF Java is updated to support writing netCDF-4 files, there could be multiple unlimited dimensions. If any of the above dimensions are not met, a new `ParsingException` is thrown. Otherwise, the row is added to `dimensions`.

When `DimensionsActivity` is first called, the sent `extras` are extracted into a `Bundle`, which is a mapping of `extras'` names to their values. In `goToVariablesActivity()`, `dimensions` is added to the `Bundle` with

`putSerializable()`. This Bundle is then added to a new Intent with `putExtras()`, rather than `putExtra()`, which then starts `VariablesActivity`:

```
public void goToVariablesActivity(View v) {
    Intent intent = new Intent(this, VariablesActivity.class);
    try {
        dimensions.clear();
        parseDimensions();
        data.putSerializable("dimensions", dimensions);
        intent.putExtras(data);
        startActivity(intent);
    } catch (ParseException e) {
        ...
    }
}
```

2.2.4 VariablesActivity

The `VariablesActivity` allows users to add variables to the netCDF file. Like the file attributes and dimensions, each variable is given its own row. However, this row also has a `LinearLayout` child that holds that particular variable's attributes. These attributes are defined in the same way as file attributes and must meet the same conditions. Variable attributes do not have to have the same value type as the variable itself; instead, the attribute's value type must agree with the value's content. The `VariablesActivity` can be seen in **Figure 13**:

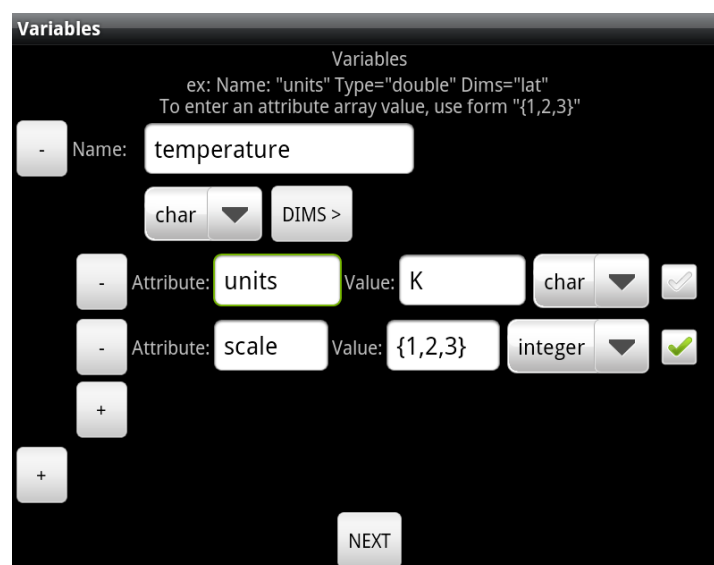


Figure 13: VariablesActivity

Each variable row has a `Spinner` to define the variable's type. The `dims>` `Button` opens an `AlertDialog` when clicked. This dialog uses the method `setMultiChoiceItems()` to populate itself with the dimensions that were passed to `VariablesActivity` through the `Intent`'s `Bundle`. This dialog is shown in **Figure 14**:

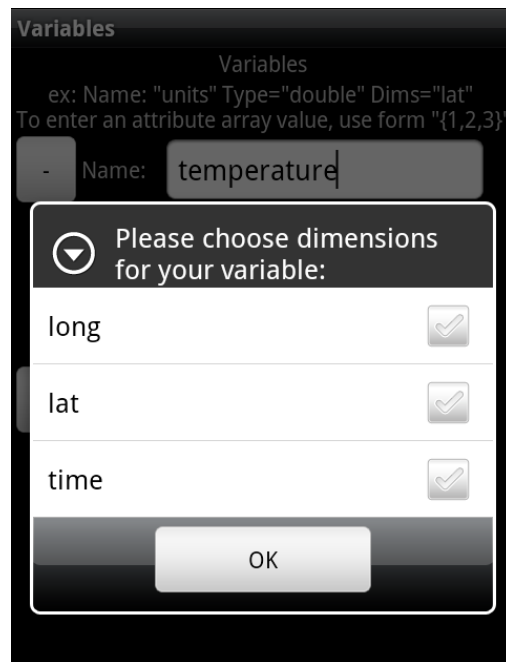


Figure 14: Dimensions Dialog

The number of dimensions selected will determine the shape of the variable. If no dimensions are selected, the variable will be a scalar. If one dimension is selected, the variable will be a one-dimensional array of the type chosen in the `Spinner`. If two dimensions are selected, the variable will be a two-dimensional array, and so on. If an unlimited dimension is selected, the variable will have a record dimension. The order of selection does not matter; when the file is created in `ConfirmWriteActivity`, the unlimited dimension will be written first.

When defining a variable's dimensions and attributes, the `Variable`'s row is used as the key in both `HashMap<View, ArrayList<String>> variableDimensions` and `HashMap<View, HashMap<String, Object>> variableAttributes`. This is so

the dimensions and attributes will be properly associated with their variable. If the variable's name was used as the key, then it is possible that the user could have two variables with the same name that would not be caught until the next `Button` is pressed. This would mean that the wrong dimensions and attributes would be associated or could be erroneously deleted. Before adding the variable data to the `Bundle`, the keys are converted to the variable's name with the methods `changeDimsToString()` and `changeAttrsToString()`. This conversion takes place only after the variables have been parsed to ensure no two variables have the same name and allows the `HashMap`s to be added to the `Bundle`. Only serializable or parceable data can be added to a `Bundle`, and `Views` are currently neither. `ConfirmWriteActivity` is then started. At least one variable should be created in `VariablesActivity`, although a variable can have no attributes.

2.2.5 ConfirmWriteActivity

The `ConfirmWriteActivity` serves two purposes: to allow the user to confirm the file's metadata (file attributes, dimensions, variables, and variable attributes), and to create the netCDF-3 file. When this activity is started, the metadata that the user has entered is parsed and displayed. First, the `Bundle` is retrieved and each `Extra` is extracted in the `onCreate()` method:

```
public void onCreate(Bundle savedInstanceState) {
    ...
    data = getIntent().getExtras();
    filename = data.getString("filename");
    fileAttributes = (HashMap<String, Object>)
        data.getSerializable("fileAttributes");
    dimensions = (HashMap<String, Integer>)
        data.getSerializable("dimensions");
    variables = (HashMap<String, String>)
        data.getSerializable("variables");
    variableDimensions = (HashMap<String, ArrayList<String>>)
        data.getSerializable("variableDimensions");
    variableAttributes = (HashMap<String, HashMap<String, Object>>)
        data.getSerializable("variableAttributes");
    ...
}
```

`ConfirmWriteActivity`'s xml layout defines several `TextView`s. Even though the text will be edited programmatically before being displayed, the user should not be able to modify the text so `EditText` is not used.

To display the filename, `setFilename()` is called, which sets the `TextView` `lblFileName`:

```
public void setFilename() {
    ((TextView) findViewById(R.id.lblFileName)).setText(filename +
        "\n");
}
```

The method `setFileAttributes()` sets the text of `confirmFileAttributesList`. Each file attribute of the `fileAttributes` map is first parsed and the appropriate `toString()` method is called. If the attribute's value is not cast first, the value does not always display properly; thus, this extra step is taken as a precaution. If `fileAttributes` is empty, then the `String` "None" is displayed.

Dimensions and variables are similarly set. An example of this screen is seen in **Figure 15**:

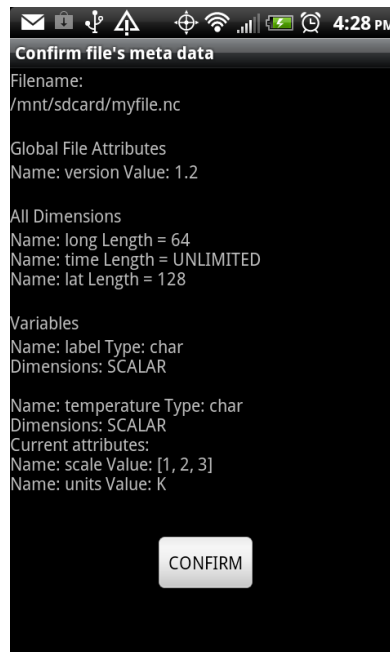


Figure 15: ConfirmWriteActivity

If the user is satisfied with the data displayed, the confirm Button is clicked. The `onClickListener()` for confirm calls `writeFile()`:

```

public void writeFile(View v) {
    try {
        NetcdfFileWriteable ncfile =
            NetcdfFileWriteable.createNew(filename, false);
        addDimensions(ncfile);
        addVariables(ncfile);
        addFileAttributes(ncfile);
        ncfile.create();
        ncfile.close();

        ...

    } catch (IOException e) {
        ...
    }
}

```

This method initializes a `NetCdfWriteable` file, `ncfile`. The method `addDimensions(ncfile)` adds every dimension to `ncfile` with `addDimension()` or `addUnlimitedDimension()`. For every variable to be added to the file, an `ArrayList<Dimension>` must be created with the appropriate dimensions, to be supplied as an argument for `ncfile.addVariable()`. This `ArrayList` is populated with values from `variableDimensions`. Once a variable is added to the file, its attributes (if applicable) are added with the method `addVariableAttributes()` that calls `ncfile.addVariableAttribute()`. File attributes are added with the method `addFileAttributes()` that calls `ncfile.addGlobalAttribute()` in turn. If any of these methods fail, it will throw a new `IOException` and the user will have to make the appropriate changes suggested in the error message. Otherwise, the file is written and an `AlertDialog` displays to notify the user of the file's success, seen in **Figure 16**:

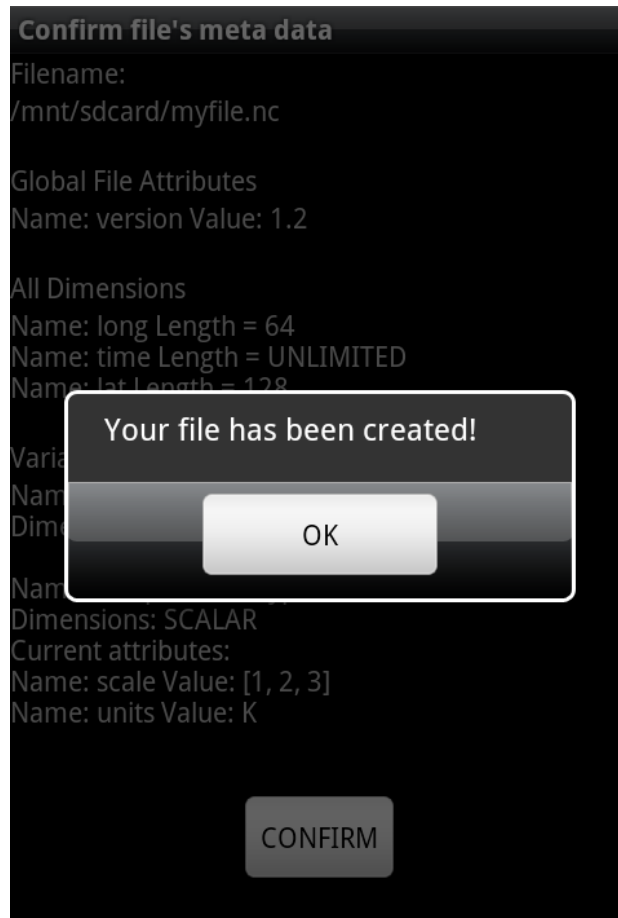


Figure 16: Creation Successful Dialog

The application then returns to the home screen, where the user may either create a new file or open an existing file for reading or writing.

2.3 Reading from Files

The reading and writing functionalities have been combined into one interface. A filename is supplied either with the filechooser or by typing the path name into an `EditText`, as seen in

Figure 17:

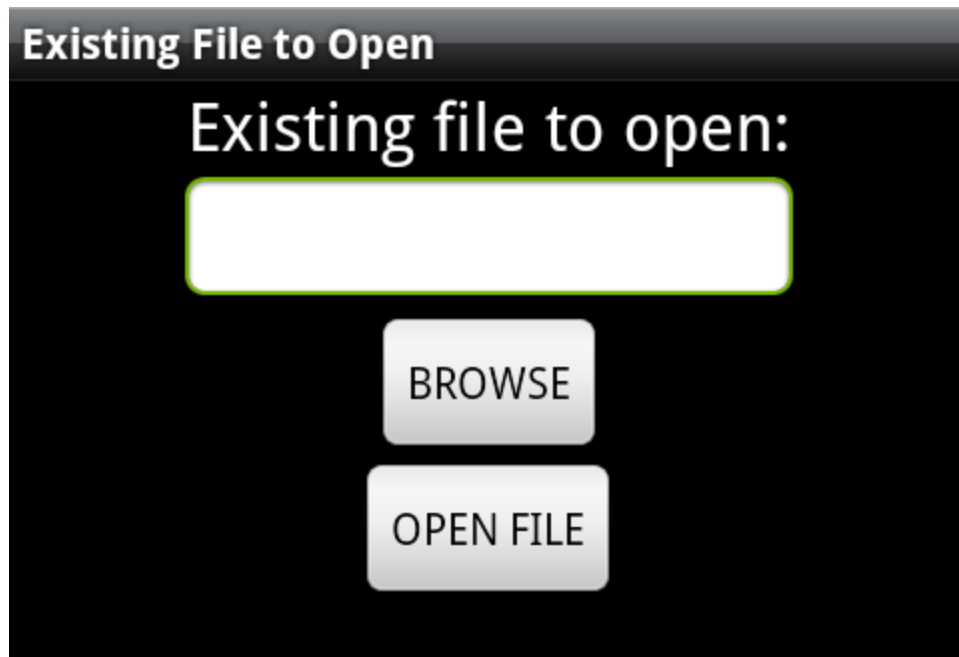


Figure 17: ExistingActivity

This is achieved by placing the editing capabilities in ContextMenus whose conditions for displaying are that the open file is in the netCDF-3 format. The condition, a Boolean called `writable`, is set in the ExistingActivity in the following manner:

```
String fileType = ncfile.getFileTypeId();
if(fileType.equalsIgnoreCase("netCDF"))
    setWritable(true);
else
    setWritable(false);
```

The file format is specified in the first four bytes of the file, but NetCDF Java can also return a human-readable format Id with the method `getFileTypeId()`. The process above uses this method to compare the file's Id with the String "netCDF," which indicates that the file follows the netCDF-3 classic format. Other common Ids are "netCDF-4" and "HDF5". [12] If the open file is a netCDF-3 file, then it can be edited by DroidCDF. Otherwise, the file is read-only.

The common read-write interface consists of a `TabLayout`. A ribbon of tabs is placed at the top of the screen with a content frame underneath. Whenever a user clicks on a tab, its

associated content is displayed. In the XML layout file, the `TabLayout` is defined with a `<TabHost />` tag whose two children are the ribbon (a `Tab Widget`) and frame (a `FrameLayout`). To create the tabbed interface, the Android developer tutorial found at <http://developer.android.com/resources/tutorials/views/hello-tabwidget.html> was initially used.

The tabs in the widget can either switch views within a single activity or switch views between two entirely separate activities. In DroidCDF, each tab is associated with a certain activity:

- `ViewFileAttributesActivity` – displays the file’s global attributes.
- `ViewDimensionsActivity` – displays the file’s dimensions.
- `ViewVariablesActivity` – displays the file’s variables and variable attributes.
- `ViewDataActivity` – displays the data for a specific variable.

Each of these four activities extends a `ListActivity` rather than a normal `Activity`. This is because a `ListView` is used to display the respective content rather than a `LinearLayout`. There could potentially be hundreds or thousands of elements to view if a file is being supplied, though only a handful can be displayed at any given time; `ListViews` recycle `Views` in order to improve efficiency. The savings are especially noticeable if a user “flings” through a long list; a `LinearLayout` would appear choppy, while a `ListView` would be much more seamless. [11] When a user is creating a file with DroidCDF, this is not as much of a concern because elements are added one at a time and the user will most likely not need the performance boost during this phase. In exchange, the `LinearLayout` code is much simpler to implement.

The `TabHost` is defined in the `OpenFileActivity`. Each tab must have some `View` for the user to click on, such as an image or text. Each tab can be referenced with an id number

that is associated with the order in which the tabs were added to the `TabHost`. For example, in DroidCDF, the `ViewFilesAttributes` tab is added first and displayed with a call to:

```
tabHost.setCurrentTab(0);
```

To demonstrate the read-write interface, data was taken from

<http://www.unidata.ucar.edu/software/netcdf/examples/files.html>. In particular, the file

`cam1_0000-09-01_64x128_L26_c030918.nc` was used as it was sufficiently large to test the application and it included a variety of attributes, dimensions, and variables. The read-write interface can be seen in **Figure 18**:



Figure 18: Read-Write Interface

Each tab's button should be assigned two images: one for when the tab is currently selected, and one for when the tab is not selected. These images can be added to a `selector`, which will automatically supply the correct image to the application [11]. Refer to code below from the Developer's Guide:

```
<?xml version="1.0" encoding="UTF-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <!-- When selected, use grey -->
```

```

<item android:drawable="@drawable/ic_tab_artists_grey"
      android:state_selected="true" />
<!-- When not selected, use white-->
<item android:drawable="@drawable/ic_tab_artists_white" />
</selector>

```

An example of this image selection can be seen in **Figure 19**:

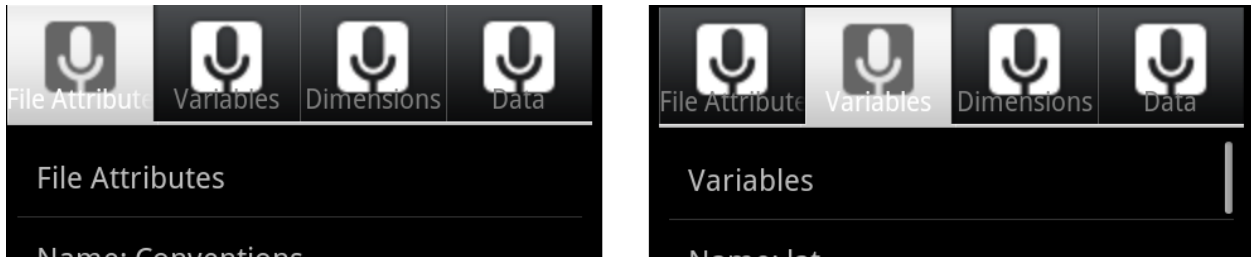


Figure 19: Image Selector

When a user selects the `ViewFileAttributesActivity` pane, its respective button is dark gray. When another tab is selected, the button becomes white.

When the file is not writeable, the `ListActivities` will simply be populated with the appropriate strings and the user will not be able to click on any individual `View` element.

However, if `writable` is true, then the file can be edited through `ContextMenus`.

2.4 Writing Files

Each `ListView` is registered for a `ContextMenu` in the following manner:

```

ListView list = getListView();
list.setTextFilterEnabled(true);
if(writable) {
    ...
    registerForContextMenu(list);
}

```

The menu can then be displayed when the user long-presses on an item in the list. The long-clicks have a special `onLongClickListener()` that calls the method `onCreateContextMenu()`. Each item has a sequential position in the list, which is necessary for defining separate context menus. For example, in the

ViewFileAttributesActivity, the first child in the ListActivity has a separate menu then all other elements and is shown in **Figure 20**:

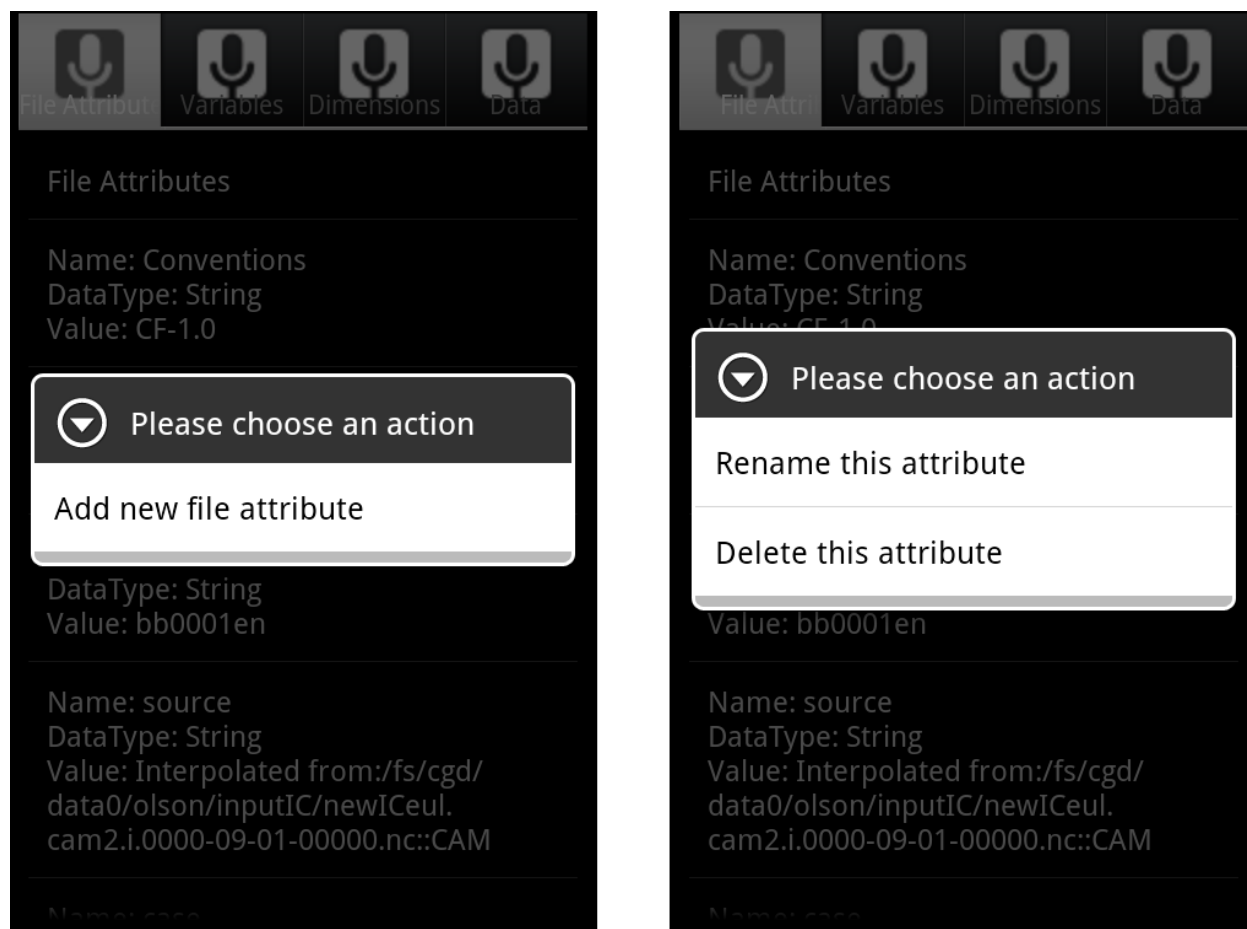


Figure 20: Two Separate ContextMenus Defined on the Same ListActivity

```

@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo
    menuInfo) {
    AdapterView.AdapterContextMenuInfo info =
        (AdapterView.AdapterContextMenuInfo)menuInfo;
    menu.setHeaderTitle(TITLE);
    if(info.position == 0){
        menu.add(Menu.NONE, ADD, Menu.NONE, ADD_TEXT);
    } else {
        menu.add(Menu.NONE, RENAME, Menu.NONE, RENAME_TEXT);
        menu.add(Menu.NONE, DELETE, Menu.NONE, DELETE_TEXT);
    }
}

```

This particular setup allows the user to perform modifications on the `ListView` as a whole, such as adding a new global attribute to the list, that do not deal with specific items. This pattern is repeated for several of the tabs.

Each item of the `ContextMenu` is given a label and unique integer identifier. When an item is selected, the method `onContextItemSelected()` is called. For example, below is the call in `ViewFileAttributesActivity`:

```
@Override
public boolean onContextItemSelected(MenuItem item){
    final AdapterView.AdapterContextMenuInfo info =
        (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
    int itemId = item.getItemId();
    switch(itemId){
        case ADD:
            addFileAttribute(this);
            break;

        case DELETE:
            deleteFileAttribute(info.position);
            break;

        case RENAME:
            renameFileAttribute(info.position);
            break;

    }
    return true;
}
```

The identifiers are perfect for a `switch` statement. Here, the appropriate method is called based on which `ContextMenu.Item` was selected.

With the `ContextMenu`, users can add, delete, modify, and rename attributes, variables, and dimensions. If a user wants to edit any information in the file header, the file must be put into redefine mode. After the edit is finished, the file must be taken out of redefine mode and the changes should be immediately flushed to minimize the risk of losing changes. In general, a user should try to set as much of the header as possible during file creation. The header is given minimal extra space (usually only until the next available file block), so any edit that changes the

size of the metadata will mean rewriting all data in the file. This can be a very expensive operation.

When an element is selected to modify, the user selects the appropriate action from the `ContextMenu`. This will generally cause an `AlertDialog` to show which guides the user through their action. For example, if the user decides to add a new file attribute to his file, he should long click on the element labeled “File Attributes.” This is the first element in the list, and has a unique menu. The user can then click on the add item, which will cause the display in

Figure 21 to show:

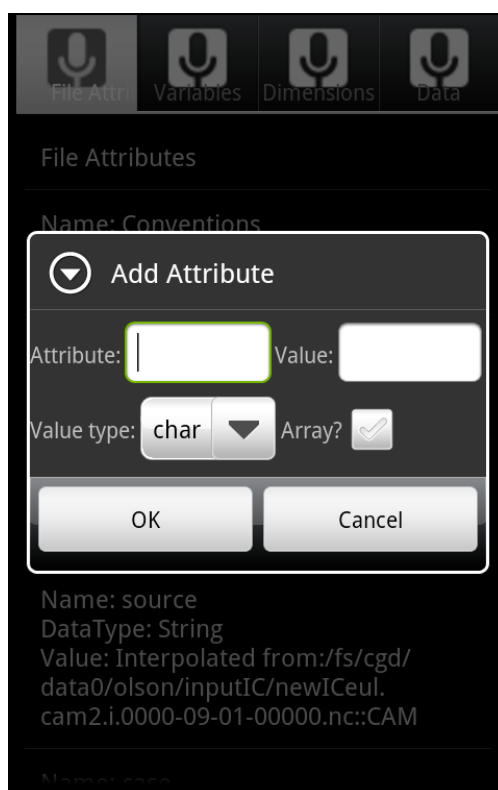


Figure 21: Add Attribute AlertDialog

This dialog behaves similar to the process of adding a file attribute during the creation phase. If the user’s values cannot be parsed or are incomplete, a `Toast` message will appear with information on the error incurred and how the user can rectify the data.

From the `ViewVariablesActivity` list, the user can select a specific variable to modify or to view its associated data. **Figure 22** shows the Variables tab:

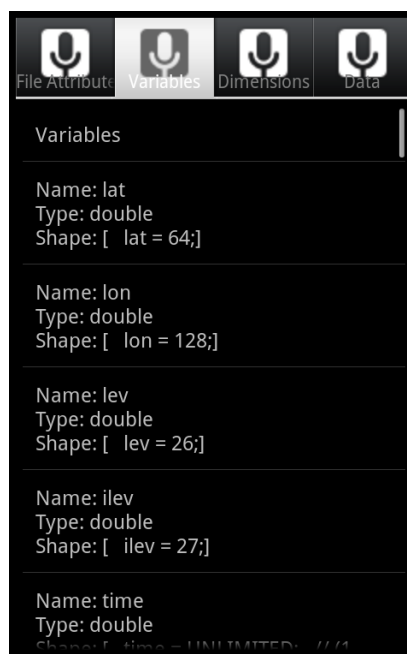


Figure 22: ViewVariablesActivity

Each item displays the variable's name, `DataType`, and `Shape` (the dimensions and dimension lengths that are associated with the variable). Each variable's attributes can also be accessed through the `ContextMenu`.

2.5 Permissions

An Android permission allows an application to utilize certain protected features of the operating system. The user must agree to these permissions in order to install the application. Each permission is declared in the `AndroidManifest`. [11] For example, `DroidCDF` requires permission to write files to the SD card; this permission is granted in the following manner:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
...
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
...
```

Chapter 3: DroidCDF Analysis

3.1 File Creation

File creation is a simple and stream-lined process. Defining the global file attributes leads to defining the dimensions, which in turn leads to defining the variables and variable attributes. Each of these elements is collected in a `LinearLayout`. A `LinearLayout` was chosen over a `ListView` for this phase due to the expected behavior of the user. Typically, a user will add elements one at a time and will have no need of quickly flinging through the list. The operating system would thus have sufficient time to seamlessly draw each `View`.

In the future, DroidCDF could extend Variables, Dimensions, and Attributes to be serializable; these objects could then be passed in the `Intents` rather than `HashMaps`. This would also reduce the sometimes redundant parsing that is necessary now.

File creation speed is generally in the low millisecond ranges. To get an average time taken, several trials were run. These included varying the number of attributes, dimensions, and variables as well as value types. **Table 4** lists recordings from varying the number of dimensions written to the header at creation time while keeping everything else constant, including the set of file attribute, dimension name length, dimension length, no unlimited dimensions, and the set of variables :

Number of Dimensions	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
One	9	7	8	6	7	7.4
Two	7	5	5	6	5	5.6
Five	6	8	8	8	8	7.6

Ten	10	6	8	7	8	7.8
Twenty-Five	7	13	10	9	13	10.4
Fifty	11	7	15	10	10	10.6
One Hundred	8	12	17	15	15	13.4

Table 4: File Creation Time with Varying Number of Dimensions

The chart shows that the average creation time slowly rises with the increase in dimensions being written to the file header. This is to be expected, as there is more input and output. However, it shows that the difference between writing one dimension or a hundred dimensions is, on average, only 6 ms. **Figure 23** shows these trends visually:

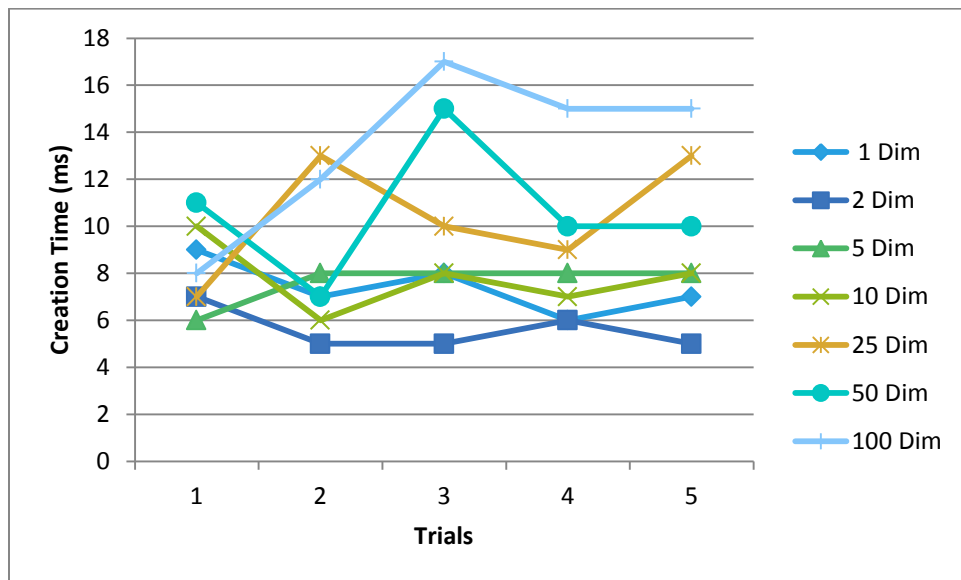


Figure 23: Graph of Creation Times with Varying Number of Dimensions

The results of trials based on attribute number and trials based on variable number were not significantly different.

3.2 Data Write

The most expensive operations in DroidCDF are editing the file header after the file has already been created. Each edit is done independently to minimize the risks of corrupting the file's data, therefore multiple trials were run of: adding a new attribute (1), changing the attribute name to a

longer value (2), adding a new dimension (3), changing the dimension to be unlimited (4), adding a new scalar variable(5), adding a new two-dimensional variable (6), and adding a new two-dimensional variable with a record dimension (7).

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Change 1	15	13	14	14	8	12.8
Change 2	6	7	9	6	8	7.2
Change 3	14	16	13	12	15	14.0
Change 4	24	22	18	17	20	20.2
Change 5	10	9	11	13	9	10.4
Change 6	11	11	7	8	14	10.2
Change 7	19	18	18	24	22	20.2

Table 5: Time to Complete Change of File's Header

This table records the time taken to complete modifications of the header file and includes leaving redefine mode and flushing the changes. The times are generally greater than the times recorded during the file creation phase, which was expected. It is interesting to note that on average it took longer to add a new element to the header than to modify an existing element; this can be attributed to the fact that the header could have enough space for a longer value but not an entirely new one. Additionally, the header elements are stored in List data structures. Retrievals run in near constant time, while adding a new element can take up to $O(n)$ steps.

Writing new values to a variable is a quick operation, but if that particular variable has already been read and is being displayed by the application, the operation sees a reasonable speed increase. This implies that it is most efficient to perform all necessary operations on a single variable before loading the next variable. A table of trials is below in **Table 6:**

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Write to new variable	23	22	25	14	17	20.2
Write to existing var.	8	9	5	9	8	7.8

3.3 Data Read

The decision to combine the reading and writing phases was a very efficient one. It cut back on code within files, redundant activities, and decluttered the workflow. Reading individual elements of the file, such as a dimension or variable, is a near constant time operation as the values are stored in List data structures. The variable's data is stored in a `ListView`, so only the necessary `Views` are drawn on the screen at any one time.

3.4 Limitations

Currently, the biggest limitation of DroidCDF is the fact that the NetCDF Java library only supports writing to netCDF-3 files. When the library is updated, DroidCDF will be able to take advantage of compression, chunking, a larger selection of primitives and reference types, and all of the other large improvements that are a part of netCDF-4. Additionally, DroidCDF can only write to one unlimited dimension, and it must always be listed first for variables that use it; these restrictions are also uplifted with the enhanced data model.

The size of a file in DroidCDF is limited by the amount of space available on the SD card. Integers can be 8-, 16-, or 32-bit, though floats can be either 32-bit or 64-bit. It is also recommended that netCDF-3 files remain under two gigabytes in size; the format is designed for portability, and some filesystems cannot handle files that are larger than this size. [1]

Chapter 4: Future of DroidCDF

This research project has been a great experience. Before I started, I had never programmed an application and knew very little about netCDF. I have now completed an entire Android application and can confidently say that I have learned an incredible amount. With the skills that I have acquired, I will continue working on DroidCDF.

Before DroidCDF is released to the open market, I will add querying functionality and an example companion application that utilizes DroidCDF. These features were not in the scope of this thesis and so were not detailed, but they are important and useful for netCDF files.

The next release of DroidCDF will include a better interface for the devices that can support it. As of now, every device sees the same interface and there is no distinction between horizontal and vertical layouts. I would especially like to take advantage of the `ActionBar` widget. This widget removes the need for any physical buttons from your application and instead relies solely on the touchscreen. The `ActionBar` generally runs across the top of the application screen and holds programmed buttons, such as “home”, “back”, “menu”, and “exit.” As the market shifts towards using new platforms, this release will become much more feasible while still staying in line with DroidCDF’s goals.

I believe that DroidCDF would make an excellent companion application to other programs. For example, a tracking application could output a netCDF file that could then be read by DroidCDF. DroidCDF could create an input file for a graph displayer or weather predictor or any number of other applications. When DroidCDF is released on the open market, any researcher or programmer will have the opportunity to help move this application forward and assist in progressing the sciences.

Licensing

Copyright 2012 Brittany Allesandro

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

References

- [1] R. Rew, G. David, S. Emmerson, H. Davies, E. Hartnett and D. Heimbigner, *The NetCDF Users Guide*, Boulder, CO: Unidata Program Center, June 2011.
- [2] L. A. Treinish and M. L. Gough, "A software package for the data-independent management of multidimensional data," *EOS, Transactions, American Geophysical Union*, vol. 68, no. 28, p. 633, 1987.
- [3] D. J. Raymon, "A C Language-Based Modular System for Analyzing and Displaying Gridded Numerical Data," Physics Department and Geophysical Research Center, R&D Division, New Mexico Institute of Mining and Technology, Socorro, 1987.
- [4] "FAQ," Open Handset Alliance, 2007. [Online]. Available: http://www.openhandsetalliance.com/android_faq.html. [Accessed 20 March 2012].
- [5] R. Rew, "NetCDF FAQ," March 2012. [Online]. Available: <http://www.unidata.ucar.edu/software/netcdf/docs/faq.html>. [Accessed 15 March 2012].
- [6] J. Caron, "NetCDF Java Library," February 2012. [Online]. Available: <http://www.unidata.ucar.edu/software/netcdf-java/documentation.htm>. [Accessed 15 March 2012].
- [7] "2008 Unidata NetCDF Workshop for Developers and Data Providers," 2008.
- [8] G. Beavis, "A Complete History of Android," Tech Radar, 23 September 2008. [Online]. Available: <http://www.techradar.com/news/phone-and-communications/mobile-phones/a-complete-history-of-android-470327>. [Accessed 7 March 2012].
- [9] "Members," Open Handset Alliance, [Online]. Available: http://www.openhandsetalliance.com/oha_members.html. [Accessed 20 March 2012].
- [10] C. Ziegler, "Android: A Visual History," The Verge, 7 December 2011. [Online]. Available: <http://www.theverge.com/2011/12/7/2585779/android-history>. [Accessed 18 March 2012].
- [11] "Dev Guide," Android Developers, 2012. [Online]. Available: <http://developer.android.com/guide/index.html>. [Accessed 10 March 2012].
- [12] "CDM File Types," Unidata, 2012. [Online]. Available: <http://www.unidata.ucar.edu/software/netcdf-java/formats/FileTypes.html>. [Accessed 30 March 2012].


APPROVAL SHEET

This is to certify that Brittany L. Allesandro has successfully completed
her Senior Honors Thesis, entitled:

Creating, Writing, and Reading NetCDF Files on A Mobile Device



Mahdi Abdelguerfi Director of Thesis



Vassil R. Roussev for the Department



Carl D. Malmgren for the University
Honors Program

May 3, 2012
Date