

Date: 6/5/2021

## Course Final

The following will document completion of the final assignment for ECE-40293, which will include the following goals:

*For your final project, combine at least THREE of the projects from the previous nine lessons into a single project.*

To accomplish this, I had planned on developing a project that leveraged the onboard sensors and a connected RTC over I2C, a Bluetooth connection over UART4, and the HAL GPIO API to interact with the LEDs, with the overall idea being that this would be a battery powered, embedded device similar to the cheap temp/ humidity sensors<sup>[1]</sup> that I had used as a reference point throughout this course and ECE-40291. I was able to accomplish most of this but failed in implementing some aspects of the external connections within the time remaining before the course ends. Rather than focusing on that aspect as a negative, this report will serve both as documentation of what it took to get to the final (mostly) functional version, as well as the issues I was unable to overcome, the methods I used to troubleshoot, and my thoughts on next steps.

Generating a default project as in previous courses, we'll want to ensure that peripherals I2C1 and I2C2 are enabled, which will be used for connecting to the external bus broken out on the Arduino headers as well as the internal bus to connect to the HTS221 sensor. Additionally, we'll enable UART4, which will need to be set to 9600 baud to interact with the HC-06 Bluetooth dongle<sup>[2]</sup> I had selected for this task.

A fair amount of the code in this project is re-used from earlier projects, for example the HTS221-related functions are identical to those developed in my ECE-40291 I2C project. We'll also see re-use with slight modifications to the CLI used heavily through both that and this course. My vision for this was to all interact over a BT-serial app on a cell phone or other mobile device, again, emulating these cheap little import sensor modules. I planned on having four functional aspects to that interface, being

1. Toggle the BLE/WiFi LEDs, as a sort of "wink" function to indicate device activity, in function `do_toggle_LED()`
2. Read the temperature and humidity from the HTS221 with `HTS221_pwr_en()`, `HTS221_get_cal_data()`, `HTS221_get_sensor_data()`
3. Get the current time from the RTC, in function `do_get_time()`
4. Set the time from the RTC, in function `do_set_time()`. This was implemented as a hardcoded "00:00:00 on Monday, 01/01/2021", with the idea being that a set/get operation would at least indicate that the device was ticking away. A second iteration of this project would include an actual programmable interface.

With the core concept fleshed out, let's jump into the code and then look at roadblocks and troubleshooting techniques employed.

Date: 6/5/2021

As mentioned earlier, this project involved a lot of reuse, or reuse with only slight modification. We start with the standard includes of `stdio.h` & `string.h` for working with the serial port. We then define the addresses of our target devices, noting that I maintained the standard of a separate R/W address for the DS3231 RTC module<sup>[3]</sup>, even though it only has the single address for both operations.

```
39 // Define R/W addresses for temp sensor
40 #define HTS221_READ_ADDRESS 0xbf
41 #define HTS221_WRITE_ADDRESS 0xbe
42
43 #define DS3231_READ_ADDRESS 0x68
44 #define DS3231_WRITE_ADDRESS 0x68
```

Next we'll define the globals used in the HTS221 calibration process. I won't go into great detail on reviewing the temperature sensor code as it has previously received a full write up. Moving on to the function prototypes and definitions, we see `do_toggle_LED()` implemented as a very simple blinky loop, again, used as an indicator that you are connected and can communicate with the device from your mobile.

```
113 static void do_toggle_LED(void)
114 {
115     for (int i = 0; i < 5; i++)
116     {
117         HAL_GPIO_TogglePin(LED3_WIFI_LED4_BLE_GPIO_Port, LED3_WIFI_LED4_BLE_Pin);
118         HAL_Delay(250);
119     }
120 }
```

Moving to `do_get_time()`, we'll see an excerpt below from the data sheet showing the time-related device registers within the RTC.

DS3231										Extremely Accurate I2C-Integrated RTC/TCXO/Crystal	
ADDRESS	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0 LSB	FUNCTION	RANGE	
00h	0	10 Seconds			Seconds				Seconds	00–59	
01h	0	10 Minutes			Minutes				Minutes	00–59	
02h	0	12/24	AM/PM 20 Hour	10 Hour	Hour				Hours	1–12 + AM/PM 00–23	
03h	0	0	0	0	0	Day			Day	1–7	
04h	0	0	10 Date		Date				Date	01–31	
05h	Century	0	0	10 Month	Month				Month/ Century	01–12 + Century	
06h	10 Year				Year				Year	00–99	

Date: 6/5/2021

This function was then implemented to write the target register to the device and then read back the register value, as seen below for reading seconds and minutes.

```
122 static void do_get_time(void)
123 {
124     // Large char buffer for strings sent over the console
125     char buffer[100] = {0};
126
127     // Register Seconds, address 0x00
128     uint8_t Seconds_Address = 0x00;
129     HAL_I2C_Master_Transmit(&hi2c1, DS3231_WRITE_ADDRESS, &Seconds_Address, sizeof(Seconds_Address), 1000);
130     uint8_t Seconds_Value;
131     HAL_I2C_Master_Receive(&hi2c1, DS3231_READ_ADDRESS, (uint8_t *)&Seconds_Value, sizeof(Seconds_Value), 1000);
132
133     // Register Minutes, address 0x01
134     uint8_t Minutes_Address = 0x01;
135     HAL_I2C_Master_Transmit(&hi2c1, DS3231_WRITE_ADDRESS, &Minutes_Address, sizeof(Minutes_Address), 1000);
136     uint8_t Minutes_Value;
137     HAL_I2C_Master_Receive(&hi2c1, DS3231_READ_ADDRESS, (uint8_t *)&Minutes_Value, sizeof(Minutes_Value), 1000);
138 }
```

Once registers 0x00- 0x06 were read in, they were formatted into a “HH:MM:SS on MM:DD:YY” string and printed out over the console connection.

*do\_set\_time()* is implemented in similar fashion, except as previously mentioned where the time and date were hardcoded in this proof-of-concept iteration.

```
187 static void do_set_time(void)
188 {
189     // Set time to 00:00:00 on Monday, 01/01/2021. Just use this as areset point for demo purposes
190
191     // Register Seconds, address 0x00
192     uint8_t Seconds_Address = 0x00;
193     HAL_I2C_Master_Transmit(&hi2c1, DS3231_WRITE_ADDRESS, &Seconds_Address, sizeof(Seconds_Address), 1000);
194     uint8_t Seconds_Value = 0x00;
195     HAL_I2C_Master_Transmit(&hi2c1, DS3231_WRITE_ADDRESS, &Seconds_Value, sizeof(Seconds_Value), 1000);
196
197     // Register Minutes, address 0x01
198     uint8_t Minutes_Address = 0x01;
199     HAL_I2C_Master_Transmit(&hi2c1, DS3231_WRITE_ADDRESS, &Minutes_Address, sizeof(Minutes_Address), 1000);
200     uint8_t Minutes_Value = 0x00;
201     HAL_I2C_Master_Transmit(&hi2c1, DS3231_WRITE_ADDRESS, &Minutes_Value, sizeof(Minutes_Value), 1000);
202 }
```

Neither of these functions seem to work as this point, again, we’ll explore that aspect later.

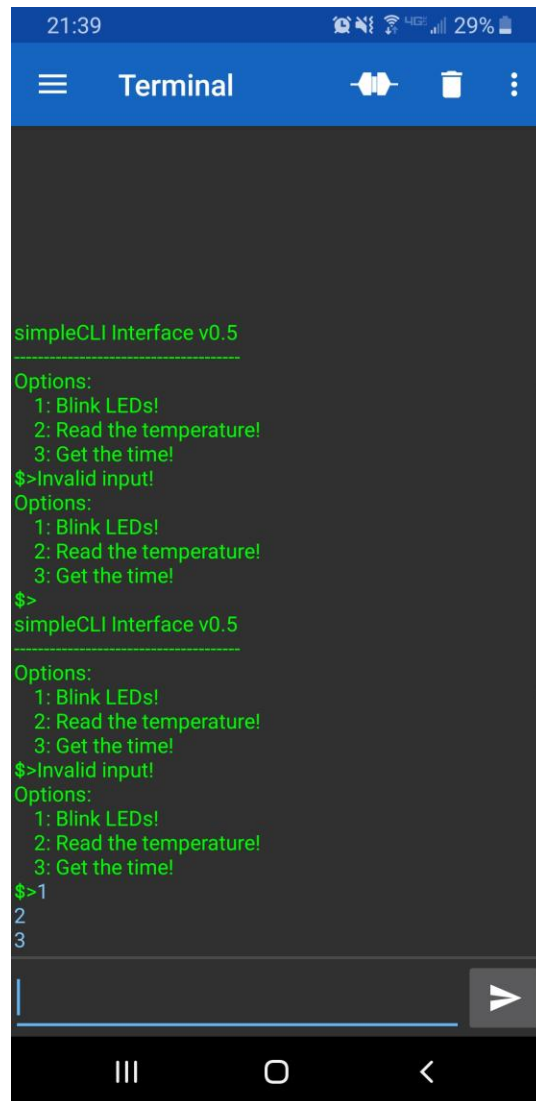
Date: 6/5/2021

Moving to the CLI, we again see re-use of the style developed in earlier projects, with appropriate modifications to represent each functional aspect.

```
450  /* USER CODE BEGIN 3 */
451
452  char* cliPrompt = "Options:\n  1: Blink LEDs!\n  2: Read the temperature & RH!\n  3: Get the time!\n  4: Set the time!\n$>";
453  HAL_UART_Transmit(&huart1, (uint8_t*) cliPrompt, strlen(cliPrompt), 1000);
454
455  char cliInput = '\0';
456  HAL_UART_Receive(&huart1, (uint8_t*) &cliInput, 1, HAL_MAX_DELAY);
457
458  switch (cliInput)
459  {
460      case '1':
461          cliPrompt = "\nToggle BLE/Wifi LED...\n\n";
462          HAL_UART_Transmit(&huart1, (uint8_t*) cliPrompt, strlen(cliPrompt), 1000);
463          do_toggle_LED();
464          break;
465
466      case '2':
467          cliPrompt = "\nReading the temperature and RH...\n";
468          HAL_UART_Transmit(&huart1, (uint8_t*) cliPrompt, strlen(cliPrompt), 1000);
469          HTS221_get_sensor_data();
470          break;
471
472      case '3':
473          cliPrompt = "\nGetting the time...\n";
474          HAL_UART_Transmit(&huart1, (uint8_t*) cliPrompt, strlen(cliPrompt), 1000);
475          do_get_time();
476          break;
477
478      case '4':
479          cliPrompt = "\nSetting the time...\n";
480          HAL_UART_Transmit(&huart1, (uint8_t*) cliPrompt, strlen(cliPrompt), 1000);
481          do_set_time();
482          break;
483
484      default:
485          cliPrompt = "Invalid input!\n\n";
486          HAL_UART_Transmit(&huart1, (uint8_t*) cliPrompt, strlen(cliPrompt), 1000);
487          break;
488  }
```

You might notice all of the UART calls are to UART1 in the above, this is a good transition into the roadblocks I found while developing this project. To communicate with the BT module, I downloaded “Serial Bluetooth Terminal” from the app store and scanned for available modules, finding and connecting to the HC-06 without issue. So far, perfect. Having already flashed the program to the Disco board, I reset it and was presented with the CLI prompt. Excited by how smooth everything had gone thus far, I slammed into a wall when I discovered that any entry to the CLI from my side resulted in hitting the default case of “Invalid Entry”, seen on the next page. Deciding to first verify that my CLI architecture was sound, I quickly switched the function calls from UART4 to UART1, reflashed the board, and opened Putty. Select option 1 and the LEDs blink away happily. Scratch head, back to UART4 and another flash and open the debugger, stepping up to the entry into the switch and monitoring the value for *cliInput*, I see that it remains ‘\0’.

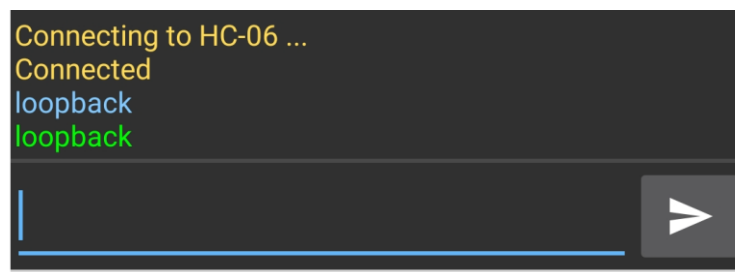
Date: 6/5/2021



A screenshot of a mobile terminal application. The status bar at the top shows the time 21:39, signal strength, 4G LTE, and 29% battery. The app's title bar is blue with a hamburger menu icon, the word "Terminal", and icons for a USB cable, a trash can, and a three-dot menu. The terminal window has a black background with green text. It displays a prompt "\$>" followed by "Invalid input!". Then it shows a menu of options: "1: Blink LEDs!", "2: Read the temperature!", and "3: Get the time!". The user enters "1", and the terminal shows "2" and "3" on subsequent lines. At the bottom, there is a blue horizontal line and a grey button with a white right-pointing arrow.

```
21:39
simpleCLI Interface v0.5
Options:
  1: Blink LEDs!
  2: Read the temperature!
  3: Get the time!
$>Invalid input!
Options:
  1: Blink LEDs!
  2: Read the temperature!
  3: Get the time!
$>
simpleCLI Interface v0.5
Options:
  1: Blink LEDs!
  2: Read the temperature!
  3: Get the time!
$>Invalid input!
Options:
  1: Blink LEDs!
  2: Read the temperature!
  3: Get the time!
$>1
2
3
```

“Ok, it has to be something with the BT module then? Let’s try loopback mode and see if Tx->Rx works!”

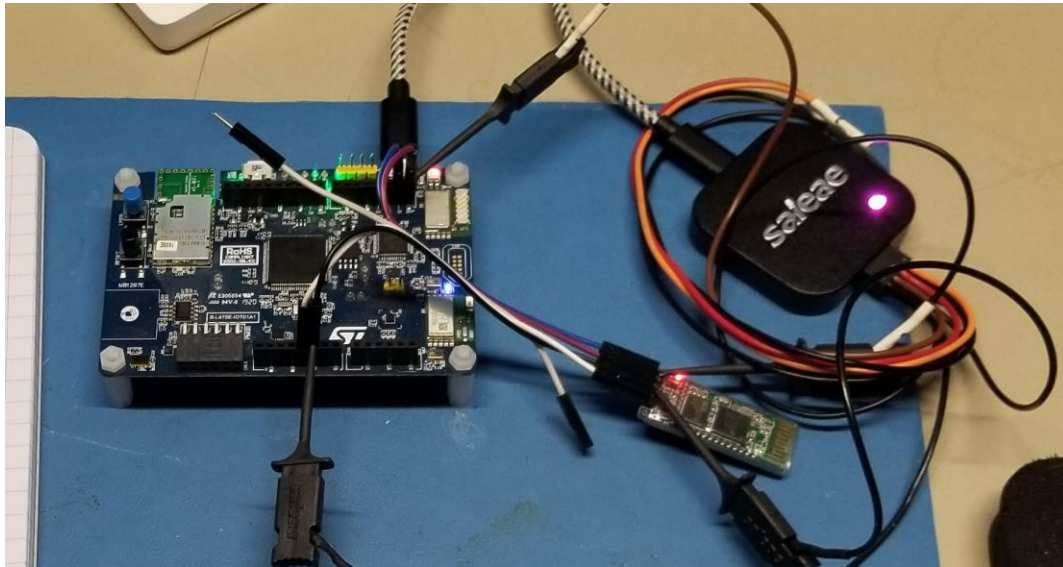


A screenshot of a terminal window with a black background and yellow and green text. It shows the text "Connecting to HC-06 ..." in yellow, followed by "Connected" in yellow. Below that, the word "loopback" is written twice in green. At the bottom, there is a blue horizontal line and a grey button with a white right-pointing arrow.

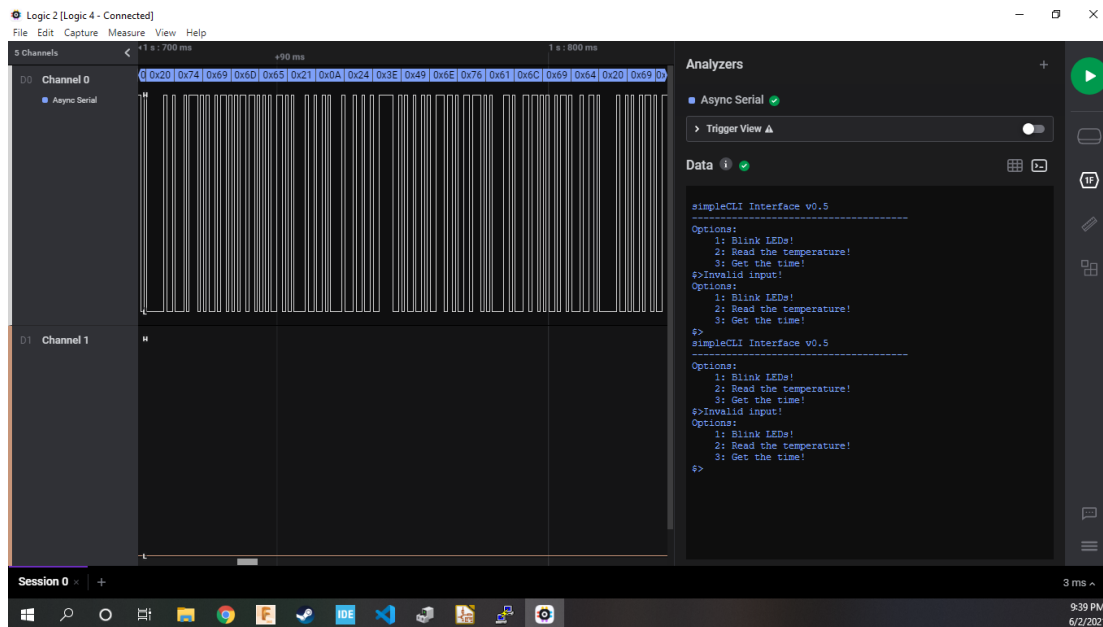
```
Connecting to HC-06 ...
Connected
loopback
loopback
```

Okay, time to break out the logic analyzer...

Date: 6/5/2021



Hooking up CH0 to the BT Tx line and CH1 to the RX line, we'll start gathering sample, hit reset, and then enter something from the CLI on the phone.

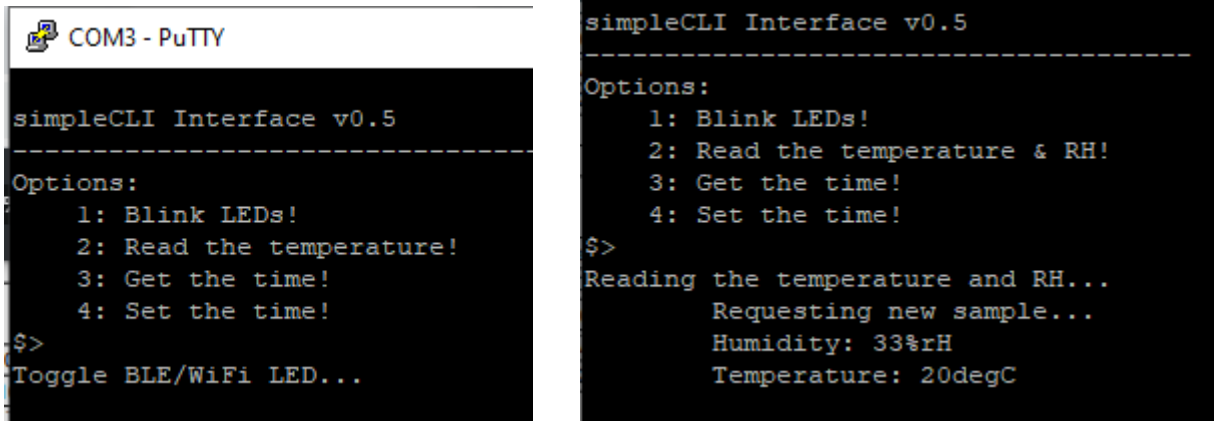


And CH0 looks perfect, while we see nothing on CH1. Additionally, something about this setup causes the HC-06 to lock up, as can be seen in the phone screen shot of 1, 2, 3 entered with no response. Being almost midnight and still needing to test the RTC, I decide to admit defeat (for now) and changed the function calls back to UART1.



Date: 6/5/2021

Re-flashed, connected to PuTTY, and rebooted, we see the console come up, and again, the LEDs toggle and the temp humidity sensor works as expected.

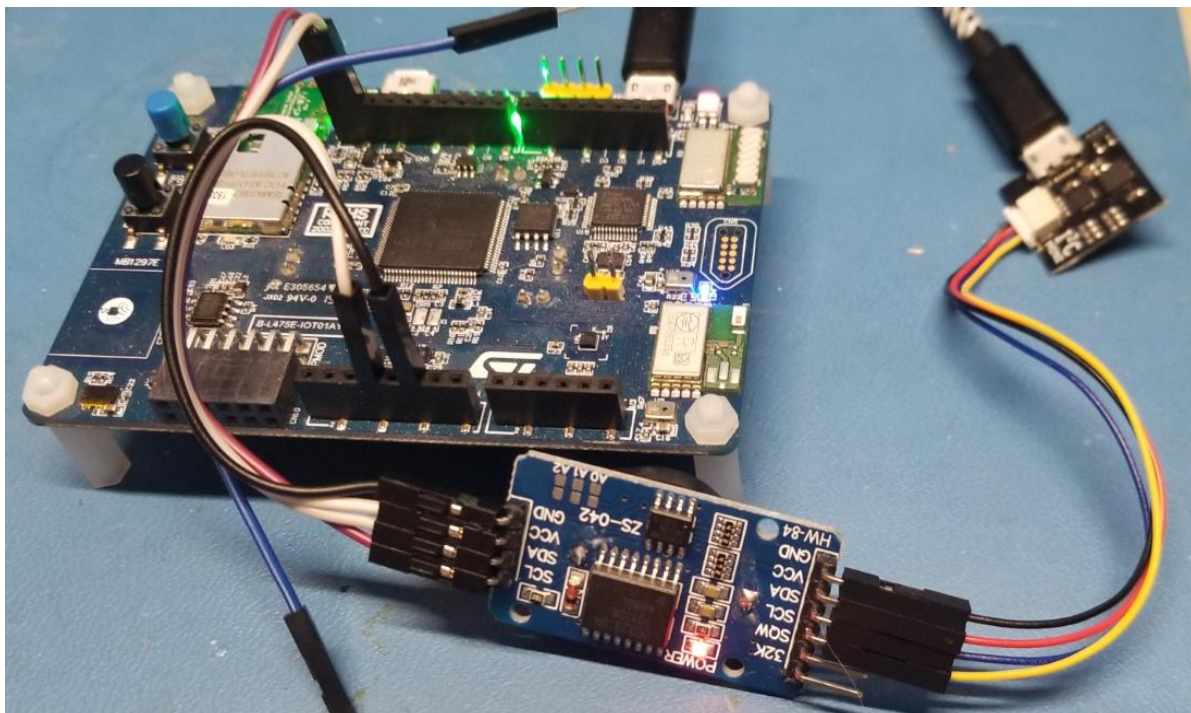


```
COM3 - PuTTY

simpleCLI Interface v0.5
-----
Options:
  1: Blink LEDs!
  2: Read the temperature!
  3: Get the time!
  4: Set the time!
$>
Toggle BLE/WiFi LED...

simpleCLI Interface v0.5
-----
Options:
  1: Blink LEDs!
  2: Read the temperature & RH!
  3: Get the time!
  4: Set the time!
$>
Reading the temperature and RH...
Requesting new sample...
Humidity: 33%rH
Temperature: 20degC
```

To test the RTC module, I used my I<sup>2</sup>CMini<sup>[4]</sup> connected directly to the RTC module. The Mini offers Python and C APIs, as well as a GUI and CLI toolset. Opening the GUI, I was able to see a device at address 0x68, as expected from the DS3231 datasheet, and then in the CLI tool, I was able to read register data from it. I'm still new to using this tool but was able to write and read back the value 0x21 to 0x06, indicating the year of 2021. However, when I tried to set registers 0x00- 0x05, I read back all 0xff. Undeterred, I chalked this up to using the tool wrong and decided to move on to testing with the board and CLI.



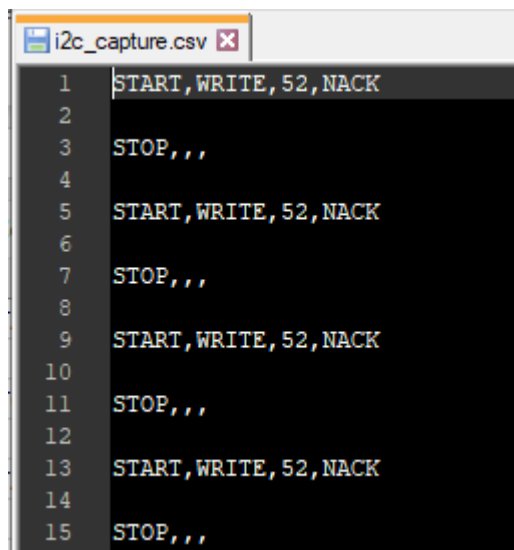
Date: 6/5/2021

Running the set time option, everything looks good. You might have noticed in my function def earlier though, that I omitted the use of *HAL\_StatusTypeDef status* in my I2C Tx/Rx calls. At the time, I was just rushing and feeling the looming submission deadline. On attempting to get the time though, I see that being under pressure shouldn't be an excuse for cutting corners...

```
Setting the time...
    Time is set to 00:00:00 on Monday, 01/01/2021!
Options:
    1: Blink LEDs!
    2: Read the temperature & RH!
    3: Get the time!
    4: Set the time!
$>

$>
Getting the time...
    Current time is 1615:800:02 on 168-01-01Options:
    1: Blink LEDs!
    2: Read the temperature & RH!
    3: Get the time!
    4: Set the time!
$>
```

So on to troubleshooting the RTC. Another feature of the I2CMini is monitor mode, where it logs all activity on the bus to a .csv file for later processing. So, in similar fashion to the BT debug, I opened monitor mode and began making set/get calls. Closing monitor mode reported 28 captures made, so opening the log file, I see there were indeed captures made, but nothing that, at the time, appeared to be of any use in debugging other than repeated NACKs, which seem to indicate that the RTC is not responding:



```
i2c_capture.csv
1  START,WRITE,52,NACK
2
3  STOP,,,
4
5  START,WRITE,52,NACK
6
7  STOP,,,
8
9  START,WRITE,52,NACK
10
11 STOP,,,
12
13 START,WRITE,52,NACK
14
15 STOP,,,
16
```



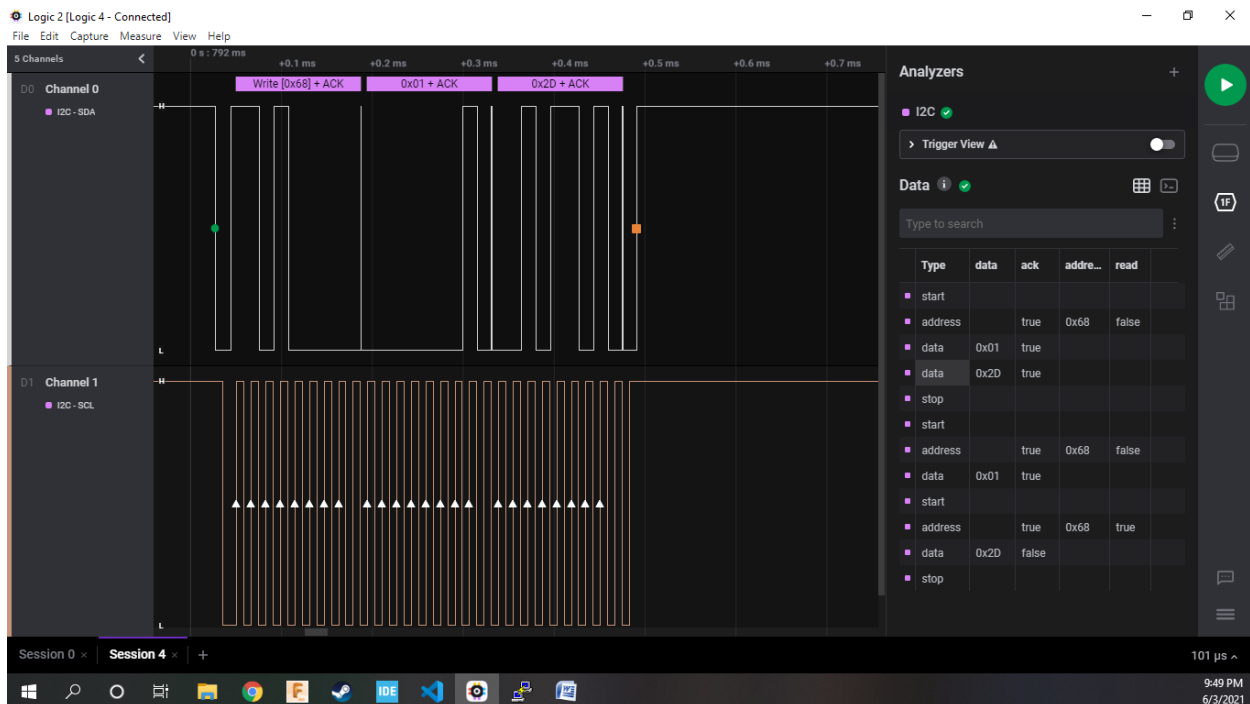
Date: 6/5/2021

The following evening, I broke out a Raspberry Pi and attempted the same operation as with the I2CMini using the i2c-tools collection. Interestingly, I was able to write and read to the time registers consistently. Back under the I2CMini, I found the same after modifying my command structure, ruling out any ideas of bad hardware.

```
PS C:\Program Files (x86)\Excamera Labs\I2CDriver> .\i2cc1.exe COM8 w 0x68 1 r 0x68 1 0x2d
PS C:\Program Files (x86)\Excamera Labs\I2CDriver>

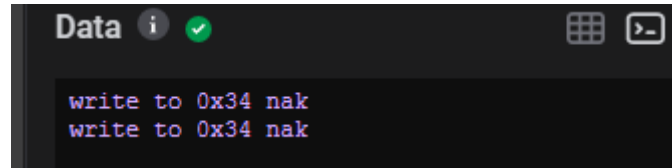
pi@raspberrypi:~$ sudo i2cget -y 1 0x68 0x01
0x2d
pi@raspberrypi:~$
```

Back to the logic analyzer, I pulled a trace of a write/ read operation to the minutes register, setting at a value of 0x2d (arbitrary number) and can see that the process is about what I would expect for this device: signal a start, send the device address with the direction bit equal to 0 for a write, send the register address, and then the data value.



Date: 6/5/2021

Changing the `do_set_time()` function to the same and re-flashing, I see this on the analyzer:



Now we're making progress! I'll qualify now that I don't recall any of the following being discussed in the lectures, and I hope it wasn't as I spent a considerable amount of time getting to this point. Seeing 0x34 is one half of 0x68, I changed the R/W addresses to 0xd0 and was able to successfully write to a register! Now searching through the HAL I2C files to find why, I ran across this parameter definition. Looking at the data sheet one more time, it is painfully obvious what the problem was.

```
1790 the configuration information for the specified I2C.  
1791 * @param DevAddress Target device address: The device 7 bits address value  
1792 * in datasheet must be shifted to the left before calling the interface  
1793 * @param pData Pointer to data buffer
```

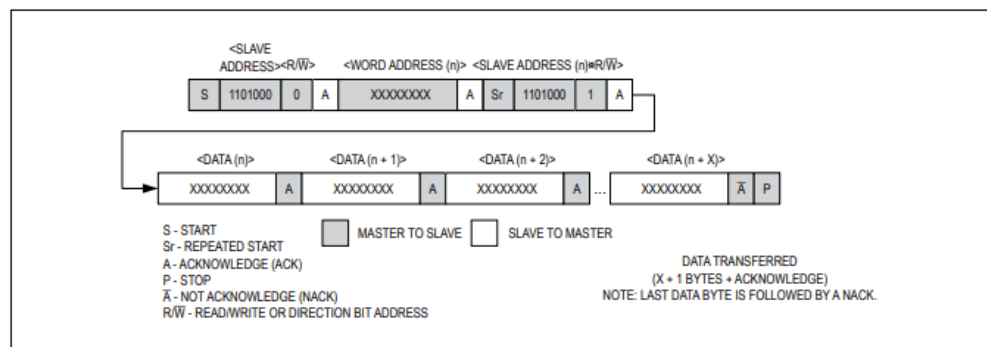


Figure 5. Data Write/Read (Write Pointer, Then Read)—Slave Receive and Transmit

My device R/W addresses should have been the DS3231 7-bit address of 1101000b or 0x68, shifted left by 1, then with bit zero set to 0 for write ops and 1 for read ops, giving me 0xd0 for a write address and 0xd1 for read. In hindsight, the I2CMini capture reporting a value of "52", or 0x34, should have clued me in but I wasn't paying close enough attention to it late at night. So with the addresses corrected, I can now read and write to the RTC. Getting the time, we see it ticking up.

```
Getting the time...  
Current time is 00:23:21 on 01-01-00  
  
Options:  
1: Blink LEDs!  
2: Read the temperature & RH!  
3: Get the time!  
4: Set the time!  
$>  
Getting the time...  
Current time is 00:23:24 on 01-01-00
```

Date: 6/5/2021

Unfortunately, setting the time still seems to fail. Back under the analyzer, I see the default 0's being written to the target registers, and reading back I see the accrued values still in place. I'm thinking that this is related to the user buffers that exist within the DS3231 not synchronizing with the values I write in but haven't been able to track the root down yet.

```
Setting the time...  
    Written time is 00:00:00 on 01-01-01  
Options:  
    1: Blink LEDs!  
    2: Read the temperature & RH!  
    3: Get the time!  
    4: Set the time!  
$>  
Getting the time...  
    Current time is 00:26:57 on 01-01-00
```

### Final Thoughts

Between the LEDs, the HTS, and the CLI, I *technically* completed all of the proscribed requirements for this assignment. With the failure to get the BT or RTC modules fully functional though, I feel that my design's implementation is lacking the most critical features. Judging by how the BT module was locking up, I'm inclined to think there is a hardware issue at play, or perhaps something related to power limitations. Given time to do so, my next step would be to power it from a bench supply instead of the Arduino headers and repeat my earlier experiments. The RTC issue I feel is entirely in software handling, and despite working through the data sheet, I haven't found a solution yet, though I do intend to spend more time digging into it outside of the scope of this project.

On the course, it has been a great learning experience to dive deeper into these various protocols than I have in the past and even to be challenged and fail in certain areas like this project's hardware. Beyond that, while taking this course alongside ECE-40291, still in a full-time capacity at work, and trying to juggle a 4-month old at home, it has been a challenge to make the time I would have really liked to devote to going beyond what is presented in the lectures or required for each assignment, again as seen in this project. Even then, this whole course has just been fun. Between the interactions with the rest of the class and getting to experiment with hardware I've not been able to before, I enjoyed all of it despite (m)any struggles. As discussed in some of the follow-ups to assignment submissions, any one of these topics could easily be a course unto themselves and I'd happily take any of them in a heartbeat.