Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

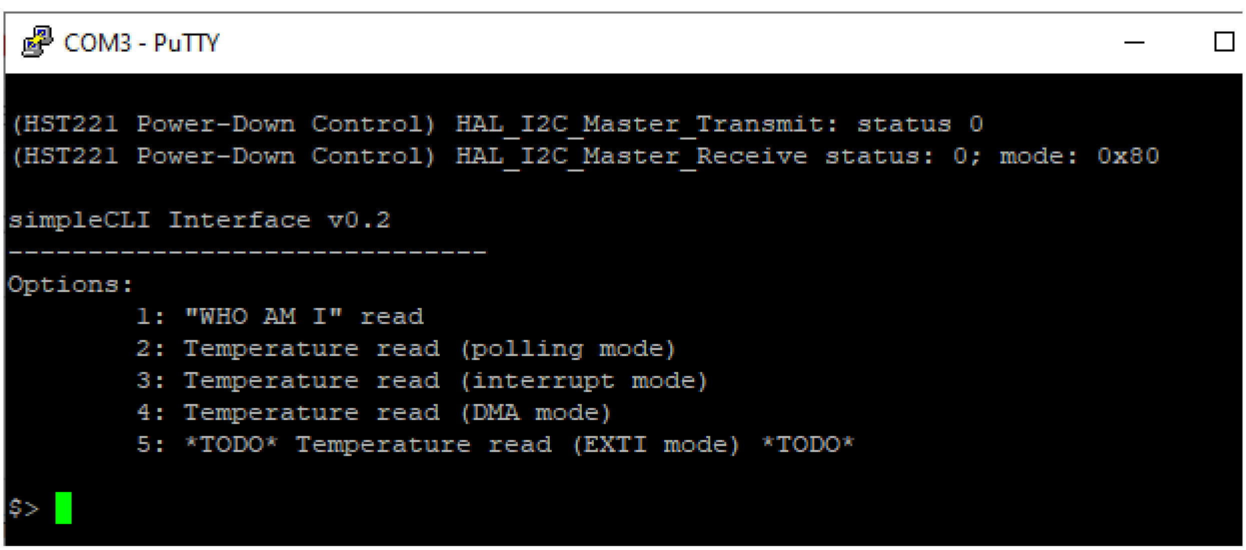
Assignment 4: I2C Hands On

The following will document completion of the fourth assignment for ECE-40293, using the onboard serial hardware to complete the following user stories:

1. Create a CLI (Command Line Interface) on UART1 that prompts you to enter a "1" for WHO_AM_I using polling, a 2 temperature read using polling, and 3 for temperature readings using interrupts, and a 4 for temperature read using DMA
2. WHO_AM_I (Polling). When the user selects 1, display contents of WHO_AM_I register from HTS221
3. Temperature (Polling). The user selects 2, use polling to read HTS221 and display temperature on console.
4. Temperature (Interrupts). When the user selects 3, use interrupts to read from HTS221 and display temperature on console.
5. Temperature (DMA). When the user selects 4, use DMA to read from HTS221 and display temperature on console.

1. Create a CLI

To start, we will open the IDE and generate our default project as in previous examples. We will leverage the code developed in the Serial Hands On assignment to prevent duplication of efforts in producing this basic interface. All project code will be attached with this report submission for reference but the result will present the following under a serial console connection:

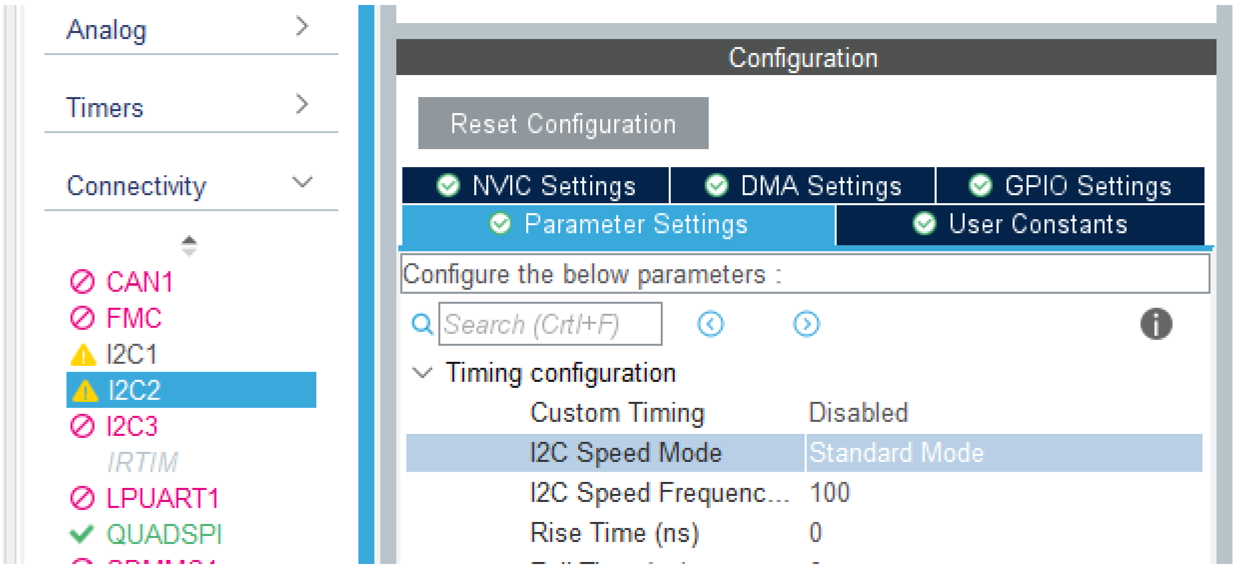


Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

2. WHO AM I

Prior to being able to interface with the HTS221 sensor, we will need to confirm under the project configuration interface that the I2C2 interface is appropriately configured, which, conveniently, is also the project default.



From there we develop our *do_who_am_i()* function definition, which will send the address of our target WHO_AM_I register to the target device, report the status of that transmission over the console, and then read back the response out over the console.

```
static void do_who_am_i(void)
{
    // Declare address of the device's WHO_AM_I register
    uint8_t whoAmIReg = 0xf;

    // Send the target register to the device and get status back
    HAL_StatusTypeDef status;
    status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &whoAmIReg, sizeof(whoAmIReg), 1000);

    // Print status results over UART1 to console session
    char buffer[100] = {0};
    snprintf(buffer, sizeof(buffer), "\tHAL_I2C_Master_Transmit status: %u\n", status);
    HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

    // Read response back to get value of WHO_AM_I register
    uint8_t data = 0xff;    // Default value should be 0xbc according to datasheet
    status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, &data, sizeof(data), 1000);

    // Print status results and response value over UART1 to console session
    snprintf(buffer, sizeof(buffer), "\tHAL_I2C_Master_Receive status: %u; data: 0x%x\n\n", status, data);
    HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
}
```

Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

As per the device datasheet, we expect to get a response of “0xBC” foor the contents of the WHO_AM_I register, and the results on the serial console correspond to that value.

```
$> 1

"WHO AM I" request:
        HAL_I2C_Master_Transmit status: 0
        HAL_I2C_Master_Receive status: 0; data: 0xbc
```

3. Temperature reading with polling

To read the values of the temperature sensor, we will need to ensure that the power-down control bit of CTRL_REG1 in the sensor is set to activate the sensor. To manage this, we will call function *HST221_pwr_en()* prior to entering the looped portion of the prompt. The results of the transmission and reception from the sensor can be seen printed out at the top of the prompt after the board is rest.

```
COM3 - PuTTY

(HST221 Power-Down Control) HAL_I2C_Master_Transmit: status 0
(HST221 Power-Down Control) HAL_I2C_Master_Receive status: 0; mode: 0x80

simpleCLI Interface v0.2
```

At this point, we can develop the actual polling function, *do_temp_polling()*, which was implemented using both example methods, which alternate between individual calls to the function from the prompt. Both methods will send a write request to the sensor’s CTRL_REG2 to enable the conversion and will then wait until the STATUS_REG reports that a temperature sample is ready to be read.

Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

main.c

```c
static void do_temp_polling(void)
{
    // Large-ish char buffer for strings sent over the console
    char buffer[100] = {0};

    // Configure control register 2 (CTRL_REG2, 0x21) bit 0 to enable one-shot
    uint8_t ctrlReg2 = 0x21;
    uint8_t ctrlData[] = {ctrlReg2, (1 << 0)};

    // Send the target register to the device and get status back
    HAL_StatusTypeDef status;
    status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, ctrlData, sizeof(ctrlData), 1000);

    // Print status results over UART1 to console session
    snprintf(buffer, sizeof(buffer), "\t(One-shot enabled): HAL_I2C_Master_Transmit: status %u\n", status);
    HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

    // Define status register (STATUS_REG2, 0x27) bit 0 to monitor for new sample available
    uint8_t statusReg = 0x27;
    uint8_t sampleReady = 0;

    // Loiter for a bit to allow time for conversion to complete and be made available
    uint8_t count = 0;
    while (count < 10)  // 10 is an arbitrary "long enough" value, this wouldn't always be great real-world practice
    {
        // Send the address of the status register and report it over the console
        status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &statusReg, sizeof(statusReg), 1000);
        snprintf(buffer, sizeof(buffer), "\t[%d] (status register): HAL_I2C_Master_Transmit: status %u\n", count, status);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

        // Read back the value of the status register and report it over the console
        status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&sampleReady, sizeof(sampleReady), 1000);
        snprintf(buffer, sizeof(buffer), "\tStatus register: 0x%02x\n", sampleReady);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

        // If the new sample is ready, report it to the console and break out of while-loop...
        if (sampleReady & 0x01)
        {
            snprintf(buffer, sizeof(buffer), "\tNew Temperature Sample Available!\n");
            HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
            break;
        }

        // Else wait for a bit, increment the counter, and keep looping
        HAL_Delay(100);
        count++;
    }

```

Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

From here, the first method will send a pair of requests for the values of the TEMP_L & TEMP_H registers as a pair of uint8_t variables that represent the reading's LSB and MSB, printing the LSB and then MSB to the console along with reports on the status of each I2C read/write interaction.

```c
    // With new sample ready, practice solutions implemented with reading sequentially from
    //  the LSB and MSB registers (0x2a and 0x2b) as well as via auto-increment
    static uint8_t toggle = 1;

    if (toggle)
    {
        toggle = 0;

        // Reading the lower half of the temperature register
        // Send target address
        uint8_t tempRegLSB = 0x2a;
        status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &tempRegLSB, sizeof(tempRegLSB), 1000);
        snprintf(buffer, sizeof(buffer), "\t(LSB): HAL_I2C_Master_Transmit: status: %u\n", status);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

        // Read response back and print over console
        uint8_t tempDataLSB = 0xff; // Junk default value
        status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&tempDataLSB, sizeof(tempDataLSB), 1000);
        snprintf(buffer, sizeof(buffer), "\t(LSB): HAL_I2C_Master_Receive: status: %u; LSB data: 0x%02x\n", status, tempDataLSB);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

        // Reading the upper half of the temperature register
        // Send target address
        uint8_t tempRegMSB = 0x2b;
        status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &tempRegMSB, sizeof(tempRegMSB), 1000);
        snprintf(buffer, sizeof(buffer), "\t(MSB): HAL_I2C_Master_Transmit: status: %u\n", status);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

        // Read response back and print over console
        uint8_t tempDataMSB = 0xff; // Junk default value
        status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&tempDataMSB, sizeof(tempDataMSB), 1000);
        snprintf(buffer, sizeof(buffer), "\t(MSB): HAL_I2C_Master_Receive: status: %u; MSB data: 0x%02x\n\n", status, tempDataMSB);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
    }
    else
```

```
$> 2

Temperature read (poll) request:
        (One-shot enabled): HAL_I2C_Master_Transmit: status 0
        [0] (status register): HAL_I2C_Master_Transmit: status 0
        Status register: 0x03
        New Temperature Sample Available!
        (LSB): HAL_I2C_Master_Transmit: status: 0
        (LSB): HAL_I2C_Master_Receive: status: 0; LSB data: 0xc8
        (MSB): HAL_I2C_Master_Transmit: status: 0
        (MSB): HAL_I2C_Master_Receive: status: 0; MSB data: 0x00
```

Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

The second method will provide a similar result, except that by ORing the eighth bit of the TEMP_L registers address with a 1 (0x80), we are able to auto increment to get the MSB back at the same time as a uint16_t in a single write/read interaction.

```
    }
    else
    {
        toggle = 1;

        // Reading the lower half of the temperature register with auto-increment enabled
        // Send target address (OR'ing with 0x80 enables auto-inc)
        uint8_t tempRegLSB = 0x2a | 0x80;
        status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &tempRegLSB, sizeof(tempRegLSB), 1000);
        snprintf(buffer, sizeof(buffer), "\t(Auto-increment): HAL_I2C_Master_Transmit: status: %u\n", status);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

        // Read response back for both registers and print over console
        uint16_t tempData = 0xbeef; // Junk default value (ALSO REALLY HOT!)
        status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&tempData, sizeof(tempData), 1000);
        snprintf(buffer, sizeof(buffer), "\t(Auto-increment): HAL_I2C_Master_Receive: status: %u; temperature data: 0x%04x\n\n", status, tempData);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
    }
```

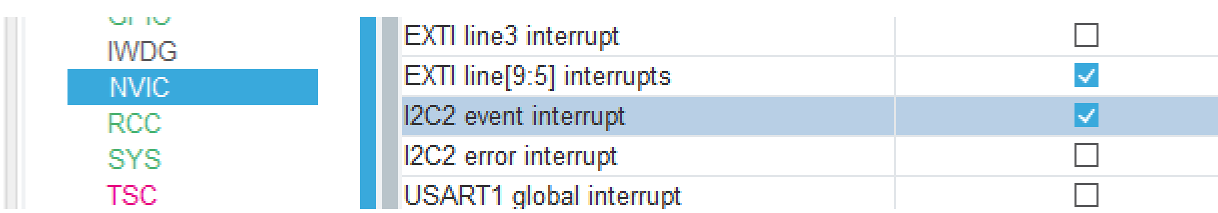
We see the difference, and I believe added clarity of this method in the console quite clearly:

```
$> 2

Temperature read (poll) request:
        (One-shot enabled): HAL_I2C_Master_Transmit: status 0
        [0] (status register): HAL_I2C_Master_Transmit: status 0
        Status register: 0x03
        New Temperature Sample Available!
        (Auto-increment): HAL_I2C_Master_Transmit: status: 0
        (Auto-increment): HAL_I2C_Master_Receive: status: 0; temperature data: 0x00c8
```

The nature of how this function was implemented means that the user experience for this option will alternate between the two per each time that this option is selected from the prompt.

4. Temperature reading with interrupts

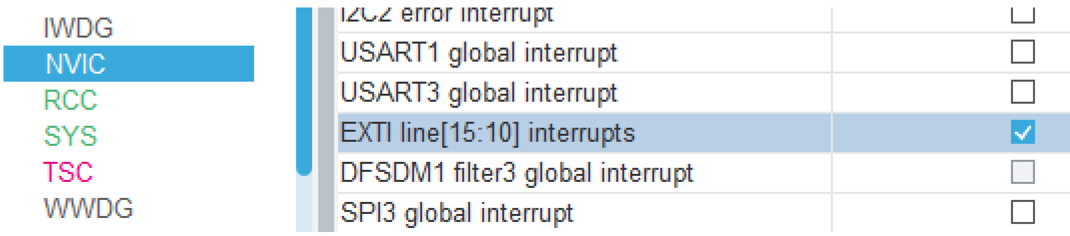
To accomplish the same with interrupts, we'll need to take the additional steps of configuring the NVIC interface to enable interrupts on the I2C2 peripheral, as seen below:



Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

If we were to use the sensor's data ready signal on EXTI15, we would need to do the same for that entry in the NVIC interface.



For the purposes of this assignment (and mainly due to personal time constraints) I will only look at the I2C2 peripheral interrupt as we have previously examined external interrupt techniques and the methods required would be a mix of those previously used with the I2C-related portions added in. As seen in previous examples, we will develop both the interrupt function and a pair of callback functions related to Tx and Rx events completing. The callbacks will be extremely simple, in this case only toggling one of the user LEDs to indicate activity on the peripheral interface and then setting a flag to indicate that the INT has completed, as seen here:

```
void HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c)
{
    // Indicate something on the board
    HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);

    // Set status flag
    irqCompleteTX = 1;
}

void HAL_I2C_MasterRxCpltCallback(I2C_HandleTypeDef *hi2c)
{
    // Indicate something on the board
    HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);

    // Set status flag
    irqCompleteRX = 1;
}
```

The *do_temp_interrupt()* code will have similarities to the polling method explored above, however we will leverage the "_IT" version of the associated function calls for I2C transmit and receive operations. Code excerpt follows but ca nbe seen more clearly by examining the *main.c* project file.

Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

```c
static void do_temp_interrupt(void)
{
        // Clear flags
        irqCompleteTX = 0;
        irqCompleteRX = 0;

    // Large-ish char buffer for strings sent over the console
    char buffer[100] = {0};

        // Configure control register 2 (CTRL_REG2, 0x21) bit 0 to enable one-shot
    uint8_t ctrlReg2 = 0x21;
    uint8_t ctrlData[] = {ctrlReg2, 0x01};

    // Send the target register to the device and get status back using the *_IT function
    HAL_StatusTypeDef status;
    status = HAL_I2C_Master_Transmit_IT(&hi2c2, HST221_WRITE_ADDRESS, ctrlData, sizeof(ctrlData));

    // Print status results over UART1 to console session
    snprintf(buffer, sizeof(buffer), "\t(One-shot enabled): HAL_I2C_Master_Transmit_IT: status %u\n", status);
    HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

    // Wait for interrupt to complete
        while (irqCompleteTX == 0)
        {
                HAL_Delay(1000);
        }

        // Clear flag
        irqCompleteTX = 0;

    // Reading the lower half of the temperature register with auto-increment enabled
        // Send target address (OR'ing with 0x80 enables auto-inc)
        uint8_t tempRegLSB = 0x2a | 0x80;
        status = HAL_I2C_Master_Transmit_IT(&hi2c2, HST221_WRITE_ADDRESS, &tempRegLSB, sizeof(tempRegLSB));
        snprintf(buffer, sizeof(buffer), "\t(Auto-increment): HAL_I2C_Master_Transmit_IT: status: %u\n", status);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

    // Wait for interrupt to complete
        while (irqCompleteTX == 0)
        {
                HAL_Delay(1000);
        }

        // Read response back for both registers and print over console
        status = HAL_I2C_Master_Receive_IT(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&tempData, sizeof(tempData));
        snprintf(buffer, sizeof(buffer), "\t(Auto-increment): HAL_I2C_Master_Receive_IT: status: %u; temperature data: 0x%04x\n\n", status, tempData);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
}
```

With this option selected from the CLI, we see the following results read back from the prompt:

```
$> 3

Temperature read (IT) request:
        (One-shot enabled): HAL_I2C_Master_Transmit_IT: status 0
        (Auto-increment): HAL_I2C_Master_Transmit_IT: status: 0
        (Auto-increment): HAL_I2C_Master_Receive_IT: status: 0; temperature data: 0x00ef
```

Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

5. Temperature reading with interrupts

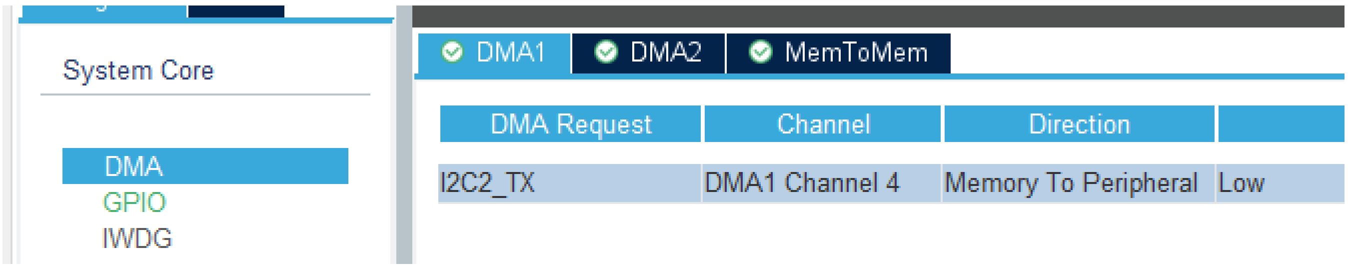
Moving into the final user story, we will leverage a function definition for *do_temp_dma()* that is extremely similar to the interrupt versio0n, of course substituting the "_IT" calls for the "_DMA" versions.

```
static void do_temp_dma(void)
{
        // Clear flags
    irqCompleteTX = 0;
    irqCompleteRX = 0;

    // Large-ish char buffer for strings sent over the console
    char buffer[100] = {0};

        // Configure control register 2 (CTRL_REG2, 0x21) bit 0 to enable one-shot
    uint8_t ctrlReg2 = 0x21;
    uint8_t ctrlData[] = {ctrlReg2, 0x01};

    // Send the target register to the device and get status back using the *_DMA function
    HAL_StatusTypeDef status;
    status = HAL_I2C_Master_Transmit_DMA(&hi2c2, HST221_WRITE_ADDRESS, ctrlData, sizeof(ctrlData));

    // Print status results over UART1 to console session
    snprintf(buffer, sizeof(buffer), "\t(One-shot enabled): HAL_I2C_Master_Transmit_DMA: status %u\n", status);
    HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

    // Wait for interrupt to complete
        while (irqCompleteTX == 0)
        {
                HAL_Delay(100);
        }

        // Clear flag
        irqCompleteTX = 0;

    // Reading the lower half of the temperature register with auto-increment enabled
        // Send target address (OR'ing with 0x80 enables auto-inc)
        uint8_t tempRegLSB = 0x2a | 0x80;
        status = HAL_I2C_Master_Transmit_DMA(&hi2c2, HST221_WRITE_ADDRESS, &tempRegLSB, sizeof(tempRegLSB));
        snprintf(buffer, sizeof(buffer), "\t(Auto-increment): HAL_I2C_Master_Transmit_DMA: status: %u\n", status);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);

    // Wait for interrupt to complete
    while (irqCompleteTX == 0)
    {
        HAL_Delay(100);
    }

    // Read response back for both registers and print over console
        status = HAL_I2C_Master_Receive_DMA(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&tempData, sizeof(tempData));
        snprintf(buffer, sizeof(buffer), "\t(Auto-increment): HAL_I2C_Master_Receive_DMA: status: %u; temperature data: 0x%04x\n\n", status, tempData);
        HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer), 1000);
}
```

Nathan Bunnell
Embedded Systems Hardware Interfacing
ECE-40293
Student ID: U08895857

Date: 5/10/2021

And again, as with the INT version, we will need to adjust our configuration to enable the I2C2_TX DMA request from the configurator:



With this option then selected within the CLI prompt, we see the following results:

```
$> 4

Temperature read (DMA) request:
        (One-shot enabled): HAL_I2C_Master_Transmit_DMA: status 0
        (Auto-increment): HAL_I2C_Master_Transmit_DMA: status: 0
        (Auto-increment): HAL_I2C_Master_Receive_DMA: status: 0; temperature data: 0x00ef
```

As a test against the accuracy of all three options, I (of camera) would rapidly call between the different options and verify that the results were the same or similar within reason, both in ambient conditions and with a space heater as an external modifier.

Closing Thoughts

Considering how ubiquitous I2C is, I didn't have too much experience in with this protocol prior to this assignment. A few years ago, I used a RPi Zero as a sniffer to troubleshooting an industrial printer at work, but I honestly don't recall accomplishing anything beyond reading back the device ID after tapping into the bus… The industrial controllers I work with daily leverage I2C heavily in communication to coprocessors, ADC/DACs, other communication interfaces, and so on but its entirely obfuscated under layers of interfaces and ladder logic. With all of that said, this assignment was a fun way to get closer to the hardware and examine both the controller and sensor interfaces at a deeper level.