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

Date: 4/28/2021

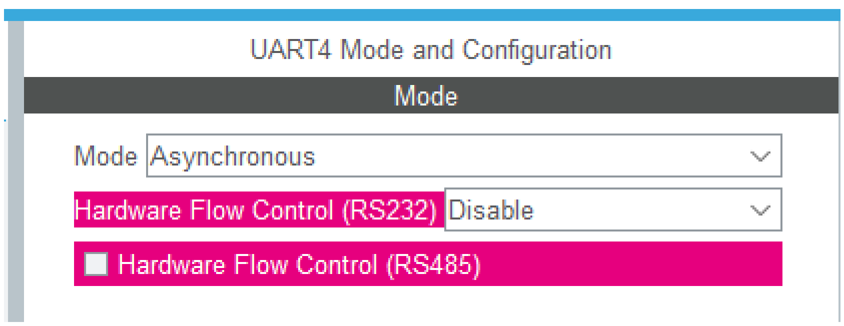
Assignment 3: Serial Hands On

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

1. CLI. Create a CLI (Command Line Interface) on UART1 that prompts you to enter a 1 for polling, a 2 for interrupt, and a 3 for DMA.
2. Polling. When the user selects 1, use polling to transmit and receive the buffer.
3. Interrupts. When the user selects 2, use interrupts to transmit and receive the buffer.
4. DMA. When the user selects 3, use DMA to transmit and receive the buffer.

1. Create a CLI

To start, open the IDE and generate a new project using the same process of selecting the Disco board and configuring all defaults that we've used on the previous project. Additionally, we will need to ensure that the UART4 peripheral is enabled and properly configured. Under the project configurator, navigate to Connectivity and select UART4. Under the mode drop down, select "Asynchronous".



From here, we can issue the build command to include the necessary support functions and files for UART4 to be used in our project. We can then jump into main.c and start developing the CLI. We'll leverage the HAL_UART_Transmit() and HAL_UART_Receive() functions to do most of the heavy lifting for this bare interface with the rest of the process being handled in the do_something() functions and whatever string formatting we need to make the interface both easy to use and look at. We'llalso need to include stdio.h and string.h in the User includes section to pull in functions like printf() and strlen().

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

#include <stdio.h>
#include <string.h>

/* USER CODE END Includes */
```

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

Date: 4/28/2021

To pretty-up the UI, I added a simple header immediately after the various init function calls. In a “real” shell or interface, this would be where I would put long form info about the program and things like shortcuts or general user instructions. For our case, a dummy title and version number is sufficient.

```
  MX_UART4_Init();
  /* USER CODE BEGIN 2 */

  // Header info for CLI
  char* cliHeader = "\r\nsimpleCLI Interface v0.1\r\n";
  HAL_UART_Transmit(&huart1, (uint8_t*) cliHeader, strlen(cliHeader), 1000);

  /* USER CODE END 2 */
```

Then, within the while(1) loop, we can build up the simple interface as seen in the code below, where we print out the options available, get an input character back, and then execute the associated action, all in a loop.

```
    /* USER CODE BEGIN 3 */

      /************************************
       * User Story 1 implementation begin:
       *
       *   Define a simple CLI with options
       *   for polling, interrupt, or DMA
       *   driven data Tx/Rx
       *
       ************************************/

    // Define strings to structure prompt around
    char* cliPrompt = "Options:\r\n\t1: polling mode\r\n\t2: interrupt mode\r\n\t3: DMA mode\r\n$> ";
    char* cliResponse = "Invalid input!\r\n";

    // Issue prompt
    HAL_UART_Transmit(&huart1, (uint8_t*) cliPrompt, strlen(cliPrompt), 1000);

    // Get the user selection
    char cliInput;
    HAL_UART_Receive(&huart1, (uint8_t*) &cliInput, 1, HAL_MAX_DELAY);

```

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

Date: 4/28/2021

If I were writing this code again, I would have moved the line that prints the cliResponse to within the case statements prior to the do_something() funcitons so that the user gets a response before the processing instead of after, as we'll see in the folllowing example screenshots.

```c
        // Evaluate input
        switch (cliInput)
        {
            case '1':
                cliResponse = "\r\nPolling mode\r\n";
                do_polling();
                break;

            case '2':
                cliResponse = "\r\nInterrupt mode\r\n";
                do_interrupt();
                break;

            case '3':
                cliResponse = "\r\nDMA mode\r\n";
                do_dma();
                break;

            default:
                break;
        }

        // Print response
        HAL_UART_Transmit(&huart1, (uint8_t*) cliResponse, strlen(cliResponse), 1000);


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

With this compiled and flashed, we can connect to our Disco board with PuTTY or minicom and see our CLI after a reset:

COM3 - PuTTY

```
simpleCLI Interface v0.1

Options:
        1: polling mode
        2: interrupt mode
        3: DMA mode

$>
```

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

Date: 4/28/2021

2. Polling

From here we can build the functions that actually do something. For function do_polling(), we'll define a pair of transmit and receive buffers of equal size that are pre-populated with the English alphabet. Then within the function, set a pair of buffer pointers to the start of each. From there, we will loop through the received data buffer and load it with a default value, '?' in this example.

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

// Define data buffers and pad with known values
static char txBuffer[] = "abcdefghigjklmnopqrstuvwxyz\r\n";
static char rxBuffer[] = "abcdefghigjklmnopqrstuvwxyz\r\n";

// Define an interrupt-complete flag
static int do_interrupt_done = 0;

static void do_polling(void)
{
    // Get pointers to data buffers
    char* txBuffPtr = txBuffer;
    char* rxBuffPtr = rxBuffer;

    // Fill RX data buffer with known value ('?')
    for (int i = 0; i < sizeof(rxBuffer); i++)
    {
        rxBuffer[i] = '?';
    }

```

This setup code out of the way, we jump into the do-while loop that makes up the majority of the functionality. Per each iteration through the loop, we'll put out a character to indicate something is happening to the user, '.' in this case. Then, and this is explicitly noted as operating with UART4 Tx/Rx connected in loopback with a jumper wire (ARD_D0 to ARD_D1), we'll send and receive whatever current value is being looked at from the Tx buffer to the current location in the Rx buffer. We then perform an error check to confirm that our destination data is the same as our source, and will print a message to the console if that is not the case. We the increment both pointers by one, have a short delay, and repeat the loop until we've traversed the length of the Tx buffer

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

Date: 4/28/2021

```c
    // Do-loop with bulk of functional code
    do
    {
        // Pipe out a heartbeat indicator ('.')
        char ch = '.';
        HAL_UART_Transmit(&huart1, (uint8_t*) &ch, 1, 100);

        // Transceive a character from the data buffers
        //
        // WARNING: This functionality is predicated on
        // UART4 being hard-wired into loopback mode!
        HAL_UART_Transmit(&huart4, (uint8_t*) txBuffPtr, 1, 100);
        HAL_UART_Receive(&huart4, (uint8_t*) rxBuffPtr, 1, 100);

        // Error check comparison of buffer values
        if (*txBuffPtr != *rxBuffPtr)
        {
            char errorBuffer[100];
            snprintf(errorBuffer, sizeof(errorBuffer), "\r\nError: 0x%02x != 0x%02x\r\n", *txBuffPtr, *rxBuffPtr);
            HAL_UART_Transmit(&huart1, (uint8_t*) errorBuffer, sizeof(errorBuffer), 100);
        }

        // Increment pointers to both data buffers
        *txBuffPtr++;
        *rxBuffPtr++;

        HAL_Delay(100);

    // Maintain loop through length of transmit data buffer
    } while (txBuffPtr < (txBuffer + sizeof(txBuffer)));
}
```

With everything compiled and flashed, we can see the following results in the serial terminal:

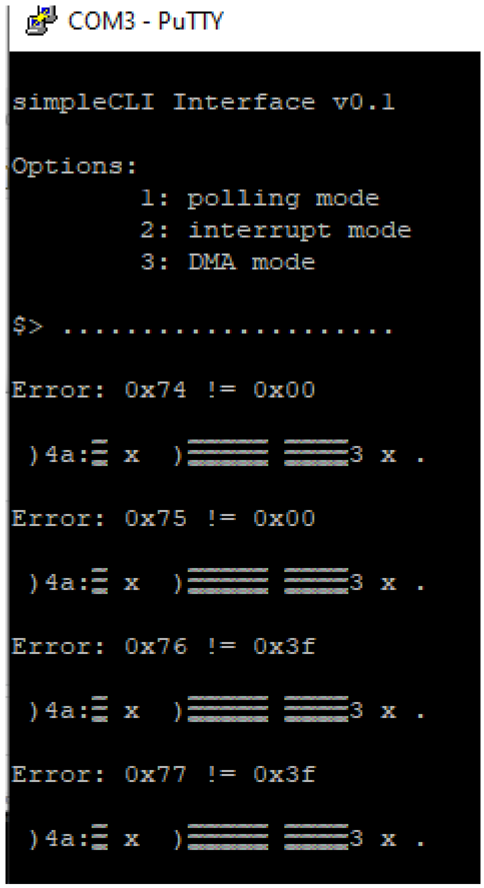
COM3 - PuTTY

```
simpleCLI Interface v0.1

Options:
        1: polling mode
        2: interrupt mode
        3: DMA mode

$> ..............................

Polling mode

Options:
        1: polling mode
        2: interrupt mode
        3: DMA mode

$>
```

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

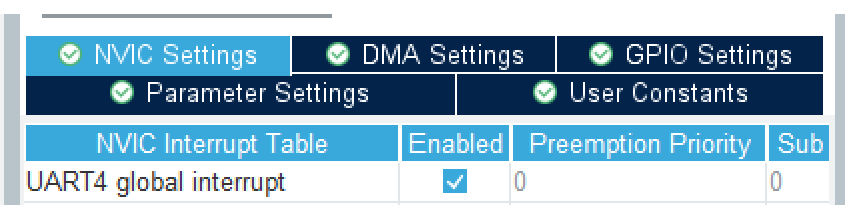
Date: 4/28/2021

Or, if there is an error in data transfer (simulated by removing the jumper here):



3. Interrupt

Moving to the third user story, do_interrupt() requires a bit of extra set up to enable the UART4 NVIC entries from the project configurator.



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

Date: 4/28/2021

We'll also need to define a global interrupt status flag and a pair of callback functions to work in tandem with the NVIC response and the do_interrupt() function. Still in loopback, the callbacks perform in a similar fashion to the polling method, pumping out the contents of the transmit data buffer to the received data buffer.

```c

// Define an interrupt-complete flag
static int do_interrupt_done = 0;

static void do_polling(void)
{

// Define our interrupt-driven callback functions
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
    // Print out a char to indicate we reached this point ('T')
    char ch = 'T';
    HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 100);
}


void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    // Print out a char to indicate we reached this point ('R')
    char ch = 'R';
    HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 100);

    // Send out the contents of the RX data buffer to UART1
    HAL_UART_Transmit(&huart1, (uint8_t *) rxBuffer, sizeof(rxBuffer), 100);

    // Set our complete flag
    do_interrupt_done = 1;
}
```

These are all wrapped up through a couple layers of HAL API abstraction to the do_interrupt() function def, seen below. This function will fill the RX buffer with a known value ('?'), send out the initial heartbeat indicator ('.'), clear our interrupt status flag, then call the HAL_UART_Receive_IT() and HAL_UART_Transmit_IT() functions for both buffers. We'll then go into a holding pattern where a '~' is printed until the interrupt flag is set by the RX callback function we defined earlier.

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

Date: 4/28/2021

```c
static void do_interrupt(void)
{
    // Fill RX data buffer with known value ('?')
    for (int i = 0; i < sizeof(rxBuffer); i++)
    {
        rxBuffer[i] = '?';
    }

    // Pipe out a heartbeat indicator ('.')
    char ch = '.';
    HAL_UART_Transmit(&huart1, (uint8_t*) &ch, 1, 100);

    // Clear the complete flag
    do_interrupt_done = 0;

    // Using the IT-specific function calls, transceive data from the TX buffer
    HAL_UART_Receive_IT(&huart4, (uint8_t*) rxBuffer, sizeof(rxBuffer));
    HAL_UART_Transmit_IT(&huart4, (uint8_t*) txBuffer, sizeof(txBuffer));

    // Indicate on UART1 that the IT functions are in progress
    while (!do_interrupt_done)
    {
        char ch = '~';
        HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 100);
        HAL_Delay(100);
    }

}
```

The results of all of this can be seen below, with a successful transmission wrapping up after a few moments and an error state hanging forever.

COM3 - PuTTY

```
simpleCLI Interface v0.1

Options:
        1: polling mode
        2: interrupt mode
        3: DMA mode

$> .~Rabcdefghigjklmnopqrstuvwxyz

T

Interrupt mode
```

COM3 - PuTTY

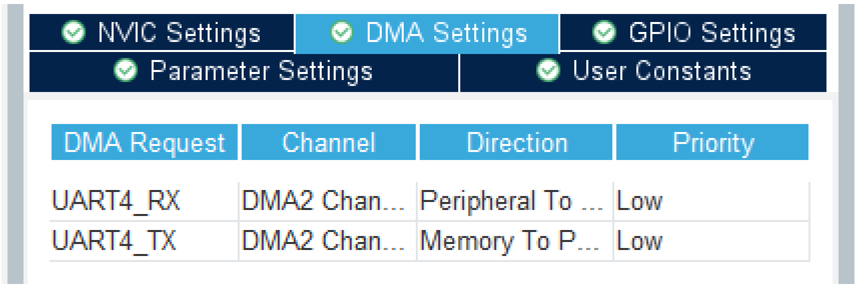
```
simpleCLI Interface v0.1

Options:
        1: polling mode
        2: interrupt mode
        3: DMA mode

$> .~T~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

4. DMA

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

Date: 4/28/2021

For the final story, we'll need to go back into the project configurator and make sure that the UART4 DMA requests are enabled and configured.



We'll also need to declare a pair of DMA Handlers in the private variables section.

```
DMA_HandleTypeDef hdma_uart4_rx;
DMA_HandleTypeDef hdma_uart4_tx;
```

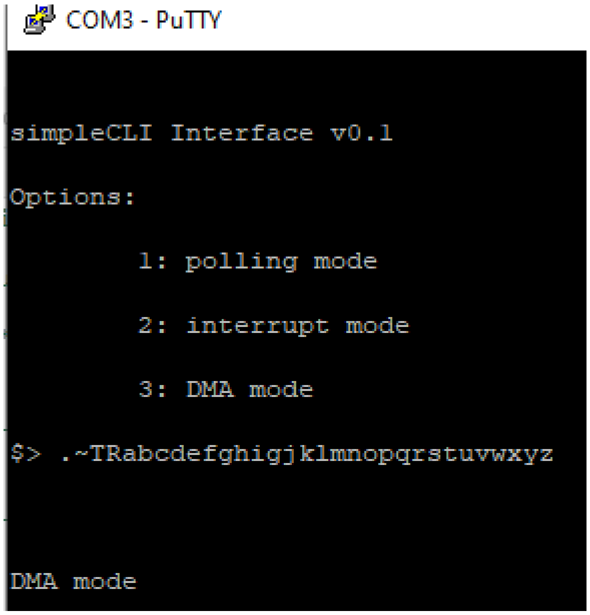
With this config done, we'll be reusing the callbacks defined in the interrupt story and essentially duplicate the wrapper function in do_dma(), with the exception that we'll use the DMS-specific calls this time.

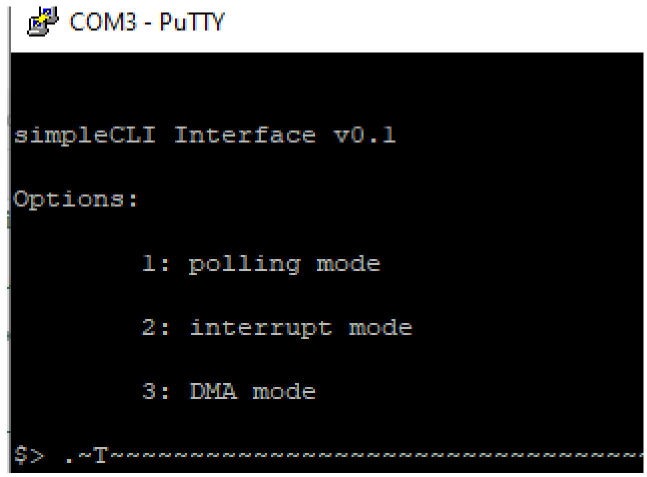
```
static void do_dma(void)
{
    // Fill RX data buffer with known value ('?')
    for (int i = 0; i < sizeof(rxBuffer); i++)
    {
        rxBuffer[i] = '?';
    }

    // Pipe out a heartbeat indicator ('.')
    char ch = '.';
    HAL_UART_Transmit(&huart1, (uint8_t*) &ch, 1, 100);

    // Clear the complete flag
    do_interrupt_done = 0;

    // Using the DMA-specific function calls, transceive data from the TX buffer
    HAL_UART_Receive_DMA(&huart4, (uint8_t*) rxBuffer, sizeof(rxBuffer));
    HAL_UART_Transmit_DMA(&huart4, (uint8_t*) txBuffer, sizeof(txBuffer));

    // Indicate on UART1 that the DMA functions are in progress
    while (!do_interrupt_done)
    {
        char ch = '~';
        HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 100);
        HAL_Delay(100);
    }
}
```

With this updated code compiled and flashed, we see similar behavior to the interrupt functions.

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

Date: 4/28/2021





Closing Thoughts

I would say that out of all the peripherals we'll study in this course, UARTs and serial communication are probably one of the ones I'm most familiar with. At work we extensively leverage various industrial protocols, including Modbus and serial-network gateway port servers. Personally, I've used serial almost exlusively over the last several years as a way to console into my Raspberry Pis or whatever Linux SBC was flavor of the month versus a keyboard and monitor solution. We didn't really need to dive into the nitty gritty of serial port configuration, for example using some fairly interesting math and bitshifts into the BRRs to set the peripheral's baud rate, but it was interesting to use DMA, which I'd not done before. A personal project that's always been on my list of one-day's is to use a small MCU like an STM8 or STM32 and build a BASIC or FORTH interpreter with access to whatever GPIO on board as a test/ development tool for hardware projects. I've never gone much beyond the point we've reached here, in getting a simple prompt up, mainly because everytime I start, I get bogged down in the tedium of doing all of this setup by hand. In this project, the usefulness of the HAL shown through clearly, where I was able to go fomr scratch to flash in about 20 minutes, with a good portion of that spend decinding how to make the interface look nice. Maybe one of these days, I'll take advantage of that and finally make some serious progress on that project.