Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

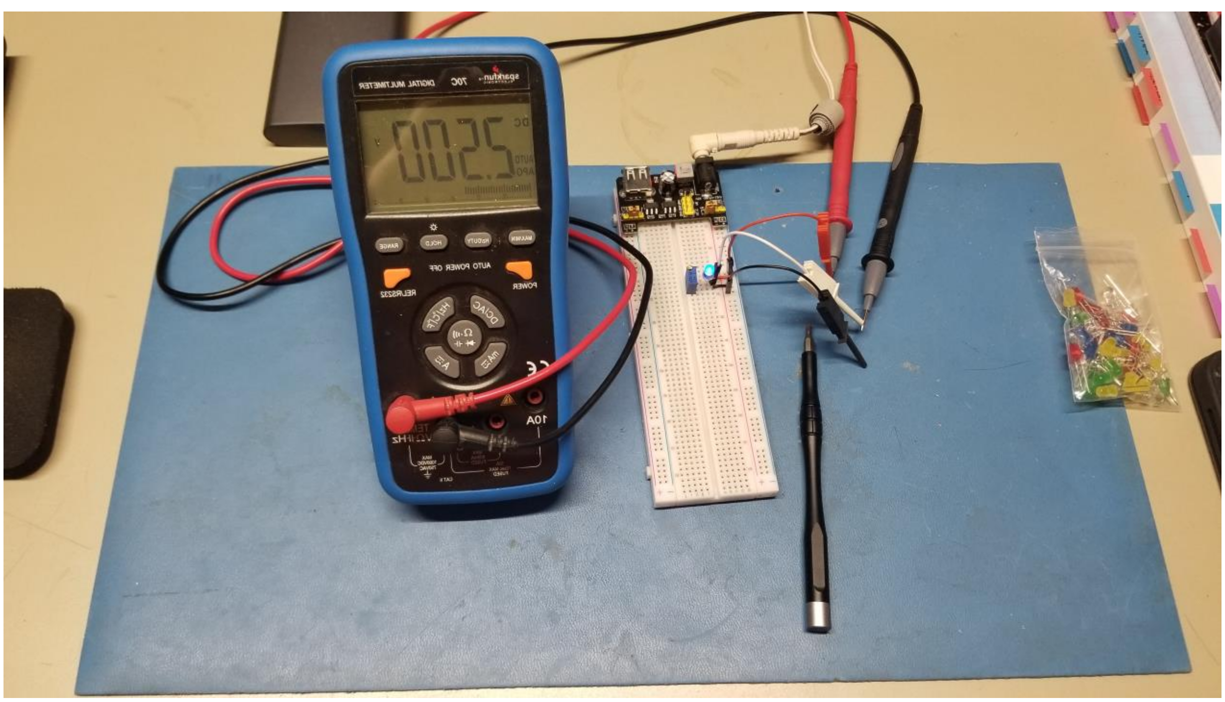
Date: 4/27/2021

Assignment 2: ADC

The following will document completion of the second assignment for ECE-40293, a study of the ADC peripheral on our Discovery boards by completing the following User Stories:

1. Connect ARD-A0 to a 1.5VDC battery and read using ADC Polling mode. Show the output to the console (HAL_UART_Transmit).
2. Connect ARD-A1 to the same 1.5VDC battery and read using ADC Interrupt mode and send the output to the console (HAL_UART_Transmit)
3. Connect ARD-A2 to the same 1.5VDC battery and read using ADC DMA Mode and send the output to the console (HAL_UART_Transmit)
4. Repeat User Story 1, 2, and 3 WITHOUT doing calibration. Do you notice a change in results?
5. Only for ADC Polling Mode, when it comes time to read, instead of reading a single time, read the ADC 12 times "back-to-back", dropping the smallest value and also the largest value. The remaining 10 values should be added and then divided by 10 to find the average. Send this averaged output to the console.
6. Compare and contrast your output from User Story 1 (single ADC reads and then display on console) compared to User Story 5 (12 reads, drop the lowest and highest, then average the 10). What values seem to "jump around" more -- those from User Story 1 or User Story 5?

For the purposes of this assignment, I used a potentiometer connected to a bench supply to provide an analog signal to be read by the Disco board. I also attached the probes of my digital multimeter to the circuit as a sanity check against my initial readings.

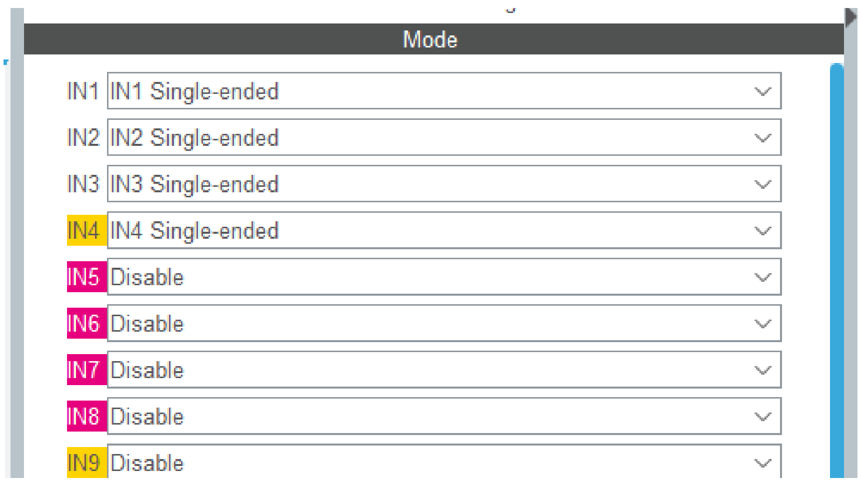


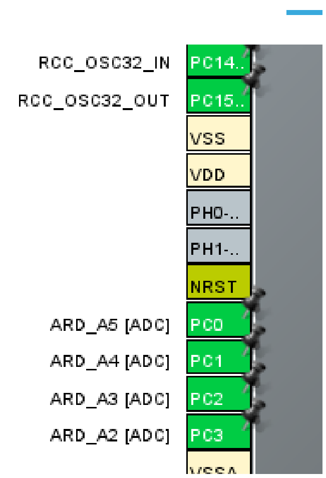
Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

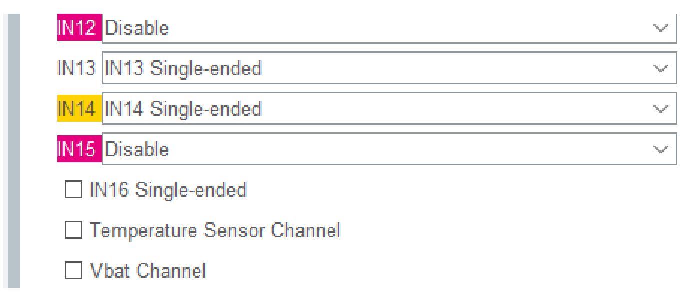
Date: 4/27/2021

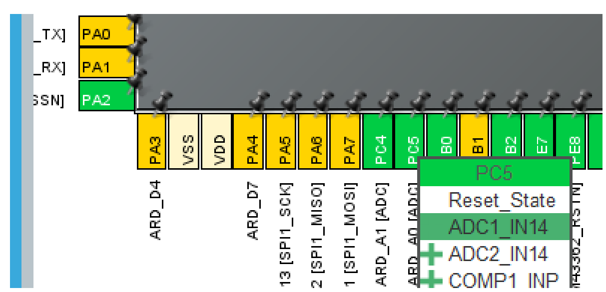
1. ADC Setup & Polling Mode

After opening the IDE and generating a default project for out Disco board as in previous projects, the ADC peripheral needs to be enabled and configured under the project manager interface. For our purposes we'll be using the Arduino analog points 0, 1, & 2 and will need to configure them in single-ended mode for our mock-up input signal. To determine the relation of the Arduino pins to the MCU's analog pins, the graphical pin manager can be consulted, for example ARD_A0 and ADC1_IN14 in the example image below.









From here, we can generate our project files and begin developing the code for each user story. The following code examples all follow roughly the same structure, where in the main() while(1) loop, we will toggle the WiFi/ BLE LEDs pin with a short delay to indicate that something is happening. In the case of this User Story, we'll then enable the ADC1 peripheral in polling mode. After receiving a value, there is a section of string formatting code that will result in the data sent out over the ST-Link's USB-Serial console connection to a terminal, at which point the loop repeats. Example code is below, with supporting comments, followed by a screenshot of the terminal output. I didn't capture this instance on camera, but for this first User Story, I also compared the value read on my meter as what should be a calibrated point of reference.

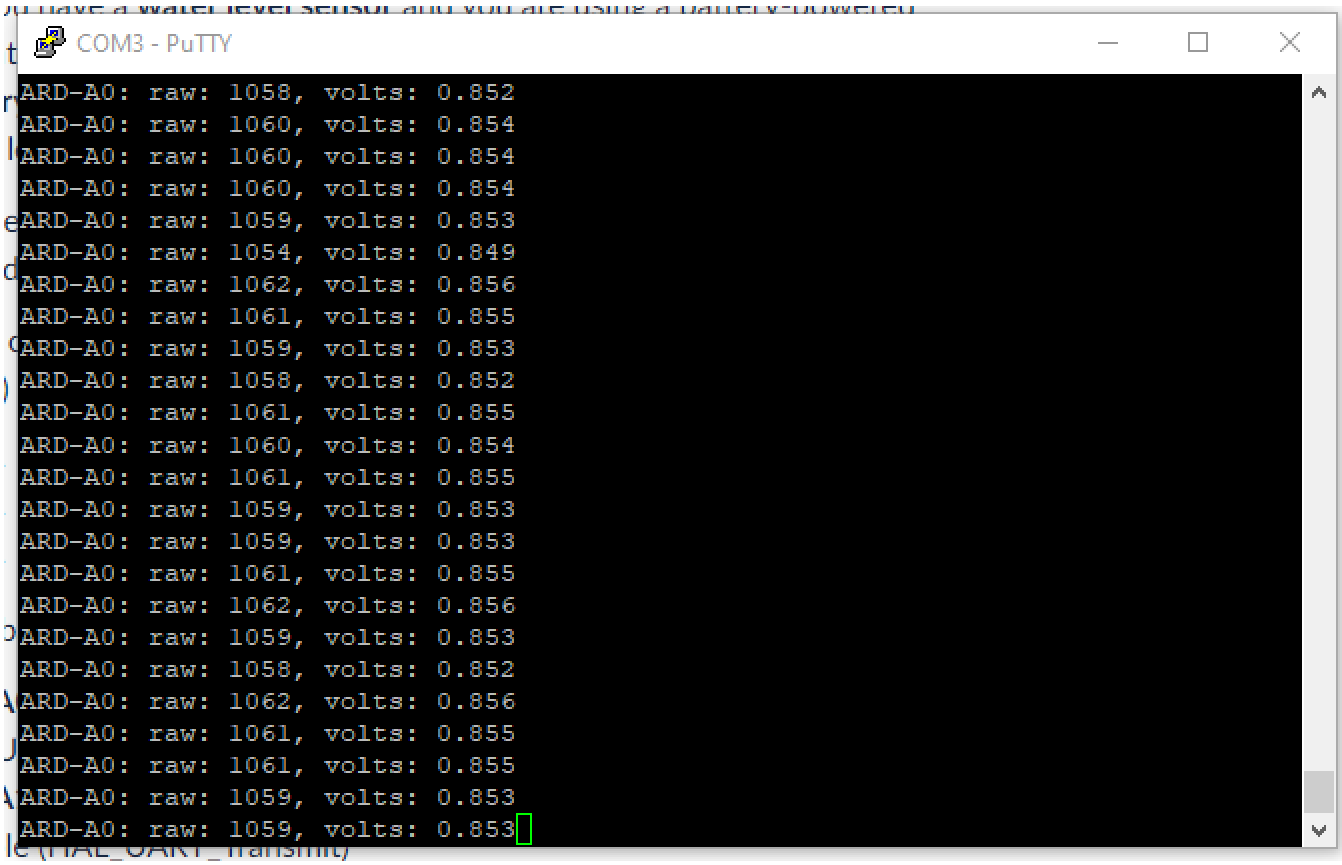
Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021

```c
/************************************
* User Story 1 implementation begin:
*
*     Connect ARD-0 to analog voltage
*     source and read using ADC polling
*     mode. Pipe output to console.
*
************************************/

// Calibrate ADC1
HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);

while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

    // Blink & delay to give indication of *something* happening
    HAL_GPIO_TogglePin(LED3_WIFI__LED4_BLE_GPIO_Port, LED3_WIFI__LED4_BLE_Pin);
    HAL_Delay(1000);

    // Enable ADC1
    HAL_ADC_Start(&hadc1);

    // Poll ADC1 for value
    HAL_ADC_PollForConversion(&hadc1, 10);
    uint16_t valueRaw = HAL_ADC_GetValue(&hadc1);

    // Define formatting strings to provide a clean serial output
    char printString[255] = "\nARD-A0: raw: ";
    char voltsString[20] = ", volts: ";

    // Store value into a buffer
    char bufferRaw[20];
    snprintf(bufferRaw, sizeof(bufferRaw), "%u", valueRaw);

    // Convert raw counts into voltage
    char bufferVoltage[20];
    float valueVoltage= (valueRaw * (3.3/4096));
    snprintf(bufferVoltage, sizeof(bufferVoltage), "%5.3f", valueVoltage);

    // Concat the formatting strings and data to an output
    // to send out over the UART1 serial port/ USB-micro
    // port to PC
    strcat(printString, bufferRaw);
    strcat(printString, voltsString);
    strcat(printString, bufferVoltage);

    HAL_UART_Transmit(&huart1, (uint8_t *) printString, strlen(printString), 1000);

    /************************************
    * User Story 1 implementation end
    *
    ************************************/
```

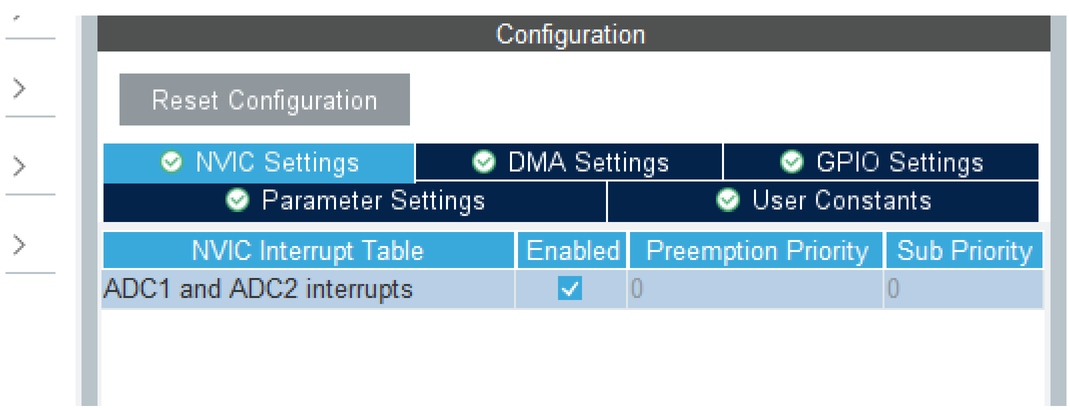
Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021



2. ADC Interrupt Mode

Moving on to the interrupt-driven example of User Story 2, the code is similar to the style used in the Polling method, except we will wrap up the bulk of it into an interrupt callback function. Prior to that, we'll also need to enable the ADC interrupts from the configuration interface and re-generate our project to include the appropriate support files.



Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021

After that, we'll develop a callback function definition in the User Code area that will serve as a wrapper for our data formatting and printing that fires once the interrupt and data conversion is complete.

```c
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

void HAL_ADC_ConvCplt_Callback(ADC_HandleTypeDef *hadc)
{
    uint16_t valueRaw = HAL_ADC_GetValue(&hadc1);

    // Define formatting strings to provide a clean serial output
    char printString[255] = "\nARD-A1: raw: ";
    char voltsString[20] = ", volts: ";

    // Store value into a buffer
    char bufferRaw[20];
    snprintf(bufferRaw, sizeof(bufferRaw), "%u", valueRaw);

    // Convert raw counts into voltage
    char bufferVoltage[20];
    float valueVoltage= (valueRaw * (3.3/4096));
    snprintf(bufferVoltage, sizeof(bufferVoltage), "%5.3f", valueVoltage);

    // Concat the formatting strings and data to an output
    // to send out over the UART1 serial port/ USB-micro
    // port to PC
    strcat(printString, bufferRaw);
    strcat(printString, voltsString);
    strcat(printString, bufferVoltage);

    HAL_UART_Transmit(&huart1, (uint8_t *) printString, strlen(printString), 1000);
}

/* USER CODE END 0 */
```

Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021

And then in our while(1) loop, the modified version of our earlier code with most of the heavy lifting moved into the callback function. Note the use of HAL_ADC_Start_IT() in this case.

```
// Calibrate ADC1
HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);

while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

  /*************************************
  * User Story 2 implementation begin:
  *
  *   Connect ARD-1 to analog voltage
  *   source and read using ADC interrupt
  *   mode. Pipe output to console.
  *
  ************************************/

    // Blink & delay to give indication of *something* happening
    HAL_GPIO_TogglePin(LED3_WIFI__LED4_BLE_GPIO_Port, LED3_WIFI__LED4_BLE_Pin);
    HAL_Delay(1000);

    // Enable ADC1 in IT mode
    HAL_ADC_Start_IT(&hadc1);


  /*************************************
  * User Story 2 implementation end
  *
  ************************************/
}
/* USER CODE END 3 */
```

After compiling and flashing the updated project, we can see the following on the serial terminal:

COM3 - PuTTY

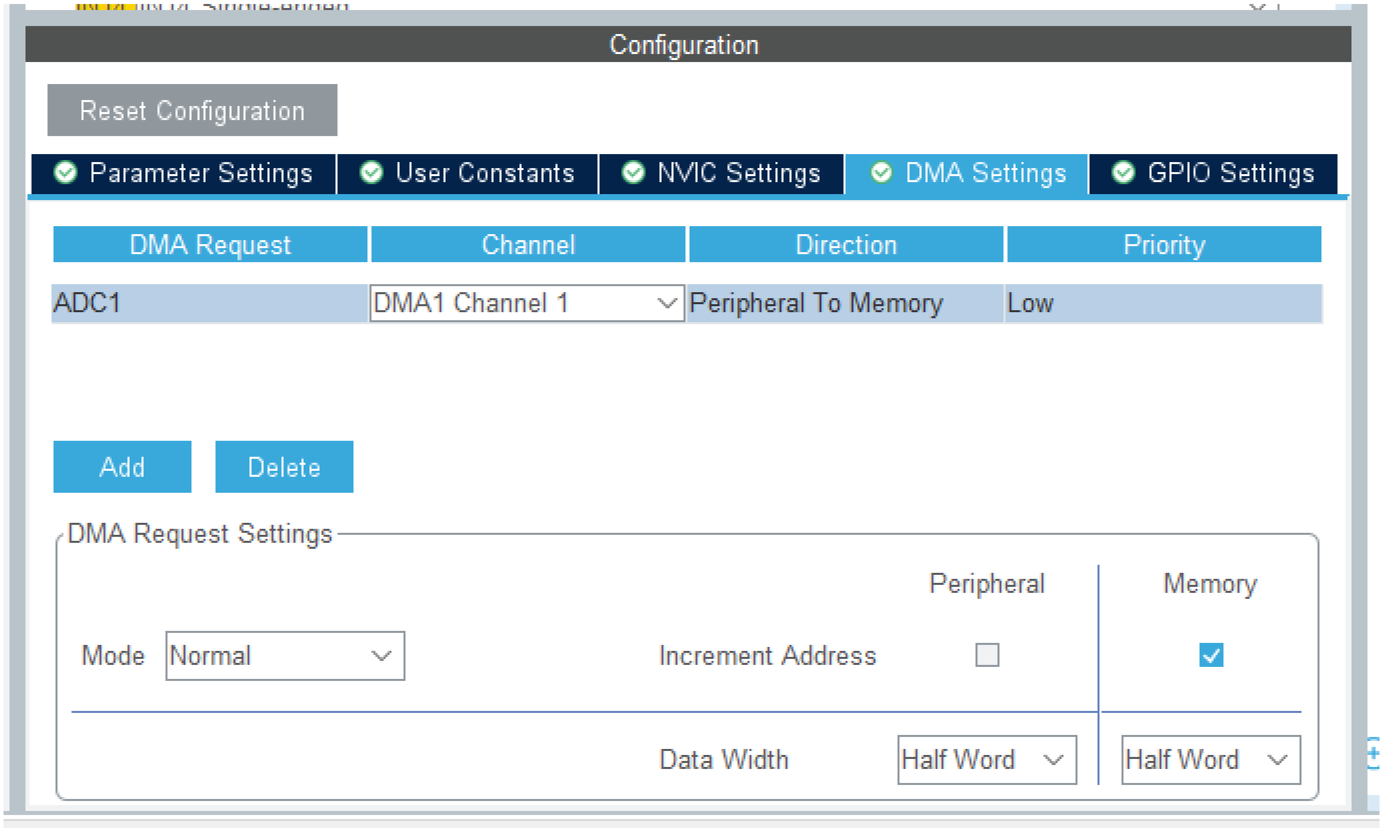
```
Starting ADC IT mode...

ARD-A1: raw: 1049, volts: 0.845
ARD-A1: raw: 991, volts: 0.798
ARD-A1: raw: 992, volts: 0.799
ARD-A1: raw: 992, volts: 0.799
ARD-A1: raw: 991, volts: 0.798
ARD-A1: raw: 991, volts: 0.798
ARD-A1: raw: 989, volts: 0.797
ARD-A1: raw: 991, volts: 0.798
```

Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021

3. ADC DMA Mode

Moving onto the third User Story, using DMA mode, we will start by going back into the project configuration and enabling DMA for ADC1.



Date: 4/27/2021

We can then modify our callback function, the key difference compared to the interrupt version is that the variable adcValue is moved to outside of the function definition to give it the scope necessary for access within main() and the callback functions.

```c
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

static uint16_t adcValue;

void HAL_ADC_ConvCplt_Callback(ADC_HandleTypeDef *hadc)
{
    // Define formatting strings to provide a clean serial output
    char printString[255] = "\nARD-A1: raw: ";
    char voltsString[20] = ", volts: ";

    // Store value into a buffer
    char bufferRaw[20];
    snprintf(bufferRaw, sizeof(bufferRaw), "%u", adcValue);

    // Convert raw counts into voltage
    char bufferVoltage[20];
    float valueVoltage= (adcValue * (3.3/4096));
    snprintf(bufferVoltage, sizeof(bufferVoltage), "%5.3f", valueVoltage);

    // Concat the formatting strings and data to an output
    // to send out over the UART1 serial port/ USB-micro
    // port to PC
    strcat(printString, bufferRaw);
    strcat(printString, voltsString);
    strcat(printString, bufferVoltage);

    HAL_UART_Transmit(&huart1, (uint8_t *) printString, strlen(printString), 1000);
}

/* USER CODE END 0 */

```

Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021

Then within the while(1) loop, modify the function call to HAL_ADC_Start_DMA(), providing ADC1 and adcValue as parameters.

```c

/************************************
 * User Story 3 implementation begin:
 *
 *      Connect ARD-0 to analog voltage
 *      source and read using ADC DMA
 *      mode. Pipe output to console.
 *
 ***********************************/

// Calibrate ADC1
HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);

while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

    // Blink & delay to give indication of *something* happening
    HAL_GPIO_TogglePin(LED3_WIFI__LED4_BLE_GPIO_Port, LED3_WIFI__LED4_BLE_Pin);
    HAL_Delay(1000);

    // Enable ADC1
    HAL_ADC_Start_DMA(&hadc1, &adcValue, 1);

    /************************************
     * User Story 3 implementation end
     *
     ***********************************/

}
/* USER CODE END 3 */
}
```

Then within in the terminal, we'll see:

COM3 - PuTTY

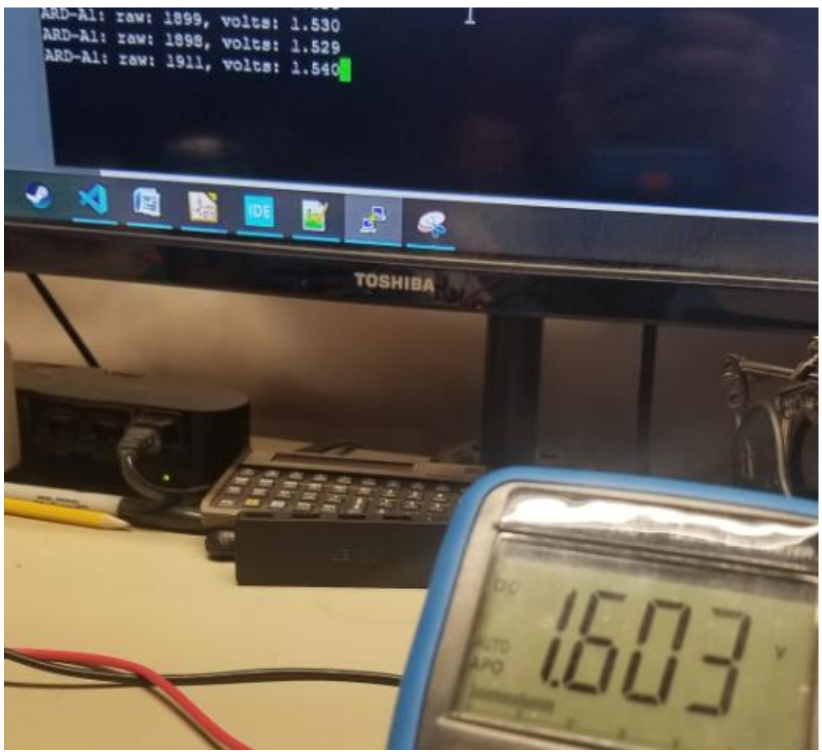
```
Starting ADC DMA mode...

ARD-A2: raw: 1056, volts: 0.851
ARD-A2: raw: 1006, volts: 0.810
ARD-A2: raw: 1005, volts: 0.810
ARD-A2: raw: 1005, volts: 0.810
ARD-A2: raw: 1007, volts: 0.811
ARD-A2: raw: 1006, volts: 0.810
ARD-A2: raw: 1005, volts: 0.810
ARD-A2: raw: 1007, volts: 0.811
```

Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021

4. Calibration

Something I haven’t mentioned it in the previous User Stories is the HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED) function call that occurs prior to entering the while(1) loop. This is supposed to be used to calibrate the ADC against known values burned into ROM during the manufacturing process. Out of curiosity, I compared the values on my meter to the readings fed back from the Disco board and found that they were close at the upper and lower ends of the signal range but off by a decent percent in the midrange values. In the example below, I was reading 1.603 V on the meter, which I’ll use as a control reference, versus about 1.530 V from the board in Interrupt mode, or roughly a 5% error. My initial thoughts were that that large of difference wasn’t very impressive but then I considered that 1) I only am using a single reading on that calculation versus several or dozens & 2) my breadboard setup with cheap jumper wires and jellybean potentiometer isn’t super accurate to begin with so I decided to reserve any condemnation of the peripheral.



Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021

Running through stories 1, 2, & 3 again with the calibration disabled, I noticed that there was a shift downward by about 100mV. I neglected to document my meter's readings during this period but the values corroborated that downward shift versus the control. Examples of the IT and DMA readings are below.

COM3 - PuTTY

```
Starting ADC IT mode with out calibration...

ARD-A1: raw: 985, volts: 0.794
ARD-A1: raw: 928, volts: 0.748
ARD-A1: raw: 929, volts: 0.748
ARD-A1: raw: 927, volts: 0.747
ARD-A1: raw: 927, volts: 0.747
ARD-A1: raw: 927, volts: 0.747
ARD-A1: raw: 928, volts: 0.748
```

COM3 - PuTTY

```
Starting ADC DMA mode without calibration...

ARD-A2: raw: 991, volts: 0.798
ARD-A2: raw: 943, volts: 0.760
ARD-A2: raw: 943, volts: 0.760
ARD-A2: raw: 944, volts: 0.761
ARD-A2: raw: 941, volts: 0.758
ARD-A2: raw: 943, volts: 0.760
ARD-A2: raw: 944, volts: 0.761
```

5. Averaged Values

The better way to compare the ADC readings versus a control reading would be to average several readings together after discarding the high and low outliers for good measure. For User Story 5, I developed the below code to take 12 readings, trim off the top and bottom ones, and then average all of them together. I should qualify that this might not be the most efficiently written code but fell squarely into the "good enough" category for meet the requirements of the assignment. It also ended up at about 100 formatted lines so I won't include screenshots of it in this document, but an excerpt file will be included with this submission as "user_story_5_code.c". With the averaging code uploaded, we see a much more consistent reading reported back on the serial terminal:

```
Average of 10 readings on ARD-A1: raw: 2546, volts: 2.051
Average of 10 readings on ARD-A1: raw: 2534, volts: 2.042
Average of 10 readings on ARD-A1: raw: 2542, volts: 2.048
Average of 10 readings on ARD-A1: raw: 2545, volts: 2.050
Average of 10 readings on ARD-A1: raw: 2535, volts: 2.042
Average of 10 readings on ARD-A1: raw: 2544, volts: 2.050
Average of 10 readings on ARD-A1: raw: 2544, volts: 2.050
Average of 10 readings on ARD-A1: raw: 2526, volts: 2.035
Average of 10 readings on ARD-A1: raw: 2543, volts: 2.049
Average of 10 readings on ARD-A1: raw: 2545, volts: 2.050
```

6. Compare single to averaged readings

Considering my previous comment about the breadboard test setup not being the most accurate tool to begin with, I found that over the course of these tests, the averaged readings absolutely would provide more consistency. Anyone who has any experience with real world sensors or even a basic understanding of statistics would likely agree that using single or "snapshot" readings for anything other than a quick point of reference is a bad idea. I can think of multiple applications in my professional

Nathan Bunnell
Embedded Controller Programming with C
ECE-40293
Student ID: U08895857

Date: 4/27/2021

experience where we would even take several thousand readings to provide an average over even just a few moments, for example in monitoring of our process water supply or natural gas flow. I can also think of times when coworkers who were either fresh out of school or who weren't exercising proper engineering discipline worked themselves into a frenzy over what ended up being single points of analog noise.

Closing thoughts

I have worked with analog signals extensively in my career, typically of single-ended 4-20mA inputs to the various PLCs in my SCADA systems. During this assignment, I looked up the number of AIs in our I/O database and found that we have somewhere in the order of 25,000 signals in some form of use throughout the site. Of course, using the ADCs on a PLC analog input card is much more high level that the exercises in this assignment so it was a great learning experience to dive down a bit deeper into the configuration and usage of an ADC peripheral than I had previously experienced. I also greatly enjoyed the opportunity to break out some of my lab tools at home that have otherwise sat unused for too long.