



ARL-TR-8584 • Nov 2018



Demonstration System for Radio-Frequency Microelectromechanical Systems Components

by Lee A Griffin, Robert R Benoit, and Ronald G Polcawich

Approved for public release; distribution is unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.



Demonstration System for Radio-Frequency Microelectromechanical Systems Components

by Lee A Griffin

Georgia Institute of Technology, Atlanta, GA

Robert R Benoit and Ronald G Polcawich

Sensors and Electron Devices Directorate, ARL

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
<p>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</p>					
1. REPORT DATE (DD-MM-YYYY) November 2018		2. REPORT TYPE Technical Report		3. DATES COVERED (From - To) 1 June–31 August 2017	
4. TITLE AND SUBTITLE Demonstration System for Radio-Frequency Microelectromechanical Systems Components				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Lee A Griffin, Robert R Benoit, and Ronald G Polcawich				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-SER-L Aberdeen Proving Ground, MD 21005-5066				8. PERFORMING ORGANIZATION REPORT NUMBER ARL-TR-8584	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT <p>The combination of piezoelectrics and microelectromechanical systems (MEMS) can result in devices with a wide range of applications, including resonators, energy harvesters, and various sensors. Currently, improvement of MEMS devices designed to operate in the radio-frequency (RF) regime has been of particular interest due to the devices' use in both commercial and military satellites and aerospace systems. The work presented characterizes and analyzes new designs for various RF MEMS devices produced at the US Army Research Laboratory. Their integration to a proposed mobile RF wireless communication system was investigated by developing a proof-of-concept demonstration of the proposed system. The RF circuitry of the proposed system consists of various RF MEMS switches and filters. A customized smartphone modification was used to demonstrate the functionality of the RF circuitry and directly compare the size/weight/performance of the MEMS devices to commercial off-the-shelf components.</p>					
15. SUBJECT TERMS radio-frequency microelectromechanical systems, RF MEMS, modular smartphone components, enhanced phone capabilities, X-Microwave demonstration, customized Android app					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 34	19a. NAME OF RESPONSIBLE PERSON Robert R Benoit
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (Include area code) 301-394-0607

Contents

List of Figures	v
List of Tables	v
1. Introduction	1
1.1 Motorola Moto Z Smartphone	1
1.2 X-Microwave Demo	1
1.3 X-Microwave Hardware	2
2. XMD App Development	4
2.1 Brief Introduction to Android Development and Android Studio	4
2.2 Installing the Moto Mods Software Development Kit (SDK)	5
2.3 UI or XML Code	5
2.4 Back-end or Java Code	8
2.4.1 Interpreting Inputs	9
2.4.2 Reset Buttons	10
2.5 Communicating Information	11
3. Firmware	13
3.1 Setting up the Environment	13
3.2 Configuring the Project	14
3.2.1 Set “Build Target”	14
3.2.2 Configuring Desired Features	15
3.2.3 Editing the Makefile	16
3.2.4 Setting the Hardware Manifest	16
3.3 Writing the Driver	17
3.4 Note on the Vendor ID and Product ID	18
3.5 Building the Firmware	19
3.6 Flashing the Firmware	19
4. Moto Mod Circuits	19

4.1	Voltage Doubler	20
4.2	Comparators	20
5.	Conclusion	21
6.	References	22
	Appendix. Additional Resources	23
	List of Symbols, Abbreviations, and Acronyms	26
	Distribution List	27

List of Figures

Fig. 1	a) Back of the Moto Z smartphone and b) components of the MDK (reprinted with permission from Motorola)	1
Fig. 2	Organization of the XMD and the PiHAT personality card (image on left reprinted with permission from Motorola)	2
Fig. 3	X-Microwave prototype plate with RF circuitry	3
Fig. 4	X-Microwave jumpers, anchors, and probes (reprinted with permission from X-Microwave)	4
Fig. 5	The UI of the app	6
Fig. 6	Activity_main.xml in design mode	6
Fig. 7	content_main.xml in design mode	7
Fig. 8	card_mod_controller_sp4t.xml in design mode	8
Fig. 9	Implementation of the onClick function	11
Fig. 10	Listener and callback function for the SPDT radio group	12
Fig. 11	Menuconfig UI	15
Fig. 12	Code defining GPIO pins added to mods.h	17
Fig. 13	Circuit for one control line; in practice, one voltage doubler's output is fed to six comparator circuits, one for each control line	20

List of Tables

Table 1	Description of each bit in REF_MOD_MESSAGE = [00000000, 00000000]	11
Table 2	Additions to the manifest	17

1. Introduction

1.1 Motorola Moto Z Smartphone

The Motorola Moto Z is an Android-based smartphone that is uniquely modular. Various devices, called Moto Mods, can be attached to the back of the Moto Z with the imbedded magnets and expand the Moto Z's functionality. The devices include camera attachments and additional batteries and communicate to the phone through metal contacts placed on the back of the phone (Fig. 1a). One of these devices, the Moto Mods Development Kit (MDK)¹ is a programmable microprocessor embedded into a plastic case that attaches to and communicates with the phone² (Fig 1b).

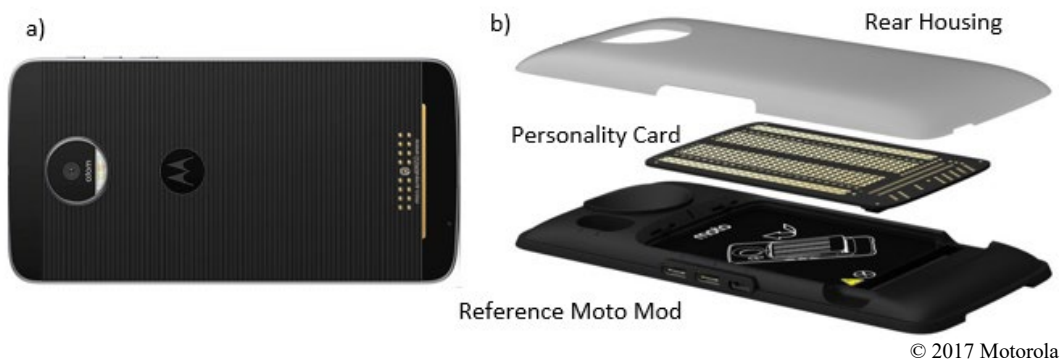


Fig. 1 a) Back of the Moto Z smartphone and b) components of the MDK (reprinted with permission from Motorola¹)

The MDK consists of the Reference Moto Mod, which contains the microprocessor and all of the computing power, an interchangeable personality card, and a plastic rear-housing piece. The MDK allows for the development of devices that expand the functionality of the Moto Z in unique ways. A proposed system would use RF microelectromechanical systems (MEMS) to create a mobile RF communication system. Such a system could theoretically be implemented and controlled by an MDK device, allowing the system to be fully integrated into the Moto Z by simply attaching the MDK device onto the Moto Z.

1.2 X-Microwave Demo

The X-Microwave Demo (XMD) app serves as a proof of concept of such a device. The XMD consists of an Android application, firmware for the microprocessor in the MDK, and a control circuit (Fig. 2). The demo allows the control of US Army Research Laboratory (ARL) RF MEMS switches and, for comparison,

commercially available RF switches. The switches are controlled through the Android application portion of the XMD app while the firmware and the circuit handle the necessary complexities needed to translate user input into various switch logic. Furthermore, in lieu of the standard personality card (pictured in Fig. 1b) the XMD uses a PiHAT personality card.

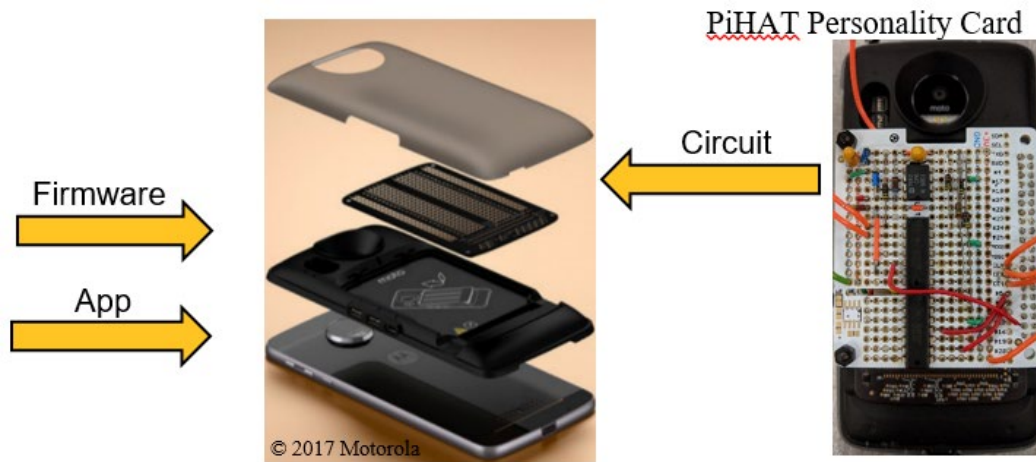


Fig. 2 Organization of the XMD and the PiHAT personality card (image on left reprinted with permission from Motorola¹)

PiHATs are commonly used with Raspberry Pi devices. Somewhat comparable to Moto Mods, PiHATs attach to the top of Pi devices via the general-purpose input/output (GPIO) pins and expand the functionality. However, the Raspberry Pi community is *much* larger than the Moto Mods community. As such, the resources available and scope of functionality developed for PiHATs and Pi devices are vast, especially compared with that of the somewhat-lacking Moto Mods. However, Pi devices lack the portability, familiarity, and existing touch screen that is inherent to a smartphone-based device such as a Moto Mod. While the personality card included with the MDK is simply a perforated board that can be used to integrate circuitry, the PiHAT card (Fig. 2) models the structure and layout of these PiHAT devices. Using the PiHAT personality card instead of the standard perforated board allows for access to the superior PiHAT community while only sacrificing some protoboard space.

1.3 X-Microwave Hardware

The RF hardware's demonstration platform is built off an X-Microwave prototyping kit with X-Microwave Blocks (Fig. 3).³ The blocks are bolted to the prototyping plate but designed to be modular and can quickly be added and removed from for rapid testing of RF–millimeter-wave circuits up through 67 GHz. Blocks are specialized coplanar waveguide (CPW) boards containing individual

hardware components. They are connected together by jumpers that are held down with anchors (Fig. 4). Signals are coupled to the blocks through probes. X-Microwave has a standard library of commercial off-the-shelf (COTS) components, can add COTS components to the library on request, and can make customized blocks to mount specialized components such as ARL's packaged RF MEMS devices. If desired, blocks can be fitted with SubMiniature version A (better known as "SMA") connectors so if a component is accepted after testing, it can be integrated into an existing RF chain.

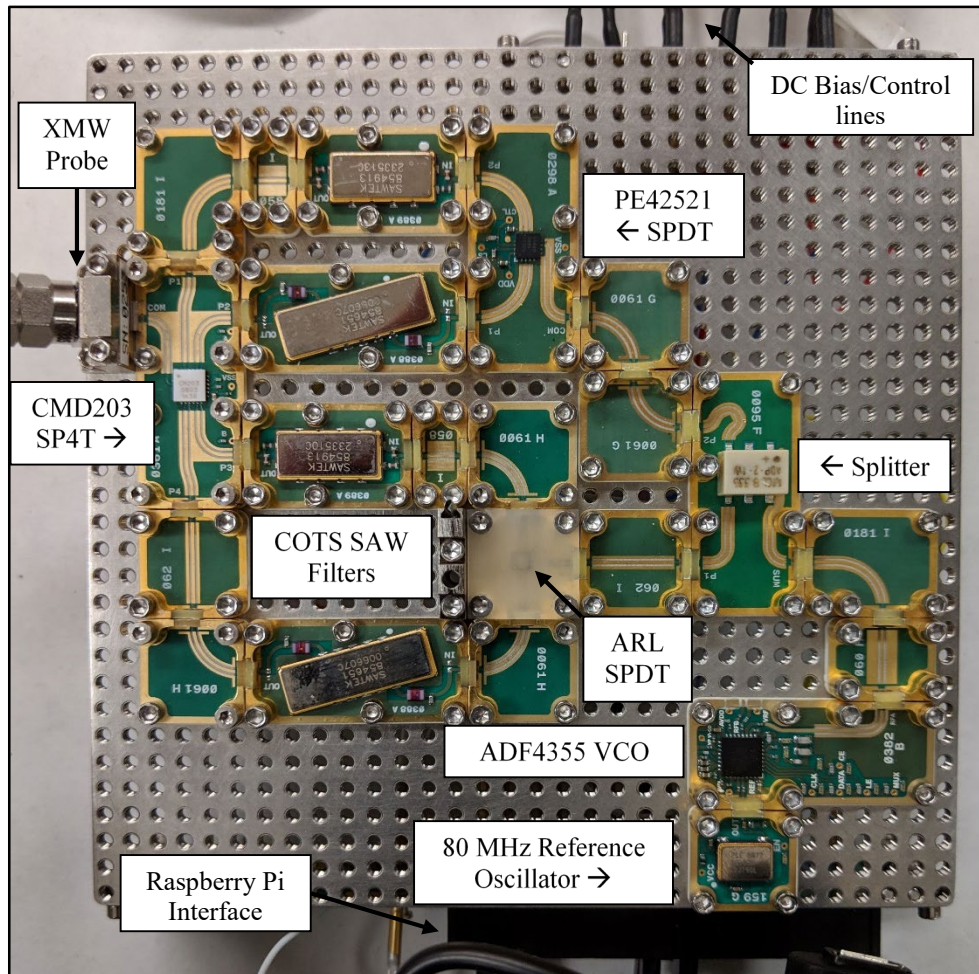


Fig. 3 X-Microwave prototype plate with RF circuitry

The demonstration platform uses an ADF4355 voltage-controlled oscillator (VCO) that can generate RF signals up to 4.4 GHz. The VCO is referenced by a SM77D 80-MHz oscillator (PLE Electronics) and operated using a RaspberryPi 3 controller outfitted by X-Microwave. The VCO is set to output a 70-MHz tone (meant to simulate an intermediate-frequency signal) and also generates a second harmonic at 140 MHz. The two tones are then passed to a -3 -dB splitter so the signals can be

passed to ARL's piezoelectric RF-MEMS-single-pole-double-throw (SPDT) switch and a PE42521 SPDT. The switches can then select between a signal path to either a 70-MHz surface-acoustic-wave (SAW) or 140-MHz SAW filter (both manufactured by Sawtek) so that one tone or the other can be filtered from the signal. A CMD203 SP4T (by Custom MMIC) switch can then be used to select the desired output signal to be viewed on a spectrum analyzer to compare the performance of the ARL switch versus the COTS part. The ability to easily swap out components makes this an ideal platform for comparing the performance of ARL's technology to what is commercially available.



Fig. 4 X-Microwave jumpers, anchors, and probes (reprinted with permission from X-Microwave⁴)

Future upgrades to this hardware demonstration could include replacing the COTS filters with ARL's piezoelectric filters or the CMD203 with a piezoelectric SP4T switch, both of which were unavailable in package form at the time of this writing. Further, as more advanced COTS components are released to the market, they can also be swapped in place of the existing COTS parts so that ARL's MEMS components are always showcased against state-of-the-art components.

2. XMD App Development

2.1 Brief Introduction to Android Development and Android Studio

The application or app is the portion of the software that is loaded onto the phone itself. The app is a traditional smartphone app and was developed in Android Studio, although it is not necessary to develop the app in Android Studio. The app has two primary goals: 1) act as the user interface (UI) receiving inputs from the user and 2) communicate the user's input to the reference Moto Mod. As one would expect from a Google product, there is a large and active community surrounding both Android software and Android Studio—meaning, most problems encountered when working with the app have already been solved and posted, in detail, online.

The app for the XMD was adapted from the open-source MDK Utility app provided by Motorola. (The MDK Utility app can be downloaded for free from the Google Play store and the source code can be accessed online.¹) Android development can roughly be divided into two categories: front end and back end (i.e., the portion the user sees and the behind-the-scenes functionality). Although not a rule, for the most part the UI is developed in Extensible Markup Language (XML) and the actual functionality is implemented with Java. The XML code can be found in the folder “res\layout” and, as stated, contains the code implementing the UI. The Java back-end or functionality code can be found in the Java folder. While it is useful to know XML, it is not entirely necessary when using an integrated development environment (IDE) such as Android Studio. However, some familiarity with Java is essential. There are numerous excellent resources online for learning Android, Android Studio, Java, and XML, so the rest of this document focuses on the specifics of this app.

2.2 Installing the Moto Mods Software Development Kit (SDK)

The app uses functions that are provided online by Motorola in the Moto Mods SDK library.²

If the SDK is not properly installed the app will not build. To install the app, simply copy the downloaded .jar file to /app/libs and the .xml file to the /app/src/main/res/values folders in the Android Studio project folder.

2.3 UI or XML Code

The XML code describe the visual aesthetics of the UI (Fig. 5). It determines which components appear and how they are laid out. As stated, this code is found in the “res\layout” folder and should contain the following:

- activity_main.xml
- card_general_info.xml
- card_mod_controller_spdt.xml
- card_mod_controller_com.xml
- card_mod_controller_sp4t.xml
- content_main.xml

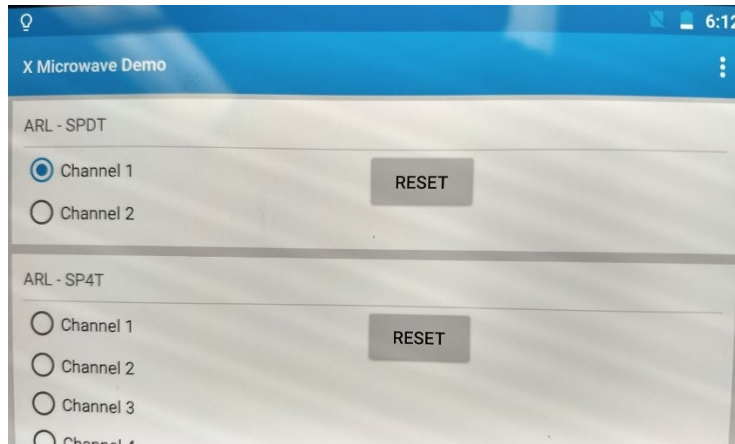


Fig. 5 The UI of the app

As with all Android Studio projects, `activity_main.xml` will be the where the code “begins”. When opening the XML in Android Studio, you can select the Text or Design view in the bottom left of the window. The autogenerated XML created by Android Studio can easily be trusted for something as simple as this. So, when making changes in Android Studio it is recommend that the Design tab is used instead of the Text tab.

Figure 6 shows there is only the toolbar and an `<include>` in `activity_main` (in the “Component Tree”, lower left). Naturally, if changes need to be made to the toolbar, make them here; otherwise, move on to the included file `content_main.xml`.

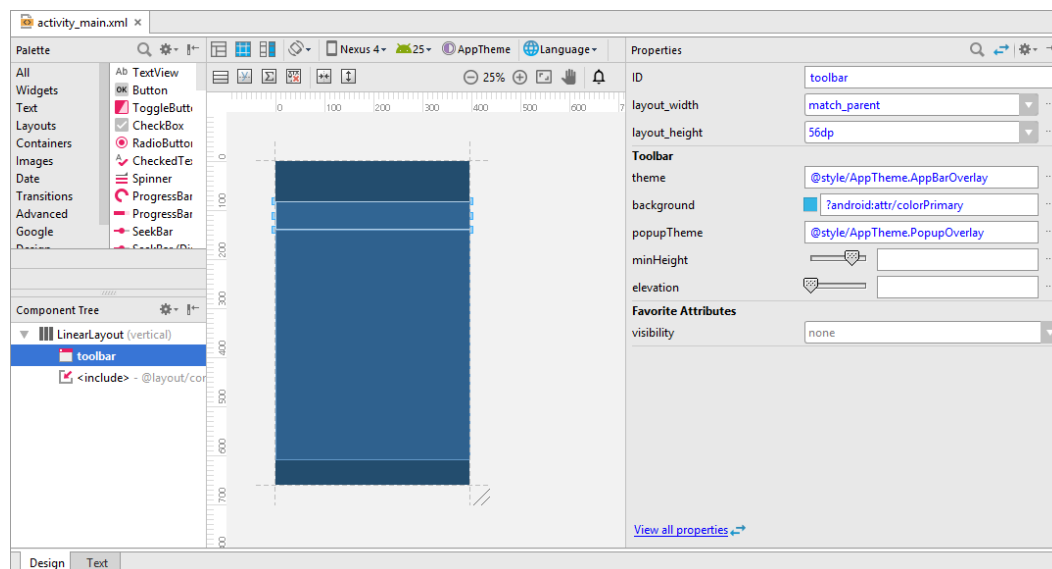


Fig. 6 Activity_main.xml in design mode

In Fig. 7, the `content_main` consists of (lower left of screenshot) four `<include>` laid out vertically and inside a scroll view. Having the four `<include>` underneath

the ScrollView component ensures the four can be “scrolled down”. These four `<include>` represent the four individual cards that are seen in the app: one card represents the SPDT controls, one represents the single pole four throw (SP4T) controls, one represents the commercial-device controls, and the last card communicates the state of the binary messages sent from the phone to the reference mod. Here one can change the cards’ order by adjusting the four `<include>` in the component tree. For example, if one wants the general-information card at the top, simply move the `<include>` - `@Layout/card_general_info` to the top in the component tree and the general-information card will appear at the top.

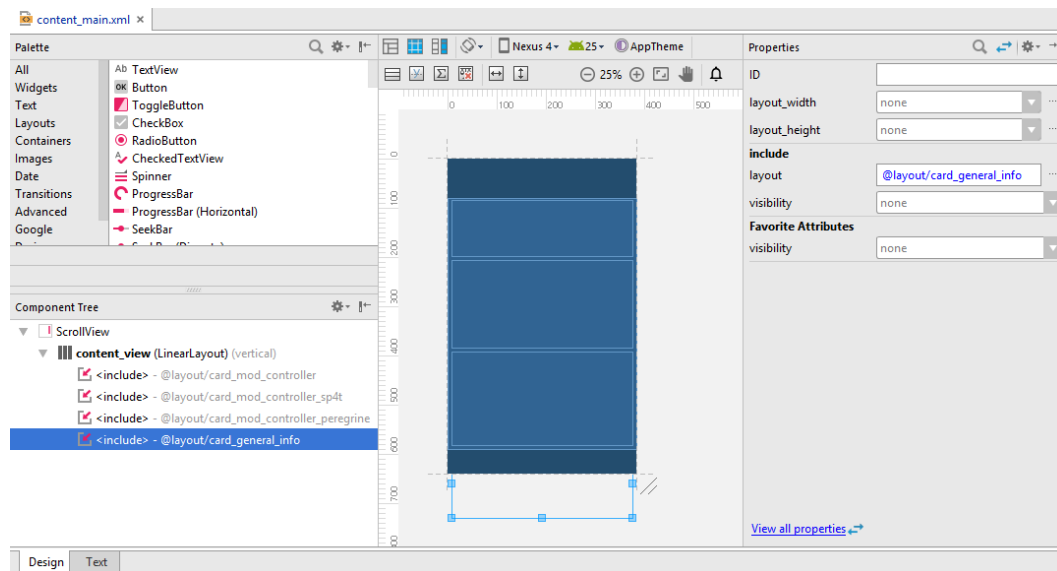


Fig. 7 `content_main.xml` in design mode

In Fig. 8, the `card_mod_controller_sp4t` file is shown. The “card_...” files contain all of the important functional components (i.e., the switches and the buttons). Radio buttons are used to represent the various channels of the switches (i.e., the SPDT has two radio buttons dedicated to it labeled Channel 1 and Channel 2). Each switch’s radio buttons are grouped with the radio-group component. Grouping the radio buttons in this way automatically forces the condition that only one button can be on at a time and allows the radio buttons to be referenced together as a radio group. The SPDT and SP4T have button components that reset the state of the switches. Finally, the general-information card has a button that resets all of the switches and also displays the current state of all the switches in binary form.

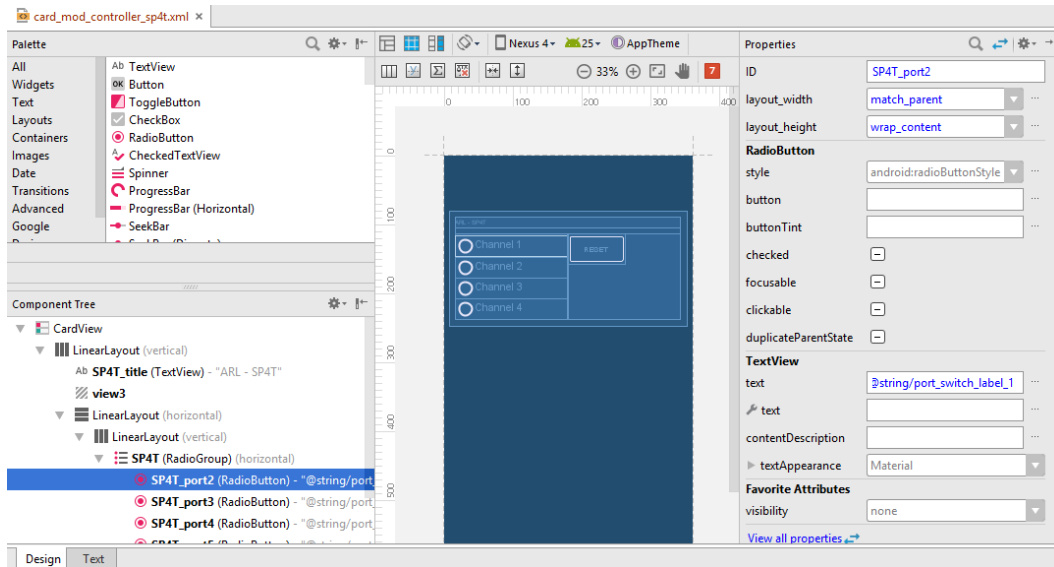


Fig. 8 card_mod_controller_sp4t.xml in design mode

The individual layouts for each card can be adjusted as needed and should not affect functionality as long as the individual component IDs (in the Properties tab) are *not* changed. If additional buttons, switches, or other components need to be added the new components *must* have unique IDs so that they can be referenced by the back-end Java code. The other properties for the components and new components could provide some useful functionality, such as automatically enabling/disabling components or implementing a timer that automatically shuts off the voltages after some time.

Finally, in the file “res/values/strings.xml” there is a list of strings that the XML code references. Essentially, this file contains all of the text that is displayed in the app and the XML code refers to this file when determine what text to display. There is no *need* to collect all of the strings into this one file and the various text can simply be edited at each individual component. However, with all of the components referencing this file, all of the text in the app can be conveniently edited with this single file.

2.4 Back-end or Java Code

The Java code is in the \Java folder. The function of the app is to take the user’s input and communicate it to the reference mod. The XML implemented the aesthetics and format of the UI; the Java code implements the functionality. The Java code must be able to recognize the user’s intent and communicate this intent to the reference mod.

2.4.1 Interpreting Inputs

In the case of this app, the user communicates their intent by clicking the radio buttons. As discussed, each switch (SPDT, SP4T, and commercial) has its own radio group. Each radio group contains a radio button to represent each channel of the switch. Furthermore, a number of buttons exist that should reset the switches. First, the Java code must be able to recognize what buttons are pressed when. This is achieved with callback functions and listeners.

Much of UI development relies on these types of functions. Essentially all of the components displayed in the UI are treated as an objects in Java. These objects come with functions that get run when various events happen, called “callback functions”, or listen for events, called “listeners”. For example, a button object has a callback function called `onClick`, which would be executed when a button is clicked. The function `onCheckedChangeListener`, which listens for any type of change to the radio button, is an example of a listener. However, these functions must first be initialized.

In `MainActivity`, there is a function called `onCreate`. Here components needing initialization are initialized, including callbacks and listeners. The radio group listeners are initialized as follows:

```
RadioGroup name = (RadioGroup) findViewById(R.id.XML_ID)

Name.setOnCheckedChangeListener(listenerName)
```

The first line identifies which radio group you are referring to and requires the ID set for the radio group in the XML. The next line initializes the On Checked Change Listener. The two lines above effectively state that when the radio group identified by `XML_ID` is changed, call the function `listenerName`. The function `listenerName` must be implemented later, outside the `onCreate` function. The initialization of the Button objects is similar:

```
Button name = (Button) findViewById(R.id.XML_ID)

Name.setOnClickListener(this)
```

The primary difference being the second line. While the radio group initialization specified `listenerName` as the function to be executed, the button initialization does not specify a particular function. Instead, when the button identified by `XML_ID` is clicked the default callback function `onClick` will be executed. The result is that each radio group has its own unique listener that is called when the radio group is changed and each button calls the same function when clicked.

The listeners are implemented just outside the onCreate function. They look as such:

```
private RadioGroup.OnCheckedChangeListener listenerName = new
    RadioGroup.OnCheckedChangeListener () {

    public void onCheckedChanged(RadioGroup group, int checkedId) {
        ...
        int port1 = R.id.XML_ID_1;
        int port2 = R.id.XML_ID_2;

        if (checkedId == port1)
        {...}
        else if (checkedId == port2)
        {...}
        ...
    }
}
```

This states that when the listener “listenerName” detects a change it runs the onCheckedChanged callback defined inside it. Note that the callback function has an input that defines the radio group (RadioGroup group) and an input that defines which radio button in the group is checked (int checkedId). Thus, inside the callback the IDs for the radio buttons can be compared to the ID of the currently checkedId to determine which button was selected, seen above in the if-else statement. Recall that the radio groups represent each of the switches. The end effect is that when the user checks a radio button, say Channel 1 of the SPDT, the listener for the SPDT group runs its respective onCheckedChanged callback. Then the callback compares the IDs of the Channel 1 and Channel 2 radio buttons to the checkedId variable to determine that Channel 1 was checked.

2.4.2 Reset Buttons

The reset buttons behave slightly differently. As discussed, the buttons forgo the individual listeners and simply use the default listener, which calls the callback function onClick. The implementation of this function is shown in Fig. 9. The onClick function has the input of View v. The ID of the button that was clicked can then be found with v.getId(). Then, the ID can be compared to the IDs for each of the buttons to determine which button was clicked. Now that the app can identify what the user did, it can react accordingly.

```

392 //On any button click
393 public void onClick (View v)
394 {
395     RadioGroup rg = (RadioGroup) findViewById(R.id.SPDT);
396     RadioGroup rg4t = (RadioGroup) findViewById(R.id.SP4T);
397     RadioGroup pere = (RadioGroup) findViewById(R.id.pere);
398     RadioGroup mmic = (RadioGroup) findViewById(R.id.mmic);
399
400
401     switch (v.getId()) {
402
403         case R.id.SPDT_reset:
404             rg.clearCheck();
405             break;
406
407         case R.id.SP4T_reset:
408             rg4t.clearCheck();
409             break;
410
411         case R.id.reset_all:
412             rg.clearCheck();
413             rg4t.clearCheck();
414             pere.clearCheck();
415             mmic.clearCheck();
416             break;
417     }
418 }
419

```

Fig. 9 Implementation of the onClick function

2.5 Communicating Information

For ease of implementation, a single global variable is maintained and edited as necessary, then sent to the reference mod. In the file Constants.java, there is a public array of two 8-bit binary strings called REF_MOD_MESSAGE. This is what is sent to the reference mod. The layout of REF_MOD_MESSAGE is shown in Table 1. This approach is used so that when the user communicates a change, the back-end code only has to update REF_MOD_MESSAGE as necessary and then tell the service manager to send out the message.

Table 1 Description of each bit in REF_MOD_MESSAGE = [00000000, 00000000]

Byte	8 (MSB ^a)	7	6	5	4	3	2	1 (LSB ^b)
Group	Reserved	Comm. SPDT		ARL SP4T				ARL SPDT
Byte 0	Bit	NA	CTRL 1 ^c	Ch. 4	Ch. 3	Ch. 2	Ch. 1	Ch. 2
	Group		Unused					Comm. SP4T
Byte 1	Bit	CTRL 2	CTRL 1

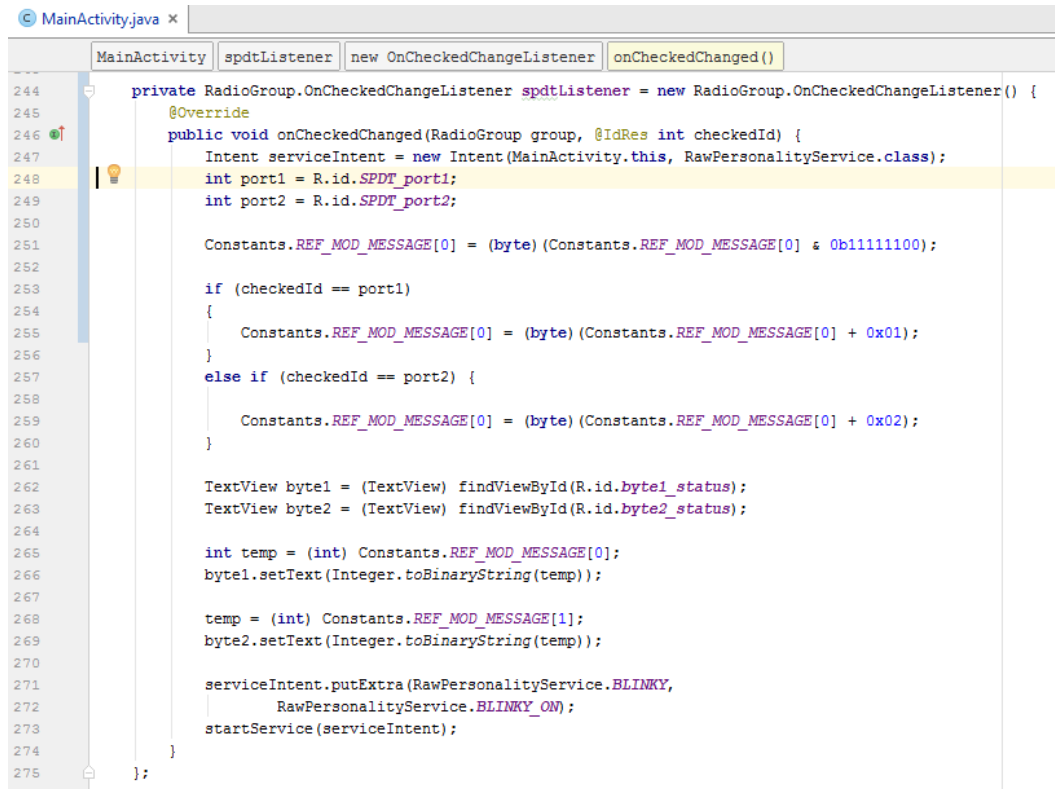
^a MSB: most significant bit

^b LSB: least significant bit

^c CTRL: control

Table 1 describes what each bit controls. In the first byte, Byte 0, the ARL switch controls and some of the controls for the commercial SPDT and SP4T are stored and sent. For the ARL switches, each channel has a dedicated bit with 1 representing closed and 0 representing open. However, the commercial switches contain built-in control schemes. For the commercial SPDT, a single bit is used because 0 represents Channel 1 being closed and Channel 2 being open while 1 represents the reverse. Similarly, the commercial SP4T is controlled with two bits. Note that the only information stored and transmitted in Byte 1 is the second control bit for the commercial SP4T. RAW_MOD_MESSAGE is edited and sent to the reference mod by the callback functions. Note that the MSB cannot be changed.

Figure 10 shows the listener and callback function for the SPDT radio group. The first line (246) in the callback function initializes a new service intent based off the RawPersonalityService class. The RawPersonalityService class is a class developed by Motorola designed to handle the communication to the reference mod. The next two lines get the IDs for the two SPDT radio buttons and the if-else statement determines which radio button was checked, as previously discussed. Next, the bits representing the SPDT are reset to zero by edited REF_MOD_MESSAGE Byte 0 with a mask. Inside of the if-else, the REF_MOD_MESSAGE is edited depending on which button was checked.



```

244 private RadioGroup.OnCheckedChangeListener spdtListener = new RadioGroup.OnCheckedChangeListener() {
245     @Override
246     public void onCheckedChanged(RadioGroup group, @IdRes int checkedId) {
247         Intent serviceIntent = new Intent(MainActivity.this, RawPersonalityService.class);
248         int port1 = R.id.SPDT_port1;
249         int port2 = R.id.SPDT_port2;
250
251         Constants.REF_MOD_MESSAGE[0] = (byte) (Constants.REF_MOD_MESSAGE[0] & 0b1111100);
252
253         if (checkedId == port1)
254         {
255             Constants.REF_MOD_MESSAGE[0] = (byte) (Constants.REF_MOD_MESSAGE[0] + 0x01);
256         }
257         else if (checkedId == port2) {
258             Constants.REF_MOD_MESSAGE[0] = (byte) (Constants.REF_MOD_MESSAGE[0] + 0x02);
259         }
260
261         TextView byte1 = (TextView) findViewById(R.id.byte1_status);
262         TextView byte2 = (TextView) findViewById(R.id.byte2_status);
263
264         int temp = (int) Constants.REF_MOD_MESSAGE[0];
265         byte1.setText(Integer.toBinaryString(temp));
266
267         temp = (int) Constants.REF_MOD_MESSAGE[1];
268         byte2.setText(Integer.toBinaryString(temp));
269
270         serviceIntent.putExtra(RawPersonalityService.BLINKY,
271             RawPersonalityService.BLINKY_ON);
272         startService(serviceIntent);
273     }
274 }
275

```

Fig. 10 Listener and callback function for the SPDT radio group

For example, say the user checks the Channel 1 radio button for the SPDT. The callback function in Fig. 10 will be run and will identify that the Channel 1 radio button was checked. Recall that first the two SPDT bits are reset to zero. Then the code for Channel 1 will be run (if (checkedId == port1) and the hex value 0x01 will be added to RAW_MOD_MESSAGE. The result of 0bXXXXXX00 + 0b00000001 is 0bXXXXXX01. As seen in Table 1, if Channel 1 in the SPDT group is checked the message should include a 1 in the LSB in Byte 0 and a 0 in the second bit and the other bits should be unaffected. Thus, the user's input is successfully translated into the RAW_MOD_MESSAGE variable. The remaining code in the function simply updates the values being displayed in the general info card and the last three lines send the message to the reference mod.

The onClick function (Fig. 9) for the buttons is much simpler because it leverages the individual radio-button listeners. As seen, the identity of the button pressed is found with a switch statement. Then, the built-in clearCheck function is called for the desired radio group, which clears all checks for the radio group; that is, if the user clicks the SPDT reset button, the switch statement will run the SPDT code and call clearCheck on the SPDT radio group, clearing any checked radio button. Since listeners listen for *any* changes to their respective object, the listener and then the callback (Fig. 8) for the respective radio group will then be called. Then, as designed, the radio-group callback will clear the bits for the radio group. However, since no radio buttons will be checked, the if-else will be skipped and nothing will be added to RAW_MOD_MESSAGE. The result is the message is cleared and sent to the reference mod.

3. Firmware

3.1 Setting up the Environment

The XMD firmware is loaded onto the reference mod. The purpose of the firmware in this application is to read the information sent to the reference mod by the Android app and turn on/off the correct GPIO pins. The XMD firmware is based off the firmware provided by Motorola located on its github.⁵ Specifically, the XMD firmware is based off the firmware developed by Motorola for the “Hello World!” or “blinky” mod.⁶ The current code for the firmware, drivers, and includes can be found in the following:

```
~/Documents/MotoMods/XCOMdemoFirmware  
BUILD_TOP/nuttx/nuttx/configs/hdk/muc/src  
BUILD_TOP/nuttx/nuttx/configs/hdk/muc/include
```

Motorola’s guide for setting up the environment⁷ walks through the setup of the Android environment and the firmware-development environment. Not all of the tools are required but they are useful. For example, OpenOCD allows for flashing the firmware to the reference mod and GNU Debugger (commonly, “GDB”) allows for debugging. (A Linux environment is necessary for firmware development.) After the tools are installed, follow this guide to download the necessary code from github, install the configuration editor, and compile the firmware.⁸ If the nuttx does not compile and mentions an error regarding “chmod 755 .version”, then it is missing the .version file. (At the time this report was being written, there was no mention of that file on Motorola’s website; however, after some research the specifics of the file were discovered.) To resolve this error, create a file in the BUILD_TOP/nuttx/nuttx directory called .version; then edit the text to match the text in Section 2 of the Appendix. Once one has verified that the boot loader and the nuttx firmware compile, the environment is ready to go.

3.2 Configuring the Project

A significant portion of developing the firmware relies on configuring the project appropriately. This includes setting the build target, enabling and disabling desired features, adding new files to the makefile, and setting the hardware manifest.

3.2.1 Set “Build Target”

Configuring desired features can be time consuming. So, to speed the process various configurations are stored in BUILD_TOP/nuttx/nuttx/configs/hdk/muc. Using the script configure.sh allows for quick switching between these configurations or *build targets*. For example, to switch to the build target defined in the folder BUILD_TOP/nuttx/nuttx/configs/hdk/muc/example/, simply execute the following command in the BUILD_TOP/nuttx/nuttx/tools folder:

```
./configure.sh hdk/muc/example
```

This command copies the configuration definitions (defconfig), makefile definitions (Make.defs), and a build script (setenv.sh) in the folder BUILD_TOP/nuttx/nuttx/configs/hdk/muc/example/ up to the BUILD_TOP/nuttx/nuttx/folder.

Naturally, if the project does not have a build-target folder, one needs to make one. Motorola has provided two base build targets: base_unpowered and base_powered. Simply copy one of those folders and rename to the desired project name. The XMD build target was based off the blinky target, which is based off the powered base.

3.2.2 Configuring Desired Features

After setting the build target with `configure.sh`, one can edit the configuration files as necessary. Executing

`make menuconfig`

in the `BUILD_TOP/nuttx/nuttx` folder opens a UI (Fig. 11) that allows control of which features and functionality are enabled and disabled. After making changes with `menuconfig`, the UI will prompt to save the configuration as `defconfig`. This will obviously overwrite the current `defconfig`, specifically the version in `BUILD_TOP/nuttx/nuttx/`. If one makes changes to the configuration one wants to save, be sure to copy the file from `BUILD_TOP/nuttx/nuttx/` back to the folder it came from.

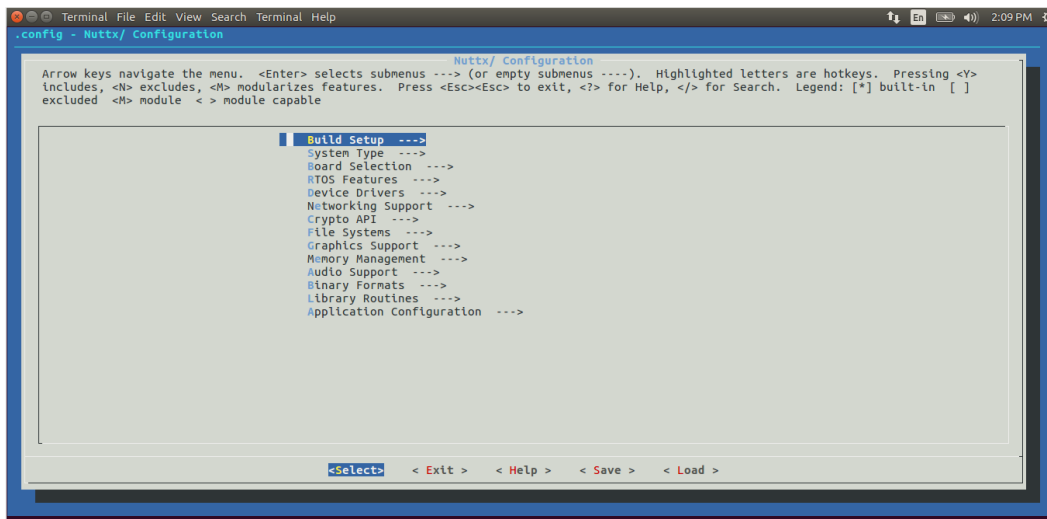


Fig. 11 Menuconfig UI

Any features enabled in the base build target should be left enabled. For the XMD firmware, Greybus RAW protocol must be enabled. This is found in “Device Drivers -> Greybus Support -> Vendor Raw Support”. If further functionality is desired, be sure that the functionality is enabled here.

The `menuconfig` can be expanded to include custom options. This can be done by editing the file `Kconfig`, which is found in `BUILD_TOP/nuttx/nuttx/configs/hdk/muc/Kconfig`. The XMD firmware adds the following to the `Kconfig` file:

```

config MODS_RAW_BLINKY
    bool "Blinky LED Mods Raw support"
    default n
    depends on GREYBUS_RAW
    select DEVICE_CORE
    select STM32_TIM6
    ---help---
        Enable Blinky LED Raw support

```

Including this in the Kconfig file specifies dependencies and other functionality that should be enabled and includes “Blinky LED Mods Raw support” as a configurable option in menuconfig under “Board Selection”. For the XMD, the “Blinky LED Mods Raw support” must be enabled. Lastly, the user *can* directly edit the defconfig file instead of using the menuconfig.

3.2.3 Editing the Makefile

Makefiles execute the various compile and link commands; the firmware for the XMD project has a number of makefiles. In particular, the makefile in BUILD_TOP/nuttx/nuttx/configs/hdk/muc/src must be edited to ensure any new driver files are properly built. Adding

```

ifeq ($(CONFIG_MODS_RAW_BLINKY),y)
    CSRCS += stm32_modsraw_blinky.c
endif

```

ensures the new XMD file stm32_modsraw_blinky.c gets compiled and linked. The “if” statement informs the makefile that this file only needs to be configured if MODS_RAW_BLINKY, defined previously in Kconfig (Section 3.2.2), is configured. The “if” statement is not necessary but is recommended.

3.2.4 Setting the Hardware Manifest

Finally, a hardware manifest for the project is necessary. As with the build target files, there are base versions provided in BUILD_TOP/nuttx/apps/greybus-utils/manifests, hdk.mnfs and hdk-powered.mnfs. For the XMD, the powered base is used and is renamed hdf-blinky.mnfs. Code must be added to the manifest. First, String-descriptor 1 is changed; this is simply a title and has no effect on functionality. Then the new raw interface and bundle must be set up, as shown in Table 2.

Table 2 Additions to the manifest

String-descriptor 1	Bundle and raw support
; Interface vendor string (id can't be 0) [string-descriptor 1] string = Motorola Mobility, LLC	; Battery related Bundle 2 [bundle-descriptor 2] Class = 0 × 08 ; RAW interface on CPort 4 [cport-descriptor 4] bundle = 3 class = 0xfe ; RAW Bundle 3 [bundle-descriptor 3] class = 0xfe

Now this manifest must be set as the active manifest. Once again this can be done in the menuconfig UI. In particular, this setting is found at Application Configuration -> Greybus Utility. Here a manifest name can be specified and then selected; a predefined manifest must be set to: Custom Manifest.

3.3 Writing the Driver

Now that the necessary features are enabled and the makefile includes the new file, the code implementing the desired functionality, `stm32_modsraw_blinky.c`, must be written. However, first add a few useful definitions.

In `BUILD_TOP/nuttx/nuttx/configs/hdk/muc/include` a few header files are stored. As shown in Fig. 12, the GPIO pins used in the XMD are defined for future use. Pins on the reference mod are specified with a letter and a number, such as Pin PG10 or Pin G 10. How the reference mod pins tie to the PiHAT pins are detailed in a description of the PiHAT Adapter Board from Motorola.⁹ Nine control lines in total are needed to operate the switches on the XMD prototype plate. Figure 12 shows that Pins C3, A1, H0, G12, G10, G9, A15, A4, and A5 are used as the controls. These correspond to PiHAT Pins 38, 35, 34, 31, 32, 30, 25, 23, and 24, respectively, and are given labels `GPIO_XCOM_DEMO_1` to `GPIO_XCOM_DEMO_9` for use in the firmware code. If the new pins need to be added or the current pins need to be adjusted, make those changes here.

```

/* ARL XCOM demo pins*/
#define GPIO_XCOM_DEMO_1    CALC_GPIO_NUM('C', 3) //Pin 38
#define GPIO_XCOM_DEMO_2    CALC_GPIO_NUM('A', 1) //Pin 35
#define GPIO_XCOM_DEMO_3    CALC_GPIO_NUM('H', 0) //Pin 34
#define GPIO_XCOM_DEMO_4    CALC_GPIO_NUM('G', 12) //Pin 31
#define GPIO_XCOM_DEMO_5    CALC_GPIO_NUM('G', 10) //Pin 32
#define GPIO_XCOM_DEMO_6    CALC_GPIO_NUM('G', 9) //Pin 30
#define GPIO_XCOM_DEMO_7    CALC_GPIO_NUM('A', 15) //Pin 25
#define GPIO_XCOM_DEMO_8    CALC_GPIO_NUM('A', 4) //Pin 23
#define GPIO_XCOM_DEMO_9    CALC_GPIO_NUM('A', 5) //Pin 24

```

Fig. 12 Code defining GPIO pins added to `mods.h`

The file `stm32_modsraw_blinky.c` acts as the driver for the GPIO pins and implements the bulk of the functionality, although functionality begins in the header file `device_raw.h`, which is found in `BUILD_TOP/nuttx/nuttx/include/nuttx`. An exhaustive description of this code is not necessary as editing it is ill advised. Suffice it to say the code allows for the user to define their own raw message-receive function, such that the defined function will be called when a raw message is received and will be passed the message in question.

In `stm32_modsraw_blinky.c`, the bottom few lines of code, Lines 127 and on, implement a few structs. These structs are of type defined in the header files `device_raw.h` and `device.h`. The purpose of these structs is to define the receive, callback, and so on functions as such. For example, in the `device_raw_type_os` struct the `.register_callback` is defined to be `blinky_register_callback`. In effect, this states that the function `blinky_register_callback` will function as the callback to the event of the reference mod registering with the phone and as such will be called when this event occurs. Note that to define the function as such the function must take the same inputs and return the same type as defined in the various header files. The most important line here is `.recv = blinky_recv`, which defines the function `blinky_recv` as the raw message-receive function (i.e., `blinky_recv` will be called when the reference mod receives a raw message and will be passed the message).

The definition of `blinky_recv` is at the top of the file. Notice that the output type and inputs are exactly as `device_raw.h` defines they should be for a receive function. The inputs are the device identifier, the length of the message, and the data of the message. The function first checks that the data actually have some length and were not received in error. Then if-else statements and bit masks are used to check the state of each individual bit. Naturally, if a bit is set to 0 the code turns off the corresponding GPIO pin, and if it is set to 1 the GPIO pin is turned on.

3.4 Note on the Vendor ID and Product ID

The Vendor ID (VID) and Product ID (PID) are, as the names suggest, hex values identifying the firmware (PID) and the creator (VID). They can be configured in the menuconfig UI under System Type. It should be noted the Android app is set up so that it may not function unless the correct VID and PID are set, namely,

VID = 0x00000042
PID = 0x00000001

In the app, these values are set in the file `Constants.java`. The same file that contains the `REF_MOD_MESSAGE` variable.

3.5 Building the Firmware

The firmware can be easily built by following the previous guide: <https://developer.motorola.com/build/tools/build-from-source>.⁸

Care should be taken to ensure the proper build target is set via the `configure.sh` script. (At the time of this writing, the proper build target should be the `blinky` folder.) Also, the “make distclean” command will remove the discussed `.version` file from the `BUILD_TOP/nuttx/nuttx` folder and the file will need to be re-added. No changes should have been made to the bootloader, however, and building it should have no adverse effects.

3.6 Flashing the Firmware

The firmware can be loaded onto the reference mod, or flashed, by using the OpenOCD tool previously downloaded. A guide for it can be found at the bottom of the page at <https://developer.motorola.com/build/tools/flashing-firmware>.

The process is straightforward. First, be sure that the USB Type C is plugged into the USB port on the reference mod, not the port on the phone. Then, if using a Virtual Box for the Linux system, be sure the port is forwarded to the Virtual Box. In the Virtual Box window on the bottom right, there is a symbol of a USB plug; clicking this allows one to forward various devices to the Virtual Box operating system. The reference mod should appear as FTDI Quad RS232-HS. To load the firmware and bootloader, simply execute the commands as shown:

From the `muc-loader` directory:

```
openocd -f board/moto_mdk_muc_reset.cfg -c "program
./out/boot_<target_name>.bin 0x08000000 reset exit"
```

From the `nuttx/nuttx` directory:

```
openocd -f board/moto_mdk_muc_reset.cfg -c "program nuttx.tftf
0x08008000 reset exit"
```

4. Moto Mod Circuits

The Moto Mod circuit (Fig. 13) is necessary because the reference mod can only output up to 3.3 V on the controllable GPIO pins and the ARL switches require approximately 10 V. The circuit has two goals: 1) generate 10 V from the available voltages and 2) use the logic generated by the reference mod to switch the 10 V for each ARL switch’s control line. The net effect of the circuit is to convert 0- to 3.3-V logic to 0- to 10-V logic.

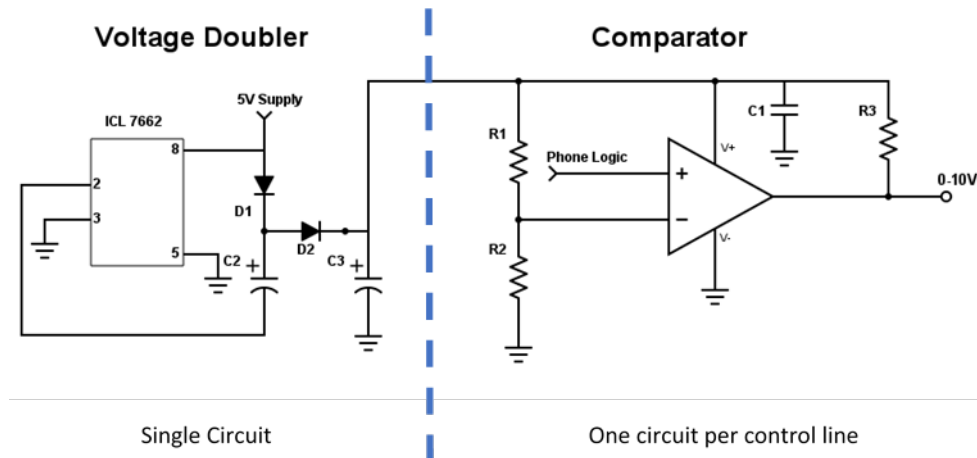


Fig. 13 Circuit for one control line; in practice, one voltage doubler's output is fed to six comparator circuits, one for each control line

4.1 Voltage Doubler

Unsurprisingly, this circuit *roughly* doubles the input voltage. The circuit uses the ICL 7662 complementary metal–oxide–semiconductor voltage converter. (The circuit is described in the documentation for the ICL 7662.) Fortunately, the reference mod has a couple 5-V supply pins, theoretically allowing the circuit to generate 10 V. However, the two diodes in the circuit each have a 0.7-V operating voltage. Thus, at most the circuit can produce 8.6 V with actual measurements ranging from 8.2 to 8.5 V, which is not ideal but sufficient. Future work could look at using lower voltage diodes.

4.2 Comparators

In Fig. 13 only one comparator is shown, but in application a comparator is necessary for each 0- to 10-V control line. The comparator compares the positive input to the negative input. When the positive input is greater than the negative input, the comparator outputs the positive supply voltage; when the reverse is true, it outputs the negative supply voltage. Thus, if one compares the reference mod logic (0–3.3 V) to a constant, say 1.6 V, then the comparator will output V+ when the control line is high and V– when the control line is low. The 1.6 V can easily be provided with a voltage divider. The desired behavior is then easily achieved by setting V– to 0 V and V+ to 10 V, or 8.5 V as in this case. The capacitor, C1, simply stabilizes the DC supply and the resistor R3 acts as a pull-up resistor.

5. Conclusion

The development of a demonstration system for RF MEMS switches and filters has been presented. A customized smartphone modification was used to demonstrate the functionality of the RF circuitry and directly compare the size, weight, and performance of the MEMS devices to COTS components. The components are mounted on an X-Microwave prototype kit and controlled by a Motorola Moto Z smartphone through use of a Moto Mod. The user interface, hardware circuitry, and control software have all been described. (The Appendix lists several resource websites.) Future work could be done to integrate the RF circuits from the prototype kit into the Moto Mod along with antennas to transmit the signal of interest to a network analyzer for a truly handheld platform.

6. References

1. MotoMods. Libertyville (IL): Motorola Mobility LLC; c2017 [accessed 2017 Aug 10]. <https://developer.motorola.com/>.
2. Moto Z. Libertyville (IL): Motorola Mobility LLC; c2018 [accessed 2017 Aug 10]. <https://www.motorola.com/we/products/moto-z>.
3. Company Overview. Austin (TX): X-Microwave; c2018 [accessed 2017 Aug 10]. <https://www.xmicrowave.com/about/company-overview/>.
4. Products: X-cessories. Austin (TX): X-Microwave; c2018 [accessed 2017 Aug 10]. <https://www.xmicrowave.com/products/x-cessories/>.
5. Motorola Mobility. San Francisco (CA): GitHub, Inc.; c2018 [accessed 2017 Aug 10]. <https://github.com/MotorolaMobilityLLC/>.
6. Hello World! Libertyville (IL): Motorola Mobility LLC; c2017 [accessed 2017 Aug 10]. <https://developer.motorola.com/build/examples/hello-world>.
7. Setup Environment. Libertyville (IL): Motorola Mobility LLC; c2017 [accessed 2017 Aug 10]. <https://developer.motorola.com/build/tools/setup-environment>.
8. Build from Source. Libertyville (IL): Motorola Mobility LLC; c2017 [accessed 2017 Aug 10]. <https://developer.motorola.com/build/tools/build-from-source>.
9. HAT Adapter Board. Libertyville (IL): Motorola Mobility LLC; c2017 [accessed 2017 Aug 10]. <https://developer.motorola.com/build/mdk-user-guide/hat-adapter-board>.

Appendix. Additional Resources

This appendix appears in its original form, without editorial change.

Approved for public release; distribution is unlimited.

1) Firmware Resources:

Main Page:

<https://developer.motorola.com/>

General Guides (Setup, Building, Flashing):

<https://developer.motorola.com/build/tools>

Example code is based on:

<https://developer.motorola.com/build/examples/hello-world>

Expanded Example:

<https://www.element14.com/community/groups/moto-mods/blog/2017/04/22/moto-mods-developer-part-1-getting-started-virtual-machine-setup-and-linux-install>

PiHAT Guide:

<https://developer.motorola.com/build/mdk-user-guide/hat-adapter-board>

Forum:

<https://www.element14.com/community/groups/moto-mods>

2) Necessary and missing .version file:

```
#!/bin/bash
```

```
CONFIG_VERSION_STRING="7.10"  
CONFIG_VERSION_MAJOR=7  
CONFIG_VERSION_MINOR=10  
CONFIG_VERSION_BUILD="85981b37acc215ab795ef4ea4045f3e85a49a7af"
```

3) App Resources:

Good Intro:

<https://developer.android.com/training/basics/firstapp/index.html>

Android:

<https://developer.android.com/guide/>

Android Studio:

<https://developer.android.com/studio/intro/>

<https://www.raywenderlich.com/154676/android-studio-tutorial-introduction>

Approved for public release; distribution is unlimited.

XML and Java:

https://www.w3schools.com/xml/xml_what_is.asp

<https://www.ibm.com/developerworks/learn/java/intro-to-java-course/index.html>

General/Debugging:

<https://stackoverflow.com/> - Stack Overflow is excellent.

List of Symbols, Abbreviations, and Acronyms

ARL	US Army Research Laboratory
COTS	commercial off-the-shelf
CPW	coplanar waveguide
CTRL	control
DC	direct current
GPIO	general-purpose input/output
ID	identification
IDE	integrated development environment
LSB	least significant bit
MDK	Moto Mods Development Kit
MEMS	microelectromechanical systems
MSB	most significant bit
PID	product ID
RF	radio frequency
SAW	surface acoustic wave
SDK	software development kit
SP4T	single pole four throw
SPDT	single pole double throw
UI	user interface
USB	Universal Serial Bus
VCO	voltage-controlled oscillator
VID	vendor ID
XMD	X-Microwave Demo
XML	Extensible Markup Language

1 DEFENSE TECHNICAL
(PDF) INFORMATION CTR
DTIC OCA

2 DIR ARL
(PDF) IMAL HRA
RECORDS MGMT
RDRL DCL
TECH LIB

1 GOVT PRINTG OFC
(PDF) A MALHOTRA

1 GEORGIA TECH
(PDF) L GRIFFIN

5 ARL
(PDF) RDRL SER L
R BENOIT
R POLCAWICH
RDRL SER W
C DIETLEIN
M HIGGINS
RDRL VT
B PIEKARSKI