

# Microprocessor Systems Lab 1

## Checkoff and Grade Sheet

Partner 1 Name: \_\_\_\_\_

Partner 2 Name: \_\_\_\_\_

Grade Component	Max.	Points Awarded		TA Init.s	Date
		Partner 1	Partner 2		
Performance Verification: Task 1	5 %				
	Task 2 10 %				
	Task 3 10 %				
	Task 4 5 %				
TA Questioning	20 %				
Documentation and Appearance	50 %				
Aide Deduction	-				
Total:					

Grader's signature: \_\_\_\_\_

Date: \_\_\_\_\_

## → Laboratory Goals

By completing this laboratory assignment, you will learn to:

1. To use the System Workbench IDE (Eclipse Based) designed for STM32 microcontrollers,
2. Program the STM32F769NI using the GNU ARM C Compiler (a.k.a. GCC ARM),
3. Produce an ANSI/VT100 terminal display in order to interact with the microcontroller connected to a computer via a serial terminal (e.g., ProComm Plus, PuTTY, picocom, etc.)
4. Perform basic digital Input/Output tasks.

## → Preparation

- Install System Workbench IDE: follow the [Install Guide](#) steps 1-3.
- Install a serial terminal on your computer. Popular options are PuTTY, RealTerm, GNU screen, minicom/picocom. Alternatively, the desktops on the lab may be used as serial interfaces, through ProComm Plus or HyperTerminal.
- Purchase a [STM32F769I-DISCOVERY<sup>1</sup>](#) board and bring it to every class.
- Review the C language, specifically the utilities contained in [stdio.h](#).
- Read suggested materials below.

## → Reading and References

- R1. [Mastering STM32](#): Chapter 1 (Introduction) and Chapter 6 (GPIO)
- R2. [32f769i\\_Discovery\\_Manual.pdf](#): Skim entirety
- R3. [stm32f769ni\\_Datasheet.pdf](#): Skim Chapters 1 and 2
- R4. [stm32f769ni\\_Errata.pdf](#): Skim
- R5. [RM0410-stm32f7\\_Reference\\_Manual.pdf](#): Chapter 6 (notably, Section 6.4)
- R6. [UM1905-stm32f7\\_HAL\\_and\\_LL\\_Drivers.pdf](#): Chapter 2 (p.35,36 only), Chapter 26 (HAL GPIO)
- R7. [VT100-ANSI\\_Escape\\_Sequences.pdf](#)
- R8. [MPS\\_Breakout\\_Board\\_Mapping.pdf](#)
- R9. [Lab01\\_ANSI\\_Term-GPIO\\_Template.zip](#): Project Template for Lab 1

---

<sup>1</sup>In this class, we will generally refer to this as the DISCO board

## ANSI PROGRAMMING TASKS FOR THE STM32

Communication between the microcontroller and a computer, for this class at least, will be achieved through a virtual serial connection provided by the USB connection. This serial connection uses the peripheral USART1 to generate a UART interface operating at 115200 baud. The code implementing this is provided by the libraries provided: `init.h/init.c`, `uart.h/uart.c`, and is called by the `Sys_Init()` function that should be called at the beginning of any program.

Input from the terminal keyboard and output to the terminal display can be done on a character-by-character basis by using the `getchar()` and `putchar()`, respectively. Additionally, `printf()` may be used to format and output strings to the display. This is essentially the same functionality you used extensively in ENGR-2350 Embedded Control.

The VT100/ANSI terminal standards allow for the terminal to be controlled more elegantly than simply writing line-by-line. This control is performed by sending special codes, or escape sequences, to the terminal. Functionalities enabled by these standards includes but are not limited to: screen erasing, cursor location control, background and font color changing and scrolling. These codes generally start with sending the character code for the `<ESC>` key (ASCII Value: Dec:27 Hex:0x1B Oct:033). A table of some of these escape codes is given in [R7](#).

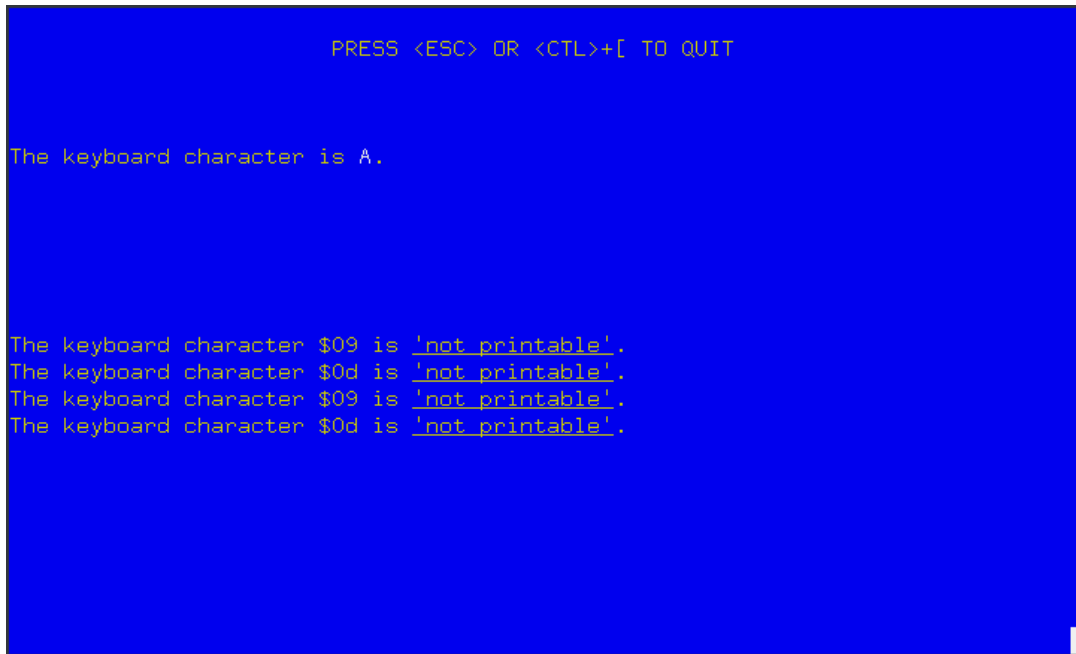
### ◇ Task 1: Introduction to the User Interface

Write a simple program to run on the STM32F769NI that outputs the text “**The keyboard character is \*.**” whenever a printable character is pressed on the terminal keyboard, where `*` is replaced with the corresponding character. The program should halt when either `<ESC>` or the key combination `^[` are pressed. Note that the notation `^[` means that the keys `<Control>` and `[` are pressed together. Display a message at the top of the screen when the program starts indicating how to terminate the program.

### ◇ Task 2: VT100/ANSI Terminal Control Sequences

To complete this task, you must modify the code from Task 1 to do the following. An example implementation of this task is given in [Figure 1](#).

1. Display yellow characters on a blue background
2. Center the program’s termination information on line 2.
3. Display the keyboard response text from Task 1 on line 6.
4. Change the color of the pressed character to red, leaving the rest of the characters in yellow.
5. For when non-printable characters are pressed, have the program beep and also add the blinking output “**The keyboard character \$XX is ’not printable’.**”. This line should initially be printed on line 12, then subsequently on the next available line. When the bottom of the terminal is reached (line 24), in order to prevent overwriting the previous message, all previous messages should be moved up, with the oldest message being removed.



```
                                PRESS <ESC> OR <CTL>+[ TO QUIT

The keyboard character is A.

The keyboard character $09 is 'not printable'.
The keyboard character $0d is 'not printable'.
The keyboard character $09 is 'not printable'.
The keyboard character $0d is 'not printable'.
```

Figure 1: Example terminal for Task 2.

#### NOTES:

1. A standard terminal is sized 80x24 characters. Depending on the terminal program you installed, your terminal may not be this size unless you manually size the window as such. You may assume for the purpose of this lab that you are targeting an 80x24 terminal.
2. The term “Escape sequence” doesn’t necessarily imply that `<ESC>` is used. For example, `\n` and `\t` are also escape sequences.
3. If the terminal doesn’t respond properly to escape sequences, it may no longer be in an ANSI compatible mode. Check your options or restart the program.
4. Scrolling of a scroll section only occurs when a newline character, `\n`, is written to the bottom of the scroll section.
5. Beep support is spotty for different terminal programs. Implementation of the beep will be considered successful as long as the functionality to produce the beep is implemented in the program code correctly.
6. It is good practice to design the program top-down, then write the routines bottom-up. The routines should be written one at a time and tested thoroughly prior to integration.

## STM32 General Purpose Inputs and Outputs (GPIO)

The STM32F769NI has 159(!) general purpose input and output pins. Unfortunately, the DISCO board that we are using does not allow us access to all of these pins. Some have been dedicated to specific uses (e.g., LCD, microSD, on-board LEDs), while a limited number (22) are routed to the Arduino standard Uno V3 connectors CN9, CN11<sup>2</sup>, CN13, CN14. Please refer to Table 5 of R2 for a complete mapping of these connectors. Table 13 in the same document provides a complete map of all I/O pins if interested. Provided in the classroom is a breakout PCB, or in Arduino terms a “shield,” that is used to provide a connection between the DISCO board Arduino connectors and the MPS Protoboards. A mapping for these connections is given in R8.

The 159 pins of the STM32F769NI are broken up into ports, labeled GPIOA through GPIOK, with the naming convention of the pins following the format  $Px\#$ , where  $x$  is the port letter (A-K) and  $\#$  is the pin number (0-15). Ports GPIOA-J are 16-pin ports and GPIOK is an 8-pin port<sup>3</sup>.

Configuration and access to the GPIO pins is enabled through the registers listed in Section 6.4 of R5. Access to each of the port registers is done through the struct pointer `GPIOx`; for example, to set the 0<sup>th</sup> bit of the `MODER` register for GPIOA, the command `GPIOA->MODER |= 0x00000001U`; could be used<sup>4</sup>.

In an effort to make use of the GPIO, among most other peripherals, easier to configure and use, the Hardware Abstraction Layer (HAL) or Low-Level (LL) drivers are given R6. These drivers provide “abstraction” functions, such that the programmer does not need to know and interface with the microcontroller registers. For this class, only the HAL driver set will be considered. GPIO configuration using the HAL drivers is done by defining a `GPIO_InitTypeDef` struct, which contains all desired configuration information for a single pin. This struct is then passed to the `HAL_GPIO_Init()` along with the GPIO port register set reference pointer (e.g., `GPIOA`) to perform all the register modifications necessary to implement the configuration defined in the `GPIO_InitTypeDef` struct. Additionally, there are abstraction functions given to read pin values, set pin values, etc.

For the GPIO functionality to work properly, each GPIO port needs its peripheral clock enabled. This may be accomplished through the register `AHB1ENR` or the HAL function `_HAL_RCC_GPIOx_CLK_ENABLE()`.

The steps necessary to read or set the voltage on a GPIO port pin are as follows<sup>5</sup>:

### Digital Input:

1. Enable the GPIO port peripheral clock
2. Set input mode
3. Internal pull-up/-down resistors
4. Read pin value

### Digital Output:

1. Enable the GPIO port peripheral clock
2. Set output mode
3. Configure output type
4. Set output speed
5. Set output value

<sup>2</sup>CN11 doesn't actually have any GPIOs connected to it, but it is part of the Arduino headers.

<sup>3</sup>Pins PJ6-11 and PK0-2 are not routed outside the chip, hence the 159 pins instead of 168.

<sup>4</sup>When reading the R5, each register's name is followed by a set of parentheses containing a notation akin to: {MODULE}-{REGISTER}, e.g., `GPIOx_MODER`. This implies that to access the register, the struct pointer {MODULE} needs to be accessed: {MODULE}->{REGISTER}

<sup>5</sup>some of these steps listed may be implied through the default configuration

### ◇ Task 3: Port Input/Ouput

Configure the 769 and wire external electronics to properly execute the following tasks. While the team should generally work together on this task, one team member should implement this functionality using only registers while the other team member uses the HAL driver (two separate programs).

- Set each accessible on-board LED (LD1-LD4) port pin to function as a digital output.
- Set the Arduino Header pins D0 through D3 to be digital inputs with internal pull-up resistors enabled.
- Wire and connect 4 external switches paired with 1 k $\Omega$  resistors on the protoboards to the D0-D3 pins. Wire the switches such that when not pressed (or toggled), the D# pin receives a value of 0V (logic LOW) and +5V when pressed/toggled (logic HIGH).
- Continuously read the state of the switches and turn on or off an LED accordingly. The switch connected to D0 should control LD1, D1 controls LD2, etc.

NOTES:

1. LD4 uses inverted logic
2. Be sure the +5V supply is NOT connected to the +3.3 V supply on the microcontroller.
3. The +5V supply originating from the Arduino headers may be used to power the switches. You should be aware, however, that the current capability of this pin is limited and you may need to use an external +5V supply in future labs.
4. Pins PA9 and PA10 are used to produce the UART link to the computer. If your UART link is not working, make sure that you have not modified these pins!

### ◇ Task 4: Input Logic Determination

Using the code from the previous task, a multimeter, and a potentiometer replacing one of the switches (connected between +5V and GND with the wiper connected to the digital input):

1. Confirm that voltages near 0V yield a logic value LOW and near +5V yield a logic value HIGH, and
2. Determine if the input logic uses a simple digital buffer or Schmitt trigger to determine these values. At what voltage level(s) does the logic value transition?