ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

Machine Learning on Microcontrollers

227-0155-00L

Exercise 4

Intro to Embedded Programming : UART, GPIO, CMSIS, Profiling

ETH Center for Project-Based Learning

Wednesday 8th October, 2025

1 Introduction

In this exercise you will learn how to effectively configure and program the STM Microcontroller (MCU) that is going to be used throughout the first half of this course. The development board for this course is the `B-L475E-IOT01A2` featuring an `STM32L475VG` low-power MCU with a maximum clock frequency of $80\,\mathrm{MHz}$, $128\,\mathrm{KB}$ of RAM and $1\,\mathrm{MB}$ of flash memory. The processor core is based on the ARM Cortex M4F, featuring advanced Digital Signal Processor (DSP) operation capabilities and a floating-point unit.

The board includes several peripheral devices such as digital microphones, several communication antennae and other I/O. In this exercise you will configure the General Purpose Input/Output (GPIO) pins for communication via Universal Asynchronous Receiver Transmitter (UART) and profile software implementations with STM32 CUBE IDE .

STM32 CUBE IDE comes with a C / C++ cross compiler and support for the ARM Cortex M hardware platforms. They also include powerful debugging tools and support for including often used software libraries such as the Cortex Microcontroller Software Interface Standard (CMSIS) and FreeRTOS.

For communication with the microcontroller we suggest to use TERA TERM or `screen`. However, if you are already experienced with UART communication, feel free to use any serial terminal software you like.

Note that all software we present in this course is only a suggestion. If you are somewhat experienced and feel more comfortable with other tools, you are of course free to use those. For this course we strongly suggest using a toolchain based on the STM CUBE MX software. STM CUBE MX allows us to configure often used modules with a GUI to speed up the development process. Further explanation of STM CUBE MX and KEIL µVISION can be found in an appendix section.

In the first part of this exercise you are going to see how to use a serial port to transfer data between your computer and the microcontroller via UART.

In the second part of the exercise you are going to learn about the specifics of the DSP instruction set used in ARM Cortex-M4F devices. Specifically you will learn about CMSIS DSP and see how you can write efficient code for ARM Cortex microcontrollers.

2 Notation

Student Task: Parts of the exercise that require you to complete a task will be explained in a shaded box like this.

Note: You find notes and remarks in boxes like this one.

3 Preparation

For the following tasks you will need STM32 CUBE IDE and TERA TERM or `screen`. Make sure all of the required software is installed and configured correctly. You can download the software from the internet.

To get started, you will configure a new project in STM32 CUBE IDE , add some code, compile it and view the output in TERA TERM or `screen`.

4 STM32 Cube IDE

STM32 CUBE IDE is a custom Integrated Development Environment (IDE) for STM32 products, based on the Eclipse IDE. It comes with the arm-none-eabi-gcc compiler, i.e. a GCC based toolchain.

Starting a new project

You can create a new project by navigating to `File→New→STM32 Project`. Then, by clicking on BOARD SELECTOR you get to the board selector interface. Either type the board name into the search bar or manually navigate the board selection to find the `B-L475E-IOT01A2` board and select it in the table. Once selected, press NEXT > in the bottom right corner of the window. Enter the desired project name and storage location for the project files and press FINISH. Click NO in the pop-up interface to start a new project without default pin assignments.

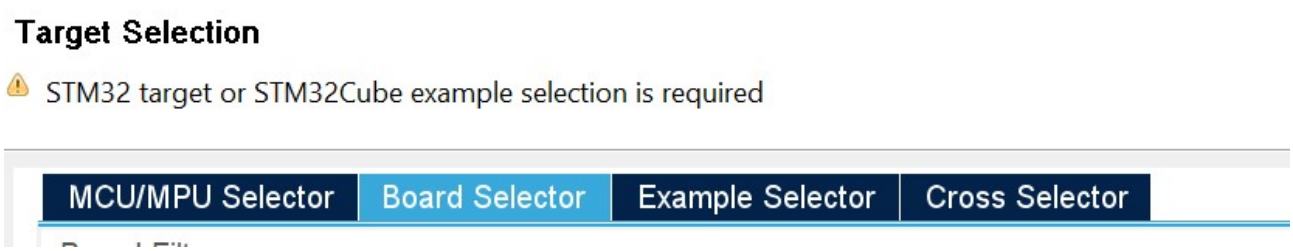

Figure 1: Board selector menu

Note 1: There is currently a bug where choosing NO in the default pin assignment dialog does not clear the pin assignment. You can still clear the pinout assignment in the Device Configuration Tool.

Changing project configurations

STM32 CUBE IDE includes an interface with STM CUBE MX , called Device Configuration Tool, so you'll be able to configure your project settings without changing software. To do this, simply select the *ioc* file and open the associated interface. You can then change settings as you would with STM CUBE MX . See the image below for guidance.

To start off a new project there are a few configurations that need to be changed here. Navigate to `Project Manager`. There you can view the Project Name, Location and Toolchain, as well as change the minimum Stack and Heap Size. Since the default values for Heap and Stack size are very low (512 and 1024 bytes), you will have to make sure that you assign enough for the software you plan on writing.

All practical projects should consider clock frequency and power consumption. For embedded systems the clock frequency should be chosen as low as possible while still meeting all timing requirements. In the `Clock Configuration` menu, you can set the core and peripheral clocks. STM CUBE MX allows you to individually set clocks for almost all modules. The clock setting for the processing core is called **SYSCLK**.

After changing settings in the Device Configuration Tool or STM CUBE MX , code has to be generated such that these settings can be properly compiled. To do this, save the .ioc file and accept the code generation popup, or navigate to `Project→Generate Code`.

> **Note 2:** The code generated by STM CUBE MX is structured by comment regions. There are pre-defined areas where user code can be inserted, and other areas where only generated code should be. This system allows you to make changes in STM CUBE MX without overwriting your own code.

Do not forget to regenerate the code after changing settings!

Figure 2: The Cube view within STM32 CUBE IDE

Pinout assignment

Since doing computations without external inputs and outputs is pointless for an embedded system, configuring input and output pins is an essential step for any project. In the `Pinout & \ Configuration` menu you can set MCU wide configurations for all GPIO pins. You can do this by either clicking on specific pins and chosing a function or by using the function menu on the left hand side.

You have to be aware that some modules offer special functionality only on certain pins. One of these modules is `USART1`, which can connect a serial interface to a connected computer via the UART protocol. The connected computer will only be able to communicate if the TX/RX pins are configured on PB6 and PB7, respectively.

Should you be unhappy with your pin assignments, you can always reset them by clicking `Pinout→\ Clear Pinouts`.

Middleware

STM CUBE MX provides support for advanced drivers and libraries such as Cube AI and software stacks for Bluetooth or Wi-Fi communication. These software components have to be selected in the `Additional Softwares` menu.

Student Task 1 (Configuring a new project with UART):

1. Create a new project with STM32 CUBE IDE . Make sure that the project name is UART-Communication.
2. Configure the USART1 module by navigating to `Connectivity→USART1`. Choose the mode to be synchronous, baud rate $115\,200\,\mathrm{Bits/s}$, word length $8\,\mathrm{Bits}$, no parity bits and 1 stop bit. Leave all other settings as they are.
3. Make sure the pins PB7, PB6 and PB5 are assigned to RX, TX and CK respectively.
4. Generate the code for the project.

Opening a project

To open a project generated by STM CUBE MX or downloaded from the course website or github, simply navigate to `File→Open Projects from Filesystem` and open the folder containing the entire project.

Configuring project settings

To configure project-wide settings, navigate to `Project→Properties`. To configure the toolchain, navigate to `C/C++ General→Paths and Symbols`. In the associated dialog window you can configure project-wide pre-processor symbols, include paths and library paths. Remember that some of these settings work differently for the ARM compiler - for example, if you want to include a pre-compiled static library, your library name will have to have a name of the format lib**name**.a. You can then import it in the `Libraries` menu, by adding **name**. You will also have to add the path to the library file in the `Library Paths` menu if it isn't located in the root folder.

Managing project files

Managing project files is very easy. Simply drag and drop the files you want to add into the folder you want to add it to. If new files were added to the file system at the project location, update the project in the IDE by right-clicking on the project and pressing `Refresh`.

Adding modules

Adding standard modules like CMSIS, basic drivers or Real-Time Operating System (RTOS) modules is done with the STM CUBE MX interface that you can open with the *.ioc* file. Add the software you want to use with the `Additional Software` dialog. Note that you might have to install the packages first - You can do this in the `Help→Manage embedded software packages` dialog.

Compiling and flashing a project

To run your code, you first have to build your project. To do this, simply click the BUILD button with the hammer icon.

If compilation succeeds, you can press the DEBUG button with the bug icon to run your code in the debugger.

Figure 3: The Build and Debug icons in the Toolbar

Using the debugger

Once the code is flashed, you can run it without the debugger. To start code execution in the debugger, press the RESUME button.

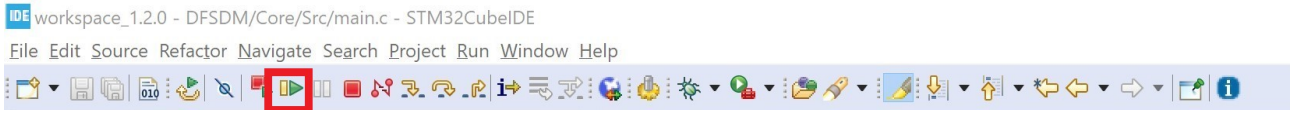

Figure 4: The Resume button in the toolbar

4.1 Printing output with UART

C as a language is designed to be portable between many hardware platforms. The standard library *stdio* is designed to implement input and output functions.

Student Task 2: Open the UARTCommunication project and add an include directive for stdio.h in the main file. Make sure you write your own code in the specified regions between
`\* USER CODE BEGIN ... *\`
and
`\* USER CODE END ... *\`.

To port this standard library to any platform and hardware interface, we need to implement one function to read a single character from a stream and one function to write a single character to a stream. We can then use the well-known *stdio* functions like `printf`. The functions needed are `__io_putchar` and `__io_getchar`.

For this exercise, we would like to transfer data from the microcontroller to a computer with the *USART1* interface. We therefore use the Hardware Abstraction Layer (HAL) functions `HAL_USART_Transmit` and `HAL_USART_Receive`.

Student Task 3: Add a function definiton for `__io_putchar` containing the following code to an appropriate section:

Implementing the `__io_getchar` function is done in the same way as `__io_putchar`. Since you will not use any formatted reading from the UART interface in this exercise, you do not have to implement it.

With the `__io_putchar` function implemented, functions like `printf` will now work and target the USART interface as output stream.

Student Task 4:

1. Add a `printf` statement in the main function body
2. Compile and flash your project onto your development board

We now have flashed the program onto our MCU and are ready to run it and observe the output.We will do this using TERA TERM or `screen`.

5 Serial Interface

Now that our software is flashed on the MCU, we can use a serial terminal to observe the output.

5.1 Windows: Tera Term

In this course we recommend that you use TERA TERM for Windows, however, you are free to use any other serial terminal you like. To use TERA TERM , open the application and chose the correct serial port. Then, configure the settings in `Setup→Serial port...`.

5.1.1 Logging

Logging in TERA TERM is quite simple: navigate to `File→Log...` and choose an appropriate save location.

5.2 Unix: `screen`

For Unix devices (like macOS and Linux), a serial terminal is usually pre-installed in the main terminal utility. Simply open Terminal and use the `screen` command as follows:

```
screen /dev/<console_port> 115200
```

where `<console_port>` is the usb device of your microcontroller. The port name will usually start with `tty.usbmodem` and can be found using `ls /dev/tty.*`. Re-plugging the device and using

autocomplete may assist in finding the appropriate device. Another way of finding the correct port is using the `lsusb` and `dmesg` commands. The `115200` in the command indicates the baud rate and is required, as the standard baud rate is 9600.

Note 3: On macOS you can enter the following command to get the name of the serial port associated with your STM32 device:
`ioreg -r -n "ST-Link VCP Data" -l | grep "IODialinDevice".`

To exit `screen`, simply press `ctrl+a` and type `:quit` and press Enter, or press `ctrl+a` and then `ctrl+k` to kill the session.

5.2.1 Logging

As it may be necessary to log the serial output of you microcontroller to a file, a quick guide is described here. Simply add the `-L` option to your screen command to enable the logging feature. Depending on your version of screen this will work a in different ways, but the main command to start logging is achieved by pressing `ctrl+a` then `H` (ensure it is capital). This will start and stop logging, writing the captured content to a file, by default named `screenlog.n`, where n is a number. The required command would then look similar to:

```
screen -L /dev/tty.usbmodem141103 115200
```

Student Task 5:

1. Set up a serial communication interface according to your OS.
2. You should now see the output you programmed in the `printf` statement from the previous task if you press the RESET button on the development board. If you do not see the output, try to debug your program, check your settings or ask an assistant for help.

6 Measuring and optimizing software performance

In this part of the exercise we will take a look at some straightforward optimizations we can use to accelerate the execution of our programs and introduce libraries which provide us with re-useable optimized code.

Note 4: We will not explore the more general techniques for optimizing and hand-tuning platform specific code, but if you are interested, there is a lecture at IIS called *Systems-on-chip for Data Analytics and Machine Learning* which explores the hardware optimization methods for machine learning applications.

For this part of the exercise we provide you with a pre-configured project. The goal of this exercise is for you to get comfortable with profiling the performance of code assets and using the CMSIS libraries.

ARM Cortex M4 and CMSIS DSP

CMSIS DSP[1] is a hand-tuned library provided and maintained by ARM which leverages the DSP instruction set of the ARM Cortex M4F processor core and employs general numerical code optimizations to achieve highly optimized code for mathematical, signal processing and control applications.

The most well-known DSP instructions are the Single Instruction Multiple Data (SIMD) extensions which allow you to process multiple data items of lower bit width at once, leading to higher throughput for applications which allow for it. Another important instruction is the Multiply-Accumulate (MAC) instruction, allowing for multiplication and addition in a single cycle.

More precisely, CMSIS offers support for vector/matrix operations, complex number operations, filtering functions including convolutions and correlations, transform functions including Fast Fourier Transformation (FFT) and Discrete Cosine Transform (DCT), PID controller and vector transforms, standard statistics functions and linear/bilinear interpolation functions.

Most functions provided by CMSIS DSP are available for floating point numbers and fixed point number in Q1.31, Q1.15 and Q1.7 format. In machine learning we often use lower bit precision to achieve higher throughput by using special hardware intrinsics.

The first part of this exercise will be to use the debugger to profile the runtime of a naive FFT implementation written by Project Nayuki using the famous Cooley-Tukey algorithm.

Student Task 6:

1. Open the FFTProfiling project. Do not unzip the Project beforehand. You can choose `Archive` when opening the project from file system.
2. Read the code and try to understand what it does. What is the input vector?
3. Start the Debugger and set breakpoints such that you can calculate the runtime of the `fft_transform` function call. You can change the *#define vectorSize* to any power of 2 you like to see how the runtime scales.

`fft_transform` runtime in cycles for vectorSize 64:

Note that STM32 CUBE IDE does not easily support reading the debug counter. We suggest using the CycleCounter.c functions.

Now that you have a runtime for a naive, non-optimized implementation, let us compare it to a more optimized implementation of the same algorithm, provided by the CMSIS DSP library.

Student Task 7:

1. Ensure the CMSIS DSP library is imported in the `Drivers` folder of the project[a].
2. Ensure *arm_math.h* is included in `main.c`
3. Comment out the naive FFT implementation and uncomment the CMSIS implementation

[1] The user manual is available at https://arm-software.github.io/CMSIS_5/DSP/html/index.html, whilst the repository can be found at https://github.com/ARM-software/CMSIS-DSP.

4. Once you compile the code, an error message might pop up. Fix the underlying mistake by adding the appropriate preprocessor symbol in the project configurations **Add the following to the preprocessor symbol: ARM_MATH_CM4 This configures CMSIS DSP to use the appropriate instructions/code.**
5. Measure the runtime of `arm_cfft_radix2_f32`, use the functions in CycleCounter.c and print them to TERA TERM

Note that you can not use access the internal debug registers like the cycle counter from within the KEIL µVISION debugger, since the debugger blocks access to these registers.

`arm_cfft_radix2_f32` runtime in cycles for vectorSize 64:

---

[a] The library should already be inlcuded in the project. In case it is not, a guide on installing the library is available at https://community.st.com/t5/stm32-mcus/configuring-dsp-libraries-on-stm32cubeide/ta-p/49637

You can see that using the CMSIS DSP code is much more efficient. Try studying the CMSIS code to see how it works and which parts of it make it run faster. There is some documentation available online for the CMSIS libraries, which can help with understanding.

Student Task 8: In the previous task the function `arm_cfft_radix2_f32` was used. This implementation is functionally the same as the `Fft_transform` implementation.

However, we can do much better. Take a look at the available CMSIS DSP implementations and try to find a more efficient FFT implementation for the given input vector.

---

7 Using the microphones on the development board

Using the peripherals correctly is important, since debugging for example an audio driver can be very difficult without a high-precision oscilloscope. For this exercise we are going to take a look at the embedded microphones on your development board and how to use them correctly. The hardware setup we are going to use consists of two digital microphones running on the same clock. To distinguish them, the right microphone (which is the one closer to the middle of the board) uses the rising edge of each clock cycle and the left microphone uses the falling edge.

In terms of encoding and conversion, the microphones use a $\Sigma\Delta$-modulator which returns a Pulse-Density Modulation (PDM) which can be low-pass filtered to get the amplitude of the signal. The exact inner workings of a $\Sigma\Delta$-modulator are somewhat complicated, so we will simply assume we receive a PDM signal from each microphone.

Once the MCU has received the PDM signal, it can be converted to the well known amplitude-time format called Pulse-Coded Modulation (PCM).

For this conversion, the pulses from the microphone are low-pass filtered by the Audio Digital Filter (ADF) peripheral using a 4th order Sinc Cascade Integrator Comb (CIC). The exact working of this filter is not important in the scope of this course, but we will need the parameters of it to calculate the final output sampling rate.

Once the ADF peripheral is done, the Direct Memory Access (DMA) is configured to efficiently save the microphone sample amplitudes into a given array for further processing.

Our audio chain has the following parameters:

- Core System Clock: $HCLK$
- Processing Clock Divider: PD
- Processing Clock: $PCLK$
- Output Clock Divider: OD
- Output Clock: $OCLK$
- Decimation Ratio: R

Figure 5 shows how the system clock is configured by the dividers and how the ADF is integrated.

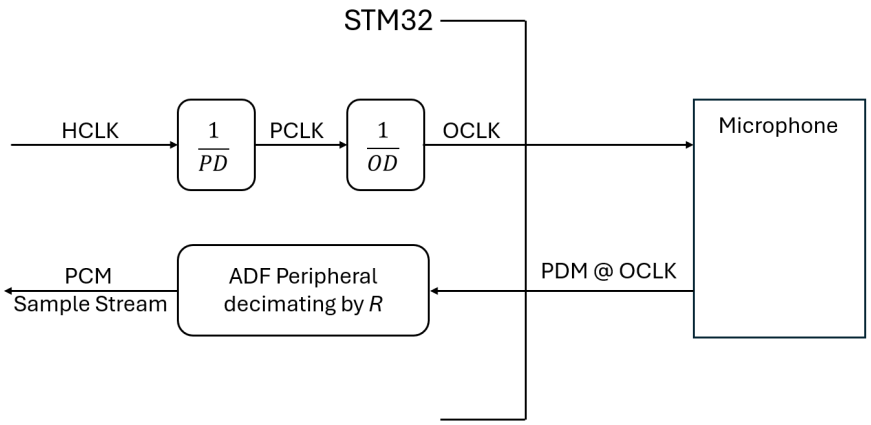

Figure 5: Audio Chain Diagram

The following equations are needed to calculate the sampling frequency:

$$\text{PCLK} = \frac{\text{HCLK}}{PD}$$

$$\text{OCLK} = \frac{\text{PCLK}}{OD}$$

$$f_{\text{PDM}} = \text{OCLK}$$

$$f_{\text{sampling}} = \frac{f_{\text{PDM}}}{R}$$

Note 5: To start data acquisition call
`HAL_MDF_AcqStart_DMA(&AdfHandle0, &AdfFilterConfig0, &AdfDmaConfig0);`
where `AdfDmaConfig0` contains the buffer configuration. This is given in the example project.

There are two interrupts that are called when the buffer is half full or full, respectively. These are `HAL_MDF_AcqHalfCpltCallback` and `HAL_MDF_AcqCpltCallback`.

The ADF and DMA process the data stream from the microphone in $32\,\mathrm{Bit}$ singed integer precision. However, this is way over standard $24\,\mathrm{Bit}$ audio bit depth. We even suggest to use lower $16\,\mathrm{Bit}$ precision, since going higher leads to slower processing in machine-learning pipelines.

Setting up a correct and sensible configuration can be quite hard. We provide you with an example project to show you some of the important configuration points.

Student Task 9:

1. Open the ADF_U5 project in the STM Cube IDE
2. Open the STM CUBE MX project file *./ADF_U5/ADF_U5.ioc*
3. Take a look at the Clock Configuration tab. What is the main clock frequency $HCLK$?
   - Main clock frequency:

   ---
4. Now switch back to the Pinout & configuration tab. Under Computing, look at the ADF1 peripheral configuration and find the following values:
   - Processing Clock Divider: PD

   - Output Clock Divider: OD

   - Decimation Ratio: R

Configuring microphones might be slightly different between MCU manufacturers or even different MCU from the same manufacturer. The general working principles of embedded microphones however is usually the same. Thus, finding these values is an important step to match your machine learning processing to the actual collected samples.

Student Task 10: Using the equations and the code in the project, find the following values:

- Output data rate:

  _____

- Output resolution:

  _____

In machine learning, data sets are the most valuable resource. It is therefore paramount that we are able to acquire data from our target platform, since different hardware might result in different enough data to throw off a trained network. Therefore it is always optimal to log data from our development board to train any algorithm.

The provided ADF project is set up to print all acquired microphone samples to the UART interface. We can use TERA TERM to log all the data for further processing.

Student Task 11:

1. Build and flash the software
2. Open your serial interface and set it up appropriately for logging
3. Reset your board and record some noises
4. Use your preferred software to visualize the data

Note 6: The log will quickly grow, so make sure you do not record for too long.

If you have any time left, we highly suggest to study the code and see how the data acquisition is started and how the data is formatted before output. Additionally, we provide a python script so you can listen to the audio your STM32 recorded! If you have any questions, talk to an assistant.

$\mathcal{E}$ **Congratulations! You have reached the end of the exercise.**
If you are unsure of your results, discuss with an assistant. $\mathcal{E}$