Reaction Tester for Nintendo Gameboy

Cameron Smith

April 24 2020

Abstract

Reaction Tester for Nintendo Gameboy is a simple reaction tester built to run on a Nintendo Gameboy with the help of a Beaglebone Black Linux board. It is built on top of a modified version of RIOS to run on the Nintendo Gameboy. This paper will go over how the reaction tester works, a simple demonstration with screenshots, currently known bugs and where to go next.

0. Introduction - How does it work?

The Beaglebone will wait a random amount of time until telling the Gameboy to tell the user to press a certain button. Once completed, the Gameboy will send back the time elapsed from when it told the user to press the button and when the user pressed it. The Beaglebone will then display the time elapsed to the user.

It works by having the Beaglebone Black wait a random number of seconds from 1 to 15. After the elapsed time, the Beaglebone will then randomly generate a number from 0 to 7 and send the number to the Gameboy. Once the Gameboy has received the value, it will prompt the user to press the button corresponding with the value.

1. Roadmap

Section 2 explains the real time software used on the Nintendo Gameboy to get this reaction tester working. Section 3 explains how I was able to get the real time software running on the Nintendo Gameboy. Section 4 explains how I was able to get the Nintendo Gameboy to communicate with the Beaglebone Black Linux board. Section 5 will go over a simple demonstration of the program with screenshots. Section 6 will go over the currently known issues and bugs. Section 7 will touch on where I'd like to go next with this project.

2. Scheduler

The scheduler I used for the Nintendo Gameboy is a modified version of pre-emptive RIOS to run on the Nintendo Gameboy. RIOS, short for Riverside-Irvine Operating System, is a lightweight real time operating system developed by Frank Vahid and Bailey Miller of the

University of California Riverside and Tony Givargis of the University of California Irvine. RIOS is a very short piece of C code with a very easy to understand scheduler.

I got this running on the Nintendo Gameboy with the help of the Game Boy Development Kit, which is often referred to as GBDK. The Gameboy Developer's Kit is a collection of tools available for Windows/DOS and Linux developed by David Galloway, Michael Hope, Sandeep Dutta, and Daniel Rammelt. Tools in this kit that I used include the C compiler, allowing me to compile C code into a binary that the Gameboy will run, and the libraries, which allow me to avoid using assembly code and instead use C.

I have RIOS set up to call the interrupt service routine 256 times per second. I have a single tick function whose job is to check and see if the user has pressed the correct button.

3. Running Software on Nintendo Gameboy

The Nintendo Gameboy runs video games on read only cartridges, so how was I able to run software on it? There exists unofficial cartridges on the market that allow you to run Gameboy binaries off an sd card. I happened to have purchased one called the EZ-FLASH Junior. Using the GBDK, I was able to compile my code into a gameboy binary that I can then put onto the SD card of the EZ-FLASH Junior.

Frequently exchanging new binarys every time you made a change would be time consuming, so I also tested my code on a Nintendo Gameboy Emulator. There are many available but I chose to use one called BGB, available for Windows or Linux running Wine. This emulator is useful because not only can you run your Gameboy binaries on your desktop, it also includes a handful of useful debugging tools, such as a vram viewer.

4. Connecting Gameboy to Beaglebone Black

This was perhaps one of the most time consuming parts of this entire project. I needed a way for the Nintendo Gameboy and the Beaglebone Black to communicate between each other. To accomplish this, I used the Nintendo Gameboy Link Cable. Both ends of this link cable have a proprietary port to connect a Gameboy to another Gameboy. Since the Beaglebone doesn't have this port connector, I had to cut open the cable, identify the wires and feed them into the Beaglebone Black. For a Nintendo Gameboy Link Cable, there are 4 primary wires; serial in, serial out, clock and ground. I happened to do this project with a Gameboy Color. The Nintendo Gameboy Color, which is a slightly upgraded Nintendo Gameboy, uses a different type of link cable that includes 2 extra wires inside. I wasn't able to figure out what these wires were for, but I ended up getting the project working without them.

I soldered each individual wire inside the link cable to a thicker, more reliable wire. On the Beaglebone, I fed the serial-in wire into an input pin, the serial-out wire into an output pin and the clock wire into an output pin. The Nintendo Gameboy outputs at 5v, but the Beaglebone Black accepts inputs of 3v. To avoid frying the Beaglebone, I purchased a logic level converter that converts 5v to 3v.

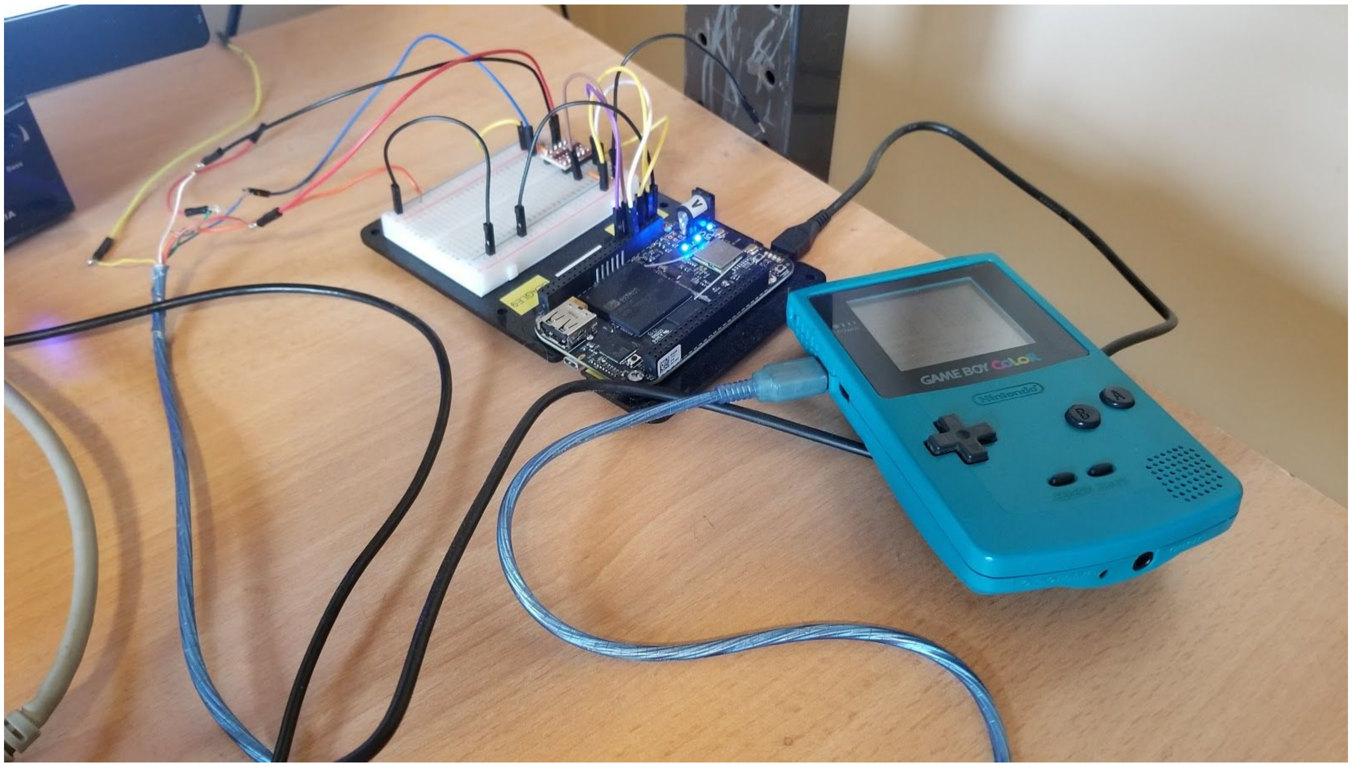


Image demonstrating how I hooked up the Gameboy to the Beaglebone Black

To transfer a bit from the Beaglebone to the Gameboy, I set the serial out pin either high or low depending on what I want to send. Afterwards I set the clock pin low, sleep for a short amount of time, set the clock pin high and sleep for a short amount of time again. For each bit transferred to the Gameboy, the clock will go low then high to tell the Gameboy the value on the serial out pin can be grabbed. Transferring a byte is the same idea, but done eight times.

While I was able to figure out how to send a value from the Beaglebone to the Gameboy, it was more difficult to figure out how to send a byte from the Gameboy to the Beaglebone. I was able to figure out how I could send a single bit to the Beaglebone, but it was difficult sending more than just a single bit because there is nothing distinguishing whether or not I have sent one bit or more than one bit. The solution I came up with relies heavily on timing. If I want to send a 0 out on the line, I would make the line high for a specific short amount of time, once the time has elapsed I will make the line go low. On the Beaglebone's end, it will record how long that line stays high for. Depending on the elapsed time, the Beaglebone will determine whether or not the value was a 1 or a 0 and record it.

Here is some example code of how I was able to pull this off:

On the Gameboy:

```
// sending a 1 on the line for .2 seconds == high bit
void sendHighBit(){
    sendBit(1);
    delay(200);
    sendBit(0);
}

// sending a 1 on the line for .4 seconds == low bit
void sendLowBit(){
    sendBit(1);
    delay(400);
    sendBit(0);
}
```

On the Beaglebone:

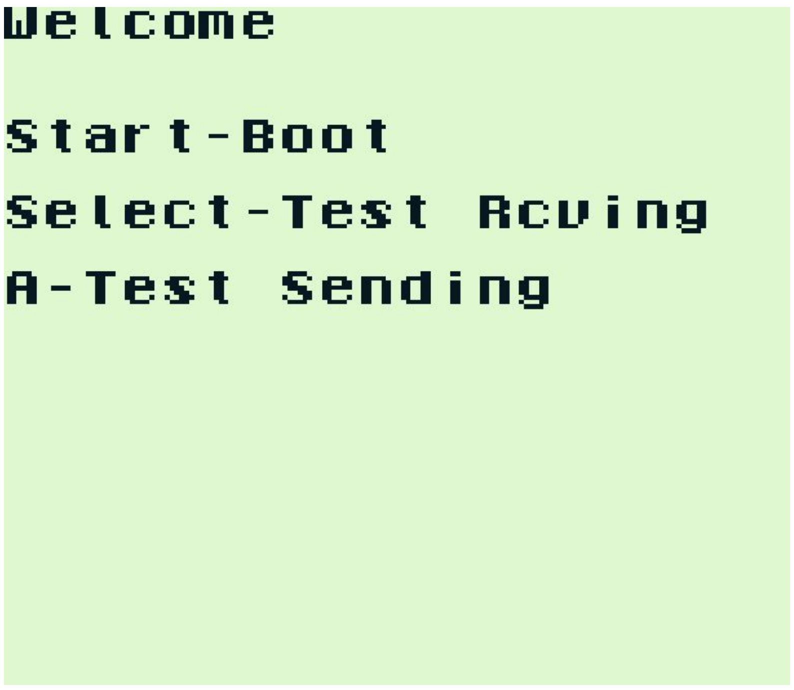
```
def receiveBitValue():
      bitToReturn = -1;
      finished = 0;
      timeOutCount = time()
      # keep checking the serialInPin for an input
      # once it goes high, count long it stays high for
      # depending on time elapsed, it will be a 1 or 0
      # if nothing comes in for 5 seconds, time out
      while finished == 0:
         bitReceived = receiveBit();
         # once we got the bit, record how long it stays on the line
         # this will determine whether it's a 1 or 0
         if bitReceived == 1:
            # start the timer, stop the timer once a 0 comes on the line
            start = time()
            while bitReceived == 1:
                  bitReceived = receiveBit()
            timeElapsed = time() - start
```

```
		#print "Time elapsed is: ", timeElapsed, "\n"
		# check to make sure there are no weird errors
		if timeElapsed > 1 or timeElapsed < .2:
			# if anything weird/unexpected comes down the line
			# label it 3, 3's will be ignored later
			bitToReturn = 3
		elif timeElapsed > .4:
			bitToReturn = 0
		else:
			bitToReturn = 1
		finished = 1
	currTime = time() - timeOutCount
	if currTime > 5:
		#print "time out\n"
		return bitToReturn

	return bitToReturn;
```

5. Demonstration

When you first boot up gbReact.gb on a flash cart, you are greeted with this screen.

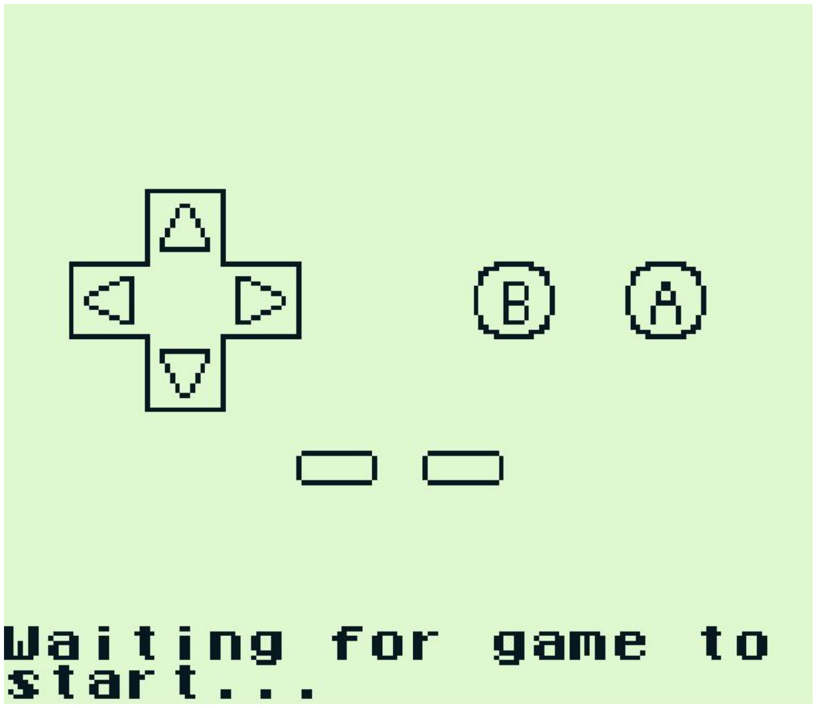


```
Welcome

Start-Boot
Select-Test Rcving
A-Test Sending
```

Pressing Start will boot the reaction tester as normal. Pressing Select will test the receiving functions. Pressing A will test the sending functions.

Pressing Start will bring you to the following screen.


Waiting for game to start...

An important note of significance is that the Gameboy will wait until you run the gbReact.py script, but if you run the gbReact.py script first, then the script doesn't wait on the Gameboy and simply runs. So you must make sure to have the Gameboy running and ready to go before running the python script.

The reaction tester will now wait on the Beaglebone to send a specific value to begin. Once the reaction tester receives the value 100 or 50 it will print "Game Start" and begin.

Once the reaction tester has begun, the Beaglebone will wait a random amount of time between 1 second and 15 seconds. Once that time has elapsed, the Beaglebone will generate a random value between 0 and 7, which happens to be the amount of buttons that the Gameboy has. That value is sent to the Gameboy, and once the Gameboy has received that value, it will prompt the user to press a button depending on the value that is sent to the Gameboy. The Gameboy will prompt the user to press that button by lighting up the button graphic on the screen. Once that button has been pressed, the reaction time will be sent back to the Beaglebone.

The way the reaction time is calculated is as follows. RIOS on Gameboy is configured to call the interrupt service routine 256 times per second. Everytime that ISR is called, we increment the number of interrupts we have encountered. Once we have encountered the instruction, we turn on the timer and we start counting how many interrupts occur before the user presses the button. Once the user presses the button, we record how many interrupts have occurred and send it to the Beaglebone. Once the Beaglebone has received the amount of interrupts, it simply divides that value by 256 (the amount of interrupts in a second) to determine how long it took for the user to press that button.

If you press Select on the boot menu, the program will test the receiving functions. This requires you to run a corresponding python file, in this case testSending.py. Keep in mind that if you are testing receiving on the Gameboy, then that means you must use the testSending.py as the Gameboy will be receiving values sent from the python script. If you are testing sending on the Gameboy, that means you must use the testReceiving.py script because you are sending from the Gameboy and receiving on the python script.

This is a very simple and straightforward test. The Gameboy will receive 10 values from the Beaglebone, from 0 to 10. If these values are any different than 0 to 10 then the test fails. Just like with starting it normally, you want to run the test on the Gameboy before running the python script.

Pressing A on the boot menu will test out the sending functions. Again, this is very similar to the previous test. The Beaglebone will send 10 values to the Gameboy, from 0 to 10. If these values are any different on receive, then the test fails. Something to keep in mind that with this test, you want to run them both at about the same time, or else it will fail.

6. Currently known bugs

When the Gameboy receives the first byte since boot up, the least significant bit will be lost. It only happens during the first byte transfer. As long as I write my code around this, the rest of the program should not be affected. For example, the Gameboy awaits the value 50 or 100 to start the game. The value 100 is the value that is explicitly sent in the python script, but due to the glitch the least significant bit is dropped and is instead 50. The sending test that you can access from the boot menu fails because of this bug.

Running the reaction tester twice in a row makes the second run not want to recognize input from the Beaglebone. Because of this, I choose to just end the program once the reaction tester finishes being called once.

When compiling, Windows has an error where the compiler doesn't like the length of where a compiler is located. If you encounter that issue, move the folder containing all the files to a higher directory closer to the root.

When grabbing a bit value from the Gameboy, the Beaglebone sometimes believes the line stayed high for a ridiculous amount of time regardless of the fact that it recorded it for less than a second. I have an error check that will discard any ridiculous time measurements. Of course this isn't an ideal solution, and could impact values received in later tests.

7. Where do I go next?

If I were to continue working on this, I would first iron out the currently known bugs. Second I would figure out a better way to send bytes over to the Beaglebone as my current solution is prone to errors. My solution is also very slow, something you will notice if you run the sending test.

The way I have test cases working right now is a little bit sloppy. I am relatively new to Python and the test cases I have created could be better. I currently have 1 test case for both sending and receiving, and inside that one test case are even more test cases to test if each value from 0 to 10 is sent/received properly. I would change this in the future so each value from 0 to 10 is a standalone Python test case.

The code is a little bit messy. Continuing to work on this further, I would clean this up and perhaps not have the RIOS code, the test functions and the reaction tester code all in the same file.

I would like to expand on this concept further, perhaps building a more robust video game to run on the Gameboy instead of a simple reaction tester. Perhaps a game that can change it's difficulty or level layout based on how a second user interacts with it. I would like to explore the idea of sending Gameboy data and information across larger distances, such as through the internet.

8. Conclusion

Even though there are still some issues I want to iron out with this reaction tester, for now I believe it is stable enough to show off. If you have any questions, please feel free to email me.