

Date: 6/1/2021

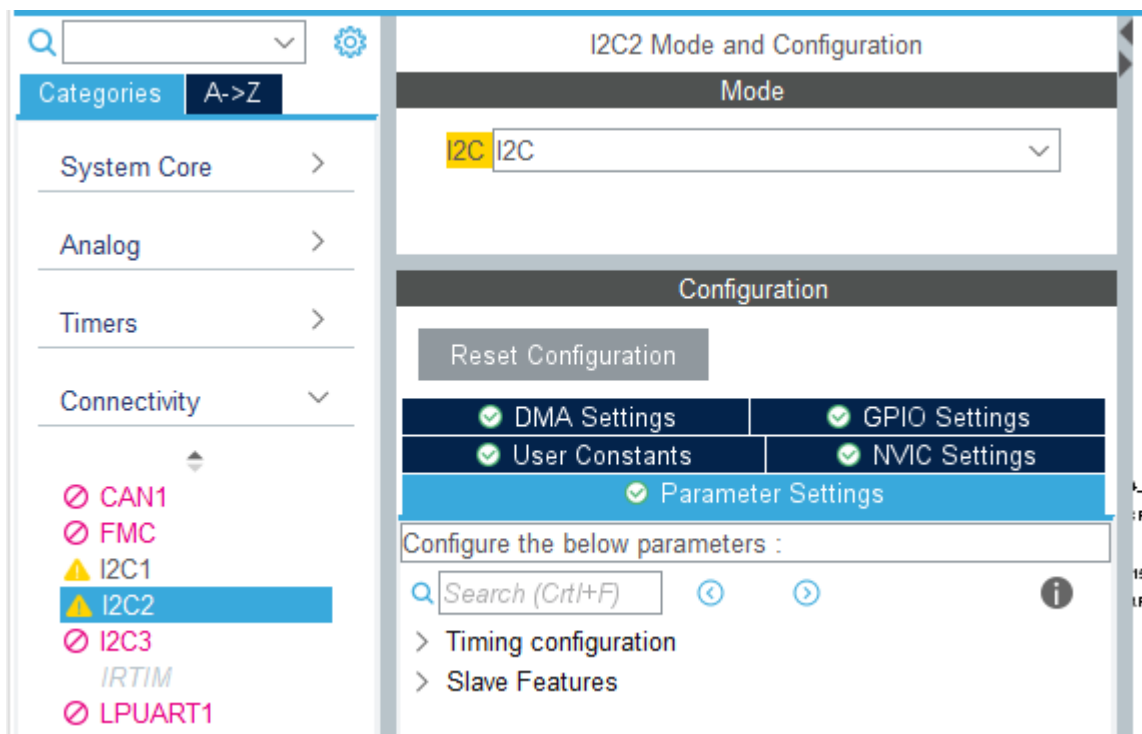
Assignment 7: I2C

The following will document completion of the eighth assignment for ECE-40291, with the stated goals of:

1. Your skills for using HAL I2C APIs to read the I2C sensor of your choice on the STM32 Discovery Board. There are several I2C sensors on the STM32 Discovery Board --- pick one that is of interest to you, and then read data using the I2C HAL APIs.
2. Display your data from your sensor on the UART1 console.

- 1. Your skills for using HAL I2C APIs to read the I2C sensor of your choice on the STM32 Discovery Board. There are several I2C sensors on the STM32 Discovery Board --- pick one that is of interest to you, and then read data using the I2C HAL APIs.**

As in all previous assignments, open and generate the default project configuration for the Disco board. To leverage the on board I2C2 bus, we'll also need to open the device configuration tool, and enable it under 'Connectivity' >> 'I2C2', as seen here. The default options are satisfactory for this project's scope.



Date: 6/1/2021

From here, the IDE will handle the addition of the appropriate init calls and other support files; we can focus on the target sensors and associated firmware from here on. For this project I chose to expand on the HTS221 sensor used in the previous assignments for two reasons. First: to attempt readings with the onboard humidity sensor in addition to reading temperature, and second: to dive further into the hardware, specifically the calibration routines. This also had the benefit of allowing some code re-use (with much expansion) from the I2C assignment in ECE-40293, which made the rest of this assignment simple to implement after enough time spent in datasheets and tech notes. Additionally, I chose to read the ambient pressure from the LPS22HB module as a supplement to taking “the easy way out” with code reused from another course. Finally, I had hoped to incorporate an RTC module on the external I2C bus broken out over the Arduino connectors and while I was able to perform basic read/write operations using my I2C Driver Mini^[1] debugging tool over my PC, I decided not to incorporate it due to time constraints with regards to the course end date approaching. On the interface level, I again reused existing code to deploy a simple CLI over UART 1, with a list of options for each sensor broken out, followed by status info and the process values as they were read back from each data point.

So, jumping into *main.c*, we’ll first take care of the boiler-plate items: including the standard libraries needed for string and print functions, followed by defining the bus addresses of each module, taken from the Disco board’s published schematics.

```
23 /* Private includes -----  
24 /* USER CODE BEGIN Includes */  
25  
26 #include <stdio.h>  
27 #include <string.h>  
28  
29 /* USER CODE END Includes */  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39 // Define R/W addresses for temp sensor  
40 #define HTS221_READ_ADDRESS 0xbf  
41 #define HTS221_WRITE_ADDRESS 0xbe  
42 #define LPS22HB_READ_ADDRESS 0xbb  
43 #define LPS22HB_WRITE_ADDRESS 0xba  
44  
45 /* USER CODE END PD */  
46
```

Coming back to the support functions that do all of the processing later, we jump into the *main()* function, insert calls to *HTS221_pwr_en()* and *HTS221_get_cal_data()*, then print out the CLI header and jump into the *while(1)* loop, where the CLI options are printed and the user can input their selection. As it has been covered multiple times elsewhere, I won’t go into

Date: 6/1/2021

greater detail beyond that on the CLI structure other than to segue into the support function definitions that each option calls. Full project code will be submitted with this report.

Option one, “HTS221 read request” will call function *HTS221_get_sensor_data()*. First though, lets jump back to the earlier setup calls to *HTS221_pwr_en()* and *HTS221_get_cal_data()*. *HTS221_pwr_en()* is an extremely simple “wake up” style function that enable the module into one-shot mode and allows us to take readings later.

```
108 static void HTS221_pwr_en(void)
109 {
110     // Configure control register 1 (CTRL_REG1, 0x20) bit 7 to enable one-shot
111     uint8_t ctrlReg1 = 0x20;
112     uint8_t CTRL_REG2_Value[] = {ctrlReg1, (1 << 7)};
113
114     // Send the target register to the device
115     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, CTRL_REG2_Value, sizeof(CTRL_REG2_Value), 1000);
116
117 }
118
```

HTS221_get_cal_data() is a bit more complicated, and involves reading the calibration values for the temperature and humidity sensors from the module into global variables for use when readings are taken at a later point. This is an example of the deeper dive I wanted to take into this piece of hardware, and while the information made available in the device datasheet was a bit vague, I found an example implementation for the Raspberry Pi written in Python^[2], followed by an ST published tech note^[3] that fleshed out the process in great detail. Breaking it out below, we pull the appropriate calibration values in from the module and store them for use in the calibration equation defined in the tech note for each data point, occurring later in the *HTS221_get_sensor_data()* call.

6. Compute the RH [%] value, by linear interpolation, applying the formula below:

$$H_{RH}[\%] = \frac{(H1_rH - H0_rH) \cdot (H_T_OUT - H0_T0_OUT)}{H1_T0_OUT - H0_T0_OUT} + H0_rH$$

7. Compute the T [degC] value, by linear interpolation, applying the following formula:

$$T[degC] = \frac{(T1_degC - T0_degC) \cdot (T_OUT - T0_OUT)}{T1_OUT - T0_OUT} + T0_degC$$

Date: 6/1/2021

```
119 static void HTS221_get_cal_data(void)
120 {
121     /*****
122     // Request humidity and temperature calibration data stored in registers 0x30 to 0x3F
123     // Reference pg. 26 of data sheet (https://www.st.com/resource/en/datasheet/hts221.pdf)
124     // for register names and definitions
125     // Reference tech note TN1218 on calibration procedures
126     // https://www.st.com/resource/en/technical_note/dm00208001-Interpreting-humidity-and-temperature-readings-in-the-hts221-digi
127     *****/
128     // Humidity calibration values
129
130     // Register H0_rh_x2, address 0x30. Divide register value by 2 for calibration value
131     uint8_t H0_rh_Address = 0x30;
132     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &H0_rh_Address, sizeof(H0_rh_Address), 1000);
133     H0_rh_Value = 0xff; // Junk default value
134     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&H0_rh_Value, sizeof(H0_rh_Value), 1000);
135     H0_rh_Value = H0_rh_Value / 2;
136
137     // Register H1_rh_x2, address 0x31. Divide register value by 2 for calibration value
138     uint8_t H1_rh_Address = 0x31;
139     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &H1_rh_Address, sizeof(H1_rh_Address), 1000);
140     H1_rh_Value = 0xff; // Junk default value
141     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&H1_rh_Value, sizeof(H1_rh_Value), 1000);
142     H1_rh_Value = H1_rh_Value / 2;
143
144     // Register H0_T0_OUT, addresses 0x36 and 0x37
145     uint8_t H0_T0_OUT_Address = 0x36 | 0x80;
146     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &H0_T0_OUT_Address, sizeof(H0_T0_OUT_Address), 1000);
147     H0_T0_OUT_Value = 0xffff; // Junk default value
148     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&H0_T0_OUT_Value, sizeof(H0_T0_OUT_Value), 1000);
149
150     // Register H1_T0_OUT, addresses 0x3A and 0x3B
151     uint8_t H1_T0_OUT_Address = 0x3A | 0x80;
152     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &H1_T0_OUT_Address, sizeof(H1_T0_OUT_Address), 1000);
153     H1_T0_OUT_Value = 0xffff; // Junk default value
154     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&H1_T0_OUT_Value, sizeof(H1_T0_OUT_Value), 1000);
155
156     /*****
157     // Temperature calibration values
158
159     // Register T0_degC_x8, address 0x32. Divide register value by 8 for calibration value
160     uint8_t T0_degC_Address = 0x32;
161     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &T0_degC_Address, sizeof(T0_degC_Address), 1000);
162     T0_degC_Value = 0xff; // Junk default value
163     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&T0_degC_Value, sizeof(T0_degC_Value), 1000);
164     T0_degC_Value = T0_degC_Value / 8;
165
166     // Register T1_degC_x8, address 0x33. Divide register value by 8 for calibration value
167     uint8_t T1_degC_Address = 0x33;
168     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &T1_degC_Address, sizeof(T1_degC_Address), 1000);
169     T1_degC_Value = 0xff; // Junk default value
170     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&T1_degC_Value, sizeof(T1_degC_Value), 1000);
171     T1_degC_Value = T1_degC_Value / 8;
172
173     // Register T1/T0_msb, address 0x35. Mask bits (0 & 1), (2 & 3) to get values of T0_degC & T1_degC
174     uint8_t T1_T0_msb_Address = 0x35;
175     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &T1_T0_msb_Address, sizeof(T1_T0_msb_Address), 1000);
176     uint8_t T1_T0_msb_Value = 0xff; // Junk default value
177     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&T1_T0_msb_Value, sizeof(T1_T0_msb_Value), 1000);
178     T0_degC = (T1_T0_msb_Value && (0b0011));
179     T1_degC = (T1_T0_msb_Value && (0b1100));
180
181     // Register T0_OUT, addresses 0x3C and 0x3D
182     uint8_t T0_OUT_Address = 0x3C | 0x80;
183     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &T0_OUT_Address, sizeof(T0_OUT_Address), 1000);
184     T0_OUT_Value = 0xffff; // Junk default value
185     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&T0_OUT_Value, sizeof(T0_OUT_Value), 1000);
186
187     // Register T1_OUT, addresses 0x3C and 0x3D
188     uint8_t T1_OUT_Address = 0x3C | 0x80;
189     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &T1_OUT_Address, sizeof(T1_OUT_Address), 1000);
190     T1_OUT_Value = 0xffff; // Junk default value
191     HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&T1_OUT_Value, sizeof(T1_OUT_Value), 1000);
192 }
193
194
195
```

Date: 6/1/2021

Jumping next to *HTS221_get_sensor_data()*, we set the module to one-shot mode, then loiter until the STATUS register updates to indicate that the new sample is ready.

```
196 static void HTS221_get_sensor_data(void)
197 {
198     // Large char buffer for strings sent over the console
199     char buffer[100] = {0};
200
201     // Configure control register 2 (CTRL_REG2, 0x21) bit 0 to enable one-shot
202     uint8_t CTRL_REG2_Address = 0x21;
203     uint8_t CTRL_REG2_Value[] = {CTRL_REG2_Address, (1 << 0)};
204
205     // Send the target register to the device
206     HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, CTRL_REG2_Value, sizeof(CTRL_REG2_Value), 1000);
207
208     // Define status register (STATUS_REG2, 0x27) bit 0 to monitor for new sample available
209     uint8_t STATUS_Address = 0x27;
210     uint8_t STATUS_Value = 0;
211
212     // Print status message to console
213     snprintf(buffer, sizeof(buffer), "\tRequesting new sample...");
214     HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
215
216     // Loiter for a bit to allow time for conversion to complete and be made available
217     uint8_t count = 0;
218     while (count < 10) // arbitrary "long enough" delay value
219     {
220         // Send the address of the status register
221         HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &STATUS_Address, sizeof(STATUS_Address), 1000);
222
223         // Read back the value of the status register
224         HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&STATUS_Value, sizeof(STATUS_Value), 1000);
225
226         // If the new sample is ready, break out of while-loop...
227         if (STATUS_Value & 0x01)
228         {
229             break;
230         }
231
232         // Update status message on console with '.' to indicate processing
233         snprintf(buffer, sizeof(buffer), ".");
234         HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
235
236         // Else wait for a bit, increment the counter, and keep looping
237         HAL_Delay(100);
238         count++;
239     }
240 }
```

Date: 6/1/2021

Once we see that flag set, we request the values of the humidity and temperature registers then pass both through their respective calibration and scaling formula before printing the final results to the console.

```
240
241 // Read the values of the humidity register H_OUT, address 0x28 and 0x29
242 uint8_t H_OUT_Address = 0x28 | 0x80;
243 HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &H_OUT_Address, sizeof(H_OUT_Address), 1000);
244 int16_t H_OUT_Value = 0xbeef; // Junk default value
245 HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&H_OUT_Value, sizeof(H_OUT_Value), 1000);
246
247 // Calculate and print value of humidity in %RH.
248 int16_t humidityValue = (((H1_rH_Value - H0_rH_Value) * (H_OUT_Value - H0_T0_OUT_Value)) / (H1_T0_OUT_Value - H0_T0_OUT_Value)) + (H0_rH_Value);
249 snprintf(buffer, sizeof(buffer), "\n\tHumidity: %d%\rH\n", humidityValue);
250 HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
251
252
253 // Read the values of the temperature register T_OUT, address 0x2A and 0x2B
254 uint8_t T_OUT_Address = 0x28 | 0x80;
255 HAL_I2C_Master_Transmit(&hi2c2, HTS221_WRITE_ADDRESS, &T_OUT_Address, sizeof(T_OUT_Address), 1000);
256 int16_t T_OUT_Value = 0xbeef; // Junk default value
257 HAL_I2C_Master_Receive(&hi2c2, HTS221_READ_ADDRESS, (uint8_t *)&T_OUT_Value, sizeof(T_OUT_Value), 1000);
258
259 // Calculate and print value of temperature in degC.
260 int16_t temperatureValue = (((T1_degC_Value - T0_degC_Value) * (T_OUT_Value - T0_OUT_Value)) / (T1_OUT_Value - T0_OUT_Value)) + (T0_degC_Value);
261 snprintf(buffer, sizeof(buffer), "\tTemperature: %ddegC\n", temperatureValue);
262 HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
263
264 }
265
```

Date: 6/1/2021

Moving on to the LPS22HB sensor code, we wrap everything into the *LPS22HB_get_sensor_data()* function call, which is structurally very similar to the HTS version, minus the cal section as that is handled internal to the module. We again configure the device for one-shot mode, wait on the status register to report that it is ready, and then send a request to get the values of 3 registers that represent the 24-bit pressure value. Each one of these is shifted into a single variable with the appropriate bit offset and then divided by 4096 as per the datasheet's example. Finally, this value is printed over the console. I should note that this device also includes a temperature sensor but I elected not to read its data as we already had accomplished that with the HTS and I felt the time would be better spent focusing on other assignments. See code excerpts on following pages.

Date: 6/1/2021

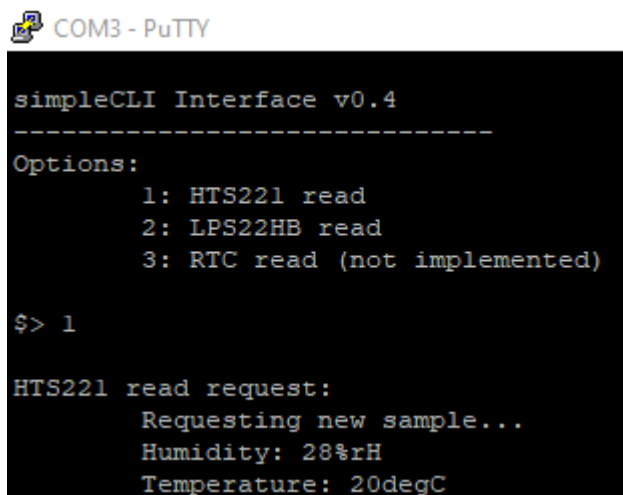
```
266 static void LPS22HB_get_sensor_data(void)
267 {
268     /******
269     // To read from the on-board pressure sensor, we will configure the control registers to one-shot mode and then
270     // read the pressure value in three byte-sized chunks, shifting them into a 32-bit word that is then converted
271     // to units of hPa, as described in the module datasheet, https://www.st.com/resource/en/datasheet/dm00140895.pdf
272     /******
273
274     // Large char buffer for strings sent over the console
275     char buffer[100] = {0};
276
277     // Register CTRL_REG2, address 0x11. Configures sensor module for one-shot mode
278     uint8_t CTRL_REG2_Address = 0x11;
279     uint8_t CTRL_REG2_Value[] = {CTRL_REG2_Address, (1 << 0)};
280     HAL_I2C_Master_Transmit(&hi2c2, LPS22HB_WRITE_ADDRESS, CTRL_REG2_Value, sizeof(CTRL_REG2_Value), 1000);
281
282     // Register STATUS, address 0x27. Bit 0 to monitor for new pressure sample available
283     uint8_t STATUS_Address = 0x27;
284     uint8_t STATUS_Value = 0;
285
286     // Print status message to console
287     snprintf(buffer, sizeof(buffer), "\tRequesting new sample...");
288     HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
289
290     // Loiter for a bit to allow time for conversion to complete and be made available
291     while (1)
292     {
293         // Send the address of the status register
294         HAL_I2C_Master_Transmit(&hi2c2, LPS22HB_WRITE_ADDRESS, &STATUS_Address, sizeof(STATUS_Address), 1000);
295
296         // Read back the value of the status register
297         HAL_I2C_Master_Receive(&hi2c2, LPS22HB_READ_ADDRESS, (uint8_t *)&STATUS_Value, sizeof(STATUS_Value), 1000);
298
299         // If the new sample is ready, break out of while-loop...
300         if (STATUS_Value & 0x01)
301         {
302             break;
303         }
304
305         // Update status message on console with '.' to indicate processing
306         snprintf(buffer, sizeof(buffer), ".");
307         HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
308
309         // Else wait for a bit, increment the counter, and keep looping
310         HAL_Delay(100);
311     }
312 }
```


Date: 6/1/2021

```
312
313 // Register PRESS_OUT_XL, address 0x28. Pressure output value (LSB)
314 uint8_t PRESS_OUT_XL_Address = 0x28;
315 HAL_I2C_Master_Transmit(&hi2c2, LPS22HB_WRITE_ADDRESS, &PRESS_OUT_XL_Address, sizeof(PRESS_OUT_XL_Address), 1000);
316 uint8_t PRESS_OUT_XL_Value = 0xff; // Junk default value
317 HAL_I2C_Master_Receive(&hi2c2, LPS22HB_READ_ADDRESS, (uint8_t *)&PRESS_OUT_XL_Value, sizeof(PRESS_OUT_XL_Value), 1000);
318
319 // Register PRESS_OUT_L, address 0x29. Pressure output value (mid part)
320 uint8_t PRESS_OUT_L_Address = 0x29;
321 HAL_I2C_Master_Transmit(&hi2c2, LPS22HB_WRITE_ADDRESS, &PRESS_OUT_L_Address, sizeof(PRESS_OUT_L_Address), 1000);
322 uint8_t PRESS_OUT_L_Value = 0xff; // Junk default value
323 HAL_I2C_Master_Receive(&hi2c2, LPS22HB_READ_ADDRESS, (uint8_t *)&PRESS_OUT_L_Value, sizeof(PRESS_OUT_L_Value), 1000);
324
325 // Register PRESS_OUT_H, address 0x2A. Pressure output value (MSB)
326 uint8_t PRESS_OUT_H_Address = 0x2A;
327 HAL_I2C_Master_Transmit(&hi2c2, LPS22HB_WRITE_ADDRESS, &PRESS_OUT_H_Address, sizeof(PRESS_OUT_H_Address), 1000);
328 uint8_t PRESS_OUT_H_Value = 0xff; // Junk default value
329 HAL_I2C_Master_Receive(&hi2c2, LPS22HB_READ_ADDRESS, (uint8_t *)&PRESS_OUT_H_Value, sizeof(PRESS_OUT_H_Value), 1000);
330
331 // Variable to hold raw pressure output from sensor. Shift each PRESS_OUT_x variable in as directed in datasheet
332 uint32_t PRESS_OUT_Value = (PRESS_OUT_XL_Value << 0);
333 PRESS_OUT_Value = PRESS_OUT_Value | (PRESS_OUT_L_Value << 8);
334 PRESS_OUT_Value = PRESS_OUT_Value | (PRESS_OUT_H_Value << 16);
335
336 // Divide by 4096 as per datasheet to get units hPa and print results to console
337 uint32_t pressureValue = PRESS_OUT_Value / 4096;
338 snprintf(buffer, sizeof(buffer), "\n\tPressure: %luhPa\n", pressureValue);
339 HAL_UART_Transmit(&huart1, (uint8_t *) buffer, strlen(buffer), 1000);
340
341 }
342
```

2. Display your data from your sensor on the UART1 console.

With all code developed, compiled and flashed without errors or warnings, we can open PuTTY and are presented with the CLI. Selecting option 1 results in the below:



```
COM3 - PuTTY

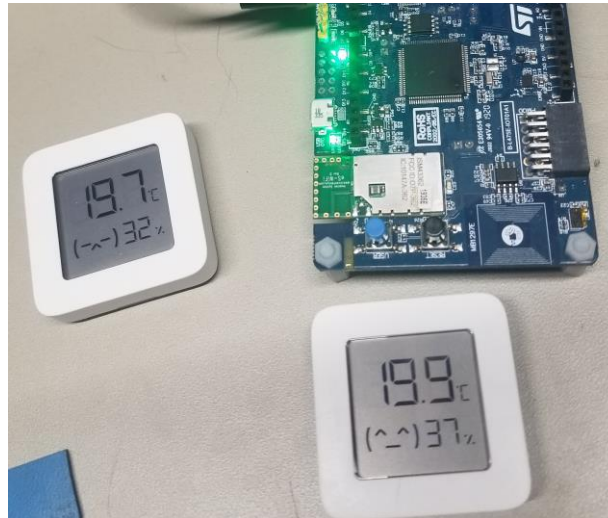
simpleCLI Interface v0.4
-----
Options:
    1: HTS221 read
    2: LPS22HB read
    3: RTC read (not implemented)

$> 1

HTS221 read request:
    Requesting new sample...
    Humidity: 28%rH
    Temperature: 20degC
```

Date: 6/1/2021

As in the other course, I sanity checked this reading against my handy temp/ RH monitors and found that this value was within a reasonable range to their, especially considering that they're not the most accurate sensors to begin with; for example: note the difference in RH even between the two units.



Selecting option 2, we get:

```
Options:
 1: HTS221 read
 2: LPS22HB read
 3: RTC read (not implemented)

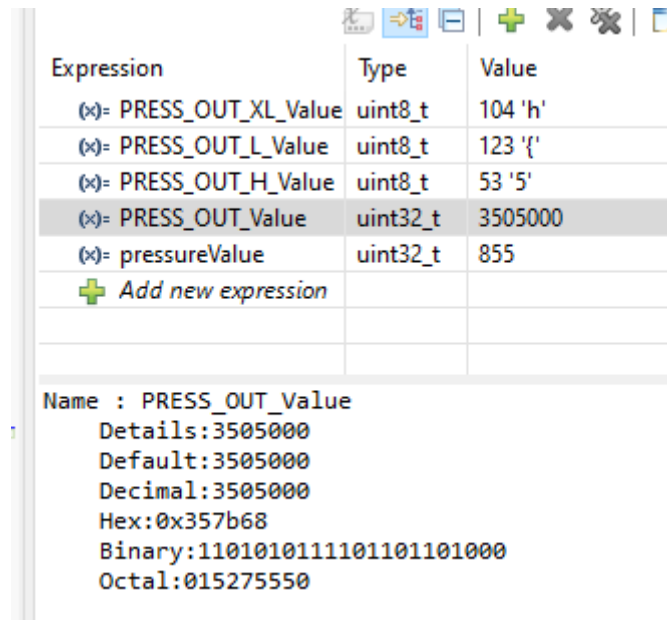
$> 2

LPS22HB read request
Requesting new sample....
Pressure: 855hPa
```

I'll say up front: this number seems **very** low compared to what I would have expected. It is entirely possible (likely?) I missed a step in the process outlined in the datasheet but not having another pressure reference handy to compare with I reviewed my code process and then compared the values of each register to those in the final converted variable in the debugger and confirmed that they at least were correct. So, in the example below, the value of *PRESS_OUT_Value* is 0x357b68, and the values of *PRESS_OUT_XL*, *PRESS_OUT_L*, &

Date: 6/1/2021

PRESS_OUT_H were 0x68, 0x7b, & 0x35, respectively. I can say with confidence that my data handling is correct, even if I might be missing something else in the overall process. Alternatively, my basement could just be at an extremely low pressure.



Expression	Type	Value
(x)= PRESS_OUT_XL_Value	uint8_t	104 'h'
(x)= PRESS_OUT_L_Value	uint8_t	123 '{'
(x)= PRESS_OUT_H_Value	uint8_t	53 '5'
(x)= PRESS_OUT_Value	uint32_t	3505000
(x)= pressureValue	uint32_t	855
+ Add new expression		

Name : PRESS_OUT_Value
Details: 3505000
Default: 3505000
Decimal: 3505000
Hex: 0x357b68
Binary: 1101010111101101101000
Octal: 015275550

Closing Thoughts

As in previous assignments in this course, having completed something similar in the sibling course made this task easy to accomplish. I greatly enjoyed jumping into the datasheets for both modules and working out the specific steps needed to pull data out. I really wish I'd had time to incorporate the RTC module but am already considering how that would be a nice feature to implement in the final coffee make project and I think I can see the code developed here finding a way to be reused in that task.