ENGR-E 399/599: Embedded systems reverse engineering
Indiana University Spring 2021
Prepared by: Austin Roach Assigned: March 25, 2021

Assignment 5: Networked speaker

*Due date:*April 30, 2021

For this assignment, you will interact with an embedded system that uses an unknown network protocol. The system is a speaker that can play audio that is streamed to it over a network connection. Your goal is to understand the protocol in order to interact with the speaker, making it play the audio of your choice.

This system is relatively complex. Noting that the goal of your reverse-engineering effort is to determine details of the protocol in order to allow interoperation with this system, you will focus your efforts to only analyze the parts of the firmware that are relevant to understanding the application-layer protocol that this system uses. You can use dynamic analysis techniques to find the part of the firmware that matters to you, then use a combination of dynamic and static analysis techniques to learn details of the protocol.

0.1 References

- OpenOCD user guide: http://openocd.org/documentation/
- STM32F427 datasheet (included in assignment repository)
- ST RM0090 reference manual (included in assignment repository)
- ST PM0214 programming manual (included in assignment repository)
- ST NUCLEO-XXXXZX schematic MB1137 (included in assignment repository)

0.2 Hardware setup

You should have the following hardware:

- ST Nucleo F429ZI development board
- USB-A to USB-micro-B cable
- Ethernet cable
- 4 male-to-female Dupont jumper wires
- Sparkfun Noisy Cricket stereo amplifier board
- Speaker with female Dupont connectors
- BeagleBone Green

Connect the USB-A end of the USB cable to the "USB Host" port of the BeagleBone Green (reference designator P3), and one end of the Ethernet cable to the "10/100 Ethernet" port of the BeagleBone Green (reference designator P5).

The micro-B end of the USB cable connects to the "USB PWR" port on the Nucleo F429ZI (reference designator CN1). This USB connection provides power to the board, an interface to the ST-Link debug adapter, an interface to a USB-to-serial device, and a virtual USB mass storage device that is used for firmware uploads. The virtual USB mass storage device is not useful to you for this assignment, and you should not interact with it. The ST-Link debug interface is the primary debug interface used for this assignment, but the USB-to-serial interface may also be useful to you.

The second end of the Ethernet cable connects to the "Ethernet" port on the Nucleo F429ZI (reference designator CN14). The board is configured with a static IPv4 address of 192.168.2.10/24. You should configure the Ethernet port of your BeagleBone Green with an address in the same local subnet, for example with the following command from a BeagleBone Green terminal:

```
sudo ip address add <IPv4 address>/24 dev eth0
```

replacing `<IPv4 address>` with your address of choice.

Jumper wires are used to connect the Nucleo F429ZI to the Noisy Cricket audio amplifier board. The jumper wires carry power and the output of the DAC to the amplifier board. The wire ends of the speaker also connect to the amplifier board. A wiring schematic is shown in Figure 1. An image of the boards with their jumper wire connections is shown in Figure 2. An image including the connections to the BeagleBone Green is shown in Figure 3.

The potentiometer on the audio amplifier board is used to control the level of amplification. If it is turned counter-clockwise to its limit, the output is completely disabled. Turning clockwise increases the level of amplification. It is possible, but unlikely, that the amplifier board could draw more power than the Nucleo F429ZI can provide. If this happens, LED LD5 on the Nucleo F429ZI, labeled "Overcurrent", will shine a bright, steady red. Turn down the amplification if this happens.

When the board is powered on or reset (using the "Reset" button, reference designatore B2), the speaker should play a series of three test tones. This is the indicator to you that the audio portion of the system is operating correctly.

0.3 OpenOCD setup

You will use `openocd` to interact with the development board for this assignment. `openocd` is already installed on your BeagleBone Green. If you would prefer to run `openocd` on your personal computer or virtual machine, you may, but the instructions here will be written assuming you are connecting from the BeagleBone Green.

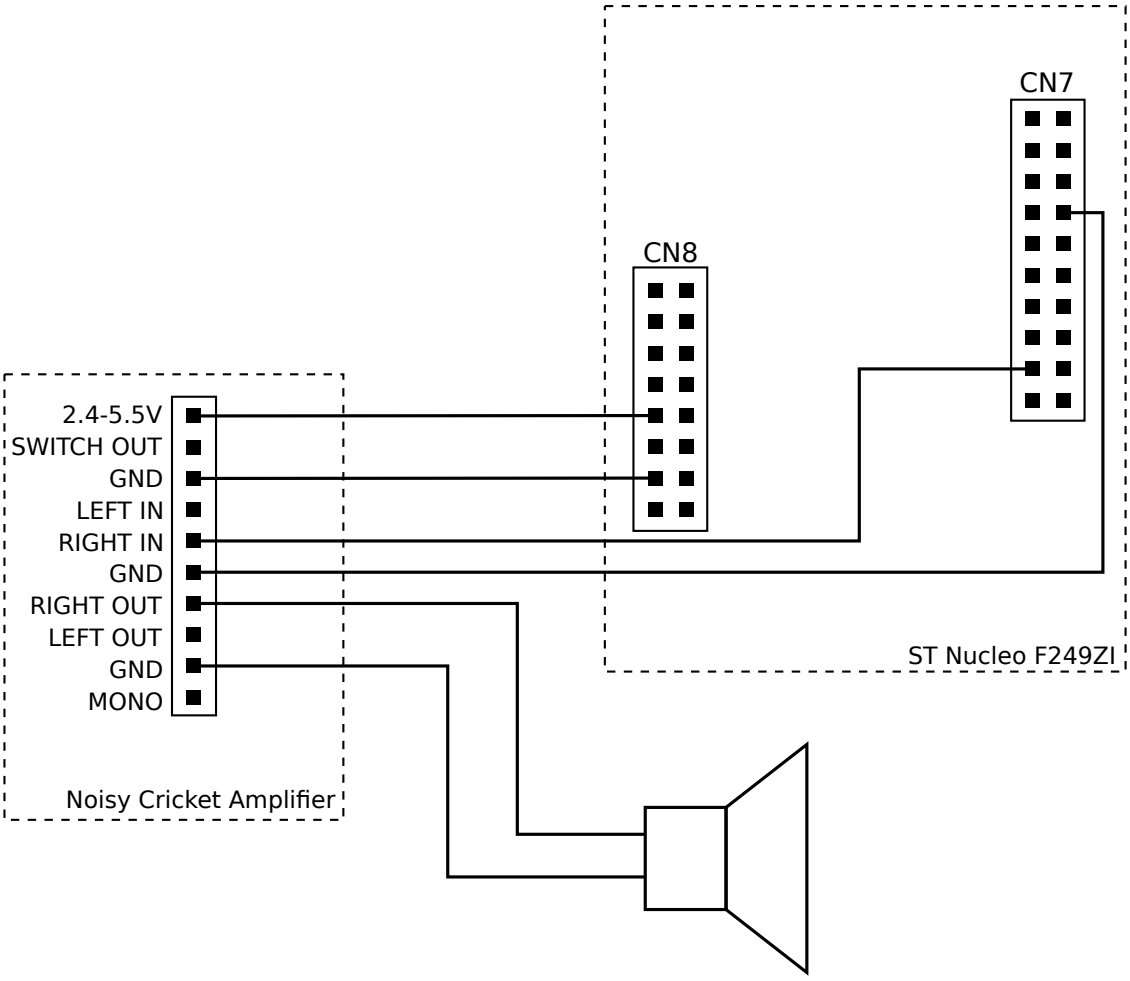

Figure 1: Wiring schematic for the Nucleo F429ZI, Noisy Cricket stereo amplifier board, and speaker

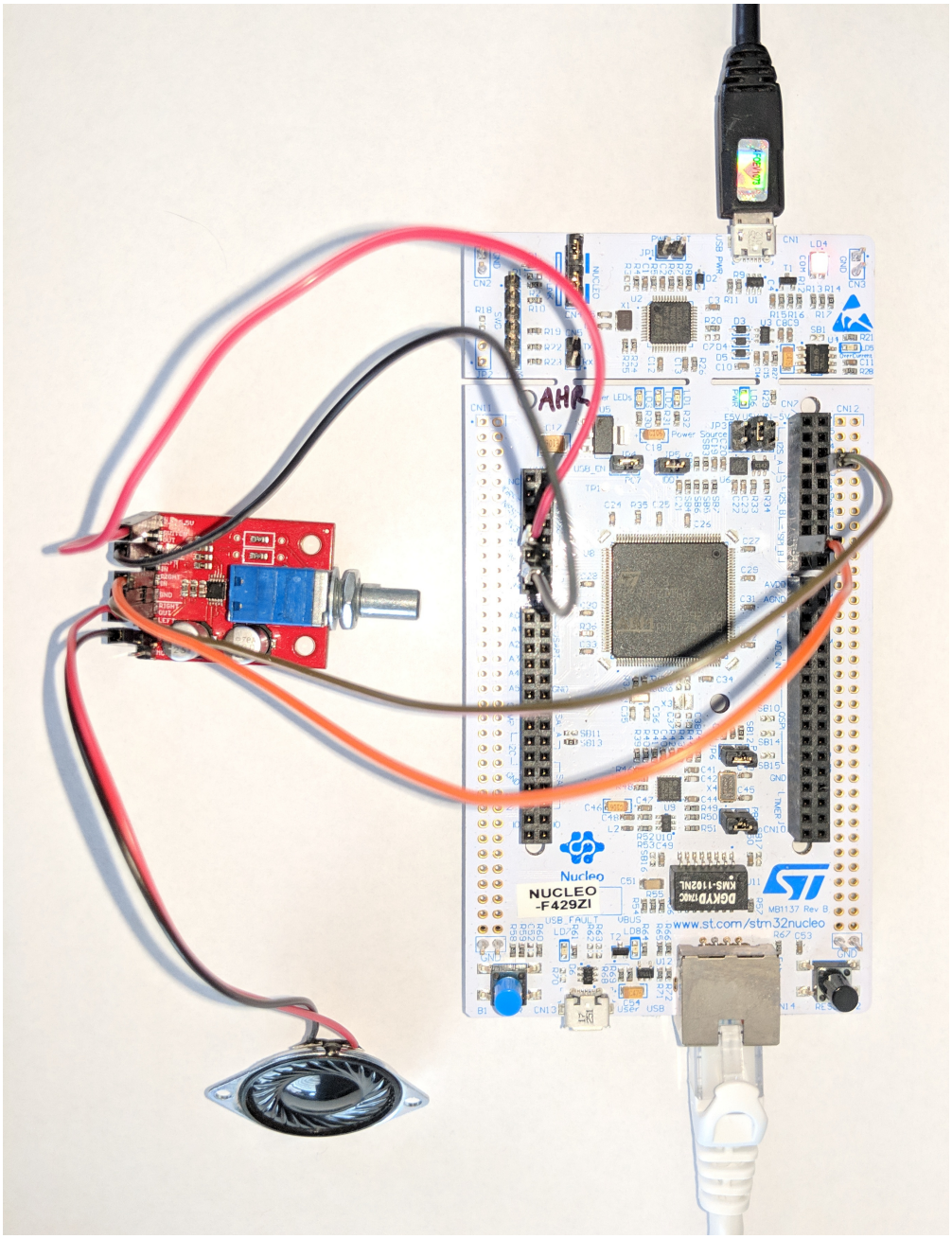

Figure 2: Image of the Nucleo F429ZI, Noisy Cricket stereo amplifier board, and speaker with jumper wires in place

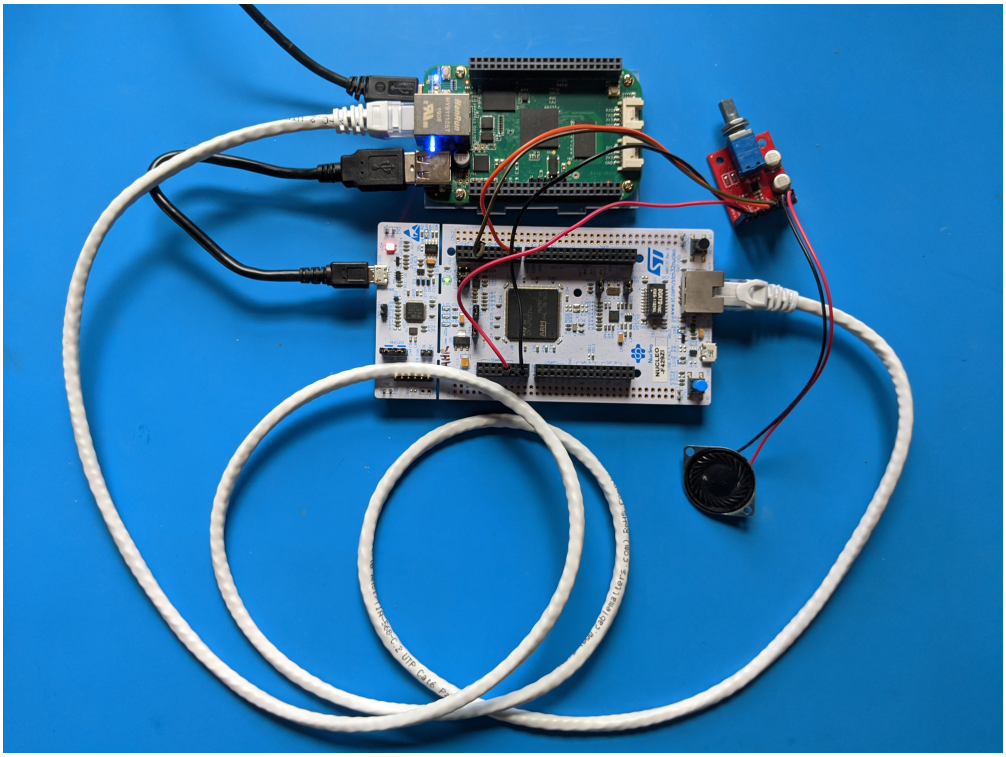

Figure 3: Image of the entire project setup, including connections between the Nucleo F429ZI and the BeagleBone Green

When the BeagleBone Green is connected by USB to the Nucleo-F429ZI, you can connect to the debug interface in the following way:

1. Run

   `openocd -f board/st_nucleo_f4.cfg`

   in one BeagleBone Green terminal.

2. From another terminal on the BeagleBone Green, connect to `openocd` over telnet:[1]

   `telnet localhost 4444`

You can interact with `openocd`'s command-line interface in the telnet session.

0.3.1 OpenOCD commands

The `openocd` command set is long and complex, and varies depending on the target with which `openocd` is interacting. The list of available commands can be found by issuing the `help` command from the command-line interface. A short list of useful commands is summarized here for your convenience:

```
help [command_name]
      Show full command help; command can be multiple tokens. (command
      valid any time)
bp <address> [<asid>]<length> ['hw'|'hw_ctx']
      list or set hardware or software breakpoint
rbp address
      remove breakpoint
wp [address length [('r'|'w'|'a') value [mask]]]
      list (no params) or create watchpoints
rwp address
      remove watchpoint
halt [milliseconds]
      request target to halt, then wait up to the specified number of
      milliseconds (default 5000) for it to complete
resume [address]
      resume target execution from current PC or address
step [address]
      step one instruction from current PC or address
reg [(register_number|register_name) [(value|'force')]]
      display (reread from target with "force") or set a register; with no
      arguments, displays all registers and their values
```

[1] Note that if you wish to connect to `openocd` from your PC or VM, you can add the `-c "bindto 0.0.0.0"` parameter when you invoke `openocd`. This will cause `openocd` to bind the listening socket to all network interfaces. Only do this if you know what you understand what you are doing.

```
arm disassemble address [count ['thumb']]
      disassemble instructions
mdb ['phys'] address [count]
      display memory bytes
mdw ['phys'] address [count]
      display memory words
mwb ['phys'] address value [count]
      write memory byte
mww ['phys'] address value [count]
      write memory word
dump_image filename address size
      dump memory contents to file
```

0.4 Ghidra import

A Ghidra script has been provided to you that will automate the naming of memory spaces and memory-mapped I/O registers. This script is named "st_f429zi_ghidra.py". You should copy this script into your `ghidra_scripts` directory.

Perform the following steps when analyzing extracted flash memory contents for this microcontroller:

- During import, select the ARM Cortex, little-endian processor variant for the Language
- In the import Options:
  - Assign a block name of 'flash'
  - Assign a base address corresponding to the address from which you extracted the flash memory contents
  - Deselect 'Apply processor defined labels'
  - Deselect 'Anchor processor defined labels'
  - Click Ok
- Click Ok to import the file
- Open the program in the Codebrowser, *but do not analyze it yet*
- Navigate to 'Window/Script Manager'
- If the script is not already in your script path, click the 'Script Directories' button in the upper right corner of the Script Manager window to add the path to the script
- Run the 'st_f429zi_ghidra.py' script

- Auto analysis should begin after you execute the script. If it does not, navigate to 'Analysis/Auto analysis' or press 'A' to execute the auto analysis scripts wih the default settings

The script should assign flags for interrupt vectors[2] and memory-mapped I/O registers. It should also provide a reasonable initial disassembly of functions in the flash memory region.

[2]Note that in this microcontroller the location of the interrupt vectors can be changed dynamically by the firmware by writing to the `VTOR` register.

1 Flash memory extraction: 3 points

The embedded flash memory in the STM32F429ZI microcontroller contains the firmware. If you know the address range to which the flash memory is mapped, you can extract the contents for analysis using `openocd`.

1.1 Determine flash memory location

Open the STM32F429ZI datasheet (stm32f427vg-956239.pdf). Navigate to the 'Memory Map' diagram, Figure 19 on page 86.

- What is the address of the beginning of the flash memory segment?
- What is the size of the flash memory segment?

1.2 Memory dump

Using the `openocd` connection instructions listed above, connect to the debug access port for the microcontroller's ARM core. Use the `dump_image` command in `openocd` to dump the flash memory, inserting the address and size that you determined from the datasheet. As part of your assignment submission, list the `openocd` command that you used to dump the flash contents.

Copy this image to your PC or VM for analysis in Ghidra.[3] Import the flash memory contents into Ghidra using the instructions provided above, and verify that you are seeing reasonable results.

[3] Using `scp`, for example.

2 Find the open TCP port: 2 points

First verify that you have correctly configured your BeagleBone Green's network interface using the instructions above. You should be able to ping the Nucleo F429ZI's IP address (192.168.2.10) from a terminal on the BeagleBone Green.

The BeagleBone Green has `nmap` installed. Use `nmap` to find the open TCP port. Which TCP port number is open? You can use a command of the form:

```
nmap -A -T4 -p- 192.168.2.10
```

3 Find the relevant part of the firmware: 5 points

The firmware from this microcontroller is too large to analyze manually in its entirety (∼50000 instructions). You need to find the specific area in the firmware that you care about so that you can perform a detailed analysis of only that region. To locate this area, you can send some data in over the network interface, find where that data is being stored in memory, and then set a watchpoint to determine which parts of the firmware are accessing that memory.

3.1 Find incoming data storage location

The STMF429ZI microcontroller has 192 KB of SRAM that could be used for data storage, not including the 64 KB of core-coupled memory (CCM).

From the memory map in the STMF429ZI datasheet, determine the address and size of the data SRAM region. Note these values on your assignment submission.

Next, send a data pattern of your choice to the TCP port that you discovered above. You may find it useful to use `netcat` do to do this. The following command would connect a TCP socket to IPv4 address 192.168.2.10 port 1234, then would send the piped string ("HELLO") through that socket:

```
echo "HELLO" | nc 192.168.2.10 1234
```

You should substitute the data pattern of your choice and the TCP port number that you found above.

Now, halt the processor and dump the SRAM contents using `openocd`'s `dump_image` command, providing as arguments to the command the SRAM data address and size you determined from the datasheet. In your assignment submission, list the full command that you ran to dump the SRAM contents.

Now search the memory dump for the first few bytes of your input pattern. You could use:

- `grep -b -o -a`
- `rafind2`
- `xxd` or `hexdump` and search the output
- A hex editor to load the file and search.

Your input might only appear once in the data area, and it could also appear at multiple locations if the data in the Ethernet device's incoming receive buffer has not yet been overwritten. Note also that only part of your input may have been copied to the data area, so consider searching for smaller substrings if your are not able to locate the full input. In your assignment submission, describe how you located your input in the memory dump.

Resume processor execution, then try this experiment several more times with a few different inputs. There should be one address where you can consistently find your input appearing in memory. List this address in your assignment submission. Remember to account for the offset from which your memory dump started when calculating this memory address. Also be sure to account for the radix of the offset where you found your input in the memory dump. (Did your search method return this offset in decimal, or in hexadecimal?)

3.2 Find the application-layer parser

Now that you know the memory address where our data is being stored, you can set a watchpoint to determine which instructions are accessing this location.

Halt the processor, set a watchpoint using `openocd`'s `wp` command on the address that you found above, then resume processor execution. Send some input to the TCP port, and if you have determined the address correctly you should see your watchpoint trigger.

Make note of the program counter address when the watchpoint triggers. Resume execution with the `resume` command, and you will see the watchpoint trigger again. Make note of both program counter addresses when the watchpoint triggers, including these program counter addresses in your assignment submission.

Look at the listing from the disassembled flash memory contents to see that one watchpoint trigger is caused by a `store` to our watched address, and the other trigger by a `load` to our watched address. The processor will have halted one to two instructions after the memory access instruction, so for each program counter address you will need to look backward in the disassembly listing one or two instructions to find the associated `store` or `load` instruction that accessed the watched address.

The `store` instruction is called by a subroutine that copies our input from the incoming Ethernet receive buffer to another part of memory where it will be accessed later. This firmware region is of little interest to us. The `load` instruction is near the beginning of the function that parses the input. What is the address of this load instruction, and what is the starting address of the function that it is a part of?

4 Protocol determination: 10 points

Now, through static and/or dynamic analysis, analyze the parser that you identified in the last section. Static analysis tends to be quicker if you can perform it efficiently, but dynamic analysis allows you to observe the execution of instructions to better understand parts of the code that might be confusing.

To perform dynamic analysis, you will probably want a program to construct messages, send them to the TCP port on the speaker, and receive and display the responses. You can then send messages to the speaker and step through the parsing of those messages to gain additional understanding of selected portions of the code. You may want to perform this activity as you are writing the client described in the next session, using your work-in-progress client to provide messages to help your analysis.

You should find that there are multiple commands that are accepted by the speaker. For each of these commands describe:

- What are the sizes of the various fields in the command message?
- What are any requirements on the values of certain fields for this command? (e.g. that a field must hold a certain constant value, or that a field can only contain a range of possible values)
- Can you identify how the data from particular fields are used? (What do these values *mean*?)
- What does each command do in the context of audio playback?
- What is the format of reply messages from the speaker?
- What information is included in replies from the speaker? (What do each of the reply fields mean?)

You can format this information however it seems reasonable to you. As an example, my description of two commands for the Arduino bootloader protocol might have included the following:

> The following command sets the address for program memory read and write commands:
>
> | **Field offset (bytes)** | **Field size (bytes)** | **Description** |
> |---|---|---|
> | 0 | 1 | Constant ‘U’ (0x55) |
> | 1 | 2 | Address (little endian) |
> | 3 | 1 | Constant 0x20 |
>
> The following command sends the data for a write command:

Field offset (bytes)	**Field size (bytes)**	**Description**
0	1	Constant ‘d’ (0x64)
1	1	Don’t care (any value acceptable)
2	1	Number of bytes to write
3	1	Don’t care (any value acceptable)
4	Varies	Data to write (size set by field at offset 2)
Varies	1	Constant 0x20 after last byte of data

For both commands, the Arduino Uno responds to the client by sending:

Field offset (bytes)	**Field size (bytes)**	**Description**
0	1	Constant 0x14
1	1	Constant 0x10

5 Write your own client: 5 points

Using what you know of the messaging protocol, write a client application to control the networked speaker. Your client should implement any commands that you discovered. At a minimum, it should implement the commands required to smoothly play an audio file on the speaker.

An audio file is provided for you in the `resources` directory of the repository. This file has no file header. Its contents are raw audio samples, stored as a sequence of 16-bit unsigned integers, little endian. The sample rate is 44100 Hz. This is the format of data that might be appropriate to send to a 12-bit DAC on a little endian system...

If you are unfamiliar with writing code using a Berkeley sockets interface, you may want to consult a tutorial for your programming language of choice, for example https://docs.python.org/3/howto/sockets.html for Python or https://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html for C.