

Date: 4/23/2021

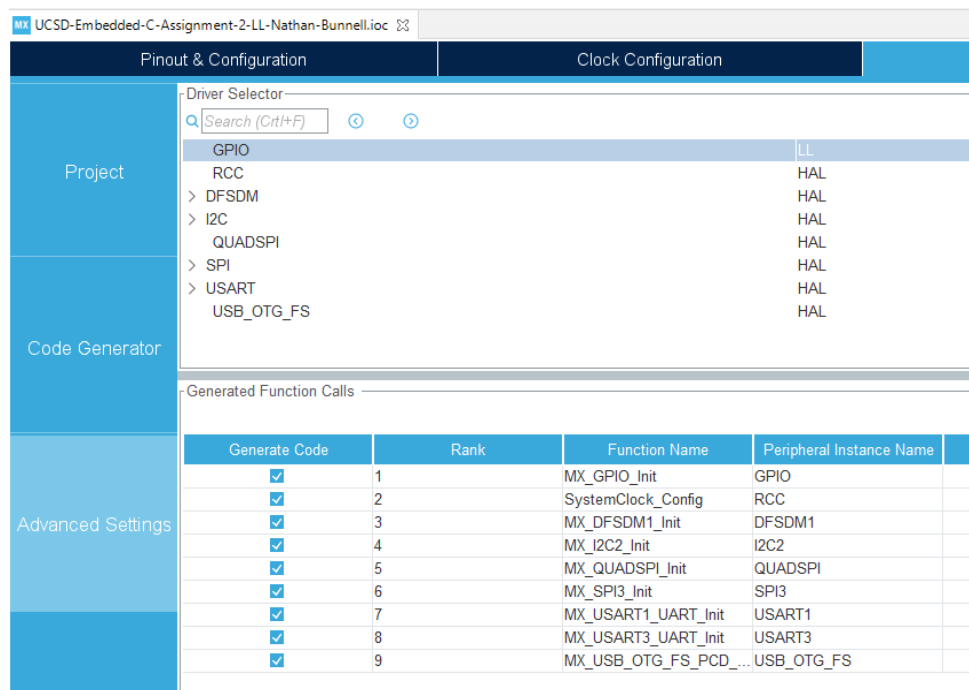
Assignment 2: Low Layer API Usage

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

1. Use STM32CubeIDE to generate the default code for the STM32 Discovery Board, being sure to include the LL option for GPIO when generating the code.
2. Within the IDE, edit, build, run, and debug the code that uses the LL API to (1) get the Flash size, LL_GetFlashSize(); (2) get the device unique ID, LL_GetUID_Word().
3. Use the LL to read the status of the Blue Button. If the Blue Button is pressed, then turn on LED2 using the LL. If the Blue Button is not pressed, the turn off LED2.

1. Generate default code with LL options enabled

To avoid repeating myself unnecessarily, this report won't cover in full detail the process of loading the IDE and creating a new project with our Disco board as the full treatment of that process was completed in Assignment 1. So, assuming we follow those same steps to create a new project for our target board with the build option to initialize all peripherals to their defaults, the first step to work with LL APIs will be to open the Project dashboard and navigate to the Project Manager tab. From here, select "Advanced Settings" and then under the "Driver Selector" area, select "LL" from the drop-down menu on the GPIO driver interface, as seen in the accompanying screenshot.



At this point, the "Build" command may be issued and default code will be re-generated with the appropriate LL files included in the project structure.

Date: 4/23/2021

2. Build project with LL APIs

With our default code and support files now in place, we can dive into main.c and begin to leverage the LL API to build useful code. We'll use the built-in functions to pull data about the MCU on our Disco board, specifically it's flash size and UID. Immediately after the peripheral initialization functions and prior to the while(1) loop, we can place the following into the *USER CODE 2* area:

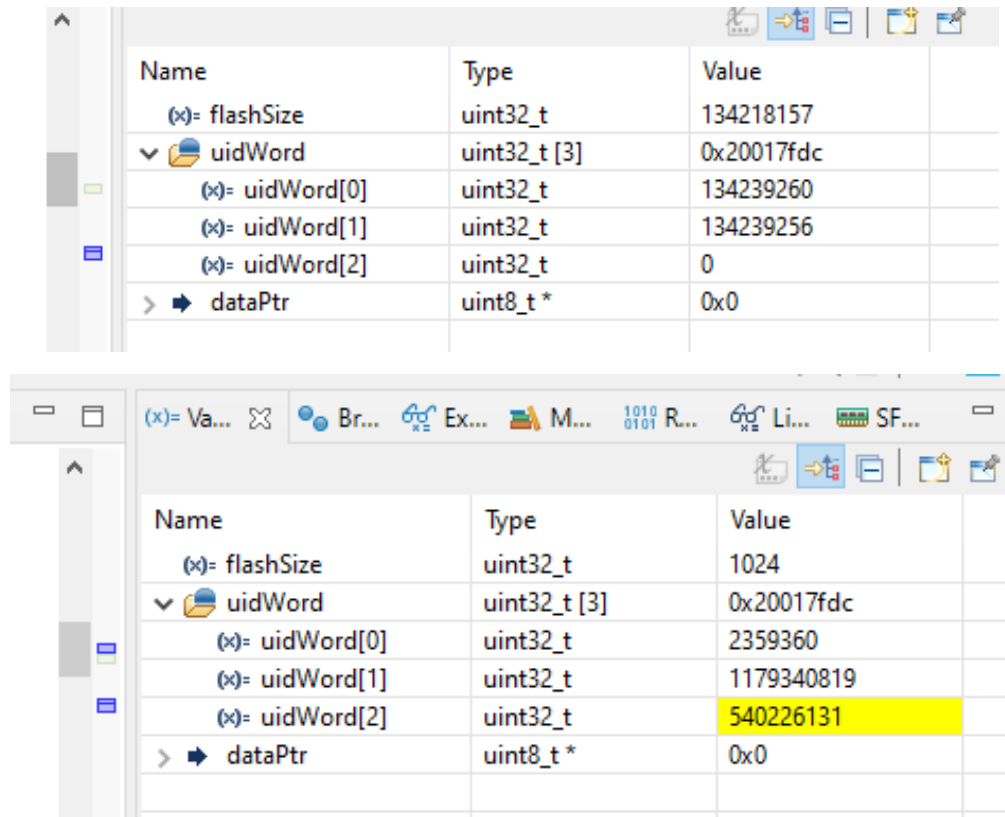
```
1  /* USER CODE BEGIN 2 */
2
3  /*
4   * Leverage LL APIs to pull the internal flash size and get the chip's UID value
5   * as a 96-byte data block
6   */
7
8  uint32_t flashSize = LL_GetFlashSize();
9
10 uint32_t uidWord[3];
11 uidWord[0] = LL_GetUID_Word0();
12 uidWord[1] = LL_GetUID_Word1();
13 uidWord[2] = LL_GetUID_Word2();
14
```

This example shows off how easy it is to use the LL functions to pull the target data. These examples are actually fairly similar to what I would expect the HAL version to be in that they don't necessarily expose the hardware-specific nature of the LL interface that can make it less portable compared to the HAL or other abstractions. As additional practice with the LL APIs, I had hoped to send this data out over the serial-USB connection that the ST-Link exposes but, as of this writing, only have a barebones implementation of the function calls and haven't spared the time getting this bonus feature working:

```
15 /*
16  * TO-DO:
17  * Send data over serial port with LL UART functions
18  *
19  * Currently non-functional
20  */
21
22 // Use a char* to push data from the uint32_T variables into the serial buffer
23 // byte-by-byte
24 uint8_t* dataPtr = &flashSize;
25
26 for (uint8_t i = 0; i < (sizeof(uint32_t)/sizeof(uint8_t)); i++)
27 {
28     LL_UART_TransmitData8(USART1, *(dataPtr++));
29 }
30
31 dataPtr = &uidWord;
32
33 for (uint8_t i = 0; i < ((3 * sizeof(uint32_t))/sizeof(uint8_t)); i++)
34 {
35     LL_UART_TransmitData8(USART1, *(dataPtr++));
36 }
37
38 /* USER CODE END 2 */
```

Date: 4/23/2021

However, with the debugger running, we can step through the code and find that these variables do populate with data, including an expected value of 1024 for the flash size and what looks like a reasonable device ID number as seen in the before/after screenshots below:



The image displays two screenshots of a debugger's variable window, showing the state of variables before and after execution.

Top Screenshot (Before):

Name	Type	Value
(x)= flashSize	uint32_t	134218157
uidWord	uint32_t [3]	0x20017fdc
(x)= uidWord[0]	uint32_t	134239260
(x)= uidWord[1]	uint32_t	134239256
(x)= uidWord[2]	uint32_t	0
dataPtr	uint8_t *	0x0

Bottom Screenshot (After):

Name	Type	Value
(x)= flashSize	uint32_t	1024
uidWord	uint32_t [3]	0x20017fdc
(x)= uidWord[0]	uint32_t	2359360
(x)= uidWord[1]	uint32_t	1179340819
(x)= uidWord[2]	uint32_t	540226131
dataPtr	uint8_t *	0x0

Date: 4/23/2021

3. Use LL APIs for GPIO interfacing

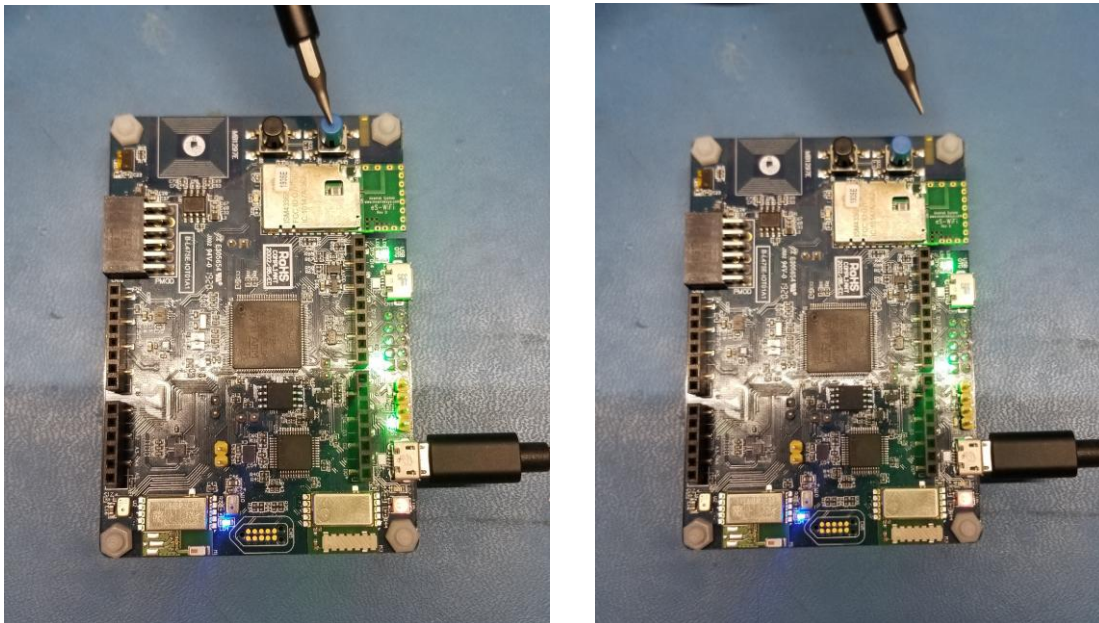
From here, we can move on to a slightly more practical example, where we see the code do something in the real world. Our goal will be to turn on LED2 when the user button is pressed and turn it off when the button is not pressed.

```
39
40  /* Infinite loop */
41  /* USER CODE BEGIN WHILE */
42  while (1)
43  {
44      /* USER CODE END WHILE */
45
46      /* USER CODE BEGIN 3 */
47
48      /*
49       * Within loop, poll for status of Blue Button, GPIOC.13 (active low).
50       * If pin is clear, set LED2, GPIOB.14 (active high)
51       * If pin is set, clear LED2
52       */
53
54      if(LL_GPIO_IsInputPinSet(GPIOC, LL_GPIO_PIN_13))
55      {
56          LL_GPIO_ResetOutputPin(GPIOB, LL_GPIO_PIN_14);
57      }
58      else
59      {
60          LL_GPIO_SetOutputPin(GPIOB, LL_GPIO_PIN_14);
61      }
62  }
63  /* USER CODE END 3 */
```

Again, this code is extremely simple, but showcases the requirements for knowledge of the target hardware to be properly implemented. While not quite down to the level of bitshifting into the GPIO registers and setting pin masks, a hardware schematic would be essential at minimum for this level of development.

With all of our code developed, we can compile and flash the project to test the real-world interface. We can see on the next page the device functioning as per our design goals on my lab bench, with LED2 (located by the micro-USB port and ST-Link debug header), toggling on and off appropriately on button press and release.

Date: 4/23/2021



Closing thoughts

This introduction to the LL interface was interesting, although perhaps a bit too brief. I was unaware the STM provided this intermediary layer between their HAL and the bare-metal environment that I am used to working in on other members of the STM32 family. The examples of getting the UID and flash information showcased the similarities to the HAL, for example: a quick check in the `stm32l4xx_hal.h` file seems to indicate that the equivalent UID function for the first word would be `HAL_GetUIDw0()`. The GPIO example showed off again how easy it is to develop with all the functions predefined but also showcased the importance of knowledge of the hardware to accomplish anything useful.

As mentioned before, I prefer a minimal, bare metal level of development but am also doing so as a hobbyist. Professionally, I work primarily with IEC-standard Ladder Diagram, which I feel is about as abstracted away from hardware as you can get in a programming language. Both have pros and cons, as with using a vendor provided HAL. I am sure that the same is true with the LL interface versus the LL, but to be honest, I don't know that this experience has been enough for me to concretely say that X or Y scenario would be the best case for either. I am looking forward to spending more time within the APIs both in this course and beyond it to grow that level of knowledge and engineering intuition.