Microprocessor Systems Lab 3

Checkoff and Grade Sheet

Partner 1 Name:______________________________

Partner 2 Name:______________________________

Grade Component	Max.	Points Awarded	TA Init.s	Date
Performance Verification: Task 1	5 %			
Task 2	10 %			
Task 3	5 %			
Task 4	20 %			
Task 5 [Depth]	10 %			

→ Laboratory Goals

By completing this laboratory assignment, you will learn to:

1. Communicate with devices using UART,
2. Communicate with devices using SPI,
3. Implement communication using either Polling (Blocking) or Interrupt (Non-Blocking) configurations.

→ Reading and References

R1. Mastering STM32: Chapters 8 (UART), 15 (SPI)

R2. stm32f769xx_Datasheet.pdf: Alternate Functions, Table 13

R3. UM1905-stm32f7_HAL_and_LL_Drivers.pdf: Chapters 62 (SPI), 66 (UART)

R4. RM0410-stm32f76xxx_Reference_Manual.pdf: Chapters 34 (USART), 35 (SPI/I2S)

R5. Lab03-Template.zip: Project Template for Lab 3

R6. STaTS_SPI_Periph_Datasheet.pdf: Example datasheet for peripheral device incorporating an SPI interface. Flashing and use instructions are near end of document.

R7. Firmware for example SPI device, Nucleo-F303K8 - See website

Serial Communication

In order to transport data into or out of a microcontroller from other digital devices (e.g., sensors, computers, other microcontrollers), various communication standards may be used, such as I^{2}C[1], UART, SPI, CAN, etc. These listed standards are all examples of *Serial Communication* techniques, where data is transferred bit-by-bit. Other standards exist but are not common in microcontroller applications, such as PATA and SCSI along with other various memory access methods, are known as *Parallel Communication*, where byte(s) are transferred over many communication lines at once[2]. For this lab, we will focus on the UART (Universal Asynchronous Receiver Transmitter) and SPI (Serial Peripheral Interface) standards.

Within the serial communication domain, there are two distinct communication methods: synchronous and asynchronous. These are differentiated by whether the data clock is shared between the devices in communication or not. In an asynchronous communication system (e.g., UART), a clock is not shared between the devices; requiring that all devices on the communication bus have knowledge of the expected data baud rate[3] For example: all labs in this course use terminal communication between the STM32F769NI and the computer which is achieved through a UART interface. When configuring the terminal on the computer, the baud rate is explicitly specified to the terminal program as 115200 bps.

Alternatively, synchronous communications are those where the data clock is shared between the devices on the bus (e.g., I^{2}C, SPI). This allows for the devices receiving the clock operate without an onboard

[1]The I3C standard has been released towards replacing I^{2}C, though adaption may take a while

[2]The STM32F769NI has a Flexible Memory Controller (FMC) which contains a parallel interface for memory addressing up to 32-bits wide.

[3]Baud rate for our purposes is the communication *bit rate* or *data rate*, measured in bits per second. In higher-order modulated communication systems, baud rate is also known as the *symbol rate*.

timebase generator but instead requires all devices to have an extra I/O pin to receive/emit the clock signal, as well as requiring an additional wire or PCB trace connected this pin on each device. Sharing the clock between the devices also increases the maximum throughput of the bus, as there is less required overhead for data packet control signals. Synchronous communication techniques commonly have a Controller/Peripheral architecture[4], where one device on the bus is configured as the controller and directly controls the communication bus[5], which includes generation and sharing of the data clock. The peripherals only communicate when addressed (I^2C) or selected (SPI) by the controller.

Universal Asynchronous Receiver Transmitter (UART)

The STM32F769NI has several on-board universal synchronous/asynchronous receiver/transmitter (USART), which may be used to generate UART, among other standards. The DISCO board has one built-in virtual UART communication channel over USB which USART1 is configured to use. Each UART interface requires only two signal lines: RX (receive data) and TX (transmit data). These lines are named the same for any device connected; therefore, for device 1 to transmit data to device 2, the TX line of device 1 must be connected to the RX line of device 2 and vice versa. The USARTs can of course be configured through the registers listed in R4, though for this lab, the hardware abstraction layer (HAL) drivers will be used. For each USART configured to operate as a UART device, a `UART_HandleTypeDef` type module handle is required, which has the following defined structure:

```
typedef struct{
        USART_TypeDef *Instance; /* UART registers base address */
        UART_InitTypeDef Init; /* UART communication parameters */
        UART_AdvFeatureInitTypeDef AdvancedInit;
                                          /* UART advanced features configuration */
        ...
} UART_HandleTypeDef;
```

The `UART_InitTypeDef Init` field is another struct containing the following parameters, which set up how the UART operates to exchange data:

```
typedef struct {
        uint32_t BaudRate;
        uint32_t WordLength;
        uint32_t StopBits;
        uint32_t Parity;
        uint32_t Mode;
        uint32_t HwFlowCtl;
        uint32_t OverSampling;
} UART_InitTypeDef;
```

These types of module configuration handles that are reused throughout the code (this includes: `UART_HandleTypeDef`, `SPI_HandleTypeDef`, etc.) should always be defined as global variables.

[4]The traditional terminology of "controller/peripheral" was "master/slave". Multiple naming conventions have been arising to replace this convension, such as "primary/secondary," "leader/follower," or Python's "parent/worker."

[5]There are multi-controller techniques, we will only focus on single-controller SPI.

→ UART Port Setup

In the basic mode, the UART is relatively easy to configure when using the HAL. Three general steps are required:

1. Enable GPIO port for transmitting and receiving
2. Populating the `UART_HandleTypeDef` fields `Instance` and `Init`.
3. Call `HAL_UART_Init()`

Note that item 3 above does not and cannot inherently accomplish item 1 as there are many different configurations as to which GPIO pins to place the UART signals. While this alternatively may have been accomplished through another field in the `UART_HandleTypeDef` (e.g., `GPIOInit`), the authors of the HAL instead have the `HAL_UART_Init()` function call back to the user space function `HAL_UART_MspInit()`, where all GPIO configuration is expected to be done. The callback function `HAL_UART_MspInit()` is provided in the template project R5 in file `uart.c`.

Transmitting and receiving data is also a simple procedure requiring only `HAL_UART_Transmit()` and `HAL_UART_Receive()`. The functionality of `printf()`, `putchar()`, and `getchar()` are all implemented using those two functions (again, see R5 `uart.c`).

◇ Task 1: Two-Terminal UART (Polling)

Write a program that will monitor two serial ports continuously. The STM32F769NI has 8 total UART capable modules, though only 5 are accessible on the DISCO board, one of these being USART1 over USB. The other port used should be USART6. There are three options for development here:

1. The USART6 interfaces between each group member's DISCO board may be connected together,
2. The "STaTS" device R6 may be used as the second unit[6], or
3. USART6 may be connected to a computer the serial port functionality available on an ADALM2000 or Analog Discovery device.

USART1 should be left as configured in the template project and USART6 should be configured to use 38400 baud and N-8-1 (no parity bits, 8 data bits, 1 stop bit). For reference, USART1 is configured as 115200 baud N-8-1.

Whenever the program detects a character coming in from either serial port, the received character should be echoed to the local serial terminal. Additionally, if the character was received from the local serial terminal, the character should also be transmitted to the other device. The end result of this configuration should be such that keyboard presses on either serial terminal should be shown on both. This requires two serial terminals active: one for each device. When <ESC> is pressed on either terminal, a brief exit message should be shown on both screens and both program should also halt (that is, not take any further input). Since continuous polling of both ports is required, `getchar()` may not be used as it will wait indefinitely for a key press.

[6]Instructions for operating the DISCO board and Nucleo board simultaneously on the same computer are listed on the last page of this document

NOTES:

1. It is not acceptable to require the programs to transmit in an alternating fashion (e.g., device 1 transmits, then device 2, then device 1). Either device should be able to transmit at any time.
2. Ensure the terminals are configured with the correct baud rate: 115,200 and 38,400.
3. Failure to connect the Grounds between the DISCO and second device will result in either no output or garbage output.
4. Remember that device 1 TX should be connected to device 2 RX, similarly: TX2-RX1.
5. Much of what you need already exists in `uart.c` and `init.c`.
6. The timeout values for `HAL_UART_Transmit()` and `HAL_UART_Receive()` may be very small.
7. Be careful when using `uart_getchar()` for this task. The function will always return a value when `HAL_UART_Receive()` timeout, and the value might not be what is expected.

Polling Versus Interrupt Operation

In Task 1, a program was made that continuously checked whether any characters were received on USART1 or USART6. This type of implementation is known as *polling*, or continuously (or periodically) checking to see if a specific event has happened. This method has the drawback that to successfully detect an event in complex code at the proper time, checks of the event need to be placed throughout the code. This both makes the code mode difficult to write and maintain but also introduces computational overhead. Alternatively, if only one check for the event is placed within the code and will not continue until the event has happened, then the program is essentially "blocked" from doing anything else.

This functionality can instead be handled by interrupts, or in a "non-blocking" mode. This frees up the required additional overhead of polling while also ensuring that the event is dealt with immediately, in cases where there is a time sensitivity (e.g., Real-Time systems). For the case of UART character reception, it is desired to trigger interrupts when a character is received by either UART port. To implement the IRQHandler for USART1, the following code should be used:

```
// Handle USB/UART Interrupts with HAL
void USART1_IRQHandler() {
        HAL_UART_IRQHandler(&USB_UART);
}
```

where `USB_UART` is the handle for USART1, defined in the template project. A similar function will need to be written for USART6. Additionally, the callback function `HAL_UART_RxCpltCallback()` will need to be written to handle character reception. Note that the handle of the USART module that triggered the interrupt will be passed to this callback function and would be used to determine the character's origin. In order receive in this fashion, `HAL_UART_Receive_IT()` will need to be used instead of `HAL_UART_Receive()`.

◇ Task 2: Two-Terminal UART (Interrupt)

Reimplement task 1 except using a non-blocking interrupt structure instead of the polling method. The while loop in the main function **cannot** have any UART commands contained within it. Further, implement several terminal control commands; where each control command starts with a '#' character and is followed by a single character. The commands should not be shown on the terminal when entered. Any invalid commands are ignored. The commands:

- `#e` : trigger the programs to behave as if <ESC> was pressed,
- `#c` : cause both terminals to be cleared.
- `#l` : toggles LED LD3 ON/OFF on the *other board only*, not the board where the command was entered.

NOTES:

1. The debugging mode of the IDE may be very useful for this task in order to determine if a character input is being handled properly.
2. Don't forget to use the NVIC to enable the interrupts.
3. `HAL_UART_RxCpltCallback()` will only be called once per one call of `HAL_UART_Receive_IT()`. The receive another character, this command would have to be reissued.
4. While UART commands are not allowed within the main program loop, *they are allowed prior to the loop*; initialization commands as well as transmit and receive commands.
5. The STaTS device is programmed to accept the above control commands on the *UART interfaces only*. The control commands are not handled if received on the SPI bus and do not need to be implemented for Task 4.

Serial Peripheral Interface (SPI)

Note: The traditional names for the devices in SPI and I^{2}C communication schemes: **master** and **slave**, have been replaced with **controller** and **peripheral**. The STM32 device documentation maintains the **master/slave** convention. See here for a discussion on this change.

As introduced already, one synchronous protocol typically available on microcontrollers is the SPI. This protocol is essentially an alternative to the I^{2}C. There are significant differences between the two, mainly arising from peripheral selection and data lines. Active peripheral selection in I^{2}C is done via a software address, where the address of a peripheral is passed first in each data packet to specify which peripheral is supposed to read from or write to the controller. The SPI uses "chip select" (CS, also labelled SS or NSS), hardware lines, one for each peripheral. Employing chip select functionality eliminates the overhead required in the I^{2}C Bus of sending information to select the peripheral, allowing for much higher achievable throughputs; however, this comes at the expense of requiring a CS line for each peripheral on the bus (hardware complexity). Of course, the hardware required to support SPI could become exceedingly burdensome if there are many peripherals on the bus, thereby requiring the use of I^{2}C instead. A generic hardware layout for each of these standards is given in Figure 1.

In addition to multiple CS lines for the SPI, the SPI also uses two data lines: *Controller In, Peripheral Out* (CIPO) and *Controller Out, Peripheral In* (COPI)[7]; as opposed to I^{2}C using only SDA. The use of CIPO and COPI allows for bidirectional communication to occur, both being clocked by SCLK, again, allowing for higher possible throughput. Data transmission and reception are always done concurrently, that is, for every byte sent, a byte is received. However, distinctions are made for SPI transmitting and SPI receiving only modes:

1. Transmit Mode: Data is transmitted on COPI while data received on CIPO is ignored.
2. Receive Mode: Dummy data is transmitted on COPI (e.g., 0x00 or 0xFF) while data is received on CIPO and stored.
3. TransmitReceive Mode: Data is transmitted on COPI while concurrently receiving and storing data on CIPO.

Unlike the I^{2}C, SPI is less standardized; which results in many different methods to communicate with a peripheral. The communication requirements of the peripheral will need to be tailored to individually. Two points of interest are the required behavior of the clock, or the **clock polarity**, and when to change/latch (read) data, or the **clock phase**. The clock polarity can either be *Idle High* or *Idle Low*, indicating the state of SCLK when not communicating. The clock phase also has two possibilities: either the first clock edge or second clock edge indicates data capture (rising or falling), with the other edge being the data change signal. Other common differences between the devices is the absence of either the CIPO or COPI lines, or bidirectional communication on only one of the lines (known as *three-wire SPI*), similar to I^{2}C.

One aspect that is consistent however is that selection of the peripheral is done via pulling the CS lines low. This is indicated by the notation in Figure 1, where "~CS" is shown: the ~ indicating that the signal is *Active Low* as opposed to *Active High*. This implies that when ~CS for the peripheral is high, the peripheral will release control of CIPO and will ignore all activity of the bus. This is also commonly denoted with a slash or overline instead: /CS or $\overline{\text{CS}}$. For the SPI modules on the STM32F769NI, only one ~CS line

[7]The signals CIPO and COPI may also be named SDO or SDI for "Serial Data Out" or "Serial Data In".

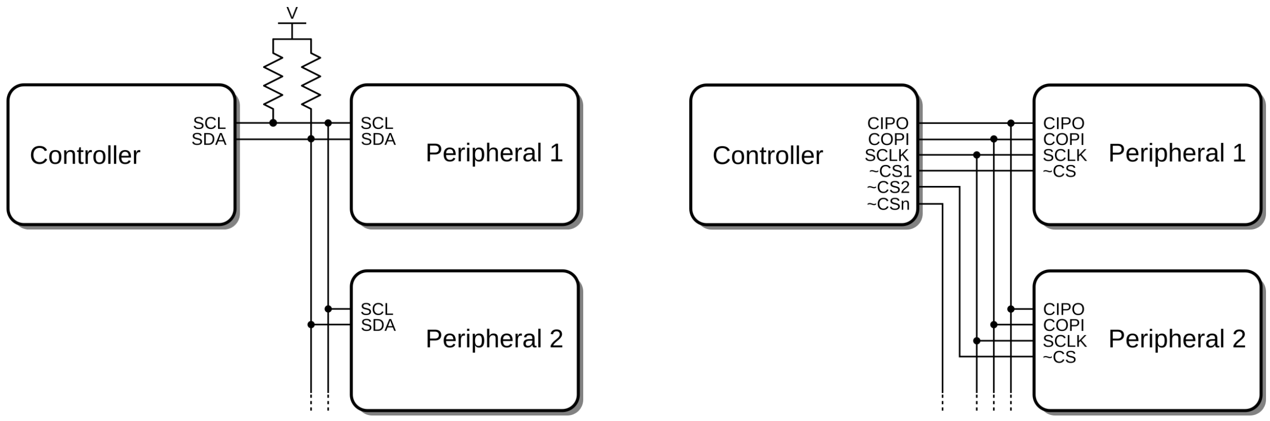

Figure 1: General architectures of (left) I^{2}C and (right) SPI Buses.

exists, labeled NSS ("*Not* Slave Select"). Therefore, the module itself is only capable of communicating with one peripheral by default if operating as a SPI controller. To add additional peripherals, GPIO pins must be configured independently to act as the required ~CS lines while the NSS line is also managed appropriately.

→ SPI Port Setup

The following steps are required to configure and use the SPI port:

1. Configure GPIO pins. These must be configured in alternate function mode. See R2 Table 13.
2. Configure SPI clock rate, wire mode (full-duplex or single bidirectional line), clock polarity and phase, and controller or peripheral mode. For this lab's purposes, CRC should be disabled.
3. Call `HAL_SPI_Init()`

As was the case for the USART modules, the completion of item 1 is triggered by item 3 through the callback function `HAL_SPI_MspInit()`. In order to send and receive data, the functions `HAL_SPI_Transmit()`, `HAL_SPI_Receive()`, or `HAL_SPI_TransmitReceive()` would be used, or their `_IT()` or `_DMA()` counterparts if operating in interrupt or DMA mode, respectively.

If the peripheral being used is controlled by the module's NSS line, then no extra commands need to be done to select the peripheral. If, however, a GPIO output pin is used to select the peripheral, the pin needs to manually be asserted low prior to calling a transmit/receive function while also disabling the NSS pin. Once the transfer is complete, the GPIO pin should again be raised high.

◇ Task 3: SPI Loopback Interface

Write a program that sets up SPI2 in 8-bit mode, monitors the terminal, echos any character received on the terminal to the SPI bus, and writes any received characters from the SPI bus to the terminal. Wire the SPI bus such that the CIPO and COPI signals of a single DISCO board are connected together (i.e., the loopback condition). The SPI port should be configured to operate at roughly 1 MHz. The terminal should be split top and bottom such that characters received from the keyboard are written on the top half and characters received from the SPI are written on the bottom half. It is not necessary to scroll both the top half and the bottom half, though the exact implementation is up to you. Implementation of this interface either in a Polling or Interrupt Mode is acceptable. The GPIO pins required to implement the SPI bus must be enabled in Alternate Function mode; where the correct mode must be identified. Tables identifying the alternate functions for each pin may be found in R2 Table 13. See `uart.c` for how this was done for the UART buses.
NOTES:

1. For this task, clock polarity and phase are not important.
2. Although SCLK is not required for this implementation, the SPI bus *will not operate* if the GPIO line is not properly configured.
3. Additionally, the NSS needs to be configured for the SPI to work; leaving the configuration blank, or filling it with the wrong value, will prevent the SPI from operating.
4. Test your interface by removing the loopback condition and tying CIPO to low or high. This should result in returned values of 0x00 or 0xFF, respectively.
5. As this is a hardware loopback operation (data sent = data received), care must be taken to ensure that the data transmission and reception occur *at the same time*. If these operations are done asynchronously, the loopback will not work properly as there is no circuitry to echo the transmitted data at a later time.
6. The STaTS required in the next task has a low maximum bit rate as compared to other SPI devices (for reasons). To support this lower rate, the input clock to the SPI2 module has been reduced within the template project by increasing the APB1 clock divider from /4 to /16, resulting in an APB1 clock of 13.5 MHz. Correspondingly, any timers on APB1 have their input clocks changed to /8! Please see R4 Figure 13 (Clock Tree) to see how this works and Table 1 (Register Boundary Addresses) to check which module is on which bus.
7. Warning: The internal clock for SPI2 is enabled for you in init.c. If you try to use a different SPI module, the corresponding clock must be activated as well. Further, be aware that a different SPI module may be on a different peripheral bus, resulting in a different input clock.

◇ Task 4: Connect to STM32-Based SPI Device

Using the datasheet provided in R6, design and implement an SPI controller that can interact with the "STaTS" device and perform the following:

1. Reliably send terminal characters from the DISCO to the peripheral device using SPI.
2. Reliably receive terminal characters from the peripheral device using SPI. The received characters should be printed on the bottom half of the controller's terminal.
3. Match the state of LD3 on the STaTS device depending on the state of a single input on the STaTS (one DP**x** line). LD3 should not respond to the other input lines.
4. Read the peripheral's firmware version upon startup.
5. Trigger a temperature measurement and retrieve the result when it is ready. The temperature should be printed on the right side of the terminal to avoid the transmitted and received terminal characters.
6. Clear or reset the peripheral terminal.
7. Change and read the device ID of the peripheral.
8. Pressing ESC in the controller's terminal should present a menu where the user can trigger sub-tasks 4-7. When this menu is active, it is acceptable to ignore characters inputted from the peripheral's terminal (sub-task 2).

Sending and receiving terminal characters (items 1,2) should happen concurrently (at the same time) and is considered the base-state of the program; that is, when the program isn't performing another function (e.g., changing the device ID), character transmission and reception should be active by default. Item 3 is also part of the base-state of the program such that LD3 changes automatically with DP**x** changes.

You must verify the SPI bus operation using a logic analyzer or oscilloscope (or Analog Discovery board, etc.) and include in your report. Using a logic analyzer or oscilliscope to debug is *highly encouraged*. If using an Analog Discovery board, etc. it is best to use the digital logic lines to watch all four (SCLK, CIPO, COPI, CS) lines at once; triggering on CS falling edges.

NOTES:

- Sub-tasks 2 through 7 may be omitted for a satisfactory checkoff; however, a deduction of 4 points will be applied for each missing task. One sub-task between 3 and 7 may be omitted without incurring a penalty.
- Triggering of the temperature measurement may be done by the terminal menu or through a timing interval. If done with a timing interval, the menu option for this may be omitted.
- A warning: the temperature measurement may not make sense as it is an internal core temperature of the STaTS, which will likely be much higher than room temperature. The value measured by the STaTS will be printed on the STaTS terminal. Simply verifying equivalency between received and printed values is sufficient, though the temperature in °F or °C must still be printed.
- Two firmware versions are provided for the STaTS on the website: one with the digital port enable and another with it disabled. There are several of the Nucleo-F303K8 boards that have failed GPIO pins as part of the digital port and they cause the digital port change indication to trigger constantly.

◇ Task 5: [Depth] Serial "Bit Banging"

In some applications, peripheral support for certain serial communication buses may not be available. The bus must be implemented in code in these situations, as opposed to using specialized hardware. This method is known as "bit banging". Bit banging effectively requires the efficient use of GPIO, timers, and interrupts to produce the bus.

For this task, generate an SPI-like bus transmit path; where the device generates a serial clock, **TX-CLK**, and a serial data output, **SDO**. The clock should only be active when transmitting bits. Similarly, provide a receive transmit path; where the device receives a serial clock, **RXCLK**, and a serial data input, **SDI**. This topology is describes in Figure 2 below.

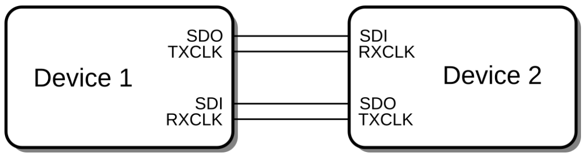

Figure 2: Topology of custom serial bus.

To demonstrate the bus, simply pass terminal characters in both directions: terminal input on Device 1 appears on Device 2's terminal and vice versa.

NOTES:

- Assume that 8 bits are transferred at a time and the devices stay perfectly synchronized (e.g., no CS lines are required to provide selection or synchronization).
- The bus should be capable of simultaneously transmitting and receiving. This implies that the individual path implementations are *non-blocking*.
- All other bus specifications are up to the programmers; such as: data rate, clock phase/polarity, etc.
- Microcontroller startup activity (e.g., GPIO initialization) may produce something that looks like a clock edge. A suggested method for dealing with this glitch is to use a timeout to discard incorrect data: e.g., once the start of a transmission is detected, it must be received within x amount of time, otherwise the receiver is reset. This implies that the designed serial buses must communicate with a minimum speed to avoid the timeout.
- Verification of this task may be done with the second device being of the instructor's design and not the group's.

Operating DISCO and Nucleo Boards on Same Computer

In order to complete most tasks of this lab, both boards will likely need to be connected to the same computer simultaneously unless using a second computer. Connection of the PuTTY, or other terminals, is fairly straight forward:

- Windows: The virtual COM port presented by each board on Windows generally stays consistent when connecting to the same USB port; therefore, tracking and determination of these ports is clear.
- Linux: The `/dev/ttyACM#` ports are assigned by first-come/first-serve. Therefore, the first device attached will be assigned to `/dev/ttyACM0` and the next will be `/dev/ttyACM1`.
- MacOS: I am unsure of MacOS's operation here - good luck!

The connection to STM32CubeIDE's programmer and debugger is less clear; however, once configured properly a single device can be tied to a single project. To do so:

1. Plug in the DISCO board ONLY.
2. Right click the project to associate the DISCO board with.
3. Select *Properties* (Bottom of context menu) →*Run/Debug Settings* (On Left).
4. Select the existing launch configuration and click *Edit...*
5. Select the *Debugger* tab
6. Enable the *ST-LINK S/N* box and click *Scan*
7. If only the DISCO board is connected, a single serial number will be available within the dropdown box next to *ST-LINK SN*. Select it and click *OK* on the bottom.
8. This specific DISCO board will now be programmed by the project. If a different DISCO board is used, the *STLINK S/N* must be adjusted.
9. With the configuration set above, the NUCLEO board may be plugged in.

Programming of binary firmware to the Nucleo Board should be done without the DISCO board connected to avoid connection to the wrong development board.