Gate 3 - Design Document Approval and Gate Participants

**Project: *Bluetooth –* *Android Intern Project***

Date(s): *Fall 2013*

Project Manager: *Blaine Tiernan*

Prepared by: *Lee Easton*

Document status: \_**X**\_ Draft \_\_ Proposed \_\_ Validated \_\_ Approved

**Change Log**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Changes Description** | **Author** | **Date** |
| 1 | First draft | L Easton | 2013,June 10 |
| 2 | Second Draft | L Easton | 2013, July 22 |
| 3 | New Version Draft | L Easton | 2013,Oct 21 |

Contents

[NOTE 2](#_Toc362346500)

[**1. Introduction 3**](#_Toc362346501)

[1.1 Intended Audience 3](#_Toc362346502)

[1.2 References 3](#_Toc362346503)

[1.3 Acronyms and Abbreviations 3](#_Toc362346504)

[1.4 Assumptions 4](#_Toc362346505)

[**2. System Design 5**](#_Toc362346506)

[2.1 System Overview 5](#_Toc362346507)

[2.1.1 Development of MCCU Android Application (*MRS1*) 5](#_Toc362346508)

[2.1.2 Development of Adaptation layer 5](#_Toc362346509)

[2.1.3 Design for application page development for test automation 5](#_Toc362346510)

[2.1.4 Totalflow.exe (Flash) 5](#_Toc362346511)

[2.2 Deployment Diagram 6](#_Toc362346512)

[2.3 Solution Architecture (TRS1, TRS2) 6](#_Toc362346513)

[2.4 UI Information Model (TRS4, TRS5) 8](#_Toc362346514)

[2.5 UI Screens Summary 13](#_Toc362346515)

[2.5.1 Phase 1 Screens 13](#_Toc362346516)

[**3. Totalflow Android Application Design 14**](#_Toc362346517)

[3.1 Application Design Overview 14](#_Toc362346518)

[3.2 Android Activity Lifecycle 15](#_Toc362346519)

[3.4 Totalflow Android Application Modules 17](#_Toc362346520)

[3.4.1 Session Manager 18](#_Toc362346521)

[3.4.2 File Manager 20](#_Toc362346522)

[3.5 MCCU Bluetooth Lifecycle (*TRS2*) 26](#_Toc362346523)

[3.6 MCCU IP Lifecycle (*TRS2*) 32](#_Toc362346524)

[3.7 Design Aspects for Application Performance 38](#_Toc362346525)

[3.7.1 Page Size and Corresponding Graphics 38](#_Toc362346526)

[3.7.2 Number of Threads 39](#_Toc362346527)

[3.7.3 Memory Usage 40](#_Toc362346528)

[3.7.4 Recovery from crash/hang 41](#_Toc362346529)

[3.7.6 Indicative/Responsive UI 41](#_Toc362346530)

[3.8 Register Request Protocol (TRS4) 42](#_Toc362346531)

[3.9 Register Grouping 44](#_Toc362346532)

[3.9.1 Tube/Tank Data Selection 45](#_Toc362346533)

[3.9.2 General Data 47](#_Toc362346534)

[3.9.3 Active Alarms 50](#_Toc362346535)

[3.10 Error Codes 54](#_Toc362346538)

[3.11 Help 55](#_Toc362346539)

[3.12 User Editable Fields 56](#_Toc362346540)

[3.12.1 View Options 56](#_Toc362346541)

[3.14 Formatting (TRS5) 57](#_Toc362346542)

[3.14.1 Fonts 58](#_Toc362346543)

[3.14.2 Error Scenario Handling 58](#_Toc362346544)

[3.14.3 Res Folder 58](#_Toc362346545)

[3.14.4 Action Bar Overflow 58](#_Toc362346546)

[3.14.5 Tab Navigation 58](#_Toc362346547)

[3.14.6 Expandable List View 59](#_Toc362346548)

[3.14.7 Graphs 59](#_Toc362346549)

NOTE

The Document Author has to remember the Gate 3 is the Agreement on Product Design. This is the most important milestone, as once this gate is passed the project moves out of into coding and testing phases. A comprehensive document must be produced, outlining the technical design.

1. Introduction

This document provides details over the technical architecture and design for implementing the ABB Totalflow Android application, MCCU. It baselines the sub-systems and implementation approach for the UI and the middleware layers which is followed during the application development.

1.1 Intended Audience

This document is targeted at team management, application developers, and testers involved in implementing the MCCU application.

1.2 References

1. ABB\_NGHLA\_Layout\_Specs\_220812.pdf
2. MRS – VSOMM with Blaine.docx
3. TRS.doc
4. <http://developer.android.com/guide/components/index.html>

1.3 Acronyms and Abbreviations

| **Term** | **Definition** |
| --- | --- |
| AGA-3 | American Gas Association Report No. 3, Orifice Metering of Natural Gas. Method for calculating gas volume across an Orifice Plate. This method requires low pressure readings, Differential Pressure (DP) and Static Pressure (SP). |
| AGA-7 | American Gas Association Report No. 7, Measurement of Gas by Turbine Meters. Method for calculating gas volume using a Pulse Meter. This method requires one pressure reading, Static Pressure (SP). |
| AJAX | Asynchronous JavaScript And XML |
| API | Application Programming Interface |
| CSS | Cascading Style Sheet |
| DES | Data Encryption Standard |
| HTML | Hyper Text Markup Language |
| HTTP | Hyper Text Transfer Protocol |
| HTTPS | Hyper Text Transfer Protocol Secure |
| IP | Internet Protocol |
| JS | Java Script |
| LAN | Local Area Network |
| MCCU | Mobile Collection Unit |
| MCCU | Mobile Collection Calibration Unit |
| MRS | Marketing Requirements Specification |
| RBAC | Role Based Access Control |
| SSL | Secure Socket Layer |
| SPP | Serial Port Protocol |
| TCP | Transmission Control Protocol |
| TLS | Transport Layer Security |
| TRS | Technical Requirements Specification |
| UI | User Interface |
| UUID | Universally Unique Identifier |

1.4 Assumptions

1. Target screens for the current scope have been based on the marketing requirement document (MRS – VSOMM with Blaine.docx) and the corresponding NGHLA Website. Any further screen modifications, addition, and deletion will need fresh analysis and review from both effort and schedule perspective.
2. All the Remote Access development and testing will consider the Android devices (and their versions) as documented in the requirement document.
3. It is assumed that no language other than English is supported for the Android application.
4. All button functionality from the Android device will perform consistently throughout the application.
5. The MCCU application will have method for exporting data to a file that can be shared externally.
6. MCCU app is only going to read the following tubes in the G4:

* AGA3
* AGA7
* AGA3SU
* AGA7SU
* APILIQUIDSU
* VCONE
* VCONESU
* CORIOLUSSU
* NOZZLE
* NOZZLESU

1. Backwards compatible for old files will be available in new app versions.
2. Devices that have previously been connected to will be saved.
3. MCCU shall be able to write back to registers on G4 Device.
4. Shall have help page that is keyword searchable and explains app functionality.
5. MCCU has read and write ability for additional Automation apps:

* Gas Lift Application
* Plunger Lift Application
* Valve Control Application
* Pulse Accumulator Application (flow rate, today vol., y’day vol.)

1. MCCU has read and write ability for other apps:

* IO Simulation
* XMV Interface

2. System Design

2.1 System Overview

This section captures the overall ABB Totalflow Android application context and associated entities based on the marketing requirement specifications listed in the document *MRS – VSOMM with Blaine.docx*. ABB Android application is developed to enable users to browse, monitor, calibrate, and share various trends and statistical values gathered by the Totalflow device.

Scope of Remote Access MCCU development work includes:

1. Development of MCCU application.
2. Development of Adaptation layer in which the application uses Bluetooth to communicate with G4 device.
3. Development of Adaptation layer in which the application uses TCP/IP to communicate with G4 device.
4. Development of Application pages for test automation.

Key Activities involved in the Remote Access MCCU implementation are described in subsequent sub sections.

2.1.1 Development of MCCU Android Application

C application code is running on the android OS which sends data request to the G4 device over Bluetooth and Network connection. The wireless connection runs as a separate thread(s) from the Android device to the G4 device containing existing registers and the trends that are hosted on the flash file system. The existing RBAC is not used for this communication.

Remote Access functionality of MCCU is developed as set of login pages. These login pages will be deployed on the Android device and will consist of java language, XML, Bluetooth SPP, and IP.

2.1.2 Development of Adaptation layer

The Adaptation layer tailors the use of Java Sockets for Bluetooth and Transmission Control Protocol (TCP/IP). As long as the Totalflow devices are running DB2 register protocol, the Adaptation layer should not be impacted.

2.1.3 Design for application page development for test automation

Application page development to support multiple Android OS versions and application page testing is automated to successfully run on various supported Android devices. Look & Feel (minor differences might be there) such as UI elements, layouts, and functionality of the page should be consistent across all the Android devices so that no functionality gets blocked or the layout gets disturbed or look misaligned.

2.1.4 Totalflow.exe (Flash)

* The flash code is enhanced to support the Application Register and File Access methods. These functions include the getting and storing of Application registers.
* Provide support for the user creation/deletion/password changing, these are used to access Remote pages.
* Changes related to implementing the retrieval of the Alarm log, Current alarms and unacknowledged Alarms.
* Changes related to implementing the retrieval of the Data units.

## 2.2 Deployment Diagram









## 2.3 Solution Architecture (TRS3)

Totalflow MCCU solution is developed as an Android application that is deployed onto the Project Wiki (https://code.google.com/p/abb-interns-2013/wiki/mainpage?tm=6). This application is accessible to end users from the Android device. The access to the G4 device data is expected to be from physically co-located Android device; however, the data can be saved and accessed again for viewing at a later time.

Remote Access enables users to monitor, change, and store configuration and statistical data from the Totalflow device.

MCCU Remote Access interfaces with other applications in Totalflow G4 device to get and update the content and data associated with all the data screens.

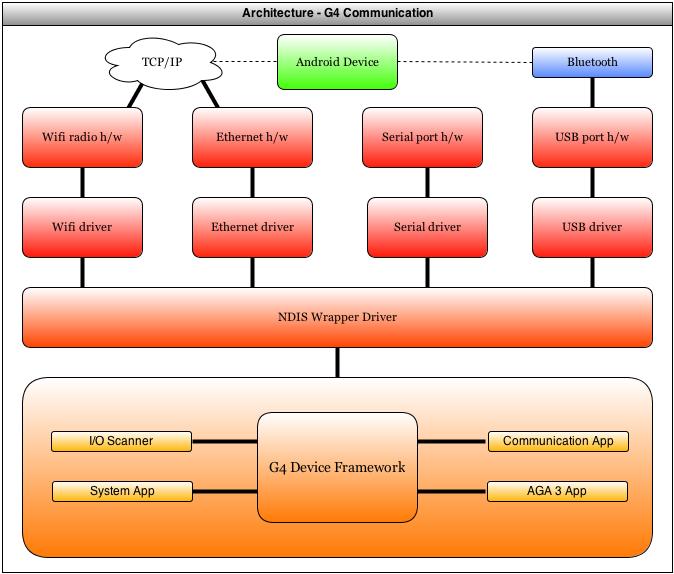


Figure 1 Solution Architecture – G4 Device

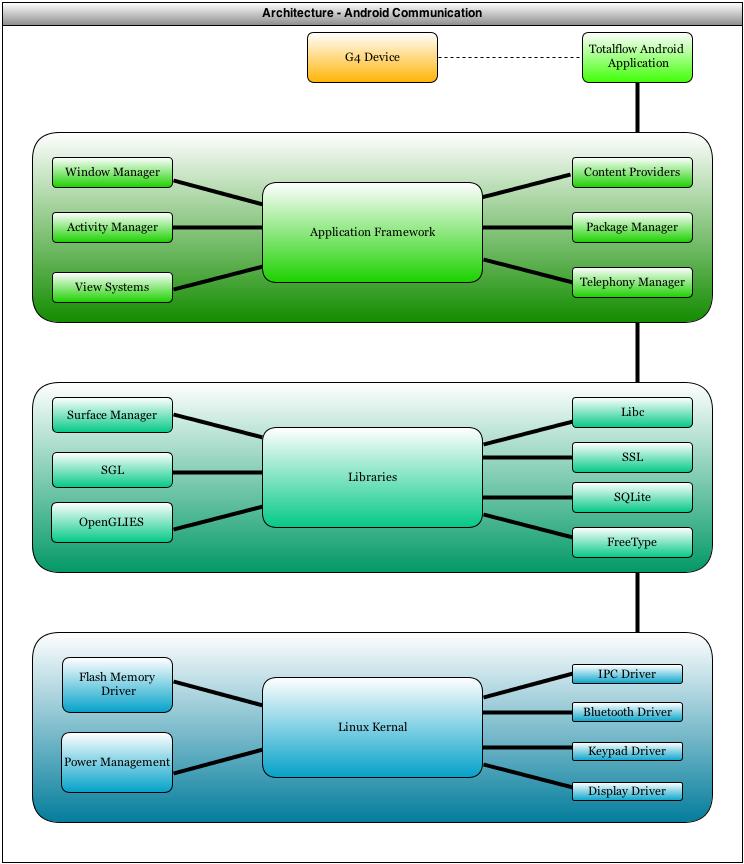


Figure 2 Solution Architecture – Android Device

## 2.4 UI Information Model (TRS4, TRS5, TRS6)

The MCCU app UI can be divided into different sets of pages:

* Home
* Wi-Fi Connection
* Bluetooth Connection
* File Explorer
* Data View
* Help
* Settings

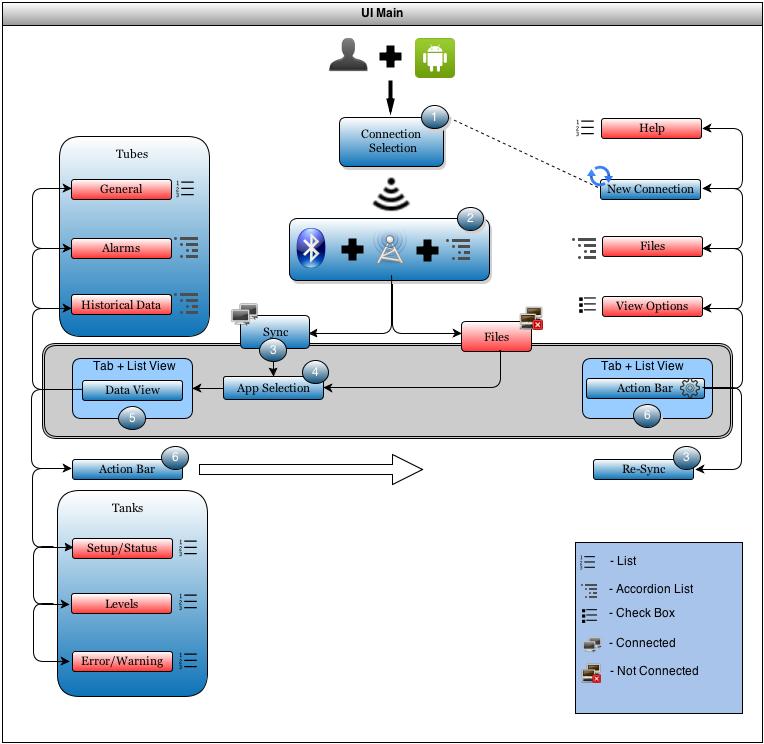


Figure 3 Login, Home Screen

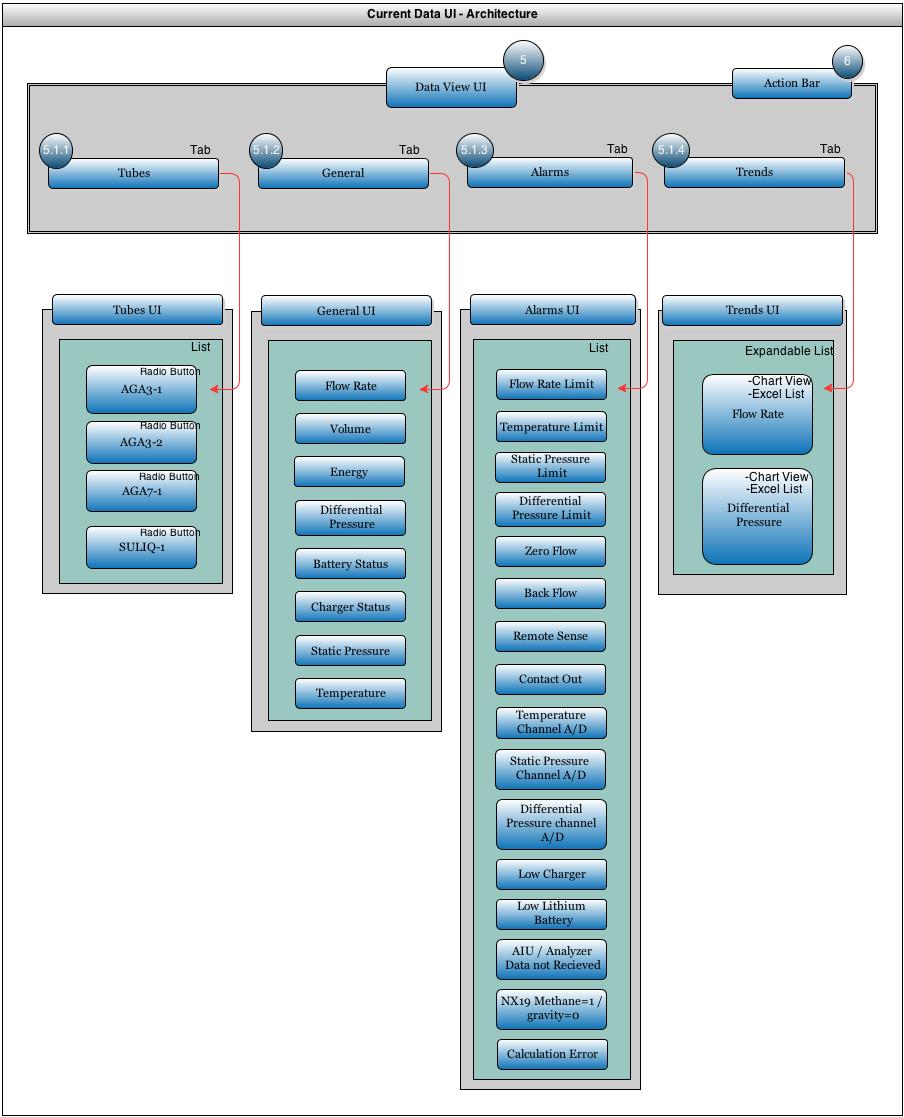


Figure 4 Data View UI

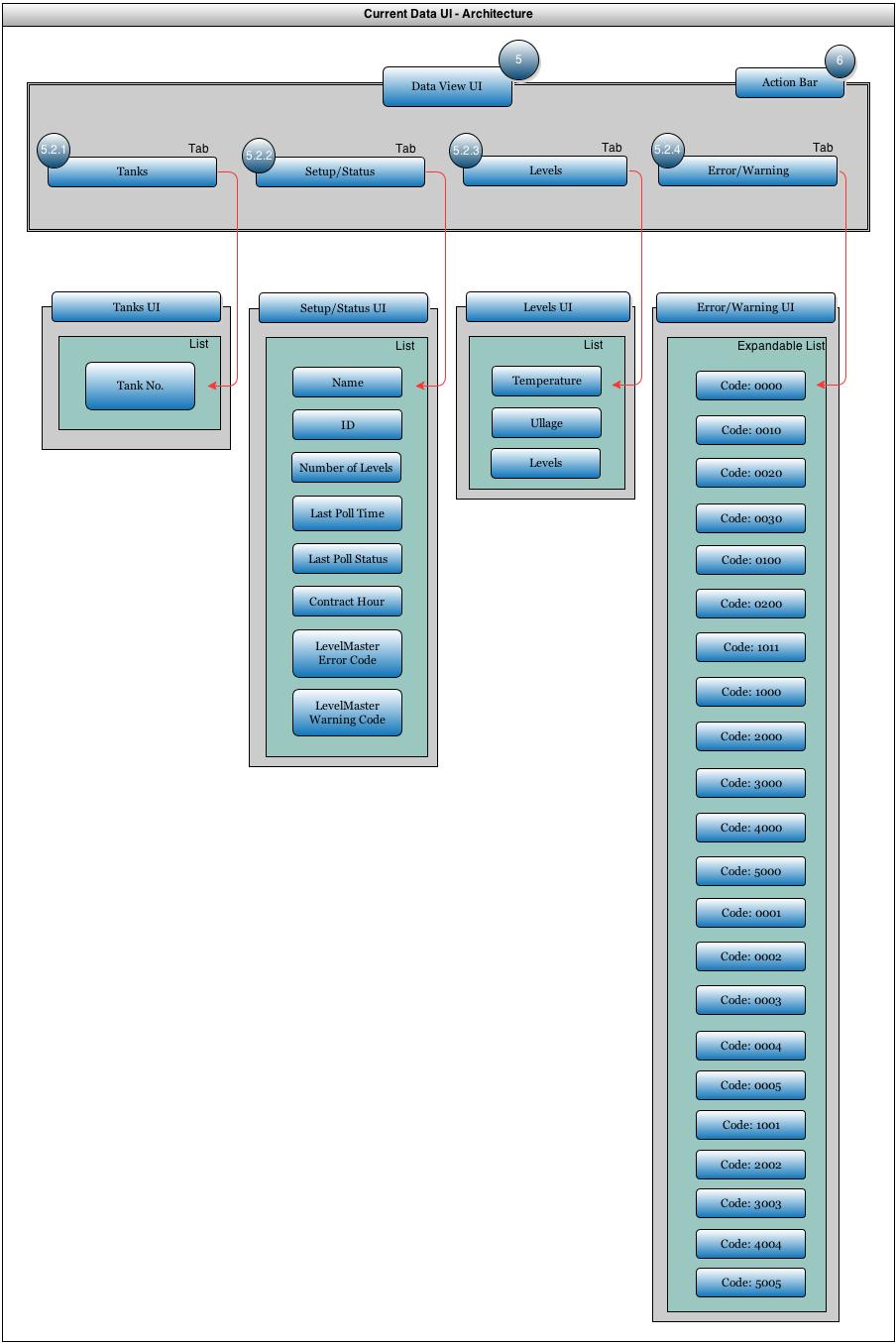


Figure 5 Data View UI (2)

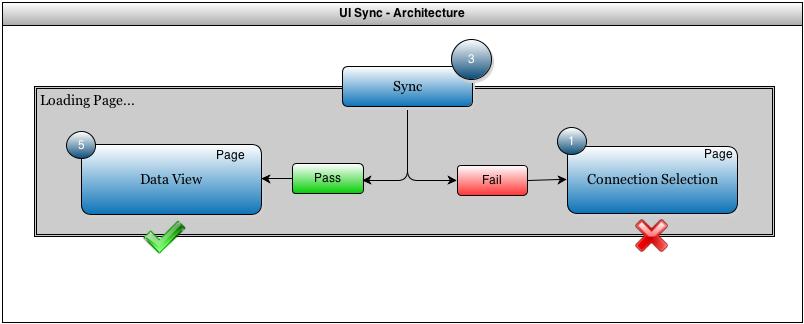


Figure 6 Sync UI

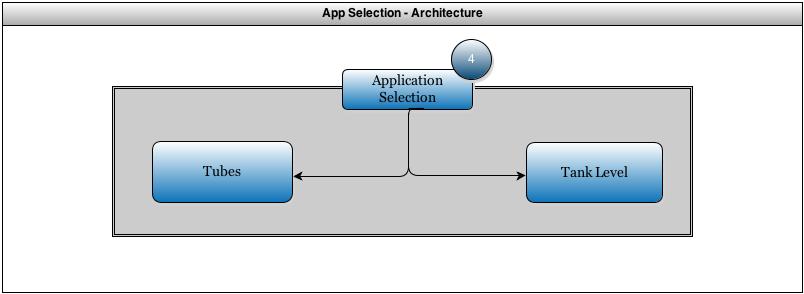


Figure 7 Application Selection UI

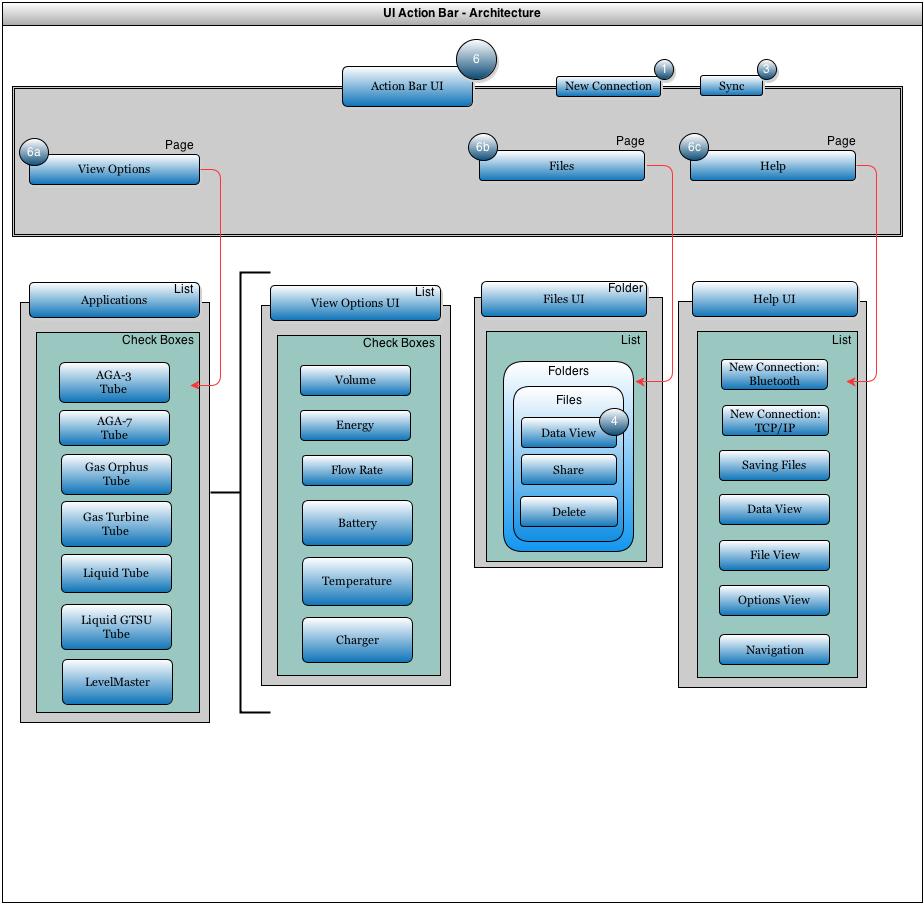


Figure 8 Action Bar Overflow UI

## 2.5 UI Screens Summary

## 2.5.1 Phase 1 Screens

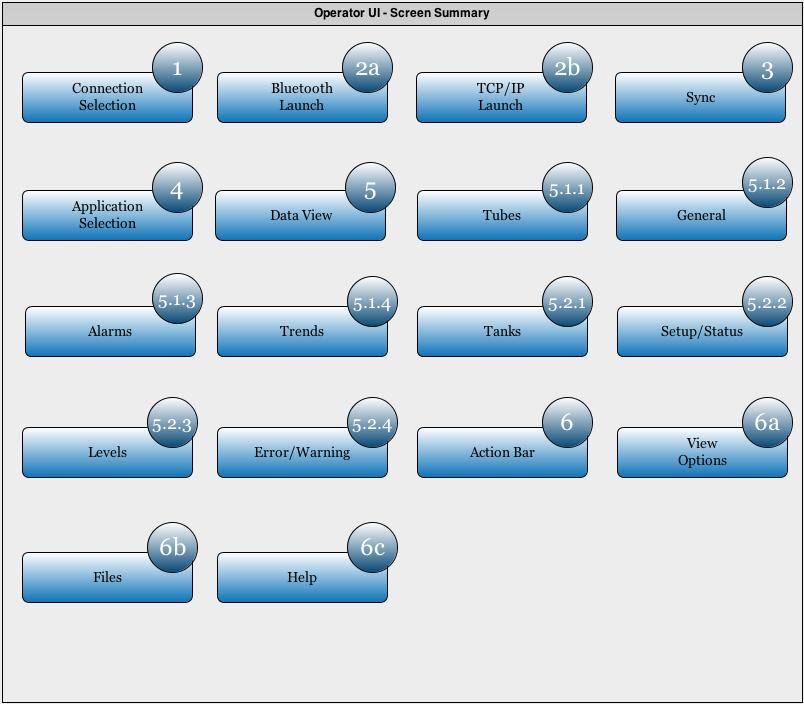


Figure 9 Remote Access UI Pages Screen Summary

3. Totalflow Android Application Design

3.1 Application Design Overview

MCCU is an application running under the context of Totalflow process, which is responsible for providing similar functionality to NGHLA Web software. This application is responsible for reading from and writing to different registers that contain current alarms and current/historical flow measurement (*TRS3*). The data is then stored on the Android’s external memory for accessing at a later time; these files shall be backward compatible to all versions of the MCCU application (*TRS1*). The MCCU application connects to the G4 devices via Bluetooth Serial Port Protocol as well as the option of TCP/IP if the G4 device sits on a network connection. After making a Bluetooth or IP connection, all settings from that connection will be saved for that specific device (*TRS2*). To help the user run the MCCU application, there shall be a task-oriented help page that provides keyword searchable results (*TRS4, TRS5*).

Scope of Design development work includes:

1. Basic Android Application Activity Lifecycle
2. Basic Android Application Indicators
3. Development of MCCU Modules.
4. Development of Class structure for Bluetooth to communicate with G4 device.
5. Development of Class structure for TCP/IP to communicate with G4 device.
6. Design for the MCCU App Array Register Access methods (Alarms and Applications).
7. Design for the MCCU File Access methods (Internal files and folders).
8. Design for the Application Page Templates.

## 3.2 Android Activity Lifecycle

The Android activity lifecycle allows for activity functionality throughout the various functions within pages of the applications.

* **Activity Launched** calls creating and starting the activity. At this point the user has selected which function they wish to use on the Application.
* **Activity Running** used after the user has selected the activity. The activity is running in the foreground and may be paused, then sent back to the start by the user if he/she shall decide to change pages and return to the activity running. There are multiple redundant paths for the user to stop the application and restart the activity. If the user decides to change or stop their activity, then the activity will be routed to **App Process Killed** and the user will then be able to create a new process.
* **App Process Killed** allows the user to begin a new activity or start the same activity from the beginning. There will only be one process running at a time, when it is killed; it is then removed from the phone’s volatile memory.
* **Activity Shut Down** is reached after the process has stopped and the user decides to close the Android application. The MCCU application only has one process running at a time, so any time they navigate pages in the app it is never destroyed and shutdown until they completely close out of the application.



Figure 10 MCCU Application Lifecycle

**3.3 Android Activity Indicators**

In the case that the Android device will reset the Bluetooth or Wi-Fi status from off/on, the indicator is shown to the user that reflects the ongoing processing by the Android application in the backend. Once the connection setting is turned on, and then the status will be displayed at the top bar of the screen

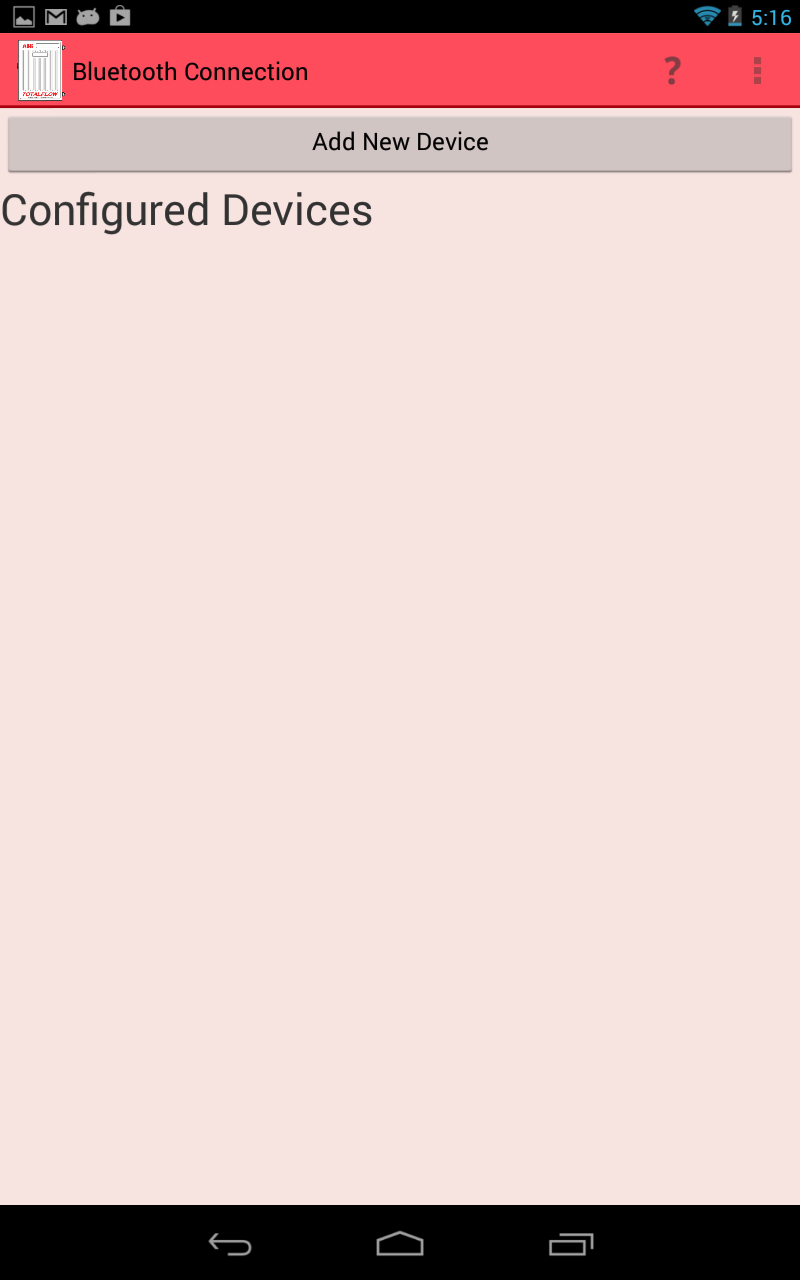
****

Figure 11 Screenshot with Bluetooth and Wi-Fi status at top right

## 3.4 MCCU Application Modules

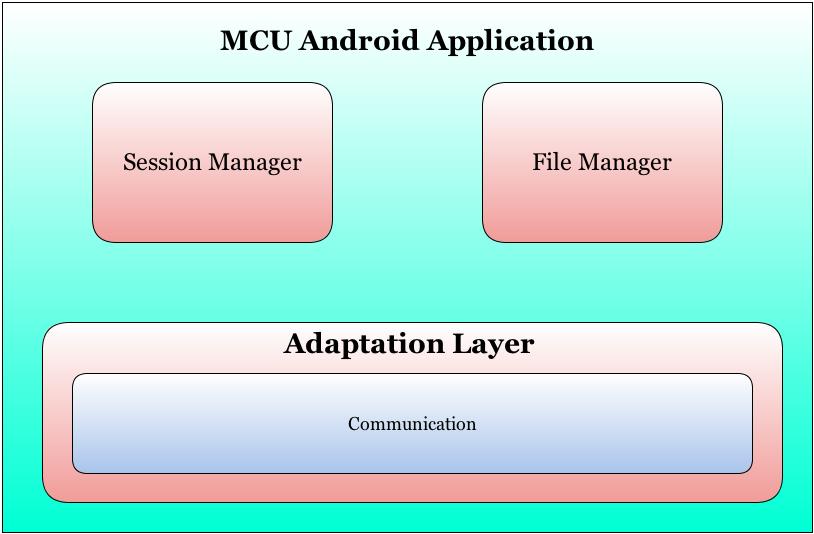


Figure 12 MCCU Application Modules

Following are the modules in MCCU Application

* Session Manager
  + Session Manager is responsible for guiding navigation throughout application pages.
* File Manager
  + File manager is responsible for providing user access to the Android device’s external storage.
* Adaptation Layer

This communication layer acts as a broker between the user interface and Android application framework and vice-versa.

* Starting the Remote connection
* Sending data requests
* Receiving data
* Ending the Remote connection

## 3.4.1 Session Manager

Upon establishing connection to a G4 device (Application Page 2a or 2b), the Navigator is configured to maintain up to one user session. The user session is limited to the duration that the user chooses to navigate within the Android Application until the application is closed. The user will be required to enter their username and password to connect to the device and navigate through the data pages. The login screens for Bluetooth and TCP/IP are mentioned in sections 3.3 and 3.4.

Navigator (section 3.5) is a class containing a list that keeps track of pages that have been accessed by the user. These pages are being cached within the Android device and accessed upon request when the user presses the back button. When the user inputs their name and password, the device retrieves the file, and then the user goes to the page to view the file data. At this point the navigator tracks the page previously visited. While the user is navigating through the application pages, the Action Bar Overflow menu and the data views are kept static within the Navigator. Any other navigation through the application is a linear process.

*public class Navigator {*

**navIndex indicates the current page that the user is on**

*private static int navIndex = 0;*

*private static ArrayList<String> prevPages = new ArrayList<String>();*

**Returns the previous page that the user visited**

public static String getPrevPageName() {

if (navIndex == 0) {

return null;

} else {

decrementNav();

String temp = prevPages.get(navIndex);

prevPages.remove(navIndex);

return temp;

}

}

When a new page is accessed, a new entry is made in this list. If a request comes for login at the time when there are already one active user, in that case request is refused and the user is shown an error message and taken back to page-1.

The **Action Bar Overflow** is another tool granting the user the maximum ease for application navigation. Any possible features that may need to be accessed at any point in the user session can be found in the Action Bar Overflow. The Action Bar Overflow is located in the upper right corner of the screen for new Android devices. The Action Bar Overflow is a physical touch button located below the screen for older Android devices. The Action Bar Overflow is represented by three dots inside of a square. The ease of use adds to the applications efficiency.

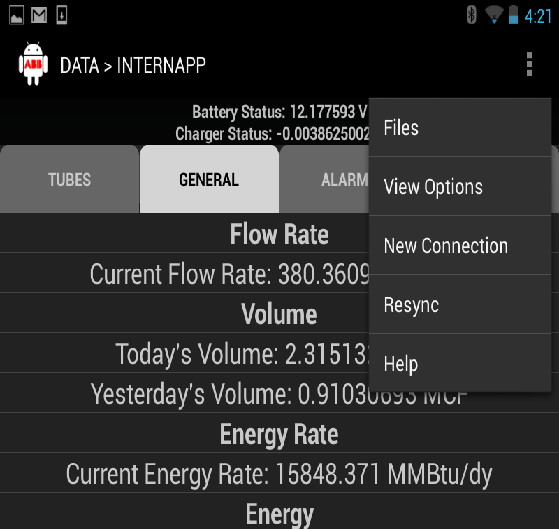
****

Figure 13 Screenshot of upper right corner where the Action Bar Overflow Tab is located

*Action Bar Overflow used throughout all of the classes:*

**// when user clicks Action Bar Overflow (Menu) Icon show the navigation menu**

*@Override*

*public boolean onCreateOptionsMenu(Menu menu) {*

**// TODO Auto-generated method stub**

*MenuInflater inflater = getMenuInflater();*

*inflater.inflate(R.menu.navigation, menu);*

*return true;*

*}*

## *Action Bar Overflow Created in the NewConnection.java class:*

public boolean onOptionsItemSelected(MenuItem item) {

switch (item.getItemId()) {

case R.id.menuFiles:

**// First, add THIS page to the navigator**

Navigator.addPrevPage(this.getClass().getName());

**// go to File explorer**

Bundle basket = new Bundle();

basket.putString("fileLevelKey", "");

Intent i2 = new Intent("com.example.g4\_MCCUv1.FILEVIEW");

i2.putExtras(basket);

startActivity(i2);

finish();

return true;

case R.id.menuViewOptions:

**// First, add THIS page to the navigator**

Navigator.addPrevPage(this.getClass().getName());

**// go to View Options page**

Intent i = new Intent("com.example.g4\_MCCUv1.VIEWOPTIONS");

startActivity(i);

finish();

return true;

case R.id.menuNewConnection:

**// do nothing**

return true;

case R.id.menuHelp:

**// First, add THIS page to the navigator**

Navigator.addPrevPage(this.getClass().getName());

**// Go to help page**

Intent intent = new Intent(NewConnection.this, Help.class);

startActivity(intent);

finish();

return true;

default:

return super.onOptionsItemSelected(item);

}

}

## 3.4.2 File Manager

MCCU has the option of saving and sharing files when the user does a long-click on a file name. The user is prompted by a pop-up message giving them the option to share, delete, view, or rename.

*Creating buttons in the FileView.java class for the different pop-up actions the user can take:*

Button bShare = (Button) optionsDialog.findViewById(R.id.bDialogShare);

Button bDelete = (Button) optionsDialog.findViewById(R.id.bDialogDelete);

Button bView = (Button) optionsDialog.findViewById(R.id.bDialogView);

Button bRename = (Button) optionsDialog.findViewById(R.id.bDialogRename);

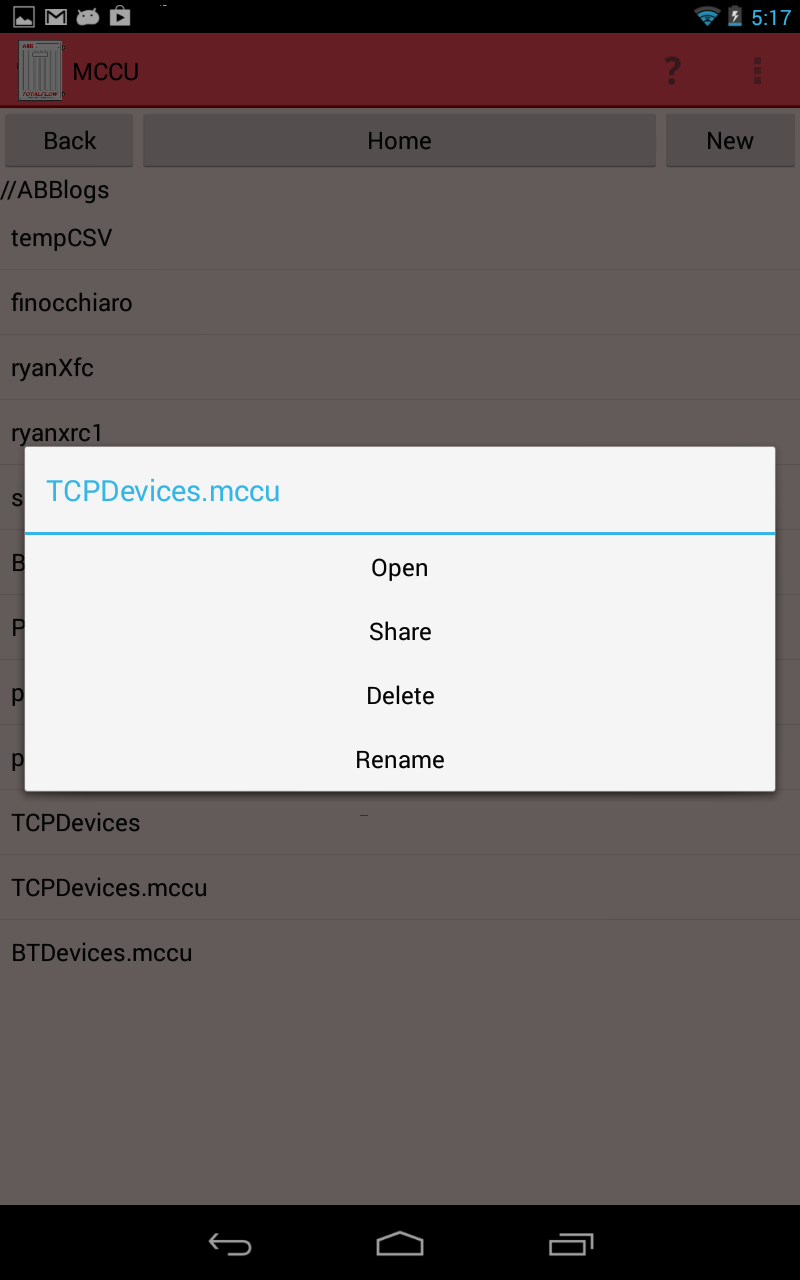
****

Figure 14 Pop-up options for managing files

**Share**

MCCU creates a set of .CSV files to share from the Android device based off of the data within the clicked file. A pop-up selection box then gives the user different options to share the .CSV file based off of the applications the user has installed on their Android device. The file is then temporarily stored on the Android’s external storage. Once the user either sends or cancels the “share” option, the file is then deleted from external storage.

*Creating the file, in the FileView.java class, to be shared:*

**// action listener for the "Share" function**

bShare.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

**//will hold all of the files that will be sent**

shareFilesList = new ArrayList<Uri>();

**//if fileLevel is empty then the user is viewing sub folders, folders cannot be formatted, error is displayed**

if (fileLevel.equals("")) {

Toast toast = Toast.makeText(context,

"Cannot send entire folder, please select file.", Toast.LENGTH\_SHORT);

toast.setGravity(Gravity.CENTER, 0, 0);

toast.show();

} else {

**//convert the file's contents from bytes to meaningful data**

convertBytes(readFile(direct + "/" + fileName));

**//uses android's Intent for sending**

Intent sendIntent = new Intent(Intent.ACTION\_SEND\_MULTIPLE);

**//for every application set of data within** allAppData

for (int i = 1; i < allAppData.size(); i++) {

chosenTubeTank = allAppData.get(i);

**//format to csv, create csv file, add its path to list of files being sent**

sendFile();

shareFilesList.add(Uri.parse(shareFile.getAbsolutePath()));

}

**//set type so intent can filter applications** that are capable of sharing .csv files

sendIntent.setType("text/csv");

**//add list of files**

sendIntent.putExtra(Intent.EXTRA\_STREAM, shareFilesList);

sendIntent.addFlags(Intent.FLAG\_GRANT\_READ\_URI\_PERMISSION);

**//display intent, with a result method**

startActivityForResult(sendIntent, 1);

}

optionsDialog.dismiss();

}

});

**Delete**

MCCU has the option to delete any file or folder within the main directory *ABBlogs*. When the user chooses to delete a new pop-up displays asking the user if they wish to confirm or cancel the deletion. If confirm is pressed, the contents are deleted. If cancel is pressed, they are brought back to the File Explorer (page-3). Files may also be deleted manually by mounting the Android device to the computer.

*Deleting any file or folder within the main directory:*

**//action listener for "Delete" function**

bDelete.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

**//Sets up another dialog that will ask the user if they are sure they want to delete the file or sub folder**

final Dialog deleteDialog = new Dialog(context);

deleteDialog.setContentView(R.layout.fileviewdeletedialog);

deleteDialog.setTitle("Confirm Delete");

Button bDeleteYes = (Button) deleteDialog.findViewById(R.id.bDeleteDialogYes);

Button bDeleteNo = (Button) deleteDialog.findViewById(R.id.bDeleteDialogNo);

deleteDialog.show();

**//action listener for "Confirm" function**

bDeleteYes.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

**//get the path to the item selected**

File toDelete = new File(direct.getAbsolutePath() + "/" + fileName);

**//delete the file, if the path leads to a sub folder the delete function will not work and return**

if (toDelete.delete() == false) {

**//in that case all files within the sub folder must be deleted first, then the sub folder can be deleted**

String filesToDelete[] = toDelete.list();

for (int i = 0; i < filesToDelete.length; i++) {

new File(toDelete.getAbsolutePath() + "/" + filesToDelete[i]).delete();

toDelete.delete();

}

}

**// updates the list of sub folders or files that were currently being displayed to reflect the changes from deletion**

folderArrList.clear();

File[] fileArr = direct.listFiles();

for (int i = 0; i < fileArr.length; i++) {

folderArrList.add(fileArr[i].getName());

}

lvFolderList.requestLayout();

adapterFiles.notifyDataSetChanged();

deleteDialog.dismiss();

optionsDialog.dismiss();

}

});

//**action listener for "Cancel" function**

bDeleteNo.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

**// do nothing, close all dialogs**

deleteDialog.dismiss();

optionsDialog.dismiss();

}

});

}

});

**View**

MCCU takes a file name from a list of files and launches to Appview so that the data can be displayed. If the File Explorer is on a list of folders, the view will take that folder name and re-seed it to File Explorer to display its files.

*Setting up the view from file list or folder list:*

**//action listener for "View" function**

bView.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

**/\* If fileLevel is empty, the user if viewing sub folders**

**\* so the item clicked is a sub folder that needs to be opened within FileView**

**\*/**

if (fileLevel.equals("")) { **// Go one file level deeper**

Navigator.addPrevPage(this.getClass().getName());

Bundle basket = new Bundle();

basket.putString("fileLevelKey", fileName);

Intent intent = new Intent(FileView.this, FileView.class);

intent.putExtras(basket);

startActivity(intent);

finish();

} else { **//The fileLevel is a sub folder so the user is viewing files. That file path needs to**

**// be sent to DataView to be displayed.**

Bundle basket = new Bundle();

basket.putString("FileNameKey", direct.getAbsolutePath() + "/" + fileName);

Navigator.setAppViewExtras(direct.getAbsolutePath() + "/" + fileName);

Intent intent = new Intent(FileView.this, AppView.class);

intent.putExtras(basket);

startActivity(intent);

finish();

}

optionsDialog.dismiss();

}

});

**Rename**

MCCU opens a new dialogue with a text box for the user to enter a new name. The dialogue also contains the confirm and cancel button. If the user presses cancel, they are brought back to the File Explorer page. If the user presses confirm, the entered file name is checked against a list of standard naming conventions. If the file name passes, the name is changed and the File Explorer is updated to display this change. If the name does not pass, a toast of the specific error is displayed and the user can try again.

**//action listener for "Rename" function**

bRename.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

**//open another dialog that will ask the user to input the new name and**

**// ask if they want to confirm or cancel the change**

final Dialog renameDialog = new Dialog(context);

renameDialog.setContentView(R.layout.filerenamedialog);

renameDialog.setTitle("Rename File");

Button bSave = (Button) renameDialog.findViewById(R.id.bRenameDialogYes);

Button bCancel = (Button) renameDialog.findViewById(R.id.bRenameDialogNo);

final EditText etNewName = (EditText) renameDialog.findViewById(R.id.etNewFileName);

**//set the EditText box to the current file name**

setNewName.setText(fileName);

renameDialog.show();

**//action listener for "Confirm" function**

bSave.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

**//get the path to the original file name**

File original = new File(direct.getAbsolutePath() + "/" + fileName);

**//get the file name entered by the user**

String newFileName = etNewName.getText().toString().trim();

**//if the user did not alter the original file name then nothing needs to be done**

if (newFileName.equals("fileName")) {

}

**// The following cases check if the name entered follows proper naming guidelines**

**// if it does not, the error is displayed, and the name is not saved**

else if (newFileName.equals("")) {

Toast toast = Toast.makeText(context, "No name entered.", Toast.LENGTH\_SHORT);

toast.setGravity(Gravity.CENTER, 0, 0);

toast.show();

} else if (newFileName.contains("<") || newFileName.contains(">")

|| newFileName.contains(":") || newFileName.contains("\"")

|| newFileName.contains("\\") || newFileName.contains("/")

|| newFileName.contains("|") || newFileName.contains("?")

|| newFileName.contains("\*")) {

Toast toast = Toast.makeText(context,

"File name cannot contain special characters:\n< > : \" \\ / | ? \*",Toast.LENGTH\_SHORT);

toast.setGravity(Gravity.CENTER, 0, 0);

toast.show();

} else if (newFileName.charAt(0) == '.') {

Toast toast = Toast.makeText(context, "File name cannot begin with '.'",Toast.LENGTH\_SHORT);

toast.setGravity(Gravity.CENTER, 0, 0);

toast.show();

} else if (folderArrList.contains(newFileName)) {

Toast toast = Toast.makeText(context, "File already exists.",Toast.LENGTH\_SHORT);

toast.setGravity(Gravity.CENTER, 0, 0);

toast.show();

} else if(newFileName.length()>201) {

Toast toast = Toast.makeText(context, "File name length too long. 200 char max.",Toast.LENGTH\_SHORT);

toast.setGravity(Gravity.CENTER, 0, 0);

toast.show();

}

**// if the name passes**

else {

**//a new file is created under that name**

File destination = new File(direct.getAbsoluteFile() + "/" + etNewName.getText().toString().trim());

**// the original file is renamed to the new name**

original.renameTo(destination);

**// updates the list of sub folders or files that were currently being displayed to reflect the changes from renaming**

folderArrList.clear();

File[] fileArr = direct.listFiles();

for (int i = 0; i < fileArr.length; i++) {

folderArrList.add(fileArr[i].getName());

}

lvFolderList.requestLayout();

adapterFiles.notifyDataSetChanged();

renameDialog.dismiss();

optionsDialog.dismiss();

}

}

});

//**action listener for "Cancel" function**

bCancel.setOnClickListener(new OnClickListener() {

public void onClick(View v) {

**// do nothing, close all dialogs**

renameDialog.dismiss();

optionsDialog.dismiss();

}

});

}

});

## 3.5 MCCU Bluetooth Lifecycle

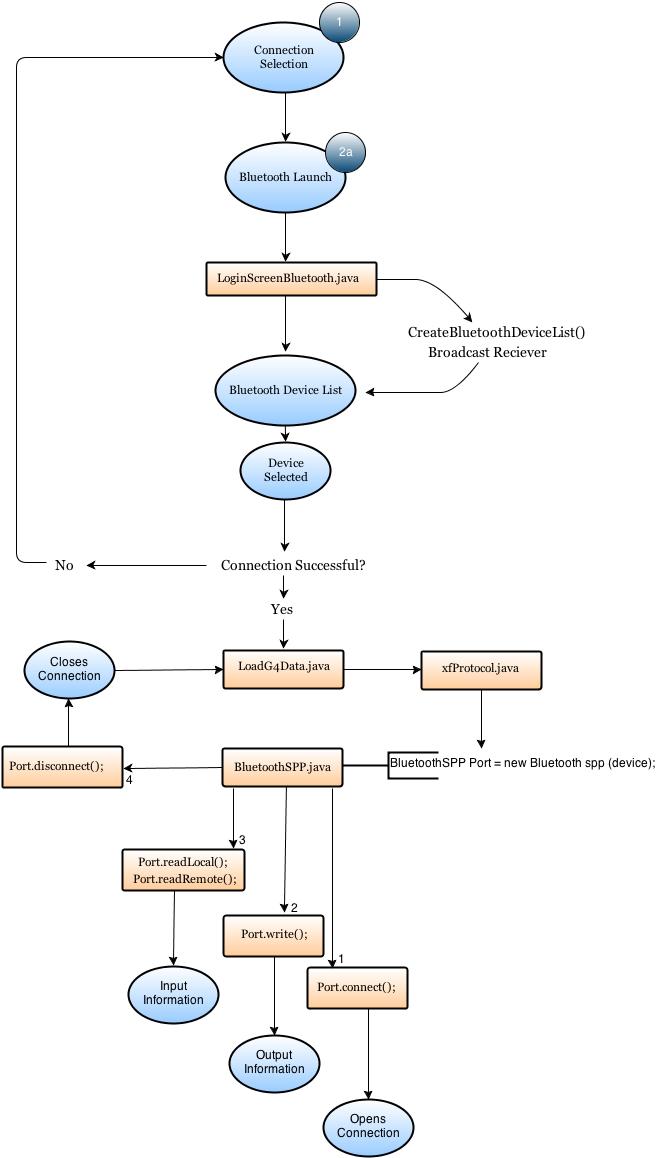


Figure 15 Bluetooth Application Lifecycle

The MCCU uses Bluetooth SPP to support connection and data request from the Totalflow G4 device (page 2a of the UI). After the user opens the Android application and chooses to connect to the G4 device using Bluetooth, the Application then checks to see if the Bluetooth setting is turned on within the Android by calling *checkBTStatus();* within ***LoginScreenBluetooth.java*** :

*checkBTStatus();*

After Bluetooth has been turned on the device then scans and populates a list of Bluetooth devices within distance. This is done so by creating the Bluetooth device list:

*showLoginScreen();*

*private void createBluetoothDeviceList()*

The Bluetooth device list then goes through and adds the devices and their address to a viewable list for the user to pick from (figure 16). When it finds a device, the MAC address and name of device will be displayed. All code coming from *LoginScreenBluetooth.java* class:

**private** **void** createBluetoothDeviceList() {

mReceiver = **new** BroadcastReceiver() {

**public** **void** onReceive(Context context, Intent intent) {

String action = intent.getAction();

**if** (BluetoothDevice.*ACTION\_FOUND*.equals(action)) {

BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.*EXTRA\_DEVICE*);

*bluetoothDevices*.add(device);

sArrBluetoothDevices.add(device.getName() + "\n" + device.getAddress());

**if** (debug) {

tvDeviceInfo.append(device.getName() + "\n" + device.getAddress() + "\n");

}

lvBTDeviceList.requestLayout();

}

}

};

After the connection list has been established, the application then jumps into ***LoadG4Data.java*** to begin the requests needed for data sets from different registers. Every Bluetooth socket connection requires a UUID from the “host” to the “client” used. The UUID in this application signifies. ***LoadG4Data.java*** calls on functions within ***xfProtocol.java*** to access the Bluetooth Serial Port Protocol. This information requested along will be obtained after the port connects via Bluetooth. The sockets created in ***BluetoothSPP.java*** use the following methods to establish the ***Port.connect(),*** the ***Port.write(),*** ***Port.readLocal(),*** ***Port.readRemote(),*** and the ***Port.disconnect().***

***private static final UUID uuid = UUID.fromString("00001101-0000-1000-8000-00805F9B34FB");***

***private BluetoothSocket socket;***

* **Setting up socket based connection to get the byte streams:**

**public** **byte** connect() {

**byte** ret=(**byte**) 255;

**try** {

socket = device.createRfcommSocketToServiceRecord(*uuid*);

socket.connect();

**if**(socket!=**null**) {

**while**(device.getBondState()!=BluetoothDevice.*BOND\_BONDED*) {

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

outStream=socket.getOutputStream();

inStream=socket.getInputStream();

ret=0;

}

} **catch** (IOException e) {

e.printStackTrace();

disconnect();

}

**return** ret;

}

* **Setting up the Port Write to write the bytes to the output stream between the devices:**

**public** **int** Write(**byte**[] departingData, **int** nBytes) {

**int** ret;

**try** {

outStream.write(departingData, 0, nBytes);

outStream.flush();

ret=nBytes;

} **catch** (Exception e) {

disconnect();

e.printStackTrace();

ret=0;

}

**return** ret;

}

* **Setting up the Port Read for Local:**

**public** **int** ReadLocal() {

**int** totalRead=0;

**byte**[] buffer = **new** **byte**[10];

**try** {

Thread.*sleep*(1750);

totalRead= inStream.read(buffer);

returningData=**new** **byte**[totalRead];

**for**(**int** i=0; i<totalRead; i++) {

returningData[i]=buffer[i];

}

}

**catch** (IOException e) {

totalRead=0;

}

**catch** (Exception e) {

totalRead=0;

e.printStackTrace();

}

**return** totalRead;

}

* **Setting up the Port Read for Remote to read in bytes from the input stream between the devices:**

**public** **int** ReadRemote(**int** nBytes) {

**int** index=0;

**int** totalRead=0;

**int** buf=0;

**byte**[] buffer = **new** **byte**[nBytes];

returningData=**new** **byte**[nBytes];

**do** {

**try** {

Thread.*sleep*(200);

buf= inStream.read(buffer);

totalRead+=buf;

**for**(**int** i=0; i<buf; i++) {

returningData[index]=buffer[i];

index++;

}

} **catch** (IOException e) {

disconnect();

totalRead=0;

**break**;

} **catch** (Exception e) {

totalRead=0;

disconnect();

**break**;

}

} **while**(totalRead<nBytes && returningData[20]!=0x01 && returningData[13]==0x00 && totalRead>=0 && returningData[0]==-1);

**return** totalRead;

}

* **Setting up the Port Disconnect to close the socket connection and the streams:**

**public** **byte** disconnect() {

**byte** ret=(**byte**)208;

**try** {

**if**(inStream!=**null**) {

inStream.close();

} **if**(outStream!=**null**) {

outStream.close();

} **if**(socket!=**null** && socket.isConnected()) {

socket.close();

}

ret=0;

} **catch** (IOException e) {

e.printStackTrace();

}

**return** ret;

}

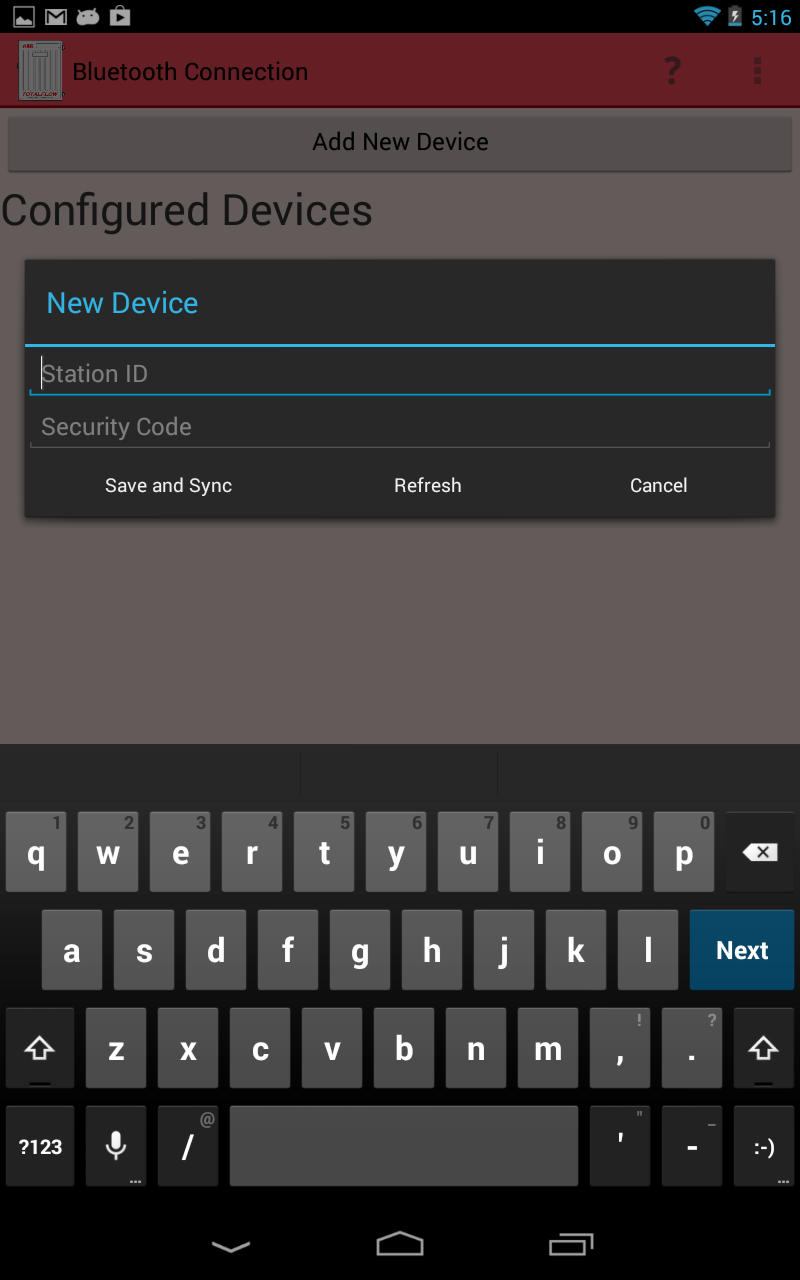


Figure 16 Screenshot of page-2a for Bluetooth log in

## 3.6 MCCU IP Lifecycle

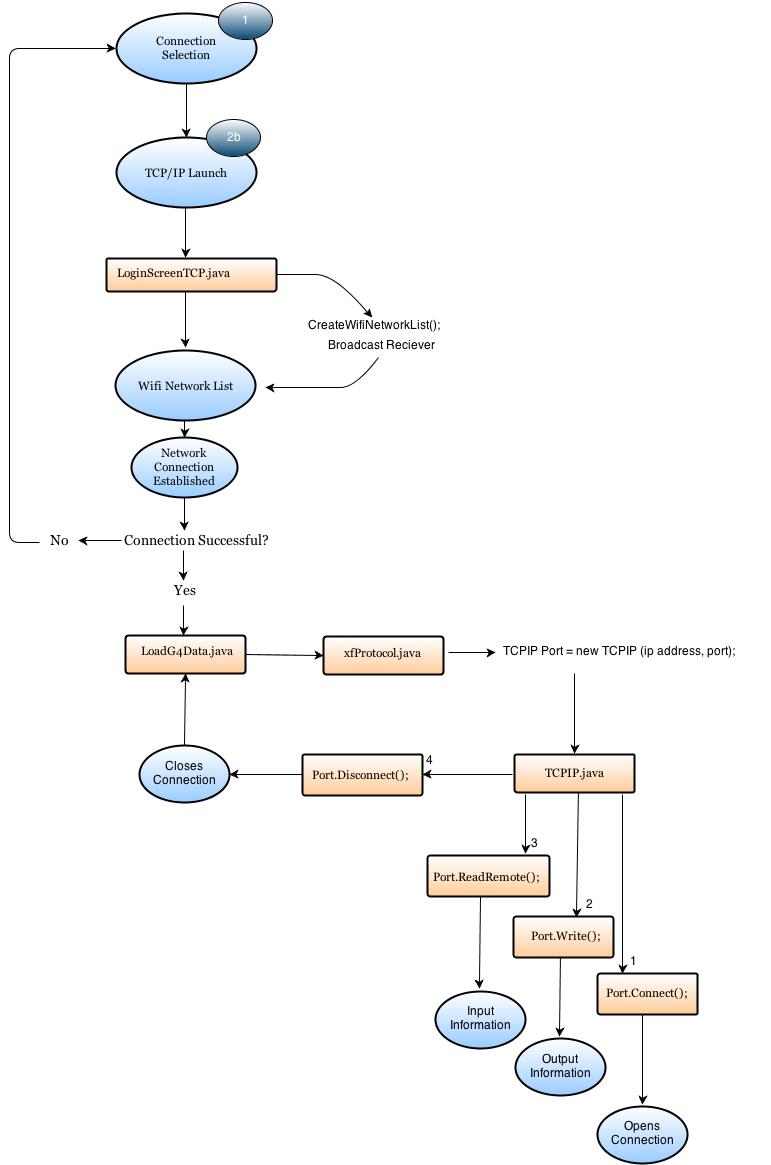


Figure 17 TCP/IP Application Lifecycle

The Android application uses TCP/IP to support connection and data request from the Totalflow G4 device (page 2b of the UI). After the user opens the Android application and chooses to connect to the G4 device using TCP/IP Connection, the Application then checks to see if the Wi-Fi setting is turned on within the Android device by calling *checkWifiStatus();* within ***LoginScreenTCPIP.java***.

*checkWifiStatus();*

After the Wi-Fi status is checked, the application turns it off then back on to refresh the list of wireless networks to connect to. This is done so by creating the Wi-Fi device list:

*showLoginScreen();*

public void showLoginScreen()

**private** **void** createWifiNetworkList() {

wifiArrList.clear();

// add a mobile data option to select

wifiArrList.add("Use Mobile Data");

mReceiver = **new** BroadcastReceiver() {

**public** **void** onReceive(Context c, Intent intent) {

//gets the scan results and copies the SSID's into wifiArrList for the user to see

wifiResults = wifiManage.getScanResults();

**for**(**int** i=0; i<wifiResults.size(); i++) {

//Only display networks with complete information

// specifically SSID so the user can identify the network

**if**(!wifiResults.get(i).SSID.trim().equals("")) {

wifiArrList.add(wifiResults.get(i).SSID);

}

}

//close receiver on finish

unregisterReceiver(mReceiver);

lvTCPList.requestLayout();

}

};

//Set up a receiver for Wi-Fi scans

IntentFilter filter = **new** IntentFilter(WifiManager.*SCAN\_RESULTS\_AVAILABLE\_ACTION*);

registerReceiver(mReceiver, filter);

lvTCPList.requestLayout();

}

The Wi-Fi Network device list then goes through and adds the network name to a viewable list for the user to pick from. When the application finds a device to be selected, the user must enter the station ID, the IP address, and name of device will be returned along with other requested information. If the user cannot connect, the application will throw the error message “Unable to connect to specified network” and the user will be sent back to the login (page-1). All code below is from LoginScreenTCPIP.java class:

**private** **boolean** connectToNetwork(NetworkInfo mWifi, NetworkInfo mMobile) {

**boolean** ret = **false**;

// if mobile data is the network of choice and is connected then return true

// no case for if mobile is disconnected, MCCU cannot activate mobile data

**if** (networkSelectedSSID.equals("Use Mobile Data") && mMobile.isConnected()) {

//turn off wifi since if both mobile and wifi are enable Android will default to wifi connection

wifiManage.setWifiEnabled(**false**);

ret = **true**;

}

// if the user is already connected to the network of choice then return true and do nothing

**else** **if** (mWifi.isConnected() && wifiManage.getConnectionInfo().getSSID() .equals(networkSelectedSSID)) {

ret = **true**;

}

// if the connection must be done manually

**else** {

//make sure wi-fi is disconnected from all networks

wifiManage.disconnect();

// set up configuration

WifiConfiguration conf = **new** WifiConfiguration();

conf.SSID = "\"" + networkSelectedSSID + "\"";

// For WEP network:

**if** (networkSelected.capabilities.contains("WEP")) {

conf.wepKeys[0] = "\"" + etWifiPassword.getText().toString()

+ "\"";

conf.wepTxKeyIndex = 0;

conf.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.*NONE*);

conf.allowedGroupCiphers

.set(WifiConfiguration.GroupCipher.*WEP40*);

}

// For WPA network

**else** **if** (networkSelected.capabilities.contains("WPA")) {

conf.preSharedKey = "\"" + etWifiPassword.getText().toString()

+ "\"";

}

**else** { // For Open network:

conf.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.*NONE*);

}

// add it to Android wifi manager settings:

wifiManage.addNetwork(conf);

// make sure the network was enabled when it was added to the list of configured networks

List<WifiConfiguration> list = wifiManage.getConfiguredNetworks();

**for** (WifiConfiguration i : list) {

**if** (i.SSID != **null** && i.SSID.equals("\"" + networkSelectedSSID + "\"")) {

wifiManage.disconnect();

wifiManage.enableNetwork(i.networkId, **true**);

wifiManage.reconnect();

**break**;

}

}

//give time to start connecting

**try** {

Thread.*sleep*(250);

} **catch** (InterruptedException e) {

}

//update connection info

mWifi = connManager.getNetworkInfo(ConnectivityManager.*TYPE\_WIFI*);

**if** (mWifi.isConnectedOrConnecting()) {

// give time to connect

**while**(!mWifi.isConnected()) {

**try** {

Thread.*sleep*(200);

} **catch** (InterruptedException e) {

}

}

mWifi = connManager.getNetworkInfo(ConnectivityManager.*TYPE\_WIFI*);

}

// if it is connected then return success

**if**(mWifi.isConnected()) {

ret=**true**;

}

}

**if** (ret == **false**) {

wifiManage.disconnect();

}

**return** ret;

}

After the connection list has been created, and the user has connected to the Wifi network, the application then jumps into ***LoadG4Data.java*** to begin the requests needed for data sets from different registers. ***LoadG4Data.java*** calls on functions within ***xfProtocol.java*** to access the Network Internet Protocol. Every Network connection requires a “client” and “host” to establish the sockets. This information requested will be obtained after the port connects. The sockets created in ***TCPIP.java*** use the following methods to establish the ***Port.connect(),*** the ***Port.write(),*** ***Port.readLocal(),*** ***Port.readRemote(),*** and the ***Port.disconnect().***

public TCPIP(String ipAddress, String port)

this.ipAddress=ipAddress;

this.port=Integer.parseInt(port);

* **Setting up socket based connection to get the byte streams:**

**public** **byte** connect() {

**byte** ret=(**byte**)255;

**try** {

InetAddress inet = InetAddress.*getByName*(ipAddress);

*socket*= **new** Socket(inet, port);

*outStream*=*socket*.getOutputStream();

*inStream*=*socket*.getInputStream();

ret=0;

} **catch**(IOException e) {

e.printStackTrace();

disconnect();

}

**return** ret;

}

* **Setting up the Port Write to write the bytes to the output stream between the devices:**

**public** **int** Write(**byte**[] departingData, **int** nBytes) {

**int** ret;

**try** {

*outStream*.write(departingData, 0, nBytes);

*outStream*.flush();

ret=nBytes;

} **catch** (IOException e) {

disconnect();

e.printStackTrace();

ret=0;

}

**return** ret;

}

* **Setting up the Port Read for Remote to read in bytes from the input stream between the devices:**

**public** **int** ReadRemote(**int** nBytes) {

**int** index=0;

**int** totalRead=0;

**int** buf=0;

**byte**[] buffer = **new** **byte**[nBytes];

*returningData*=**new** **byte**[nBytes];

**do** {

**try** {

Thread.*sleep*(100);

buf= *inStream*.read(buffer);

totalRead+=buf;

**for**(**int** i=0; i<buf; i++) {

*returningData*[index]=buffer[i];

index++;

}

} **catch** (IOException e) {

disconnect();

totalRead=0;

**break**;

} **catch** (Exception e) {

totalRead=0;

disconnect();

**break**;

}

} **while**(totalRead<nBytes && *returningData*[19]!=0x01 && *returningData*[12]==0x00 && totalRead>=0 && *returningData*[0]==1);

**return** totalRead;

}

* **Setting up the Port Disconnect to close the socket connection and the streams:**

**public** **byte** disconnect() {

**byte** ret = (**byte**)255;

**try** {

**if**(*inStream*!=**null**) {

*inStream*.close();

} **if**(*outStream*!=**null**) {

*outStream*.close();

} **if**(*socket*!=**null**) {

*socket*.close();

}

ret=0;

} **catch**(IOException e) {

e.printStackTrace();

}

**return** ret;

}

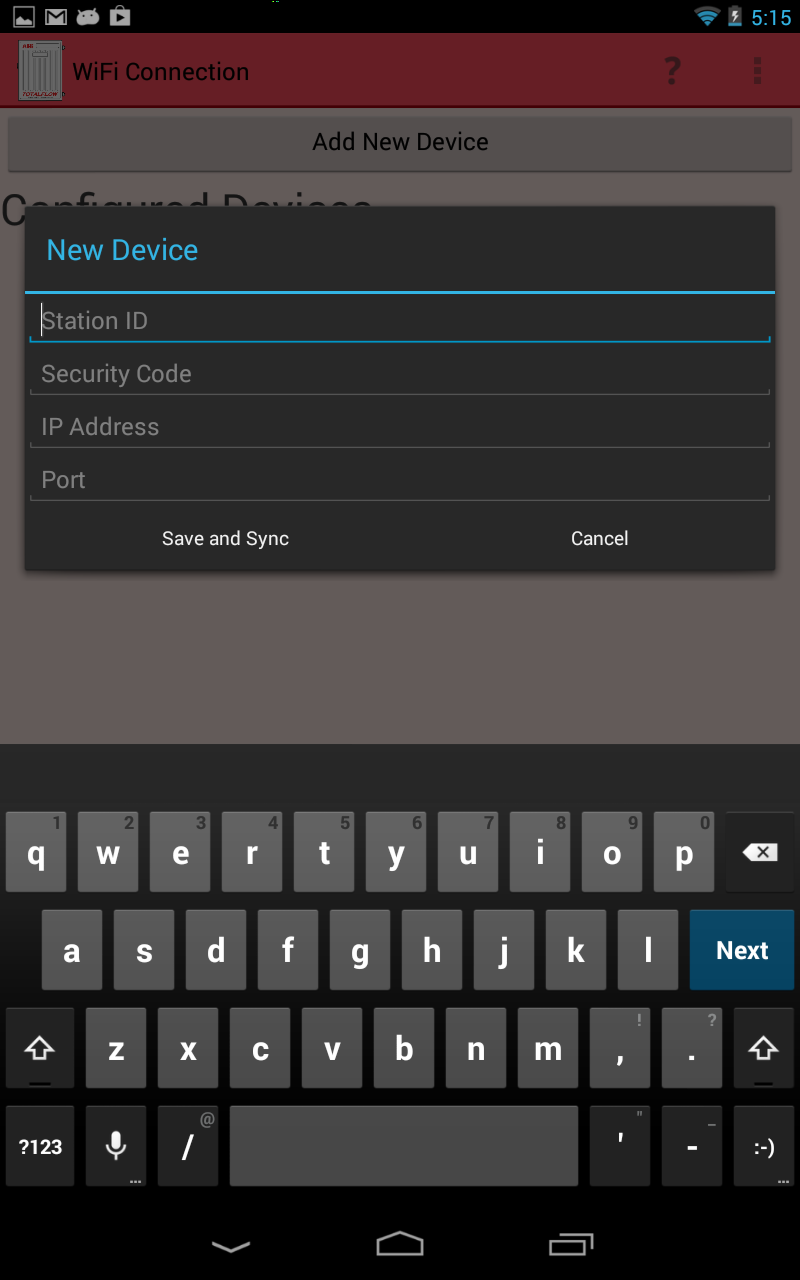


Figure 18 Screenshot of page-2b for TCP/IP connection

## 3.7 Design Aspects for Application Performance

The MCCU application will be used on have limited resources in terms of memory and CPU. In order to achieve the time for individual page loading to around one second and comparable to the time taken by similar screens on MCCU, following parameters need to be considered. The other aspect that needs to be considered is the hardware capability of the Android device used to run the application.

The **Navigator** (section 3.2.1) always keeps the user at square one, allowing for every page to open seamlessly with no hanging pages. This may not be necessary on every page, but it is more efficient to keep track of all pages in order to optimize user navigation throughout the application.

## 3.7.1 Page Size and Corresponding Graphics

Number of graphics and size of all graphics on a page have a direct impact on the response time for that application page. To improve the performance of any application page, the quality and number of graphics needs to be minimized. In order to reduce the size of the graphics used on any application page there are some techniques like compressing the content, reducing the image sizes to the desired resolution. By placing multiple .PNG files with varying pixel counts, according to size in different folders, the application gains efficiency.

## 3.7.2 Number of Threads

The Android application has one thread dedicated to the user interface. The Android OS forces all network communications onto a separate thread. A new thread is created when the Android device begins receiving register data from the G4. After register data is obtained, the thread is finished. If the transmission is canceled then an error is forced into the protocol causing it to finish the thread early in order to maximize performance. A thread is created on page 2b to connect to a Wi-Fi network. A progress dialog spinner is displayed on the UI thread while the other thread finishes it course.

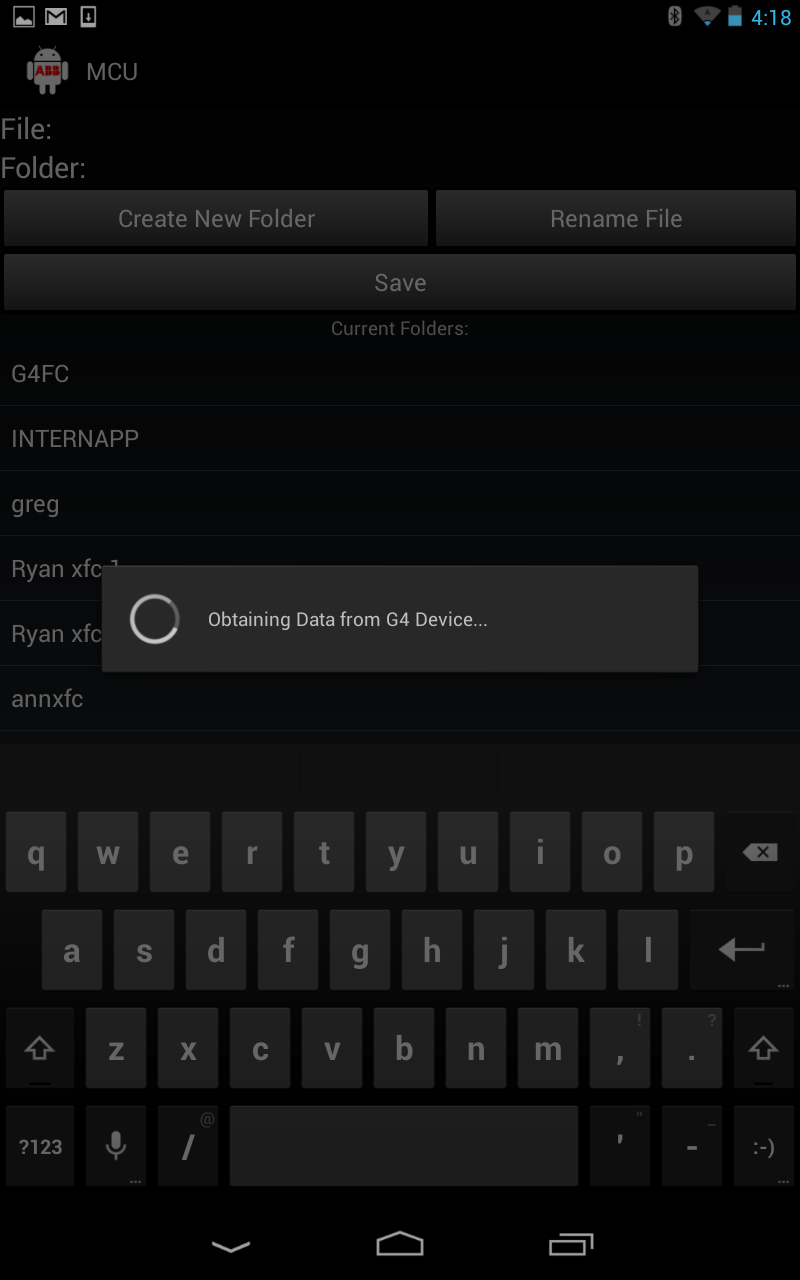


Figure 20 Spinner displayed to represent thread for obtaining Wi-Fi

## 3.7.3 Memory Usage

The Android application uses internal and external memory to maximize application performance. When navigating from page to page, the previous page is stored as a variable. When the user presses the “back” button to navigate to the previous page, that variable will be accessed. This functionality is shown as follows:

*Navigator.java class should be called when user goes to a new page. The page the user visited previously should be entered in the form of <package name>, <Activity previously visited, or class name>:*

public static void addPrevPage(String prevPage) {

prevPages.add(prevPage);

incrementNav();

}

private static void decrementNav() {

navIndex--;

}

private static void incrementNav() {

navIndex++;

}

public static ArrayList<String> getArray() {

return prevPages;

}

For File Manager, refer to section 3.4.2.

## 3.7.4 Recovery from crash/hang

If the Android application crashes due to some problem/bug and causes Android MCCU to exit; the user must re-open the Android application and start again at the Connection Selection (page-1). If the application crashes during Re-Sync (page-3) or during the storing of a file, the data is not stored on the Android device. If the application crashes during View Options (page-5a), the user preferences are not saved. If the application crashes during Re-Sync (page-3), the port is left hanging and the user must manually reset the G4 device. If the Android device does not get a response from the G4 device, it will throw a timeout error.

## 3.7.6 Indicative/Responsive UI

To ensure that user is always aware of the UI processing and events feedback, progress indicators is used so that user knows that processing is in progress and screen is not in a frozen/blocked state.

After the user inputs their information for connecting to a device via Bluetooth or IP, the application will display the dialogue spinner to let the user know that their action is being processed.

*Spinner being created in LoadG4Data.java class:*

private void createSpinner() {

**//Sets up the up the view of the dialog**

progDialog = new ProgressDialog(this);

progDialog.setProgressStyle(ProgressDialog.STYLE\_SPINNER);

progDialog.setMessage("Obtaining Data from G4 Device...");

progDialog.setCanceledOnTouchOutside(false);

progThread = new ProgressThread();

progThread.start();

progDialog.setOnCancelListener( new OnCancelListener() {

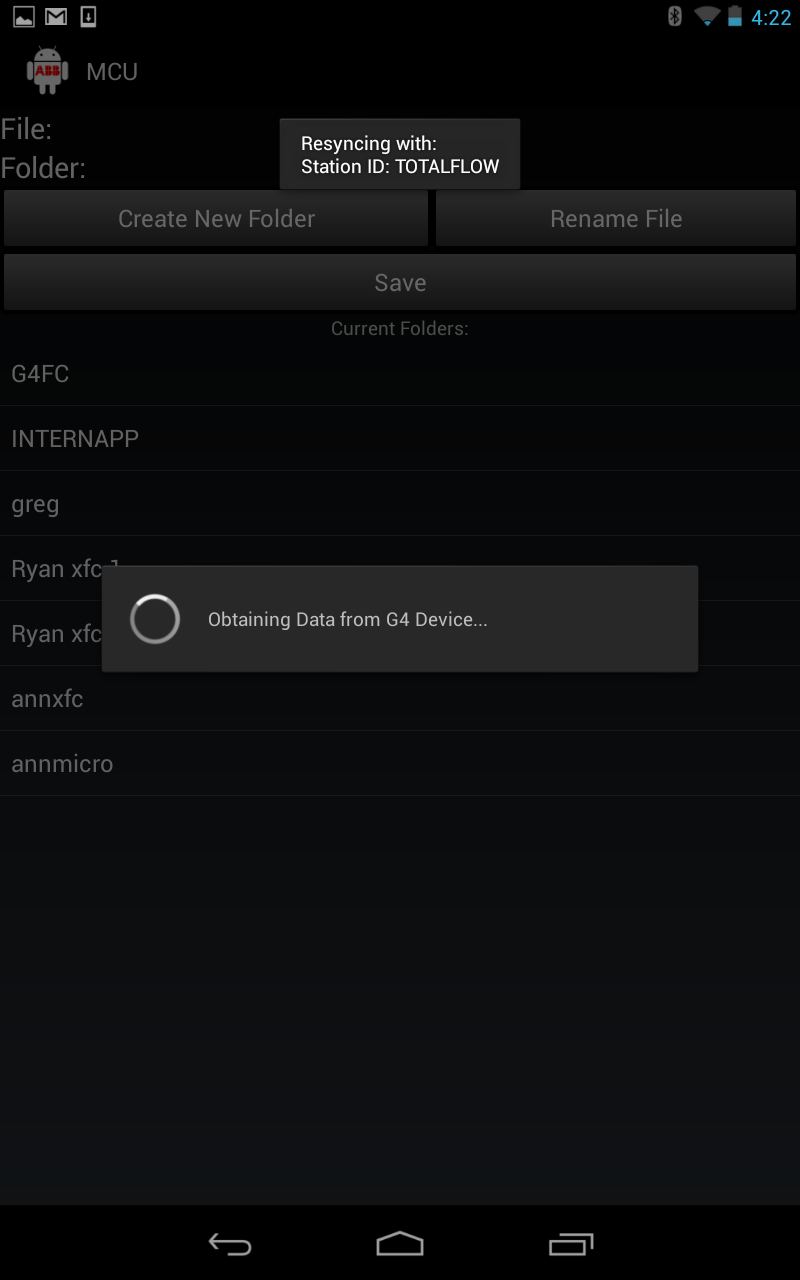
****

Figure 21 Screenshot example for “spinner” used when loading data

Android OS has built in identifiers to indicate Wifi and Bluetooth status. Refer to section 3.3 for indicator information.

## 3.8 Register Request Protocol (TRS4)

Only the predefined registers are accessed by the Android device when the user connects to the G4 device (TRS4). All of the register types are read based and are defined in the format *app.array.index*. The data can be stored into the Android device’s external memory to be shared or to be displayed (TRS4) in lists and charts from the device.

Requests are sent out in the format (*Application*, *Array*, *Index*, *# of registers*, *data size (in bytes)*)

*Register requests from LoadG4Data.java class:*

**// register request for system apps, System is always in slot 0.**

xfRequest appTable = new xfRequest(AppView.SYSTEM, 3, 1, 253, Byte.SIZE / 8); numReg++;

numBytes += 253;

**// Adds all the requests to a list**

xfRequest[] registerList = new xfRequest[numReg];

int index = 0;

registerList[index] = appTable;

index++;

// **This will hold the returned data**

registerData = new RegisterData(numStrings, numBytes, numFloats, numInt32, numInt16,numUInt8, numInt8);

**// This runs the protocol down the right path with the connection** information, the list of requests, the data container,

**// and whether this is the first time through the protocol**

db2Protocol = new xfProtocol(code, stationID, registerList, registerData, device, ipAddress, port, connectType, isFirstProtocol);

**// begins the transaction**

db2Protocol.ReadRegister();

registerData=db2Protocol.getRegisterData();

return db2Protocol.getStatus();

}

*Connection and register request protocol from xfProtocol.java class:*

public void ReadRegister() {

**// Calculates how many bytes are going to be sent and formats a queue so they can all be sent at once**

// **1(Sync Byte), 4(Sync Frame), Supervisory Frame, Single Header, and (Record\*the number of xfRequests)**

totalSendBytes=1+4+xfFrame.SIZE+xfHeader.SIZE+(xfRecord.memSIZE+xfRecord.crcSIZE)\*(registerList.length);

departingData= new byte[totalSendBytes];

**//Assume unsuccessful**

Status=xfTIMEOUT;

**//Establish connection with device for the port based off the chosen connectType**

if(connectType==LoadG4Data.BLUETOOTHconnect) {

**//Creates the Bluetooth Port and then tries to connect**

Port = new BluetoothSPP(device);

Status = ((BluetoothSPP) Port).connect();

} else if(connectType==LoadG4Data.TCPIPconnect) {

**//Creates the TCP/IP Port and then tries to connect**

Port = new TCPIP(ipAddress, port);

Status = ((TCPIP)Port).connect();

}

**/\*If the connection was successful and it is a Bluetooth connection then the device will need to**

**\* have the protocol switched from Local to Remote. However, this only needs to happen once with the**

**\* first set of requests.**

**\*/**

if(Status==0 &&isFirstSet &&connectType==LoadG4Data.BLUETOOTHconnect) {

switchToRemote();

}

if(Status==SUCCESS) {

**// Create & add Frame, Header, and Records to departingData**

SendXFrame();

SendXHeader(XSINGLE, XREADREQ);

SendXRecords();

**// Send the message, then check if send was successful**

if(SendAllXData()) {

**//Read the returned message**

Status = ReadAllXData();

**//Parse the Frame, Header, and Records from the returnedData, while checking for any errors**

if(Status==SUCCESS) {

Status = ReadXFrame();

if(Status==SUCCESS) {

Status=ReadXHeader();

if(Status==SUCCESS) {

Status=ReadXRecords();

}

}

}

} else { //If there was an error while sending the message

Status=xfTIMEOUT;

}

}

**//Break the connection once all steps have finished or if an error broke them out of the steps**

**//for the port based off the chosen connectType**

if(connectType==LoadG4Data.BLUETOOTHconnect) {

((BluetoothSPP) Port).disconnect();

} else if(connectType==LoadG4Data.TCPIPconnect) {

((TCPIP) Port).disconnect();

}

return;

}

## 3.9 Register Grouping

Registers do not have to be grouped in any orderly fashion when sending a request. After the data has returned for the group of register requests, MCCU categorizes the data for a user friendly display.

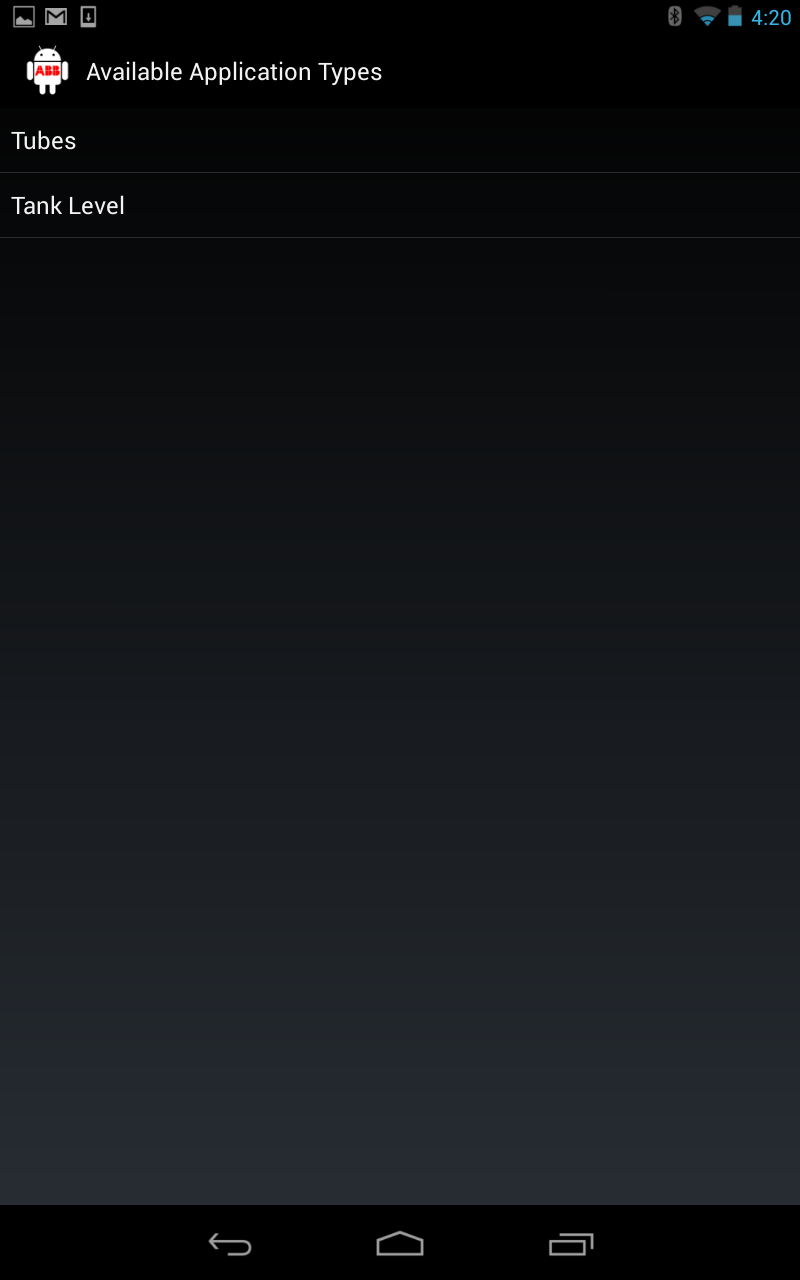


Figure 22 Application Selection Page

## 3.9.1 Tube/Tank Data Selection

When the user views a set of data from the G4 device, they will obtain a list of tubes or tanks depending on their application selection (Page 4). By clicking on a member of that list the following tabs will be filled with its corresponding information. The user must re-sync from MCCU application to further update their data taken from the G4 device.

The list of tubes derives from the instances of tube applications on the device. It is one tube per application. The list of tanks derives from not only the instances of level master applications but the number of tanks per level master application. The number of tanks within a LevelMaster application has to be figured first through a separate register request.

// request for the number of tanks within the specific level master application

xfRequest numTanksReg = new xfRequest(tank,107,0, 1, int16.sizeof()); numInt16++; numReg++;

// request for the number of set of registers per number of tanks

private byte getNumTankRegister(int tank) {

**//resets the counters**

numStrings=0;

numBytes=0;

numFloats=0;

numInt32=0;

numInt16=0;

numUInt8=0;

numInt8=0;

numReg=0;

**//the register requests**

xfRequest numTanksReg = new xfRequest(tank,107,0, 1, int16.SIZE); numInt16++; numReg++;

**// Adds the requests to a list**

xfRequest[] registerList = new xfRequest[numReg];

int index = 0;

registerList[index]=numTanksReg; index++;

**// will hold the all the returning data**

registerData = new RegisterData(numStrings, numBytes, numFloats, numInt32, numInt16,numUInt8, numInt8);

**// This runs the protocol down the right path with the connection information, the list of requests, the data container, and**

**// whether this is the first time through the protocol**

db2Protocol = new xfProtocol(code, stationID, registerList, registerData, device,ipAddress, port, connectType, !isFirstProtocol);

**//begins the transaction**

db2Protocol.ReadRegister();

registerData=db2Protocol.getRegisterData();

return db2Protocol.getStatus();

}

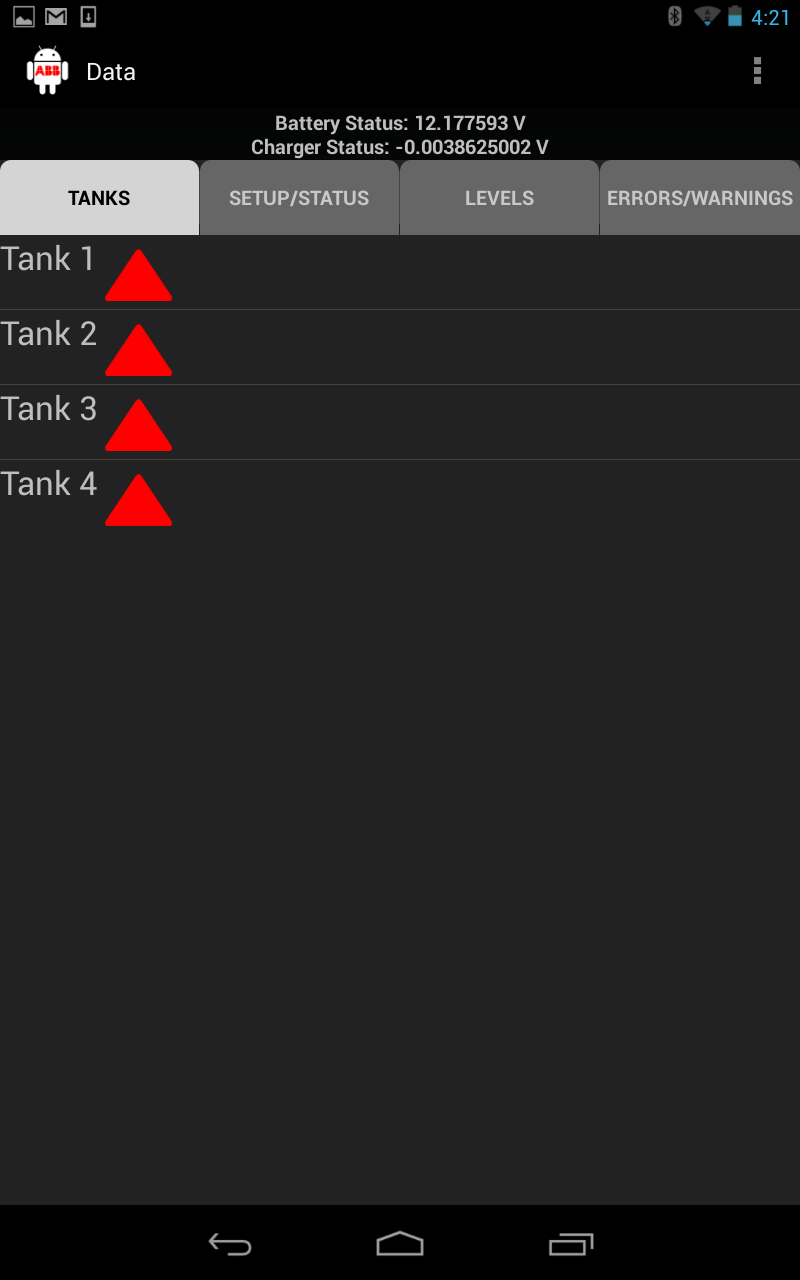


Figure 23 Screen shots for tank selection

## 3.9.2 General Data

The General Data covers a range of information that is displayed simply as:

‘*category description:’ ‘retrieved data’ ‘unit’ (if applicable).* See figure below.

For **Tubes**, general data is specifically set to display flow measurement. When reading flow measurement from the G4 device, the user will be able to see current data as well as the entire day, yesterday, and accumulated for both gas and liquids. Flow measurement categories include: volume, flow rate, energy, energy rate, static pressure, differential pressure, temperature, pulse, and mass (TRS4). When reading flow measurement from the G4 device, the user will be able to see all data with corresponding units of measurement. Since the units can be changed for certain registers, additional register request must be made to get the correct unit.

*Units register requests made from LoadG4Data.java class:*

***// For US unit tubes***

xfRequest unitV\_FR = new xfRequest(aga3,0,42, 1, 1); numBytes++; numReg++;

***//For Selectable Unit tubes***

xfRequest frUnit= new xfRequest(aga3su,11,20,1,65); numStrings++; numReg++;

xfRequest vUnit= new xfRequest(aga3su,11,18,1,65); numStrings++; numReg++;

xfRequest erUnit= new xfRequest(aga3su,11,30,1,65); numStrings++; numReg++;

xfRequest eUnit= new xfRequest(aga3su,11,28,1,65); numStrings++; numReg++;

xfRequest dpUnit= new xfRequest(aga3su,11,3,1,65); numStrings++; numReg++;

xfRequest spUnit= new xfRequest(aga3su,11,2,1,65); numStrings++; numReg++;

xfRequest tUnit= new xfRequest(aga3su,11,5,1,65); numStrings++; numReg++;

xfRequest mUnit= new xfRequest(aga3su,11,13,1,65); numStrings++; numReg++;

xfRequest ivUnit= new xfRequest(aga3su,11,23,1,65); numStrings++; numReg++;

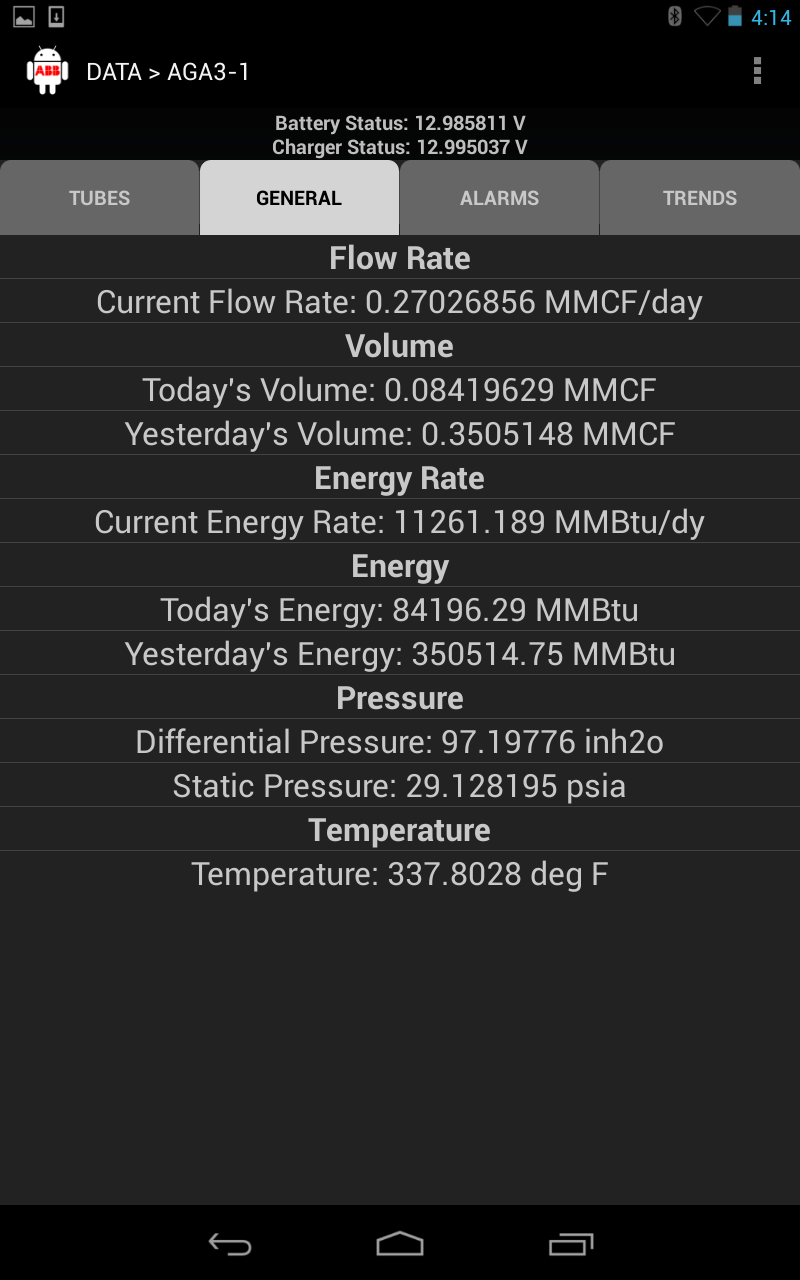
****

Figure 24 Screenshot example for Flow Measurement Data collected

For **Tanks**, two tabs are used to display the general data for the device. The setup/status tab shows information regarding tank identification and polls. The levels tab shows information regarding tank measurements. Tank measurement categories include: level volume, level inches, temperature, and Ullage. The units for LevelMaster are static.

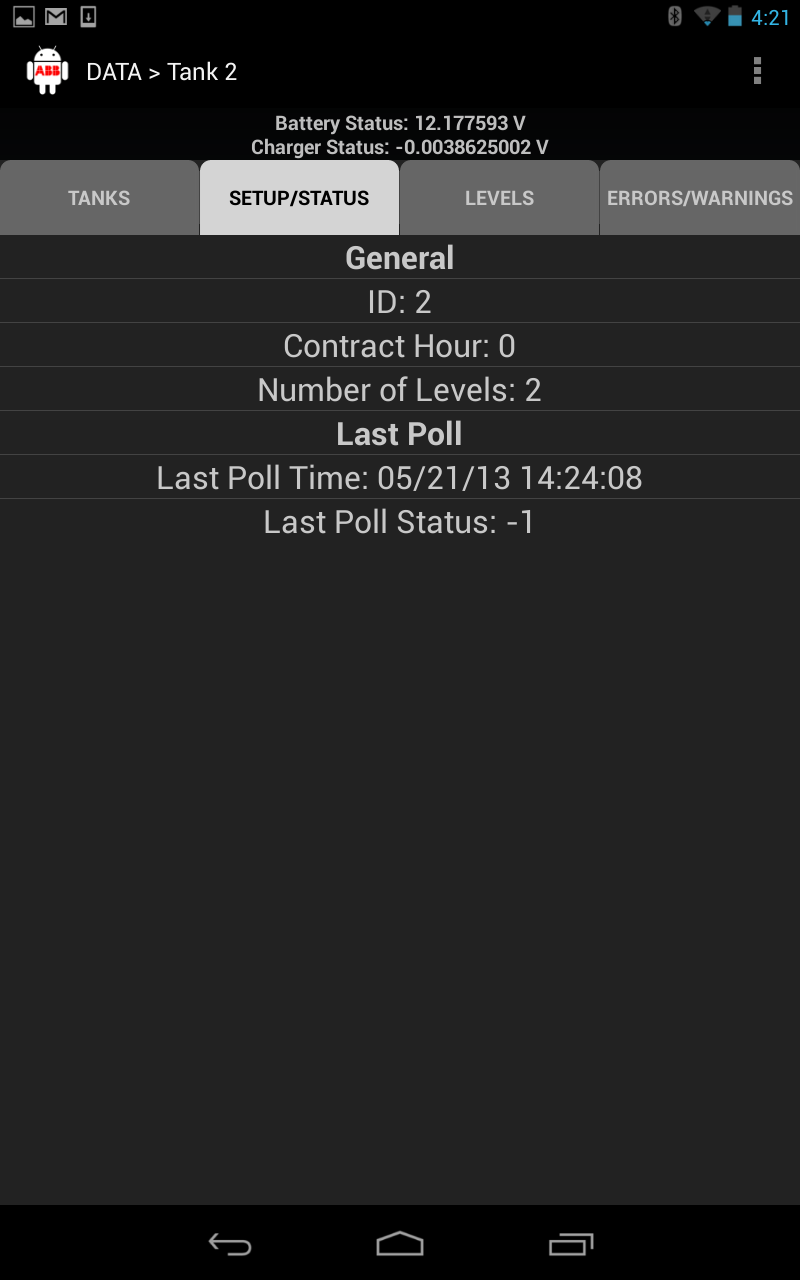


Figure 25 Screenshot example of Setup/Status Tab Data

****

Figure 26 Screenshot example of Levels Tab Data

## 3.9.3 Active Alarms

When reading active alarms from the G4 device, the user will be able to see any unacknowledged alarms. This will not allow the user to acknowledge the alarm from the Android device. After the alarm has been reset on the G4 device, and the user re-syncs the Android application to the G4 device, the alarms will no longer show up as unacknowledged.

The Android application is enhanced to provide the list of current Active Alarms based on the specific application. A register request corresponding to the Tubes current alarms is packaged with the other registers for that data transaction. The response is decoded by taking the 32-bit integer returned and converting it into binary. Each bit correlates to a specific alarm within a set of 20 total alarms. If the bit is equal to 0, the alarm is off. If the bit is equal to 1, the alarm is on. The two character code that corresponds to that bit is shown if it is on. The user can toggle between the two character code and the full length description on the page.

A register request corresponding to the tank current errors and warnings are packaged with the other registers for that data transaction. The response corresponds to a single error in an enumeration of errors/warnings. The error code is displayed and the user can toggle between the error code and the full length description on the page.

xfRequest alarms = new xfRequest(aga3, 9, 0, 1, int32.sizeof()); numInt32++; numReg++;

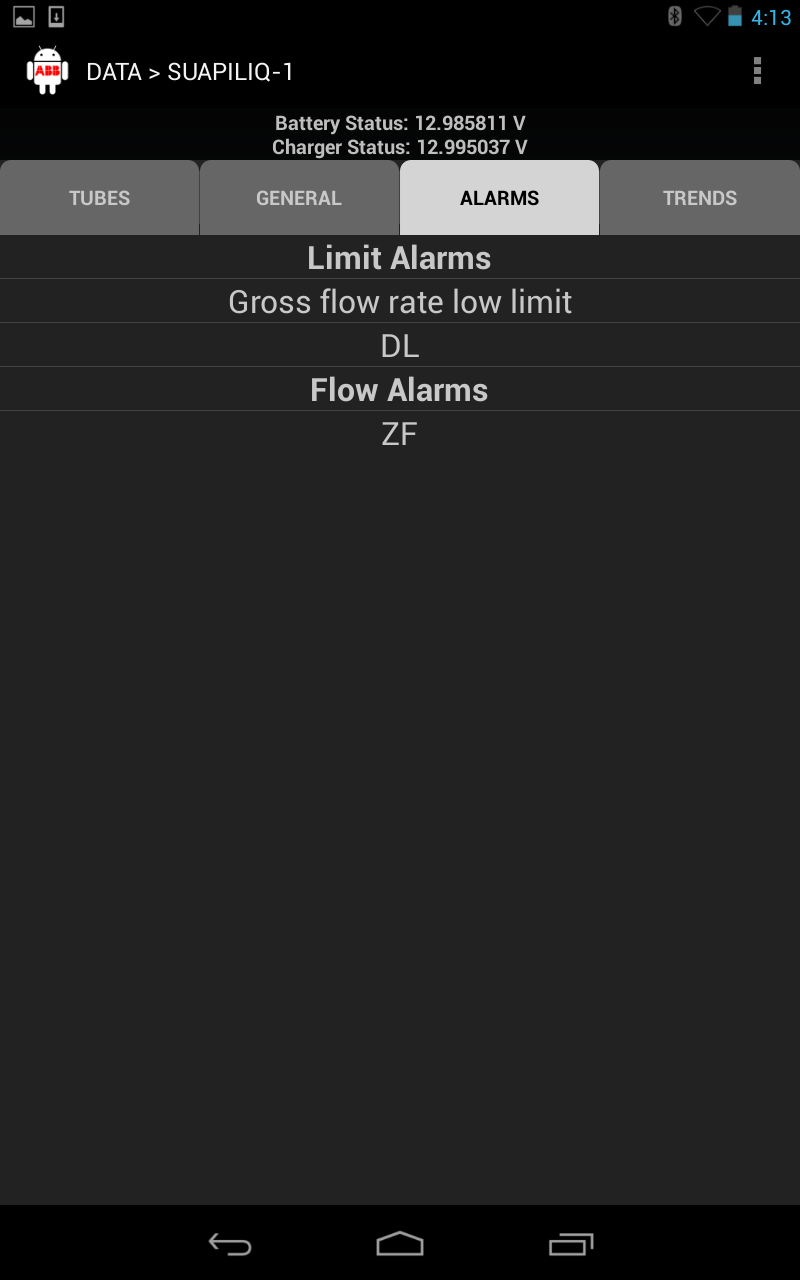
****

Figure 27 Screenshot of Alarm Tab Data

## Alarm Log

## MCCU does not provide for an updated list of Alarm logs, only current active alarms. Any alarm viewed from the application is an active alarm request. All alarms that are viewed and unacknowledged can be saved as a file into the external thread and then viewed by the user later. Essentially the user is creating their log of alarms.

**3.9.4 Trends (TRS4.6)**

Trend data is gathered with a single register request. The G4 device has designated arrays that contain historical data for certain categories. In the register request, the array containing historical data is accessed at index 0 and polled through index 13 to gather the past fourteen days of data.

xfRequest volume14=new xfRequest(aga3,225,**0, 14**, floatSize); numFloats+=14; numReg++;

All fourteen days are displayed in an Expandable List View in the format: ‘*category description:’ ‘retrieved data’ ‘unit’ (if applicable).*

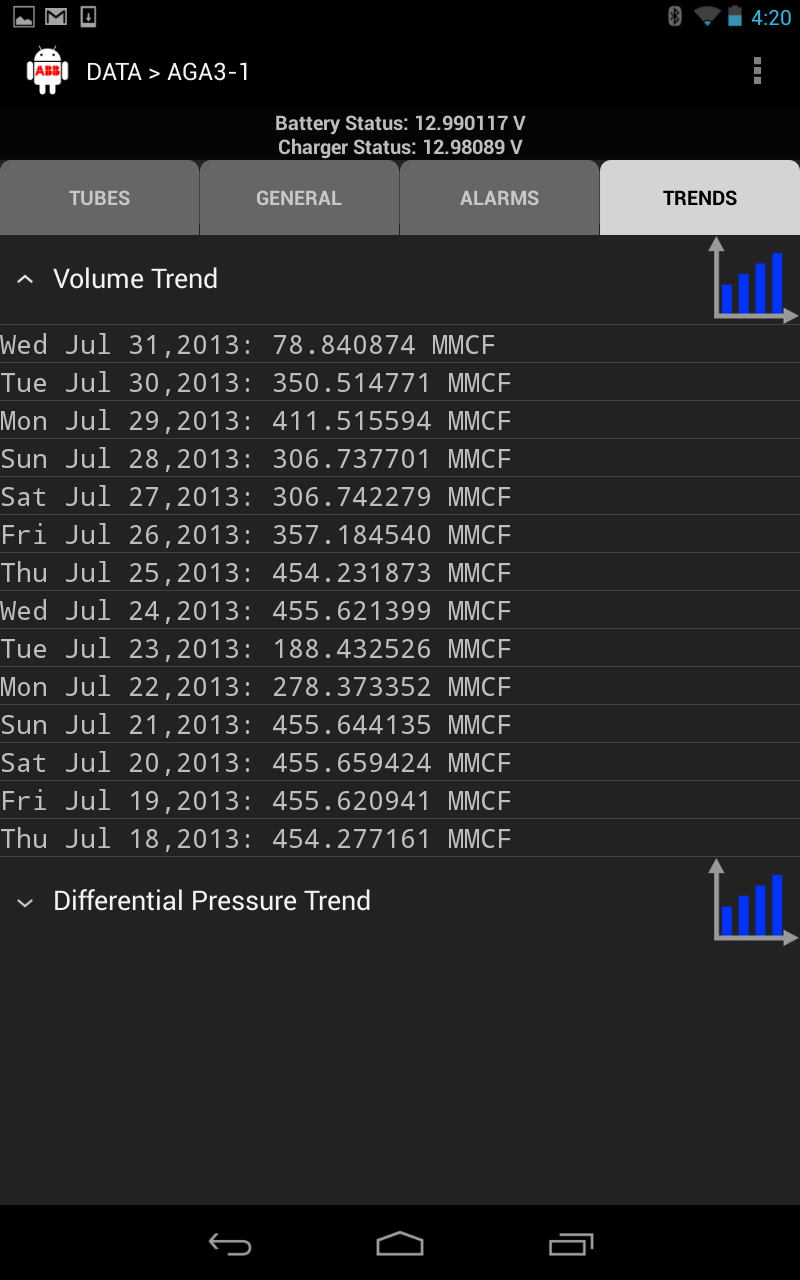
****

Figure 28 Screenshot of Trends Tab Data

Next to the category name that contains the past 14 days is a button that will launch the user to an interactive bar graph. The graphs are used to visualize the various parameters and monitor them over a period of time. There will be X and Y axis which will represent the parameters that are being compared against each other – measured data being the Y axis and duration of time being the X axis. A legend is also shown to describe multiple parameters in case multiple parameters are being used in the graph.

For implementing the graph in Trends screens, ‘ChartEngineDemo’ library is used.

The following is called from the TrendsBarChart.java class:

//Adding color to be used for the Bars

**int**[] colors = **new** **int**[] {0xffff6c00};

XYMultipleSeriesRenderer renderer = buildBarRenderer(colors);

//Adding Renderer Style, chart title, labels, X-axis boundaries, Y-axis boundaries, and label text color

setChartSettings(renderer, "Historical Data for the past 14 days", "Day", " "+floatUnit, 0.5,

7.5, 0, max, 0xff989898, 0xff989898);

renderer.getSeriesRendererAt(0).setDisplayChartValues(**true**); //Adds values to the Bars

renderer.~~setChartValuesTextSize~~(22); //Changes text size of the Bar values

renderer.setLabelsTextSize(18); //Changes text size of the Axis labels

renderer.setXLabels(7); //Sets spacing for X labels

renderer.setYLabels(5); //Sets spacing for Y labels

renderer.setXLabelsAlign(Align.*LEFT*);

renderer.setYLabelsAlign(Align.*LEFT*);

renderer.setYLabelsColor(0,0xff989898); //Sets label colors to Grey

renderer.setXLabelsPadding((**float**)8); //Shifts the X-axis labels down from chart

renderer.setYLabelsPadding((**float**)30); //Shifts the Y-labels left

renderer.setPanEnabled(**true**, **false**); //allows panning along the X-axis

renderer.setPanLimits(**new** **double**[] {0.5, 14.7, 0, max}); //Limits X-axis panning to 14.7 days

renderer.setZoomEnabled(**false**);

renderer.setZoomEnabled(**false**, **false**);

renderer.setBarWidth(45); //Changes the width of each bar

renderer.setLegendTextSize(28); //Changes legend text size

//Utility method for creating chart views or intents

**return** ChartFactory.*getBarChartIntent*(context, buildBarDataset(titles, values), renderer,

Type.*STACKED*);

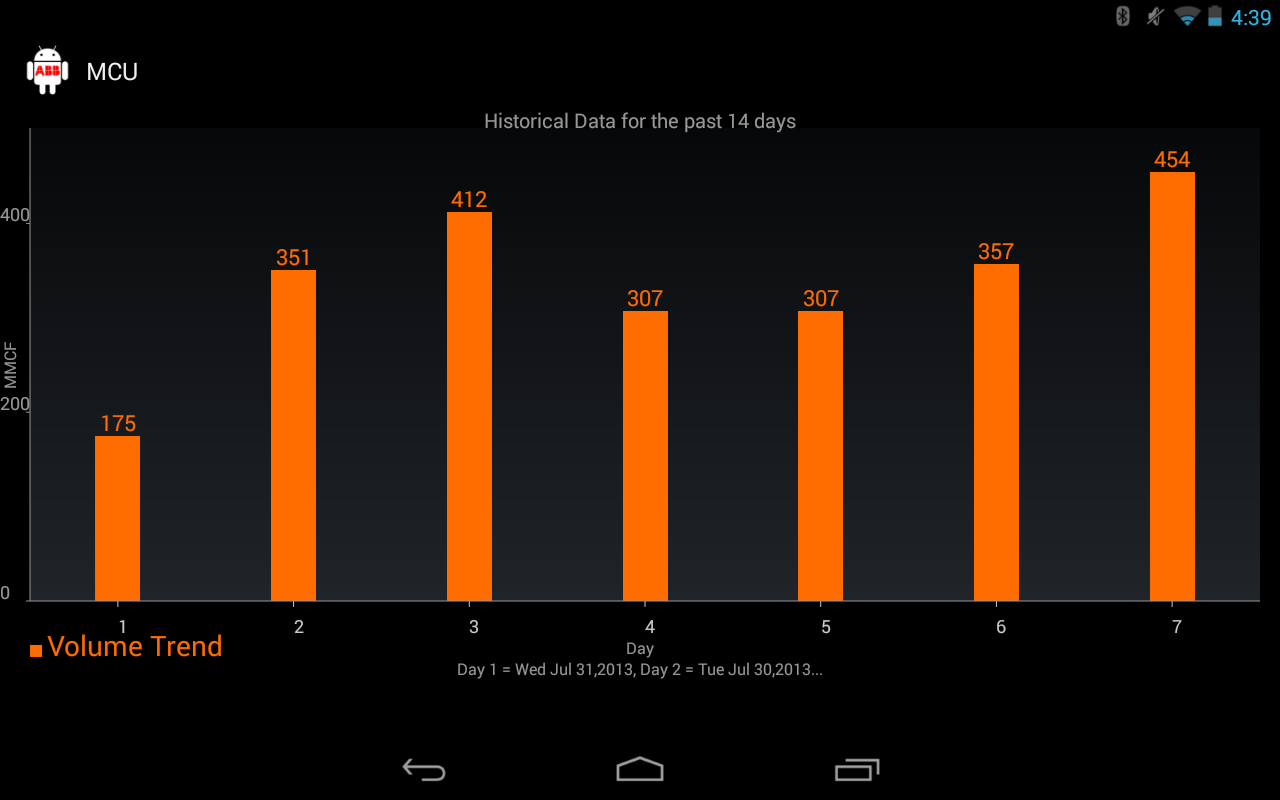
****

Figure 29 Screenshot for an example of historical data

## 3.10 Error Codes

When using MCCU application there are cases where the application may not complete a request made by the user. MCCU is designed to notify the user when there is an error during connection. When an error is caught, MCCU runs through the following cases to match the error code with a description. The user is brought back to the connection selection (page 1), and the error is displayed over the page.

private String getErrorMessage(int error) {

switch (error) {

case 200:

return error + ": Unable to locate applications for tubes.";

case 201:

return error + ": Unable to save and retrieve data.";

case 203:

return error + ": Data request cancelled.";

case 102:

return error + ": The device received a bad xfHeader CRC.";

case 104:

return error + ": The device received a bad xfRecord CRC.";

case 106:

return error + ": The device received an invalid operation request.";

case 126:

return error + ": Illegal register read or write.";

case (byte) 0x81:

return error + ": Undefined error.";

case (byte) 0x88:

return error + ": xfFrame already in use.";

case (byte) 0x96:

return error + ": Invalid application slot.";

case (byte) 251:

return error + ": xfLENERR";

case (byte) 252:

return error + ": xfCRCERR";

case (byte) 253:

return error + ": xfNAKCODE";

case (byte) 254:

return error + ": xfINVALID";

case (byte) 255:

return error + ": xfTIMEOUT";

case (byte) 207:

return error + ": No files";

case (byte) 206:

return error + ": No Bluetooth ports found on device.";

case (byte) 205:

return error + ": Connection to device failed.";

case (byte) 204:

return error + ": Unable to change communication protocol.";

default:

return error + ": Unknown error number was returned.";

}

}

**3.10.1** **No Data from Register**

In case the device register has no value at a particular moment and the request to get the data from device register returns no data, user is prompted with pop-up dialog box displaying ‘xfCRCERR’. This behavior follows the Error Scenario Handling and all the error codes mentioned in section 3.10.

## 3.11 Help (TRS4, TRS5)

Help options are available within the Action Bar (top of screen). The help page will allow the user to navigate information on topics regarding connection for Bluetooth/IP, saving files, data view, File Explorer, options view, and navigation.

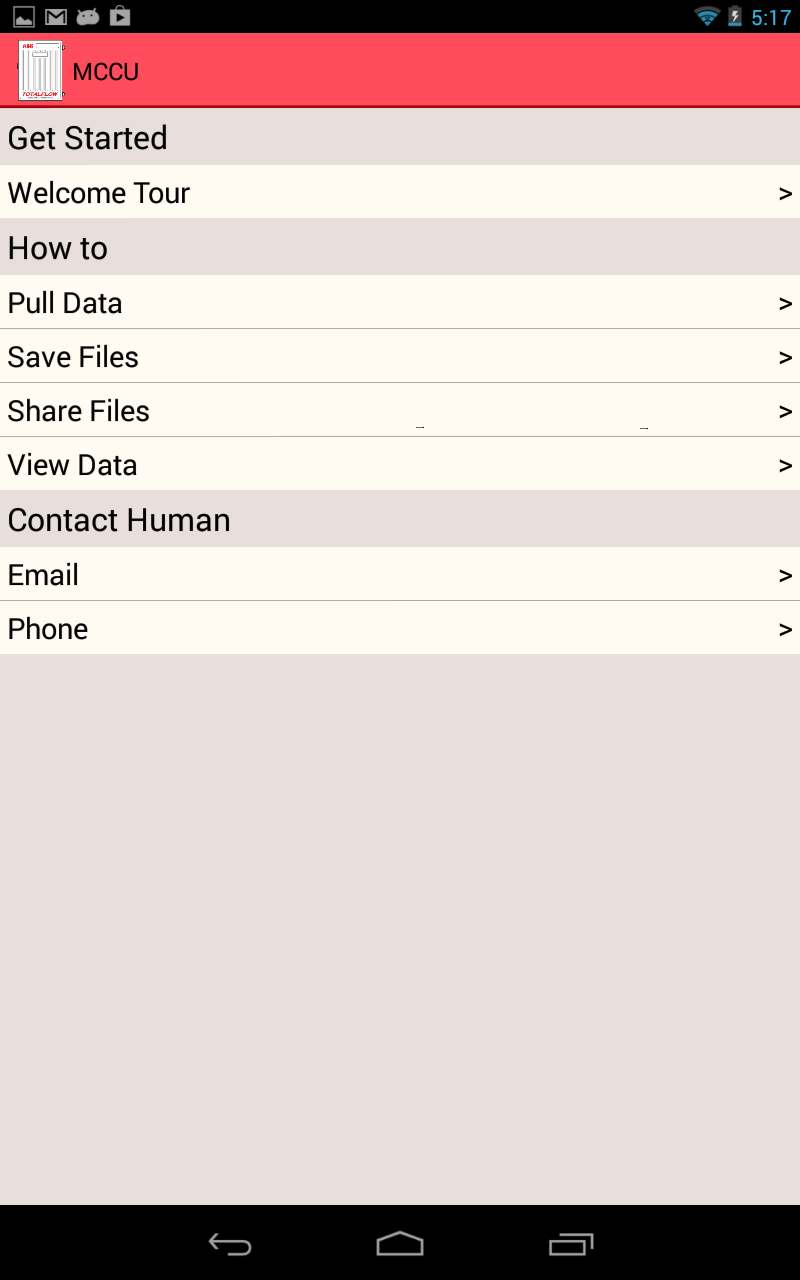


Figure 31 Screenshot from Help menu:

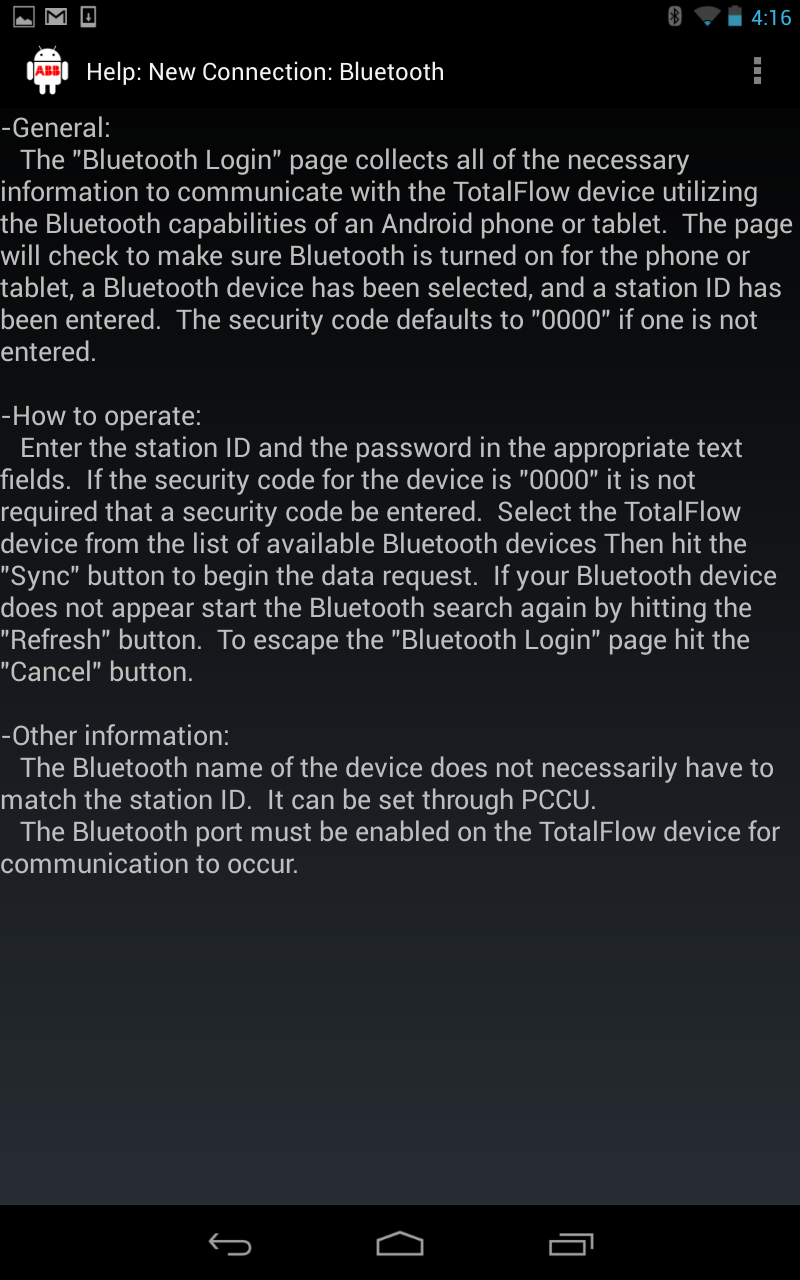


Figure 32 Screenshot example of selecting a Help option

## 3.12 User Editable Fields

MCCU has the option to allow the user to make adjustments to the look, feel, and functionality of the application.

## 3.12.1 View Options

When displaying data on MCCU, the user may want to limit or maximize the items to be displayed based on their own user preferences. The user has the capability of selecting which items to be displayed using checkboxes in the View Options selection from the Action Bar Overflow (section 3.4.1). Items under the General and Trends tab can be manipulated for a on or off view. The selection of items to be edited is created in an expandable list, allowing the user to select what type of measurement they want to edit. The expandable adaptor is built of the base list for the expandable adaptor is an inner, meaning it cannot be re-used. This is another page that utilizes AppView.java class to create the list of expandable items. This allows the list to populate based upon the tube type that the user selects from the Application Selection (page 4).



Figure 33 Screenshot example of View Options

## 3.14 Formatting (TRS5)

MCCU follows the NGHLA guidelines for specific colors, fonts, and layouts used on different application pages. The built in Android theme, Holo, has been modified slightly to meet the NGHLA coloring format. When applying styles that only modify a View’s text properties the style is applied via the Text Appearance property. Styles that modify non-text properties are applied via the *style* property. By default, the Android Holo theme uses TextAppearanceMedium which is scalable based on the type of Android device and the pixel count for the screen size. The user can edit these options in the settings of the Android Device.

Holo Theme Content:

* Screen Background Dark: Black (#000000)
* Text Size: 18 scaled pixels
* Text Font: Holo

## 3.14.1 Fonts

In general, Holo Font, default theme is used as the ‘font’ across the screens to display various text. On all screens for non-value labels, font is non-bold. For Trends tab, MonoSpace font is used to make all of the characters the same width. Using MonoSpace makes the Trends dates organized so each day is lined up and the user can scan down the list of days with ease.

Custom Trends Content:

* Text Scale: 1.0
* Text Size: 20 scaled pixels
* Text Font: MONOSPACE

## 3.14.2 Error Scenario Handling

In case there is an error while saving data or running through the protocol then an error message is displayed to the user. The error message takes into account all of the format and font settings for the Android Holo Theme. This error message displays brief error description along with possible recovery step (if required). This error message helps the user to be aware of any possible error and then recover from it (section 3.10).

## 3.14.3 Res Folder

Styling information for the user interface part is maintained as separate files. There are two ways to set a style:

* To an individual View, by adding the *style* attribute to a View element in the XML for the layout.
* To an entire Activity or application, by adding the *android:theme* attribute to the <activity> or <application> element in the Android manifest.

All such information is maintained as XML files, which are applied as required at multiple places. This also ensures that styling information is not duplicated. A lot of the features on MCCU utilize the *android:theme* attribute.

## 3.14.4 Action Bar Overflow

The Action Bar Overflow appears at the top of an activity’s window when the activity uses one of the system’s descendant themes that are set by default. The Action Bar Overflow is populated with various tabs to allow the user for navigating throughout the application.

Holo Theme Content:

* Screen Background Dark: Black (#000000)
* Text Size: 18 scaled pixels
* Text Font: Holo

## 3.14.5 Tab Navigation

User can choose to browse across various screen sections using the different Tabs. Every tab contains the name that describes the content on screen. The name changes color when user clicks on that particular tab name.

Following are the key states of the tab names section on screen:-

1. Selected (This tab is the currently selected tab and displays the content that belong to that screen).
2. Un-Selected (This tab is not the active tab)

This navigation section conforms to following specification:-

1. Un-Selected

* Font Style: Holo
* Font Size: 18 scaled pixels
* Width : Theme Default
* Height: Theme Default
* Font Color: Light Grey (#c8c8c8)
* Background Color: Very Dark Grey (#666666)

1. Selected

* Font Style: Holo
* Font Size: 18 scaled pixels
* Width : Theme Default
* Height: Theme Default
* Font Color: Black (#00000)
* Background Color: Very Light Grey (#d4d4d4)

## 3.14.6 Expandable List View

The Data View page contains the Trends Tab with data that is organized in Expandable lists. The Expandable list is a view control that allows the user to provide multiple panes and display them when user clicks the list. It is like having several [panels](http://www.asp.net/ajaxlibrary/AjaxControlToolkitSampleSite/CollapsiblePanel/CollapsiblePanel.aspx) where every Expandable lists can be expanded on user click. Every Expandable list represents a logical grouping of the data. Expandable lists make it easy to display large number of data in a limited screen space. Expandable lists will conform to following specification:-

Expandable lists Section Content

* Font Style: Holo
* Font Size: 18 Scaled Pixel
* Font Color: Very Light Grey (#d4d4d4)
* Background Color: Very Dark Grey (#222222)

## 3.14.7 Graphs

The graphs used for representing trends are controlled by the ‘renderer’ class. The labels, panning, zoom, bar width, legend, text, and colors are all set here. The coloring follows the same guidelines mentioned for charts in the ABB\_NGHLA\_Layout\_Specs\_220812.pdf document.

Expandable lists Section Content

* Font Style: Holo
* Background Color: Very Dark Grey (#666666)
* Bar Label Text Size: 22
* X Label Text Size: 7
* Y Label Text Size: 5
* Y Label Color: Light Grey (#989898)
* X Label Color: Light Grey (#989898)
* Bar Width: 45
* Legend Text Size: 28