Introduction to STM32 Nucleo

Karthik Ganesan*

Introduction

This lab will introduce you to the 32-bit Nucleo platform from ST Microelectronics that you will use for the labs in ECE342. You will also learn how to use the different software tools you will need to connect to the Nucleo board.

NOTE: Unfortunately, Keil μVision is only available for Windows. If you have a Mac, you may try using the *STMCubeIDE* developed by ST Microelectronics, available here. While we expect it will work very similarly to Keil μVision , we cannot guarantee this and therefore the teaching team might not be able to support you if you run into issues using *STMCubeIDE*.

Everything you need for ECE342 is already installed on the DESL lab room computers. However, we **strongly recommend** you still download and install them on your own computer, for two important reasons:

1) The labs are only open for limited time periods. Having the software on your own computer makes it easy for you to work on the labs outside of scheduled lab sessions.

2) The version installed in the labs has a code size limit. This means, that the biggest program you can run is 32KB. While this is not an issue for the labs, some of you may have larger programs for your projects. This document provides instructions to install a version which does not have this code size limit.

NOTE: Start without the board or the USB cable connected to your computer.

1. Installing software

The software you will need for ECE342 is:

1. **Keil μVision** : The main IDE you will use to download programs to and debug the STM32-Nucleo platform.
2. **ST-LINK Drivers**: The drivers needed by the Nucleo platform to connect with your computer.
3. **Hercules SETUP utility**: This program will let you see messages sent from your Nucleo board on a connected computer.

We will now see how you can install each of these.

1.1 Keil μVision

Keil μVision is a popular IDE used in industry to work with embedded systems. We will be using the Keil **Microcontroller Development Kit (MDK)** software to program the ARM-M4 CPU on the STM32 Nucleo board. MDK uses the Keil μVision Integrated Development Environment (IDE).
Go to this website and fill in the required details. This is only required for Keil to gather statistics on who is using their software. What you fill in does not affect anything; you will still be able to download the software. Once you fill in the details, you will see a page titled **MDK ARM**. At the bottom, you will see *MDK_541.EXE*. Download this to your computer and install Keil μVision . You do not need to open Keil μVision yet; we will do that in a later step.

1.2 ST-Link Drivers

You will connect to the Nucleo board over a USB cable. For this, you will also need to install the USB drivers for the Nucleo board, which you can get from this website. **NOTE:** Make sure to download and install the drivers **BEFORE** connecting the Nucleo board to your computer. Once the drivers are installed, you may need to restart your computer for the drivers to get loaded.

*The original version of these lab documents was developed by Fabian Torres Alvarez

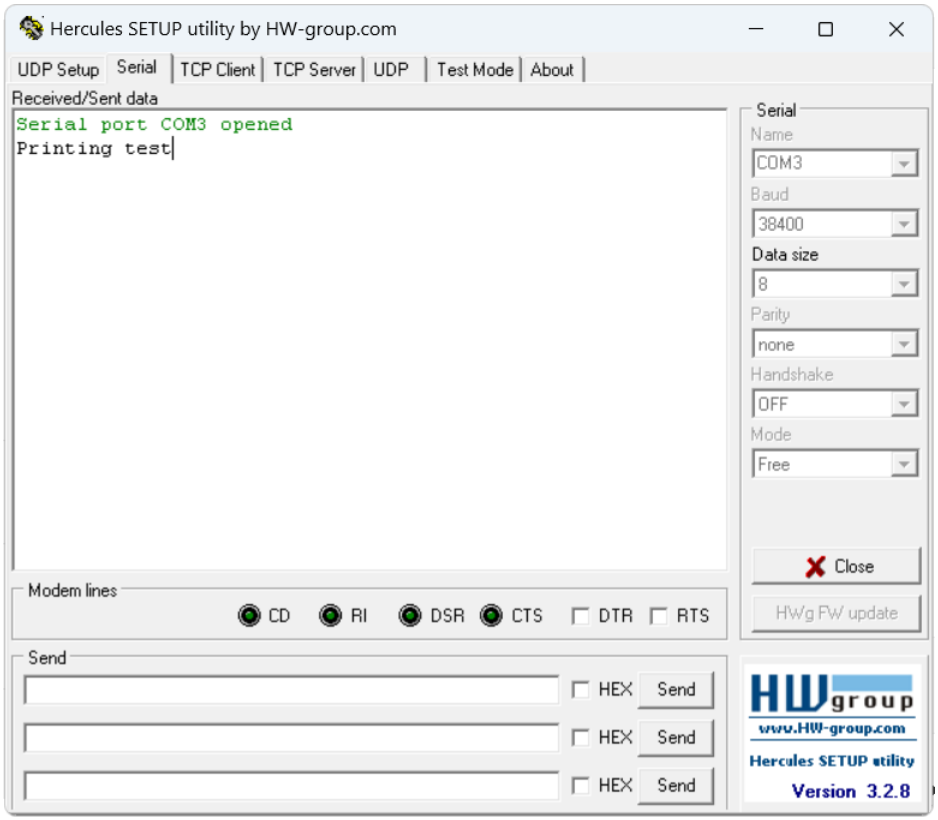

Figure 1. Hercules SETUP utility, showing the 'Serial' tab.

To verify the drivers were installed correctly, you can now connect your board to your computer using the USB cable. Your computer should show a folder titled 'NUCLEO', which means that the drivers were installed correctly. If you do not see this window, it may mean that you have a 'power-only' USB cable.
Next, you will need a program that lets you communicate with the Nucleo platform.

1.3 Hercules SETUP utility

To start with, let us see why you need this program.
Using the COM port for printing. You will be doing the majority of your programming using *C* in ECE342. On a regular computer, you can use `printf` statements to debug your code. However, embedded platforms do not have a screen so we need another way to use `printf` statements for debugging. Embedded systems frequently use a `Universal Synchronous/Asynchronous Receive Transmit (USART)` serial communication protocol to send messages from the embedded system to a regular computer. `USART` is connected to your computer via a `COM` port. You will learn how *USART* works in lectures, later in the term. Many decades ago, computers typically came with `COM` ports but modern computers do not. The Nucleo board, therefore sends this data over the same USB cable. However, your computer still 'sees' this as a separate `COM` port.
Reading COM data. Now, we need a way to read the data sent over the `COM` port and display it. For this, we will use the *Hercules* program as our 'screen' to print messages from the board to a computer. Download the program from the website here. *Hercules* is a standalone program, that can simply be run without having to be installed first.
Open the *Hercules* program and click on the *Serial* tab at the top. You will see a large window as shown in Figure 1. This is your 'terminal'; messages printed from your Nucleo board will appear here. You will not need to change anything right away; simply leave the program open for now.

2. Running your first program

Start by downloading the sample *blinky* project provided on the course Quercus. *Blinky* is often the first program everyone runs when learning to use a new embedded platform, similar to *hello world* when you learn a new programming language. This very simple program will toggle the blue LED on the board on and off at a set rate. Unzip the provided *lab0.zip* file on Quercus.

Then, navigate to *MDK-ARM/blinky.uvproj* to start Keil μVision .

The Keil μVision IDE interface: At first you will only see the *Project explorer* window on the left. This shows you the hierarchy of your entire project. As you can see, even for a very simple program, a large number of files are included in the project. Expand the *Src* folder and open *main.c*. You should see the window shown in Figure 2.

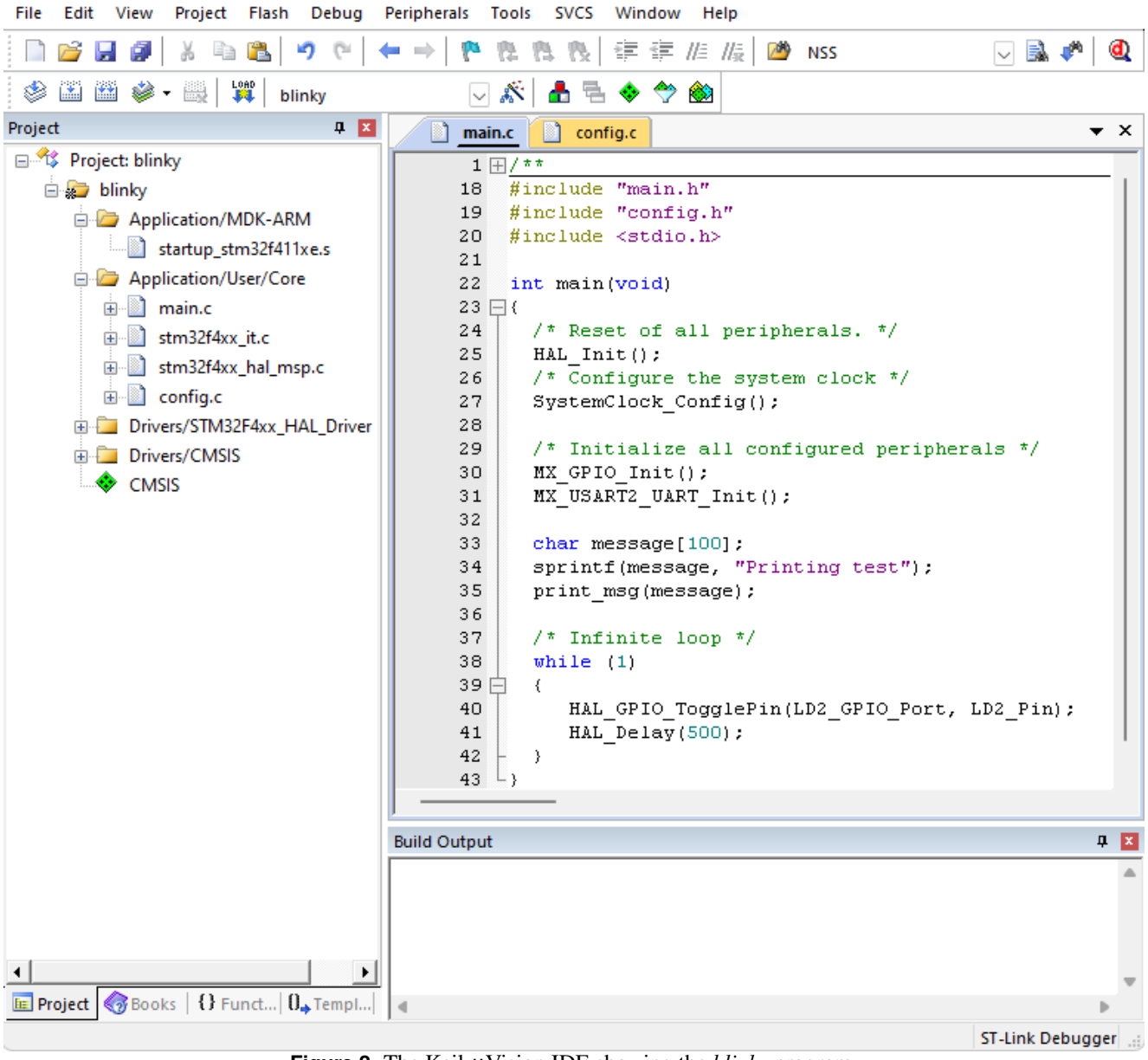

Figure 2. The Keil μVision IDE showing the *blinky* program.

Understanding the provided code: We will now go through and explain the code shown in Figure 2. After including the necessary header files, `main.c` has prototypes for functions declared later in the file. In `main()`, there are several functions with initialize different board peripherals. We won't go into the specifics of each of them but to give you an idea of what each of them does (and to give you a preview of what you will learning throughout the course), they are:

1. **HAL_init()** initializes the `Hardware Abstraction Layer` or HAL. The HAL is a set of functions provided by the board vendor (i.e., ST Microelectronics), that allows for easy access to the board's registers and peripherals.
2. **SystemClock_Config()**, provided further down in the code, configures the clock circuitry for the board, to provide the clock that our system uses.

3. **MX_GPIO_Init()** configures the `General Purpose Input/Output` (GPIO) pins on the board. The GPIOs are the main way for us to connect the board to external peripherals so we will be using the GPIOs function a lot this term.
4. **MX_USART_UART_Init()** configures the USART (or Serial) communication protocol that allows us to 'print' from the board to the *Hercules* utility.

All of these functions are defined in `config.c`. There is also code which creates a string, which we can use for printing. We will look at printing in more detail later.

Code inside an infinite loop: You will see that `main` ends with a `while(1)` loop. Code running on embedded systems typically uses such an infinite loop. As the programs we will run do not have an OS, there is nowhere for the *main()* function to return to. So, any code you want to run continuously should be placed inside this *while(1)* loop. For the *blinky* program, you will see two lines inside the loop.

Toggling the LED: The first line toggles the blue LED, which is labelled *LD2* on the board itself. As the LED is outside the actual chip, it is connected to our system via a GPIO. So, we use the *HAL_GPIO_TogglePin()* function to toggle the LED; if it is on, we turn it off and vice-versa. The toggle function takes two parameters: 1) The port that the GPIO is part of and 2) The specific GPIO pin connected to the LED. Both of these are provided as *#defines* in the *main.h* file.

Adding a delay: Since the board is running at several megahertz, if we toggled the LED every clock cycle, we would just see it being on all the time. To slow down the rate of toggling, we need to add a small delay, which we do using the *HAL_Delay()* function. This function takes in a time (in milliseconds) and causes the CPU to wait for that long before continuing.

Running on a board: To compile the `blinky` project, select *Project* > *Build Project* or press the *F*7 key. You will see the compilation output in the *Build Output* window, at the bottom of the IDE. This is where you check if you had any compile errors that you need to fix. Once this is done, you can connect your Nucleo board to your computer using the provided USB cable. To download your code to the board, select *Flash* > *Download* or press the *F*8 key. Your code should now be downloaded onto the Flash memory of the Nucleo board.
You should now see the blue LED on the board flashing at a fixed speed. Try changing the value inside the *HAL_Delay()* to change the rate of flashing. You can try running other experiments to better understand this code.
At this point, Keil μVision may say that the firmware on your board is dated and it will ask to update it, which you can go ahead and do.

Throughout the labs for ECE342, we will suggest ideas for your to try for yourself, which will be presented in boxes like the one shown below:

> **Try it yourself:** Try flashing different Morse code patterns on the LED, similar to Lab 5 in ECE241. You can find the full Morse Code alphabet here. How can you store the sequence of dots and dashes for each letter? Can you flash a word or even a whole sentence?

2.1 Printing from the board

Setting up Hercules: Now that you have run your first program, let's see how you can send information from your board to your computer. We will use the `Universal Synchronous/Asynchronous Receive Transmit (USART)` peripheral for printing. The Nucleo board we are using has several `USART` interfaces. To use for printing, we must use `USART2`, which is connected via USB to the computer.
In the provided code, `config.c` already configures `USART2` for you. Similar to the Nucleo board, computers also support many 'COM' ports. So we will need to find out which COM port the Nucleo board is connected to. Open the `check-com-port.bat` file, included in `lab0.zip`. If your board is connected (and your computer can communicate with your board), you will see the COM port the board is connected to on the terminal window that appears.
Next, open the `Hercules` program and click on the `Serial` page at the top. Configure the Serial port as follows:

1. **Name**: Select the COM port you saw above.
2. **Baud**: 38400
3. **Data Size**: 8
4. **Parity**: None
5. **Handshake**: None
6. **Mode**: Free

You must **exactly** match these settings between your Nucleo board and the hercules program. If the settings do not match, you will see random characters when trying to print. Configure the Hercules program as indicated and click 'open'.

How we print from the board: Now let us see how the code in `main.c` performs printing. We can print any array of characters but it can be a hassle to manually convert values to be strings. This is one of the advantages of using `printf`; we can print variables of different types (e.g., int, float) along with text using specifiers such as `%d` or `%f`, as you learned about in APS105.
To do the same thing on this board, we can use the `sprintf` function – which operates in the same way as `printf` – but 'prints' the message to a character array instead of the terminal (i.e., stdout). To use this function to print, we need the following:

1. An array of type `char` to hold the message we want 'printed'.
2. Include `stdio.h` at the top, so we can use the `sprintf` function.
3. Use `sprintf` to write to the character array.
4. Call the `print_msg` function by passing in the character array we just created.

The `print_msg` function itself is very simple; you can see this function in 'config.c'. `print_msg` simply takes a null-terminated string (i.e., a character array that ends with `\0`) and prints each character one by one. You can continue to use the `print_msg` function to help you debug your code for the rest of ECE342.

Using printing carefully: While having the ability to print is a useful debug tool, you must be careful when doing this. While running `printf` on a regular computer does not incur much overhead, this is not the case on embedded systems.
Printing long messages very often can slow down your board significantly. Keep your messages as short as possible and print only the bare minimum information. Keep in mind you can also use the LED to debug you code. For example, you can set the LED to blink at a specific rate at a certain point in your code; this way, the blinking LED will tell you that your program successfully ran to that point.

Exploring the project: The Keil IDE provides you with a lot of tools you can use to help write, test and debug your programs. For example, to take a look at *main.h*, you can just click on the file in the *Outline* window shown on the right.

You can also try running the program in 'Debug' mode, which allows you to set breakpoints and view register and memory values after each instruction. However, keep in mind that 'Debug' mode only affects the CPU and does not affect the behaviour of peripherals. You should be careful when using 'Debug' mode as you may not be able to reproduce a failure you see in the regular 'Run' mode, when using 'Debug' mode. You should try this out now so you are familiar with it for your labs, when it will come in handy for you to debug stubborn issues.