STM32F407 Discovery Kit Familiarization

Contents

Introduction to STM32F407 Discovery Board

STM32F407 Discovery Kit Data Brief:
https://www.st.com/resource/en/data_brief/stm32f4discovery.pdf

STM32F407 User Manual:
https://www.st.com/resource/en/user_manual/um1472-discovery-kit-with-stm32f407vg-mcu-stmicroelectronics.pdf

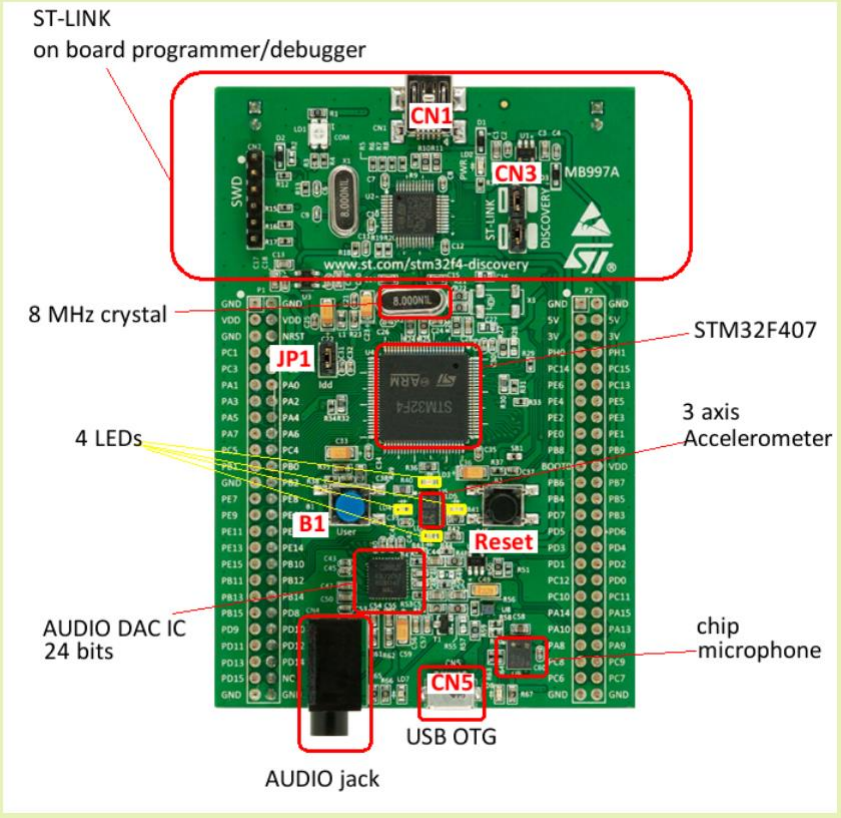
STM32F407 Datasheet:
https://www.st.com/resource/en/datasheet/stm32f405rg.pdf

STM32F4 Reference Manual:
https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

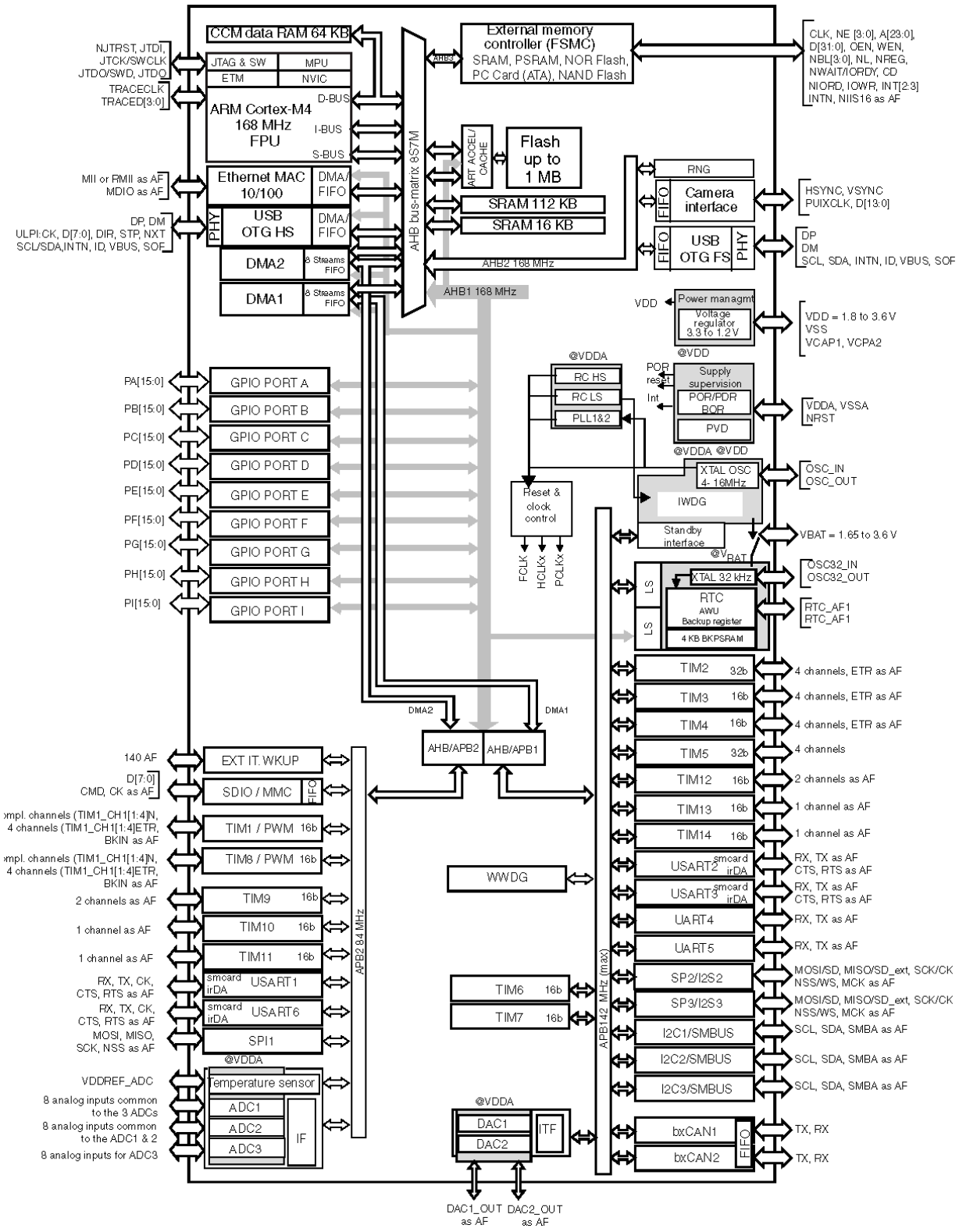
STM32F407 Discovery Kit schematic:
https://www.st.com/resource/en/schematic_pack/mb997-f407vgt6-b02_schematic.pdf

HAL (Hardware Abstraction Layer) User Manual:
https://www.st.com/resource/en/user_manual/um1725-description-of-stm32f4-hal-and-lowlayer-drivers-stmicroelectronics.pdf

Photo of STM32F407 Discovery board



Block diagram of STM32F407 Discovery board

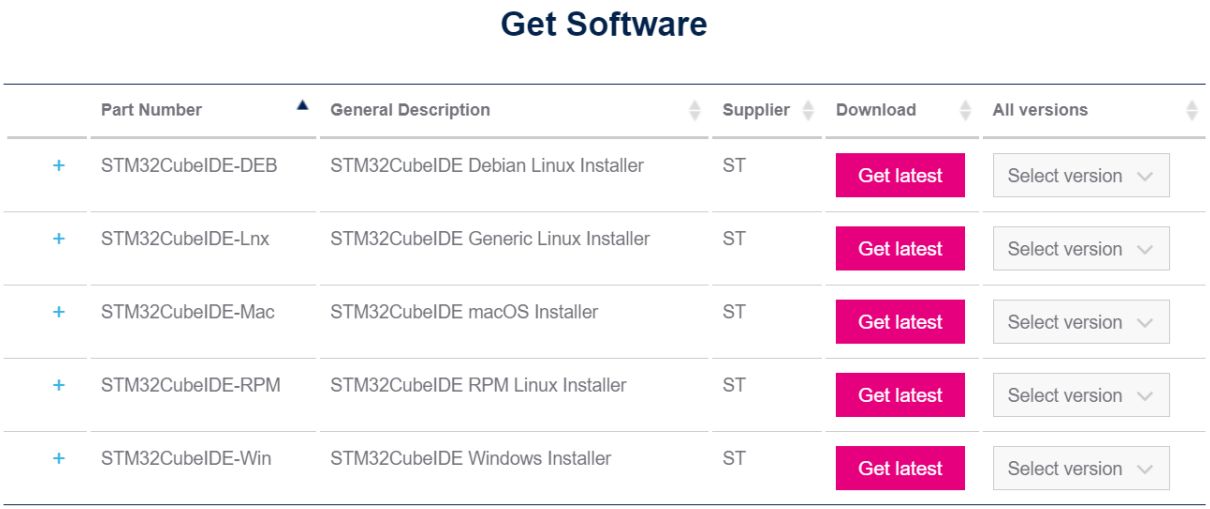


Setting up STM32CubeIDE

STM32CubeIDE is the development environment for all STM32 processors.

Register at https://www.st.com/en/development-tools/stm32cubeide.html. This will enable to you to access all STM32 software.

After signing in, Get Software for Windows at https://www.st.com/en/development-tools/stm32cubeide.html and install.



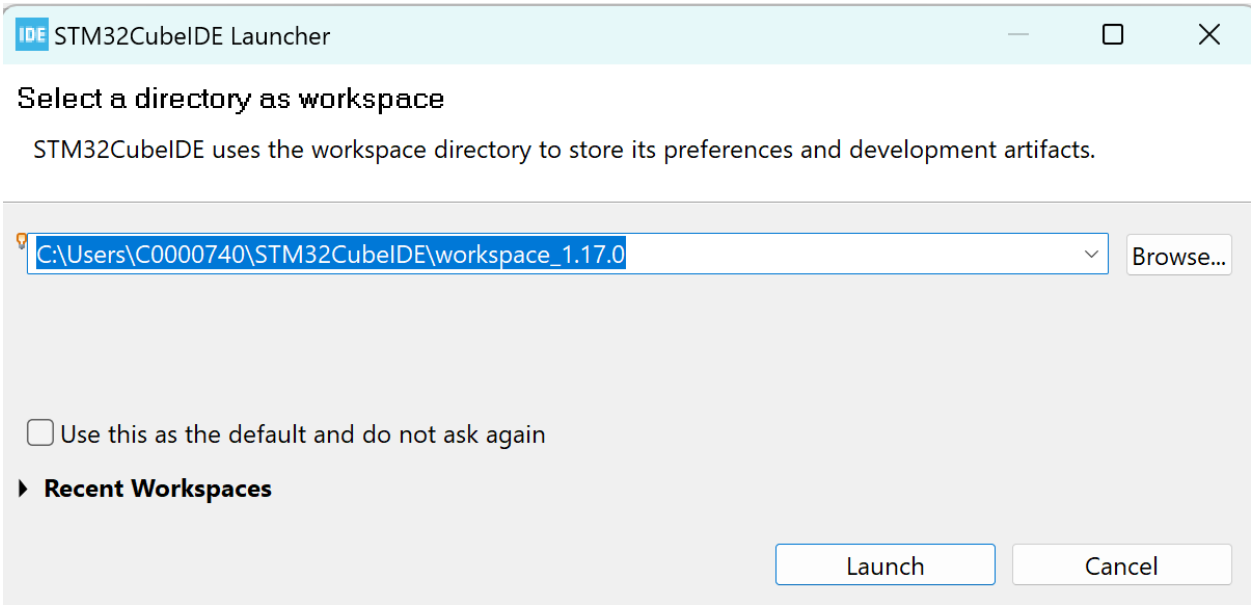
First project: `blink_LED`

Open STM32CubeIDE by double-clicking on the icon.

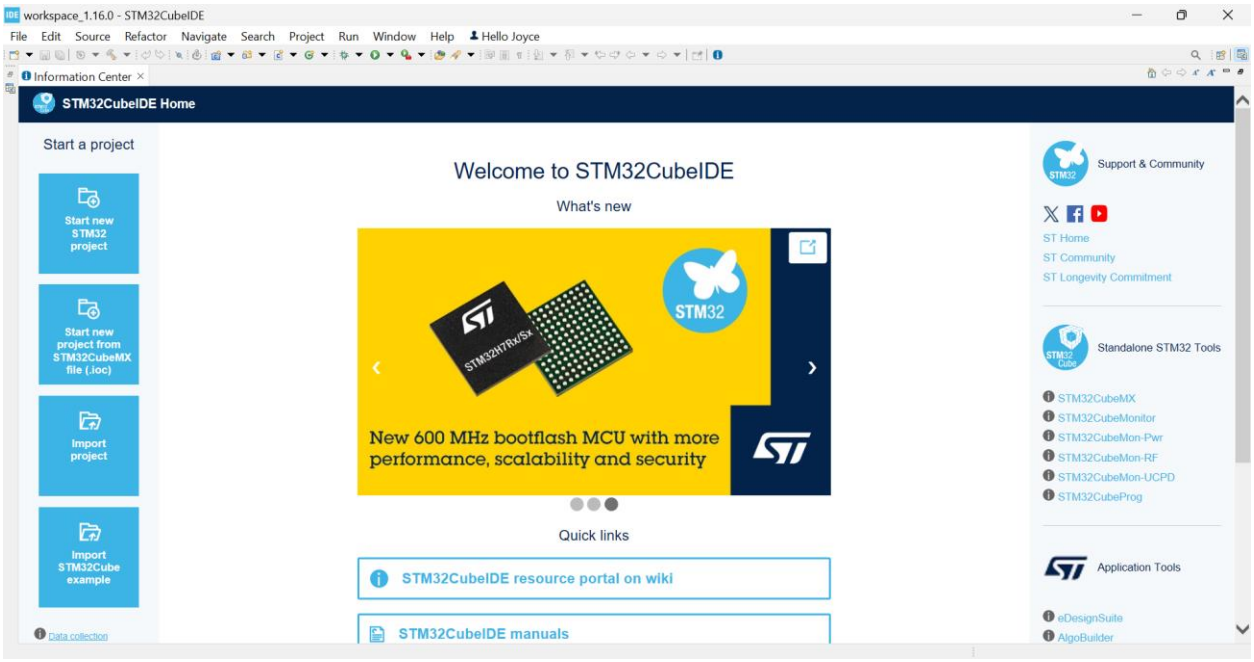
You will be prompted to choose a workspace. The default location is:

`C:\Users\<username>\STM32CubeIDE\workspace_1.17.0`

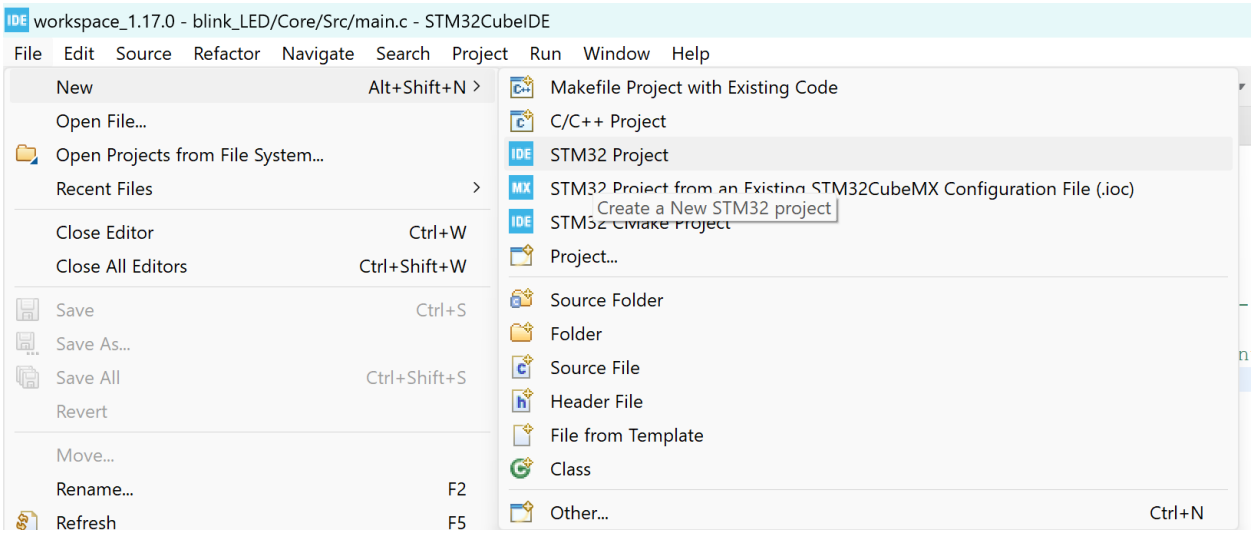
where the number at the end is the version of STM32CubeIDE.



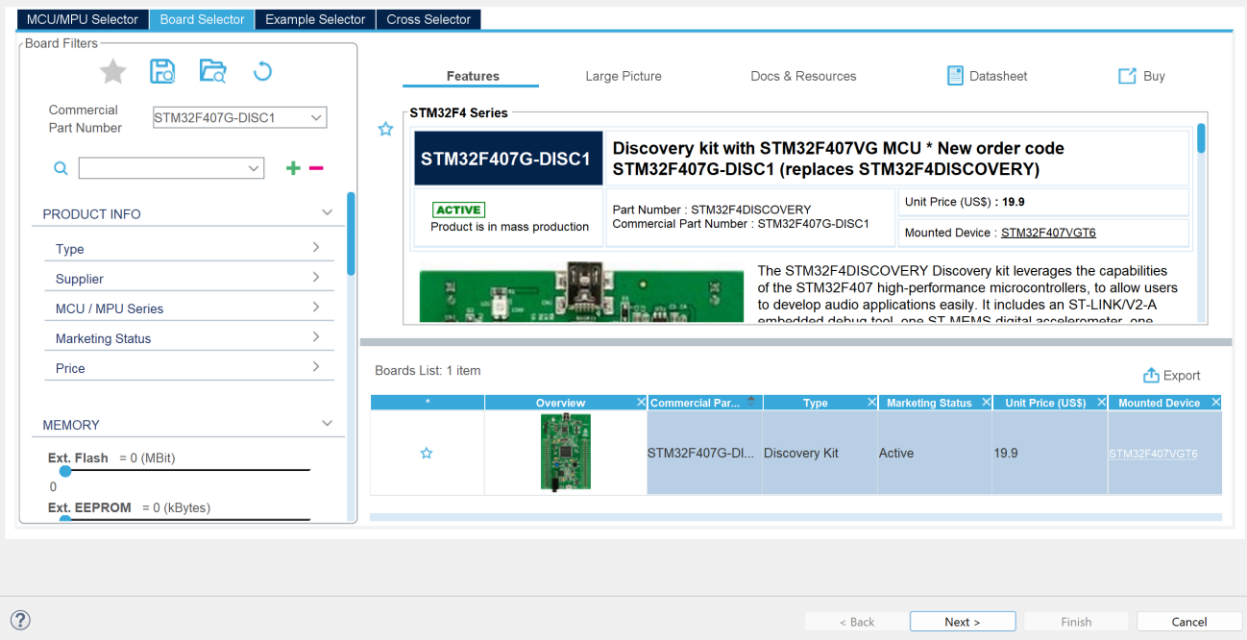
Click Launch.



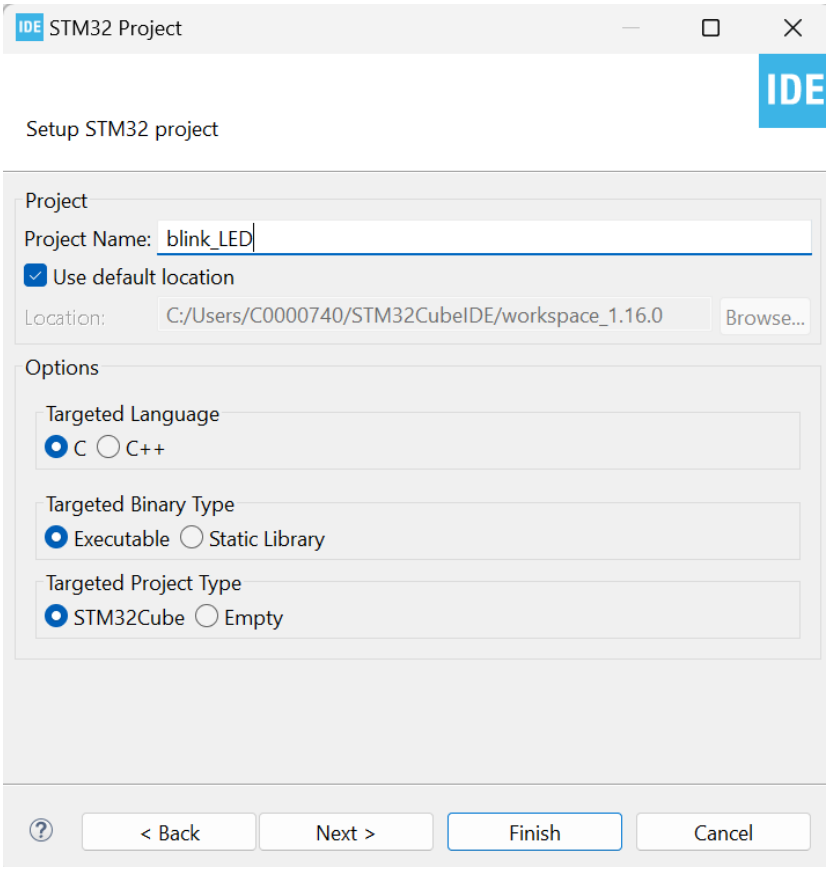
To start a new project, either click on the blue tile “Start new STM32 project” or choose File – New – STM32 Project.



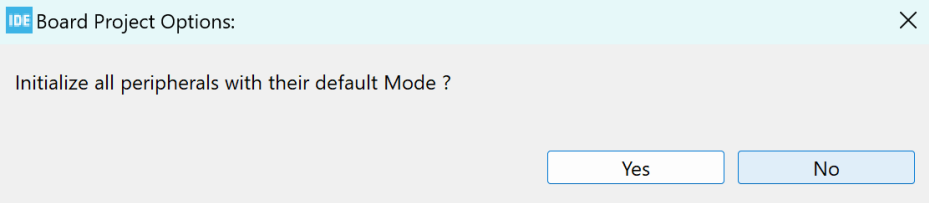
After a few moments a Target Selection window will appear. On the Board Selector tab, type 407 into the Commercial Part Number textbox, and select the STM32F407G-DISC1. Click on the image of the board on the right to select it, and click Next.



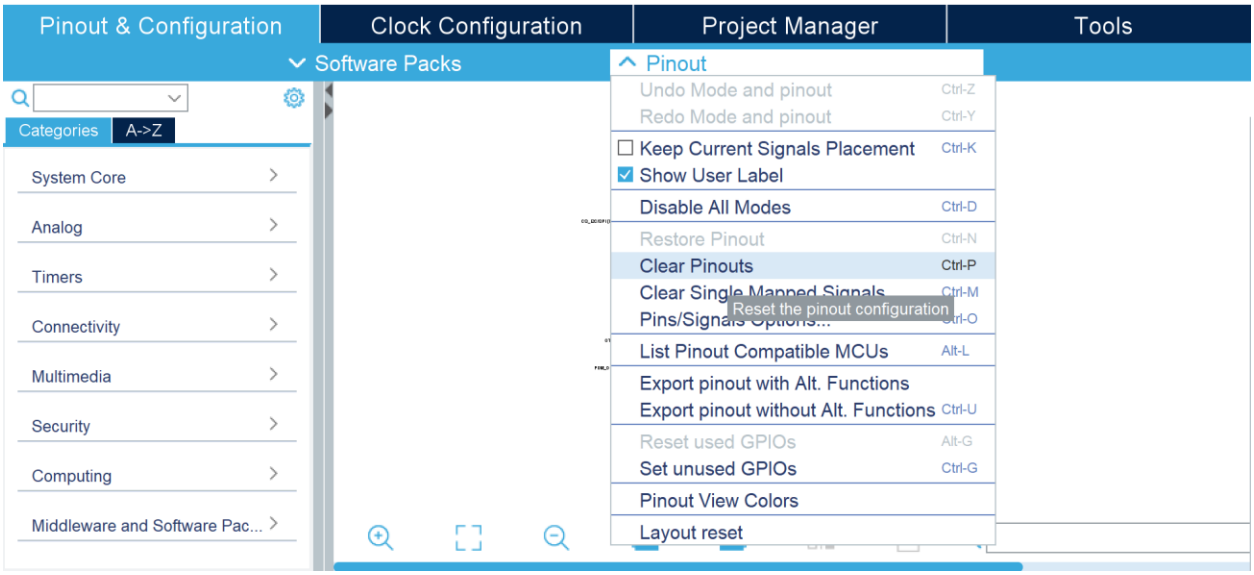
Type in a Project Name, for example, `blink_LED`, and click Finish.

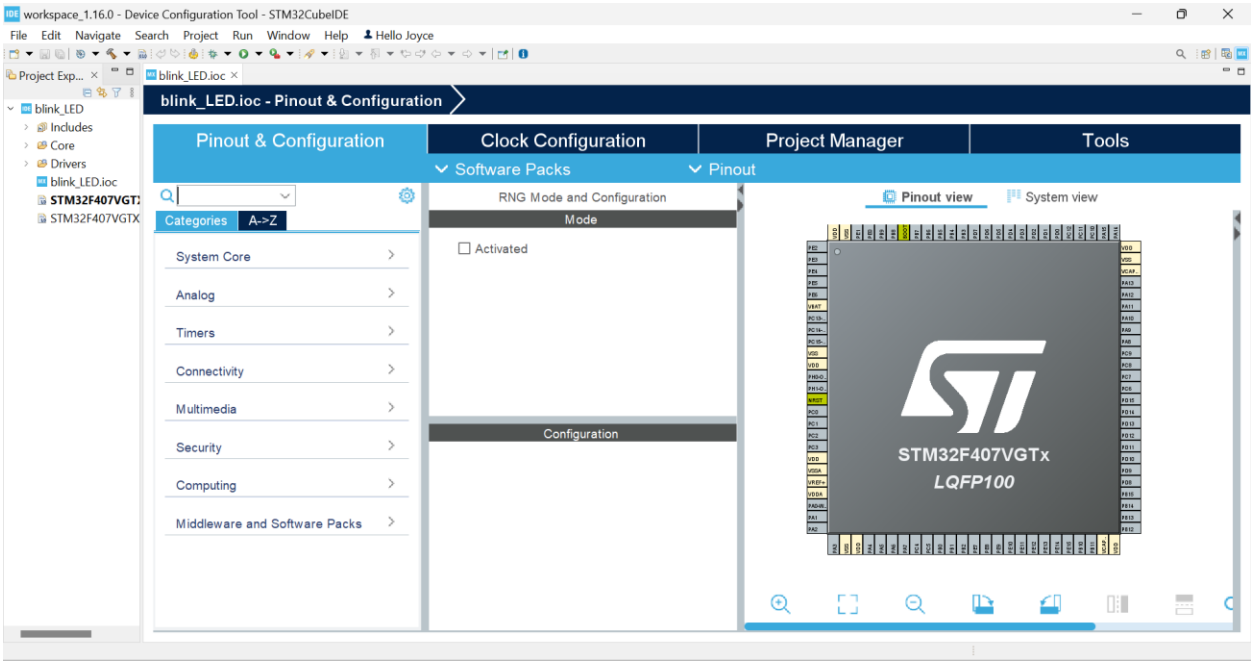


When asked if you want to initialize peripherals with default values, say no.



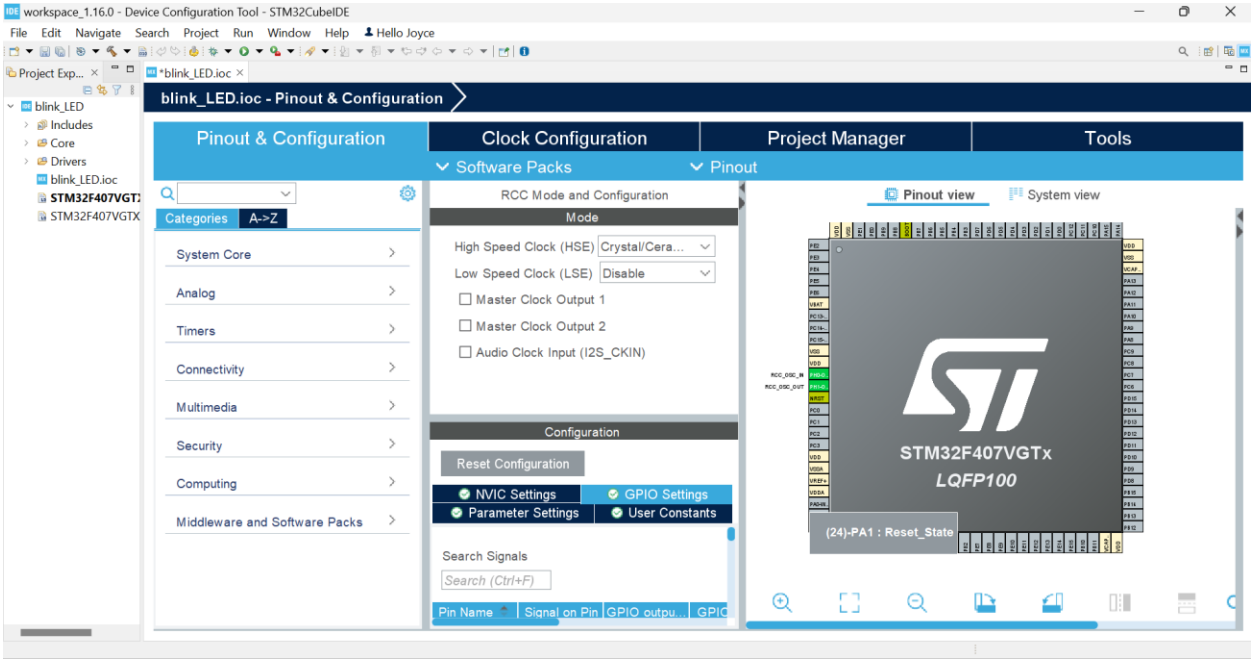
After a few moments, a device configuration file will open, called `blink_LED.ioc`. Pinch if you want to zoom the graphic; double-tap if you want to move it. Under the Pinout dropdown, select Clear Pinouts, so that you will begin with no pin assignment.



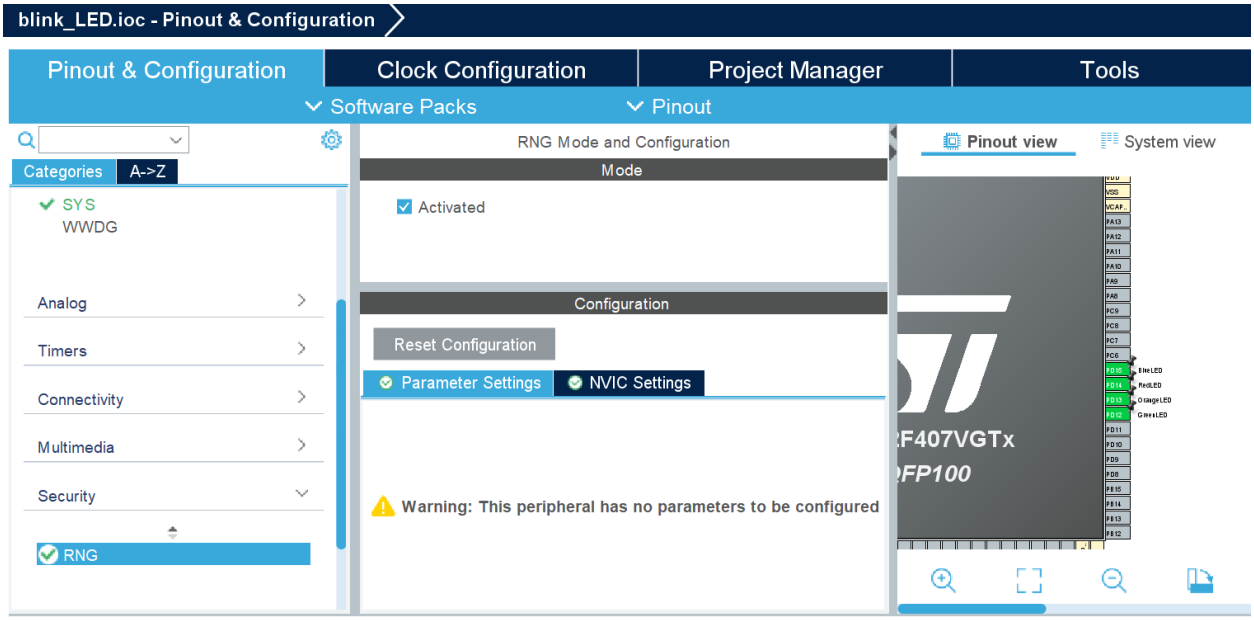


The graphic on the right represents all the pins on the STM32F407 microcontroller. Peripherals can be selected and configured using this tool, and C code for the peripherals will be automatically generated.

Expand the System Core list and click on RCC (reset and clock controller). For the High Speed Clock (HSE), select Crystal/Ceramic Resonator.

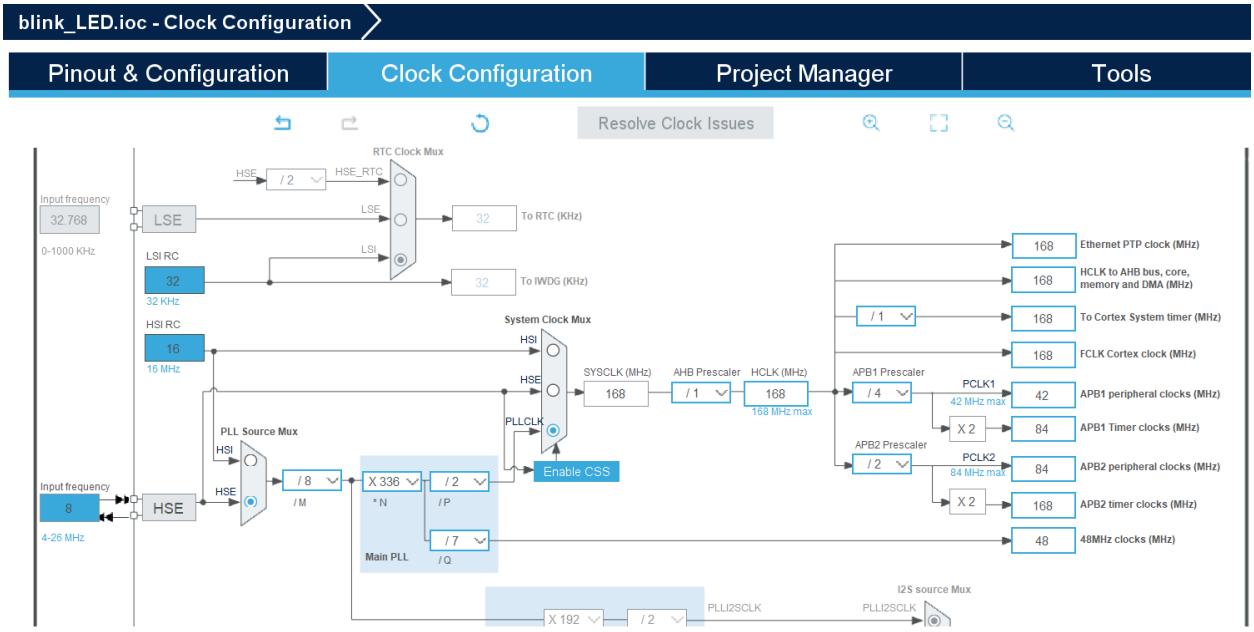


Expand the Security list and click on RNG. Check off Activated for RNG Mode. This is one way to create a clocking pulse.



Click on the Clock Configuration tab. The maximum clock frequency for the STM32F407 is 168 MHz. In the HCLK box, type 168. Select the HSE radio button under PLL Source Mux. Verify dividers for PLL in the Main PLL box. If you receive any warnings, allow the Clock Wizard to find a resolution.

The final clock solution should look like:

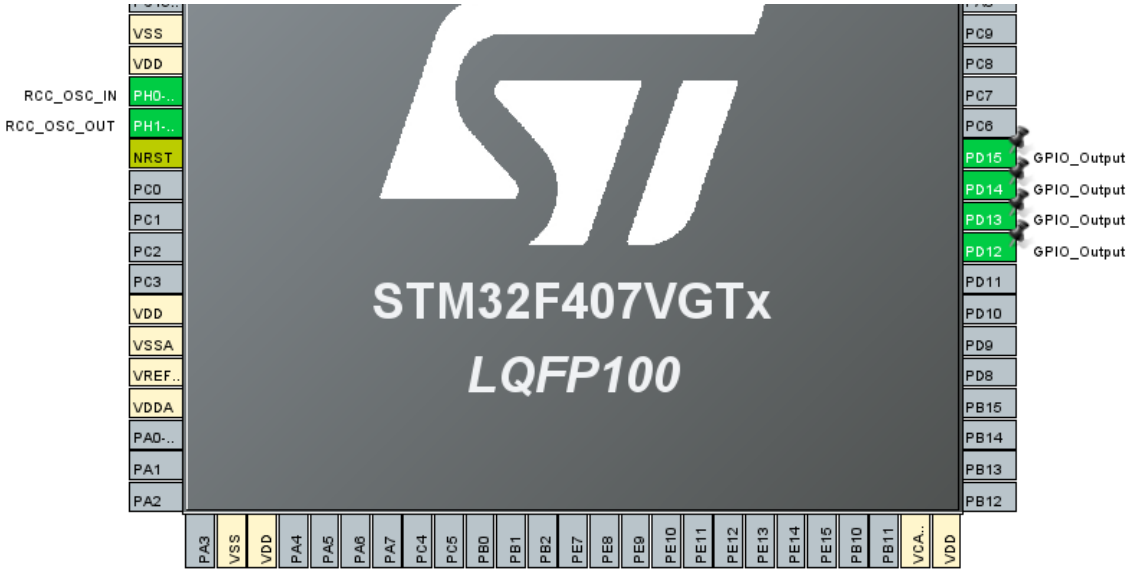


Click on the Pinout & Configuration tab.

The STM32F407 Discovery Kit has four user LEDs. Section 6.3 of the User Manual for the STM32F407 Discovery Kit states that the user LEDs are connected as follows:

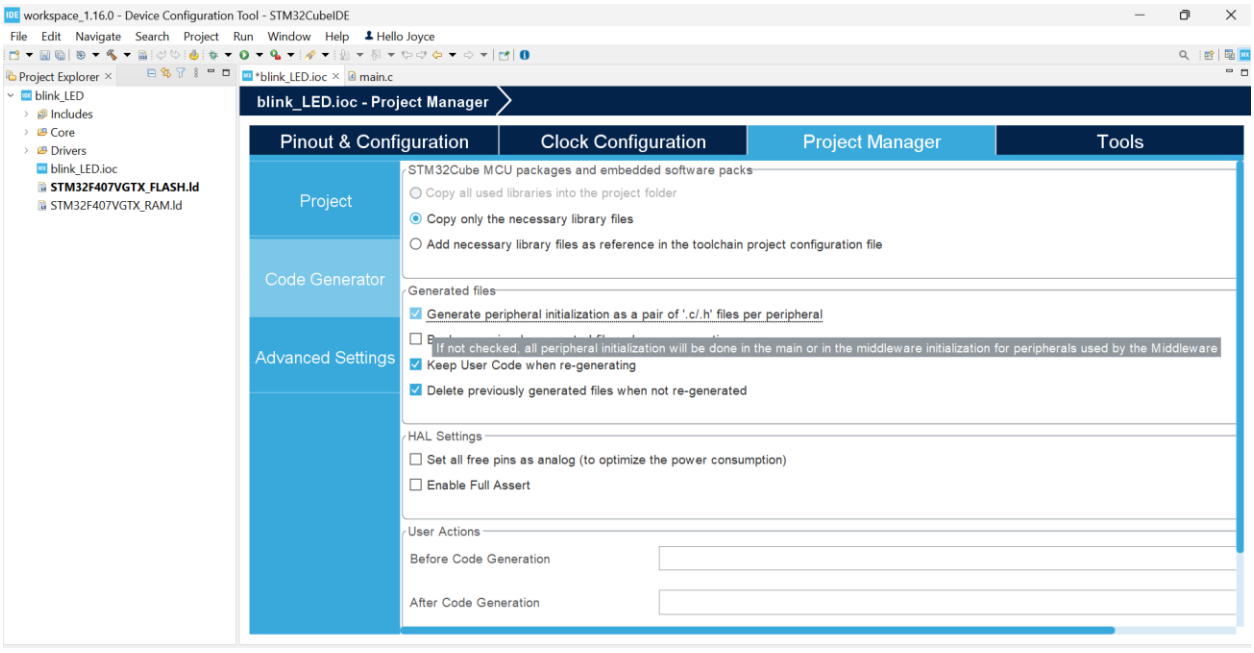
GreenLED connected to PD12
OrangeLED connected to PD13
RedLED connected to PD14
BlueLED connected to PD15

For each of these four pins, select GPIO-Output.

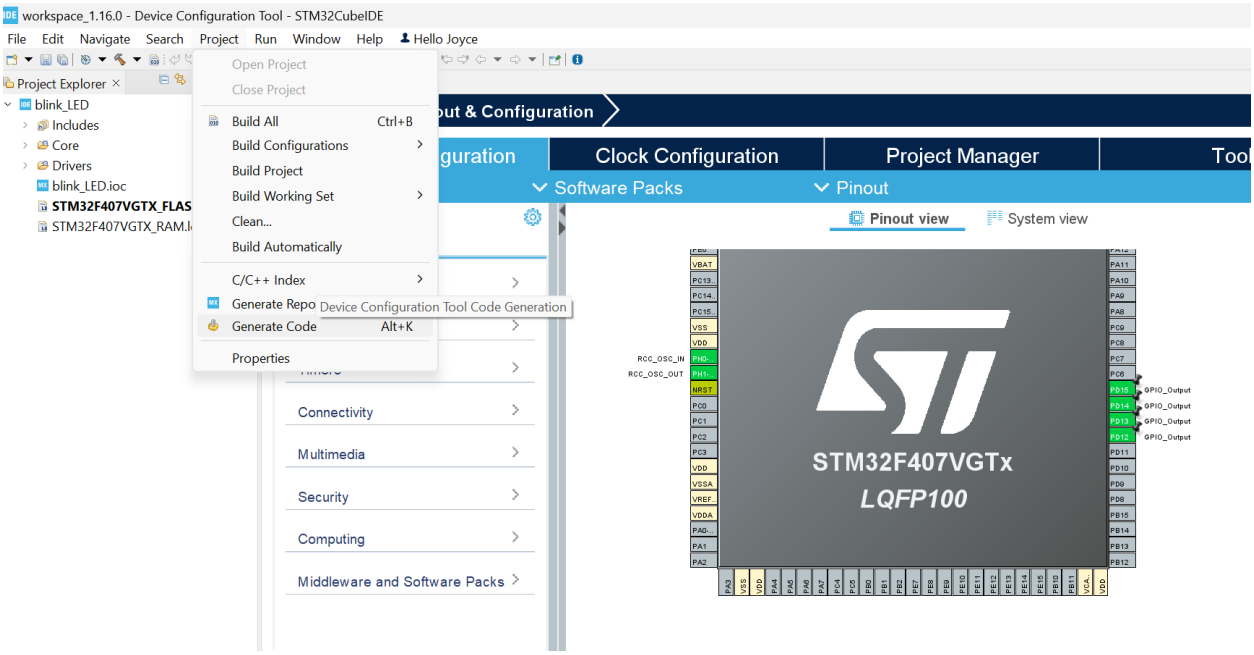


Right click on PD12, PD13, PD14, and PD15 and choose Enter User Label. Enter labels GreenLED, OrangeLED, RedLED, and BlueLED. These user-defined labels replace the default labels LD4, LD3, LD5, and LD6.

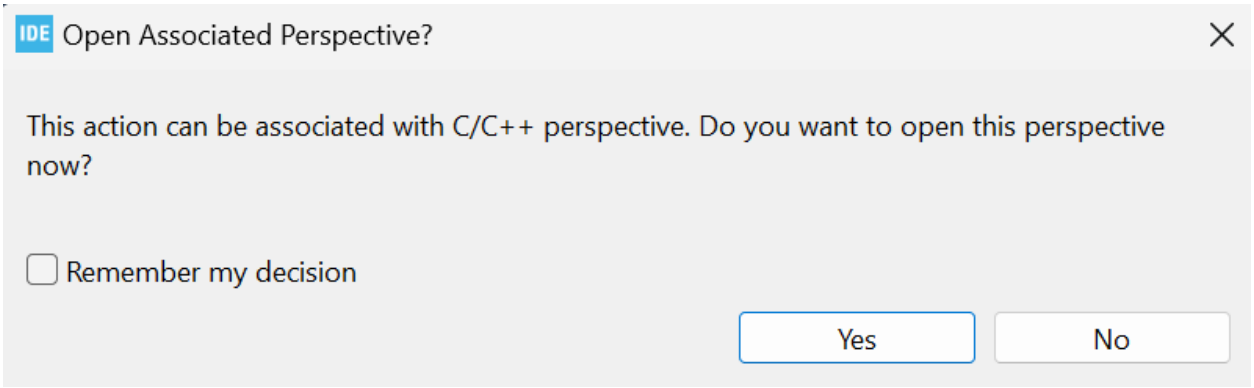
Click on the Project Manager tab. Click Code Generator. Check off “Generate peripheral initialization as a pair of ‘.c/.h’ files per peripheral.”



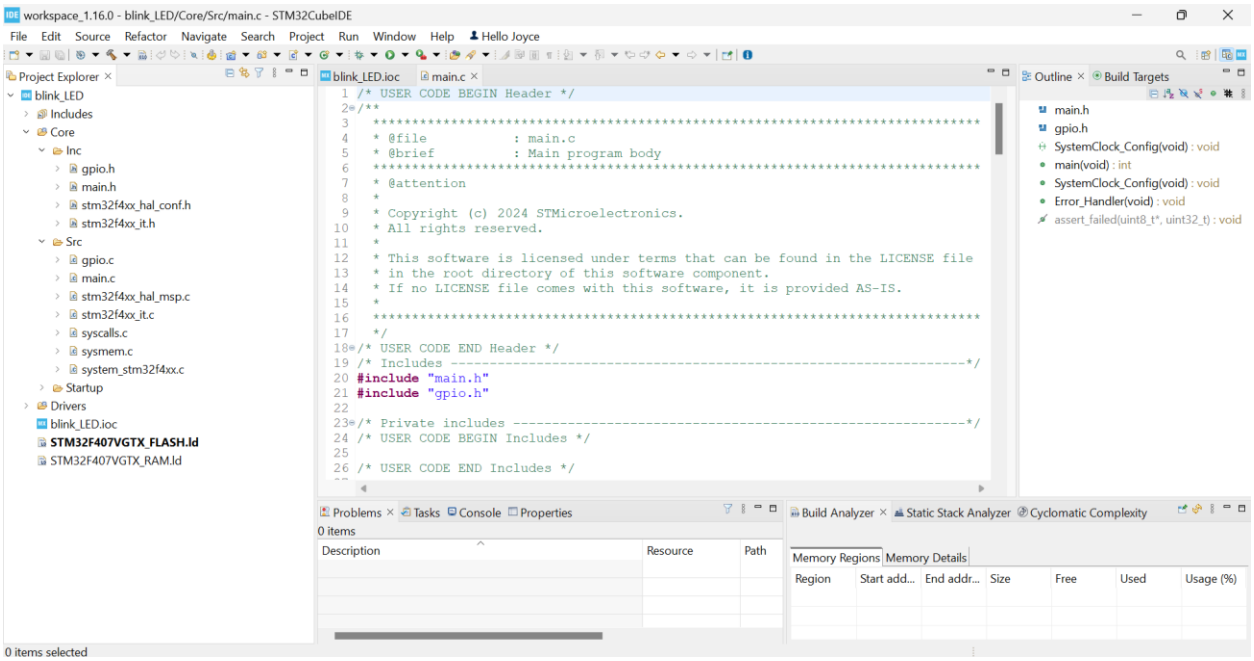
Select Project – Generate Code, or click the icon  in the toolbar.



Agree to open the C/C++ perspective. The device configuration tool will generate code according to the .ioc file.



In the left pane, you will find the Project Explorer. Double-click on the Core folder, and then on its subfolders Inc and Src. Notice the files `gpio.h` and `gpio.c`, created by the user LED peripherals. All of your project files are located on your drive in the workspace directory you selected.



If you double-click on `main.c`, it will open in the main editing window. All of the initializations for the clock and the user LEDs are taken care of.

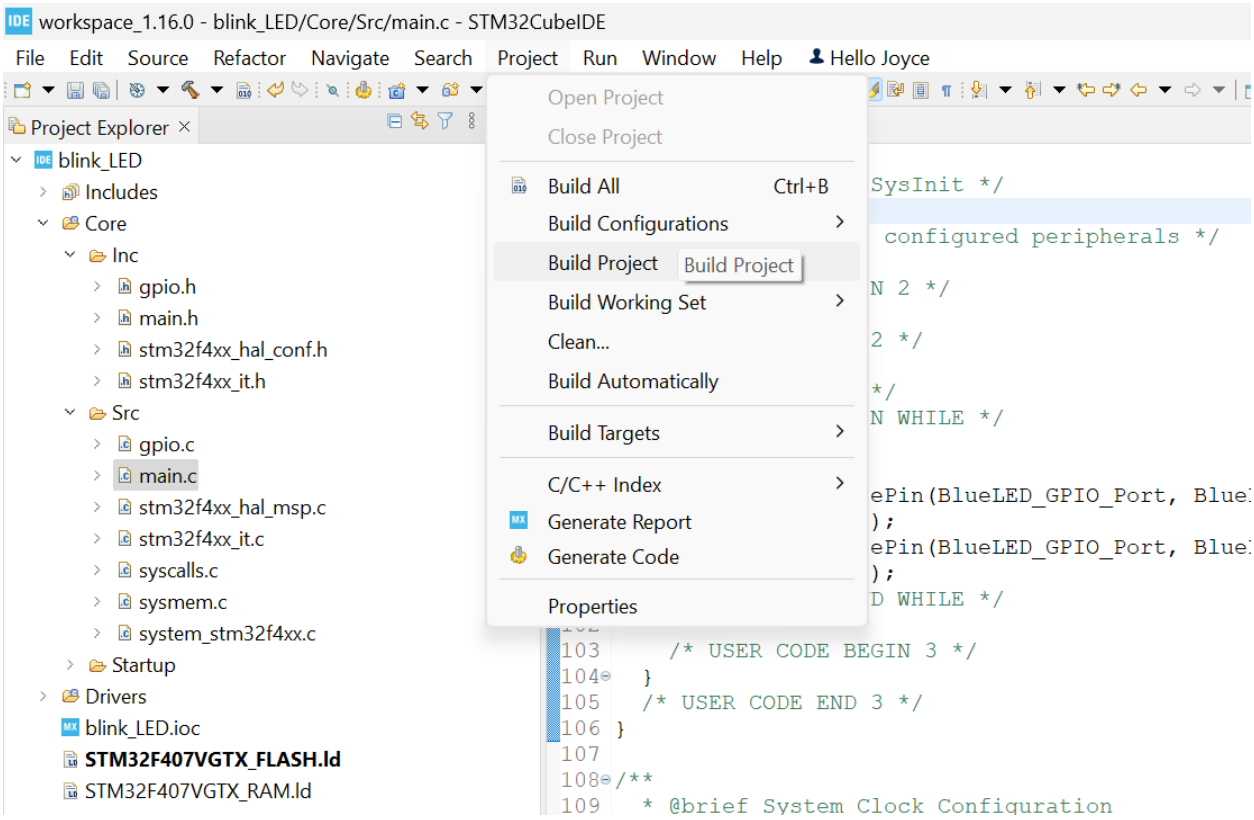
Important: When you add code of your own, you must add it in a legitimate `USER CODE` section. If you do not, your code will disappear when you Generate Code.

Add the following lines to the `while(1)` loop in `main.c`:

```
93   /* Infinite loop */
94   /* USER CODE BEGIN WHILE */
95   while (1)
96   {
97       HAL_GPIO_WritePin(BlueLED_GPIO_Port, BlueLED_Pin, GPIO_PIN_SET);
98       HAL_Delay(500);
99       HAL_GPIO_WritePin(BlueLED_GPIO_Port, BlueLED_Pin, GPIO_PIN_RESET);
100      HAL_Delay(500);
101    /* USER CODE END WHILE */
102
103    /* USER CODE BEGIN 3 */
104  }
105  /* USER CODE END 3 */
```

Tip: If you start typing a command and type Ctrl + Space, you will see a listing of commands that fit. This is useful if you cannot remember the function name or the function arguments exactly.

To compile your program, choose Project – Build Project, or click the hammer icon in the toolbar.



Verify that your project compiles with 0 errors and 0 warnings.

Tip: If the windows seem to reorganize, choose Window – Perspective – Reset Perspective...

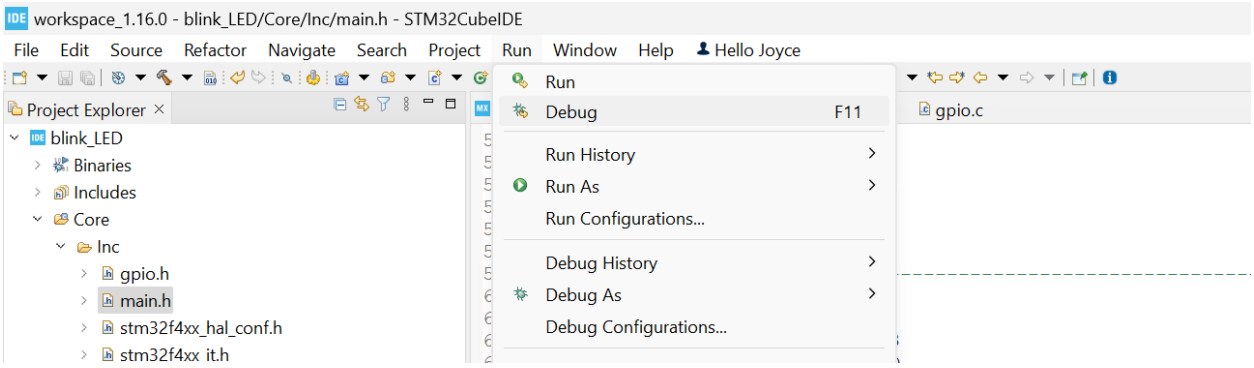
If you open `main.h` from the `Core/Inc`, you will see your user labels for the four LEDs, which are referenced in your `while(1)` code. The `HAL_Delay` function argument is a delay in msec.

In `gpio.c` (found in `Core/Src`), you will find in the `void MX_GPIO_Init(void)` function the line that starts the program with all LEDs off:

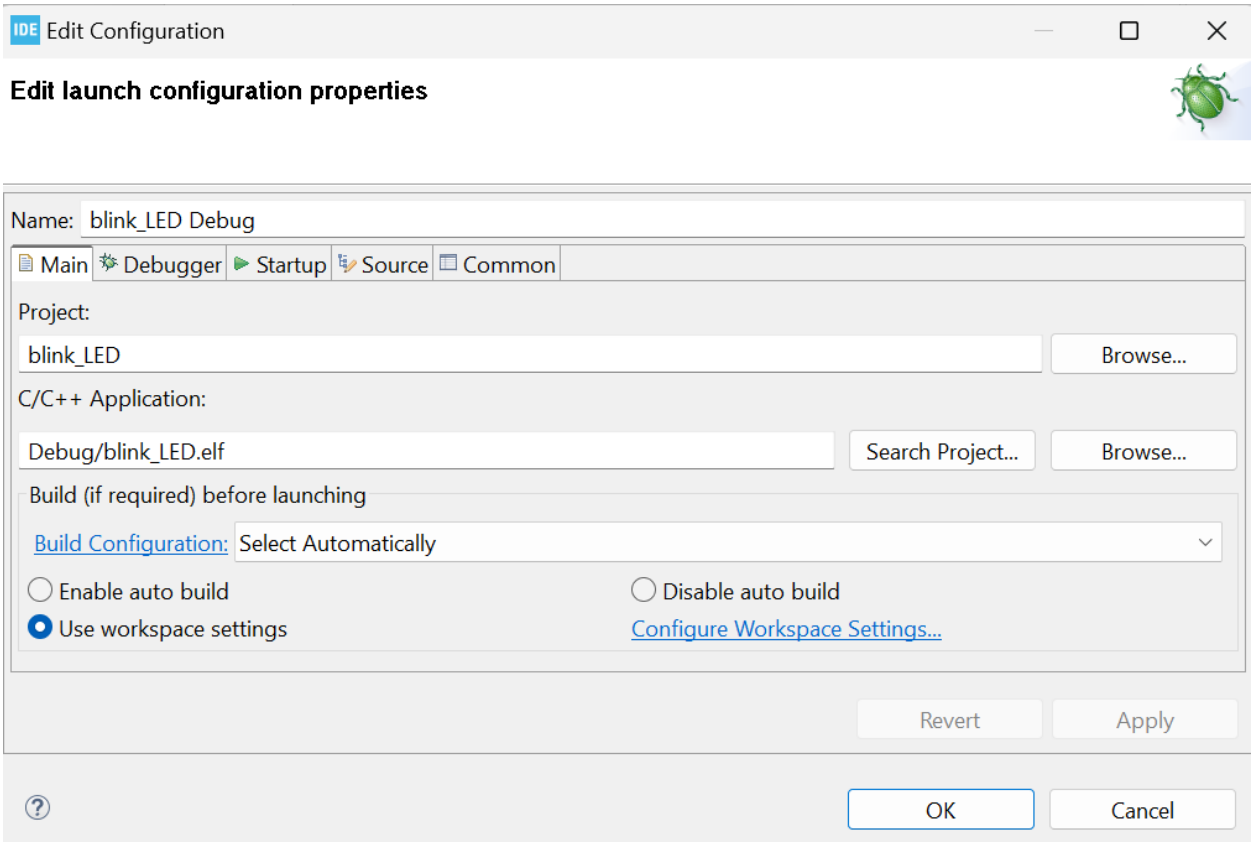
```
HAL_GPIO_WritePin(GPIOD,
GreenLED_Pin|OrangeLED_Pin|RedLED_Pin|BlueLED_Pin, GPIO_PIN_RESET);
```

Connect a mini-USB to USB cable between the STM32F407 Discovery board and your PC.

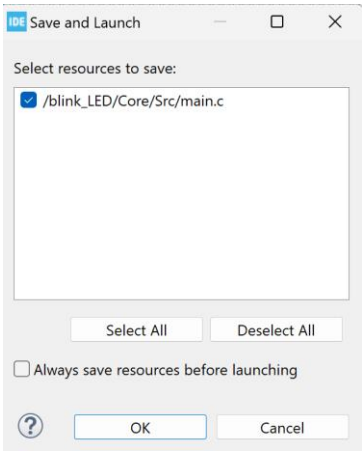
Choose Run – Debug, or click the bug icon in the toolbar.



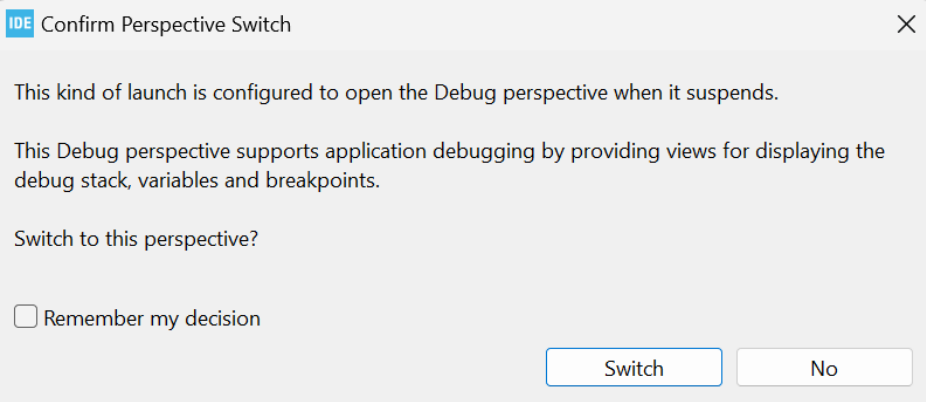
The first time you debug a project you will see an Edit Configuration window. Click OK.



Every time you make a change to your code you will be prompted to save:

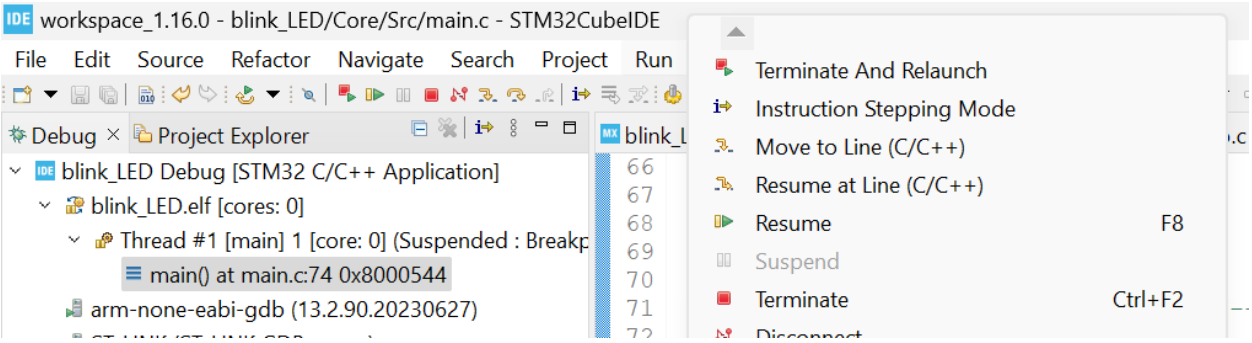


Confirm a Perspective Switch.



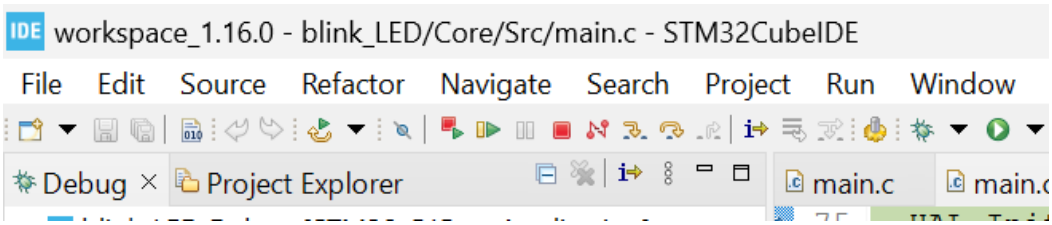
You will see a “Download in Progress,” “Verifying,” and finally, “Download verified successfully.”

Click the Play – Resume button on the toolbar, or choose Run – Resume.



Verify that you see the blue LED blinking on and off every 0.5 sec.

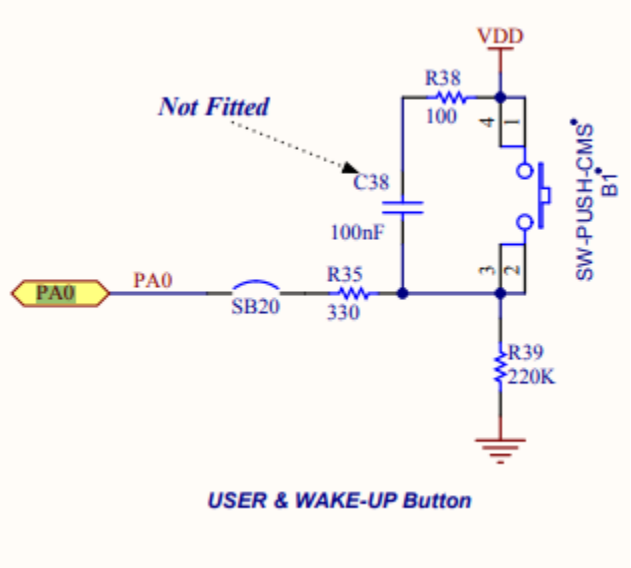
When you want to stop debugging, click on the red square.



Challenge: Can you make the four LEDs light up in turn for 0.25 sec each, in a clockwise direction?

Second project: `user_button`

The STM32F407 Discovery board has a blue user button, connected to pin PA0. As the schematic for the board shows, the button has a pull-down resistor. Thus, when the button is not pressed, PA0 is low; when the button is pressed, PA0 is high.

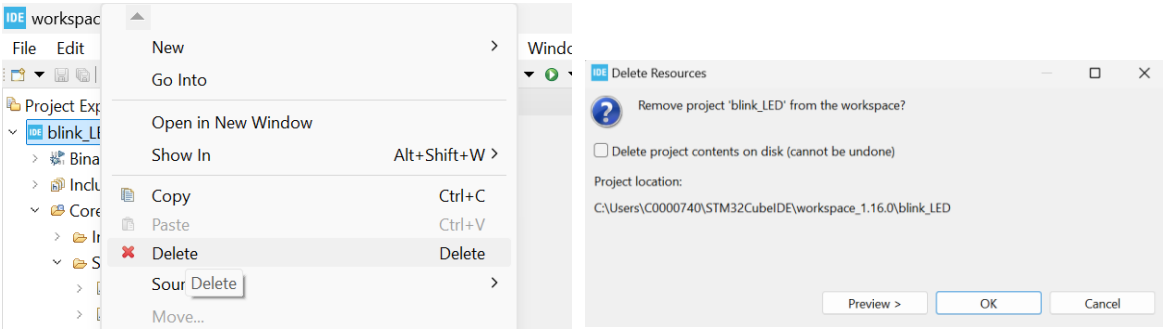


In this project, you will use the user button to change the behaviour of the LEDs.

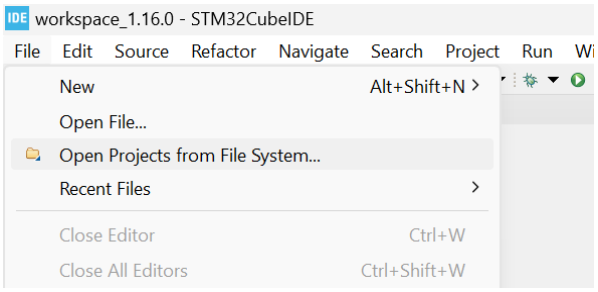
It is possible to begin with a blank new project, as you did for the `blink_LED` project, but for this project it will be more convenient to work from your existing project. This technique can be used whenever you begin a project that has similarities with another.

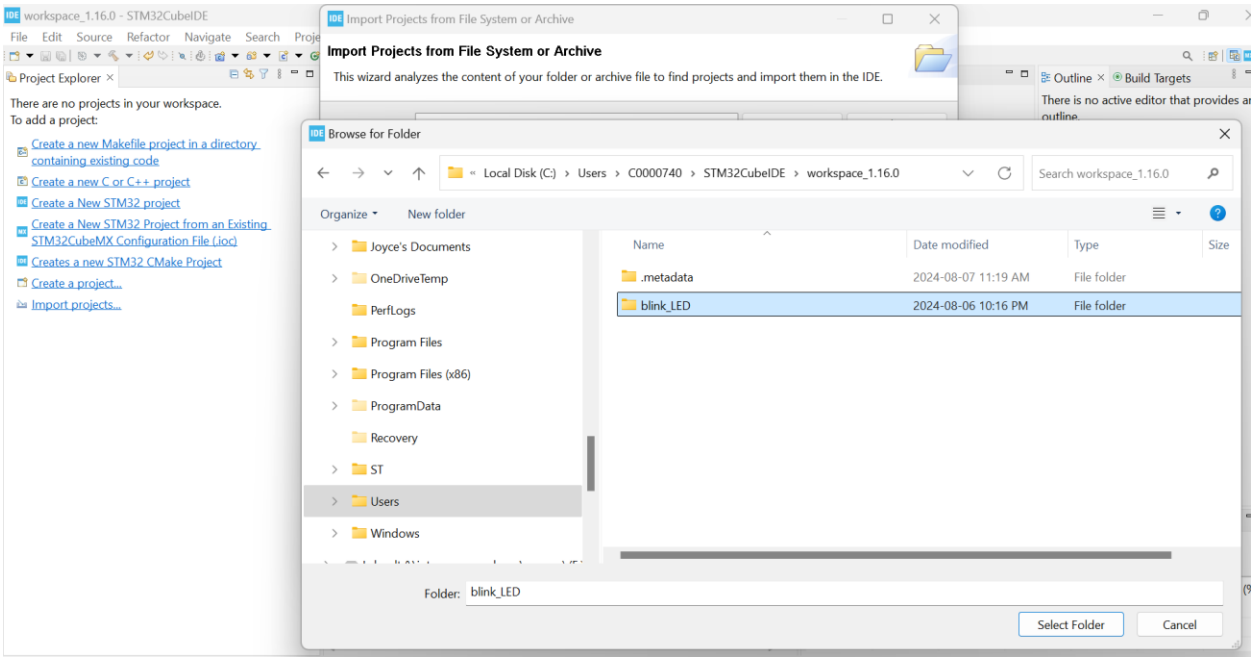
Your `blink_LED` project can remain open in your project explorer. It will be handy to have it available to borrow code. Double-clicking on the project name will compress or expand the project file listing.

If you wish to remove a project from the Project Explorer, you can right click on the project name and choose Delete. Click OK (do not delete project contents).

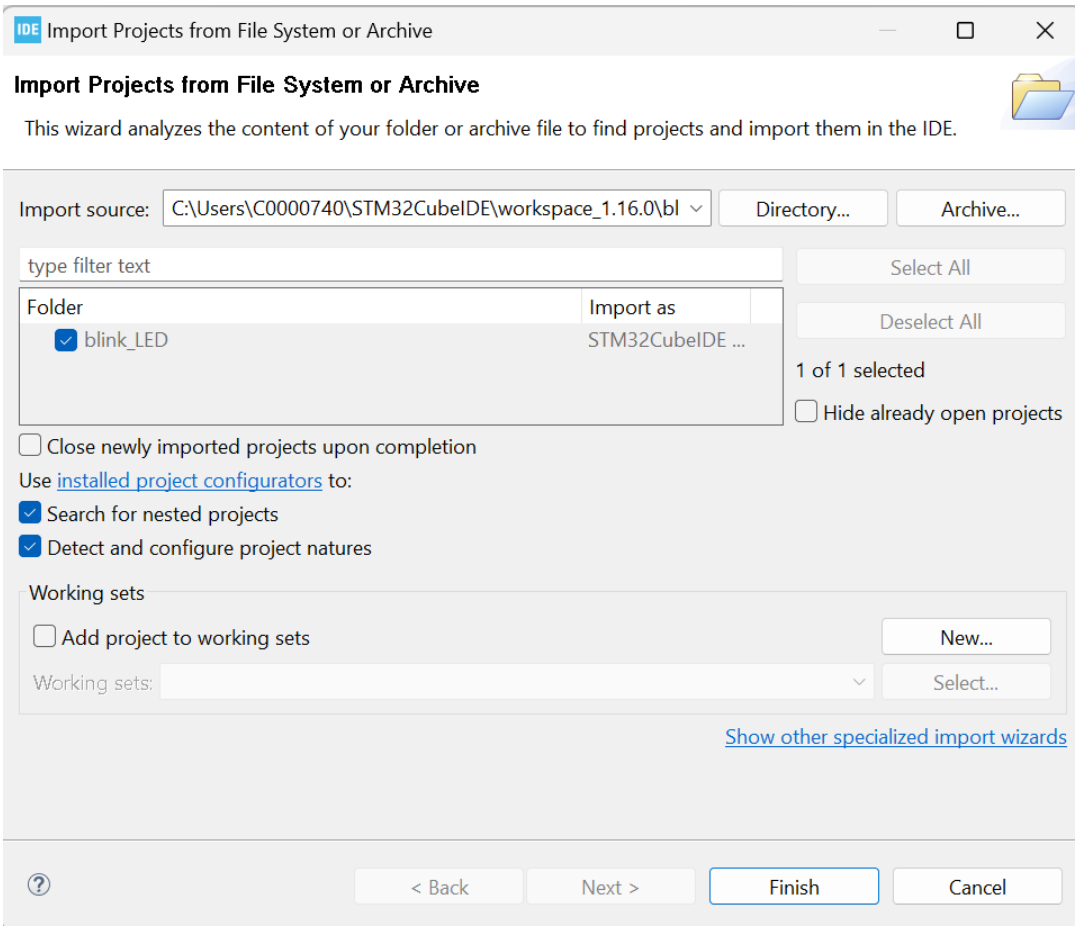


To open any existing project, choose File – Open Projects from File System... and browse for the project folder.

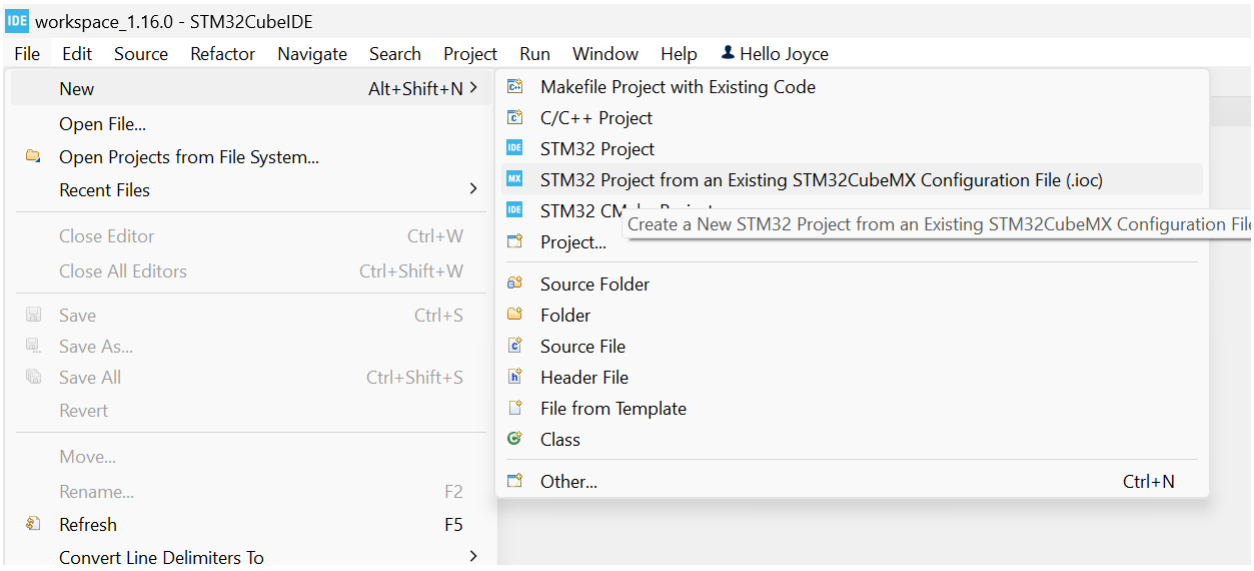




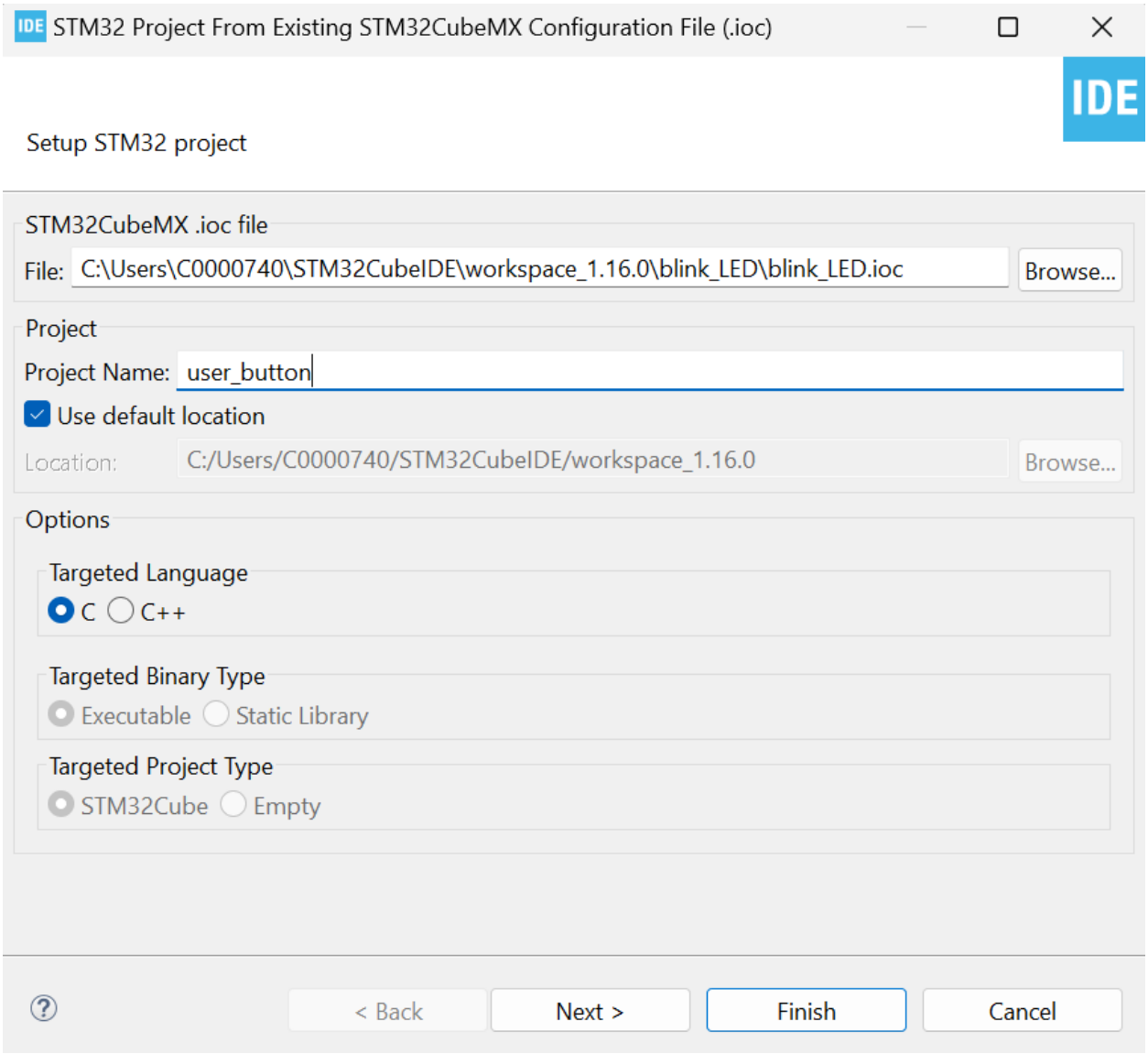
Once the project has been located, click Finish.



To begin the `user_button` project, choose File – New – STM32 Project from an Existing STM32CubeMX Configuration file (.ioc)



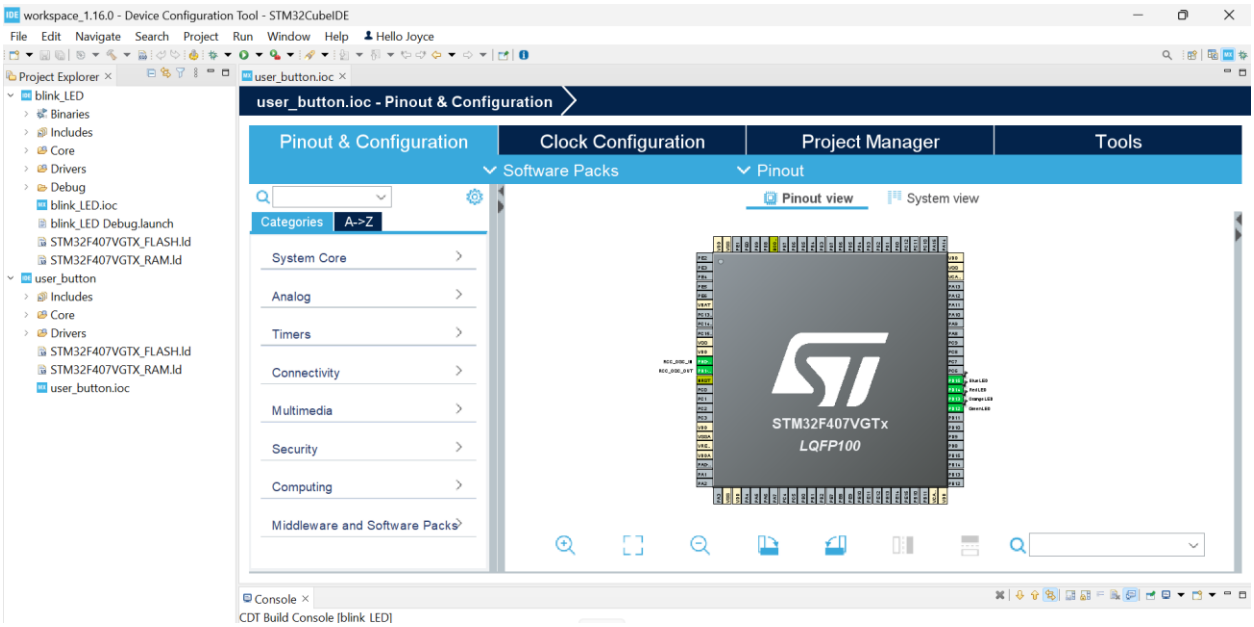
Browse to find `blink_LED.ioc`, the `.ioc` file you wish to copy and choose a new project name, `user_button`. Click Finish.



Once the project has been created, its folder will appear in the Project Explorer at the left, and its `user_button.ioc` configuration file will be open. Notice the green pins for RCC and the four user LEDs, which were configured for the `blink_LED` project.

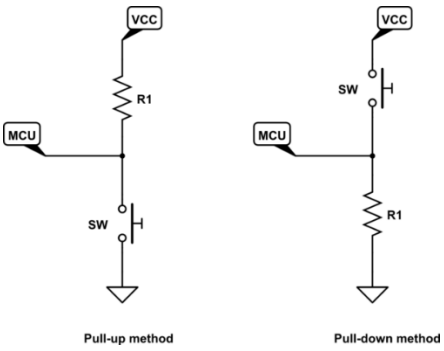
Note that it is not possible to simply copy and paste a project and give it a new name, as the files in the project folder will retain the original project name.

Tip: If too many files are open in the editor window of STM32CubeIDE, Ctrl + Shift + W will close all the open windows.

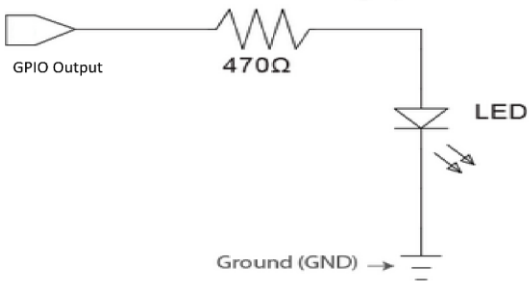


Since the blue user button is connected to pin PA0, click on this pin and choose GPIO_Input. Right click on the pin to enter the user label `UserButton`.

Note that other pins on the STM32F407 can act as GPIO inputs or outputs. In this project you are using the button and LEDs on the board, but external buttons or LEDs can instead be connected to the GPIO pins on the board, if desired. Sample circuits are shown. Most pins on the STM32F407 have multiple possible functions. Some settings will preclude others. If you make conflicting assignments, the pins in question will turn red.



Pushbutton circuits

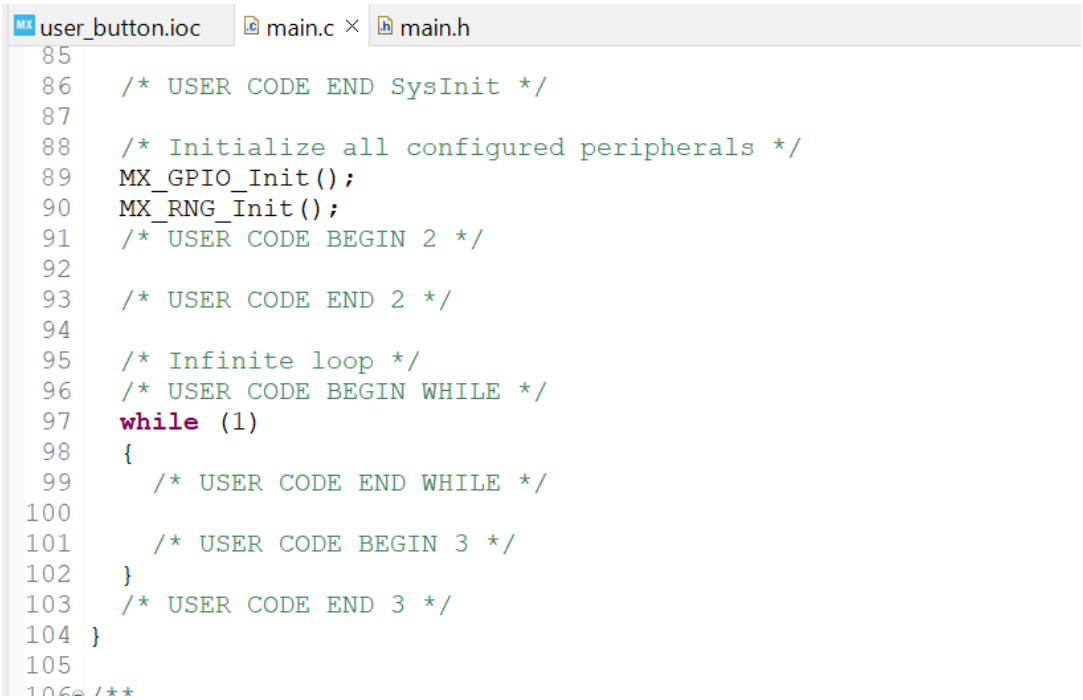


LED circuit

Generate code for the `user_button` project, and switch perspective.

Open `main.h` from `Core/Inc`. Notice that there are now entries for `UserButton`. You can also have a look at `stm32f4xx_hal_conf.h`, which indicates which modules are enabled in the project, among other things.

Although `user_button` was based on the `blink_LED` project, only the device configuration is the same. At the start, the `while(1)` loop is entirely empty; C code must be added to `user_button`.

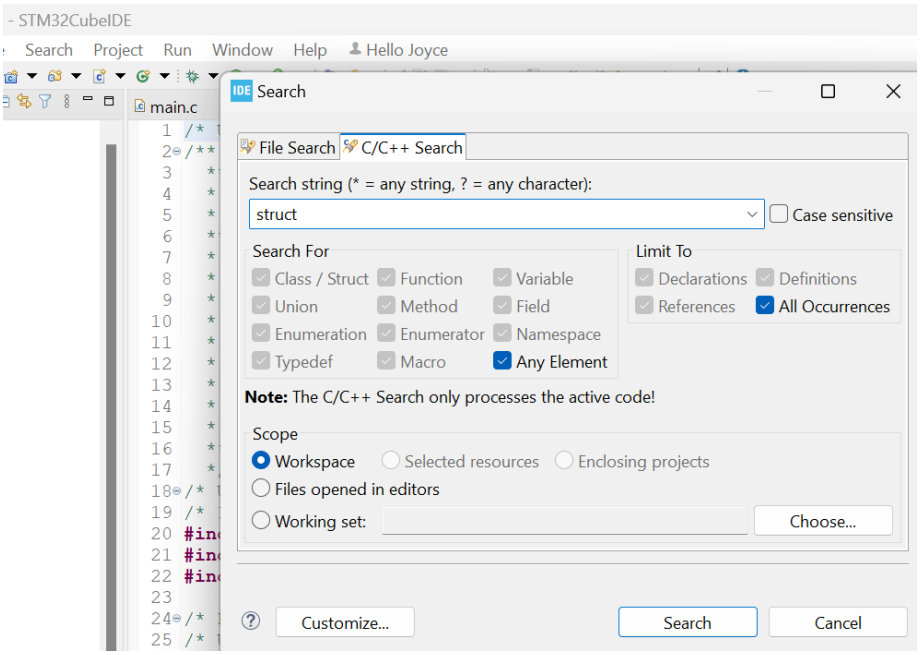


Open `main.c` from your `blink_LED` project. Copy your code from the first project into the `while(1)` loop of your `user_button` project. Remember to paste your code into a legal `USER CODE` area.

Compile your code to verify it is error-free.

Tip: When multiple projects are open, make sure you click on the project name in the Project Explorer to be sure you are compiling your current project.

Tip: You can search for text in the entire workspace or an individual project. To have the option of limiting your search to one project, first click on the name of the project in the Project Explorer.



Run – Debug, switch perspective, and when download is successfully verified, Play – Resume. Verify that your project behaves as you expect.

The state of the user button may be read using the command:

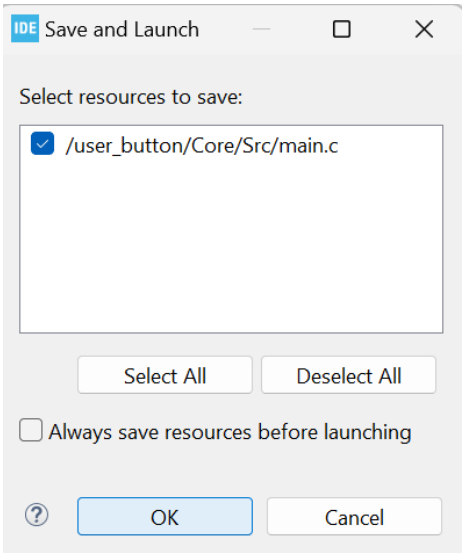
```
HAL_GPIO_ReadPin(GPIOx, GPIO_Pin)
```

where `GPIOx` refers to the port for the user button and `GPIO_Pin` refers to the pin for the user button. Refer to `main.h` to find the correct port and pin. Studying the way you set and reset LEDs will be helpful. As with the user LEDs, the user button is either `GPIO_PIN_SET` or `GPIO_PIN_RESET`.

Tip: Hovering over a variable, variable type, or function will pop up details.

Tip: If you right-click on a variable, variable type, or function, and choose Open Declaration, you will jump to the place in the project where the declaration occurs.

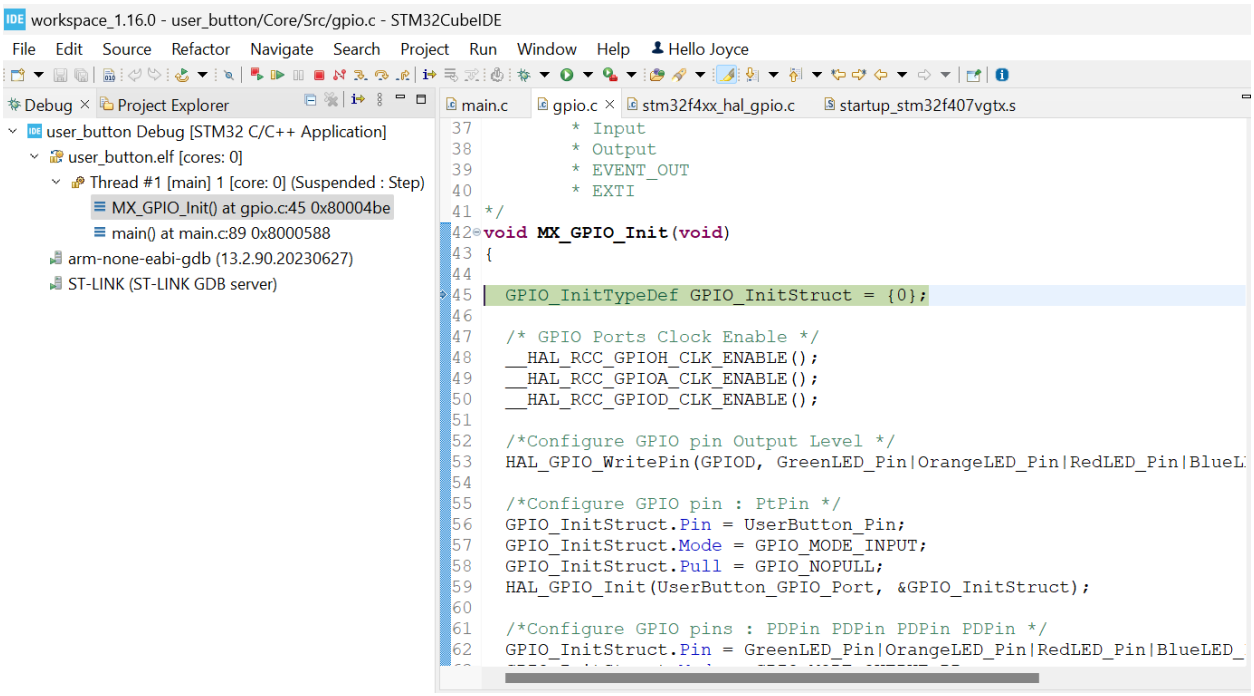
Tip: Whenever you choose Run – Debug with unsaved changes to a project file, you will be prompted to save.



Challenge: Modify your code so that the four user LEDs light in a clockwise direction when the blue user button is pressed, and in a counterclockwise direction when the blue user button is not pressed.

Experiment with some debugging techniques: After choosing Run – Debug for your program, do not click Play – Resume. Instead, notice that the current instructor pointer is pointing at the first line of code inside `main()`. If you choose Step Over, you will jump to the next function call. If you choose Step Into, you will dive into the function.

For example, if you Step Into `MX_GPIO_Init();`, you will see:



You can step through the function, or choose Step Return to return to the calling function.

Double-clicking in the blue vertical bar will place a breakpoint on a line. Clicking Play – Resume will cause the program to execute until the breakpoint instruction is reached. From there, you can step through code from the breakpoint onwards, or you can click Play – Resume again to complete one loop of the program and return to the breakpoint again.

You can create many breakpoints. To remove a breakpoint double-click on it. Other breakpoint options are available in the Run menu.

Third project: `adc_poll`

Create a new STM32 project called `adc_poll` that is based on the existing configuration file `user_button.ioc`.

Generate code and compile.

The ability to print variable values is very helpful for debugging. The steps needed to add print capability to an STM32 project are provided in Appendix 1.

Challenge: Follow the directions in Appendix 1. Add the following lines to your `while(1)` loop to verify print is working.

```
 95   /* Infinite loop */
 96   /* USER CODE BEGIN WHILE */
 97   while (1)
 98   {
 99       printf("testing\n");
100       HAL_Delay(500);
101     /* USER CODE END WHILE */
102
103     /* USER CODE BEGIN 3 */
104   }
```

When you are satisfied print is working, feel free to delete the two lines of test code.

In the `adc_poll` project, you will configure an ADC on the STM32F407 to read in voltages from an external source. This external source could be, for example: a function generator, the voltage from a variable voltage divider circuit, a thermistor, and so on.

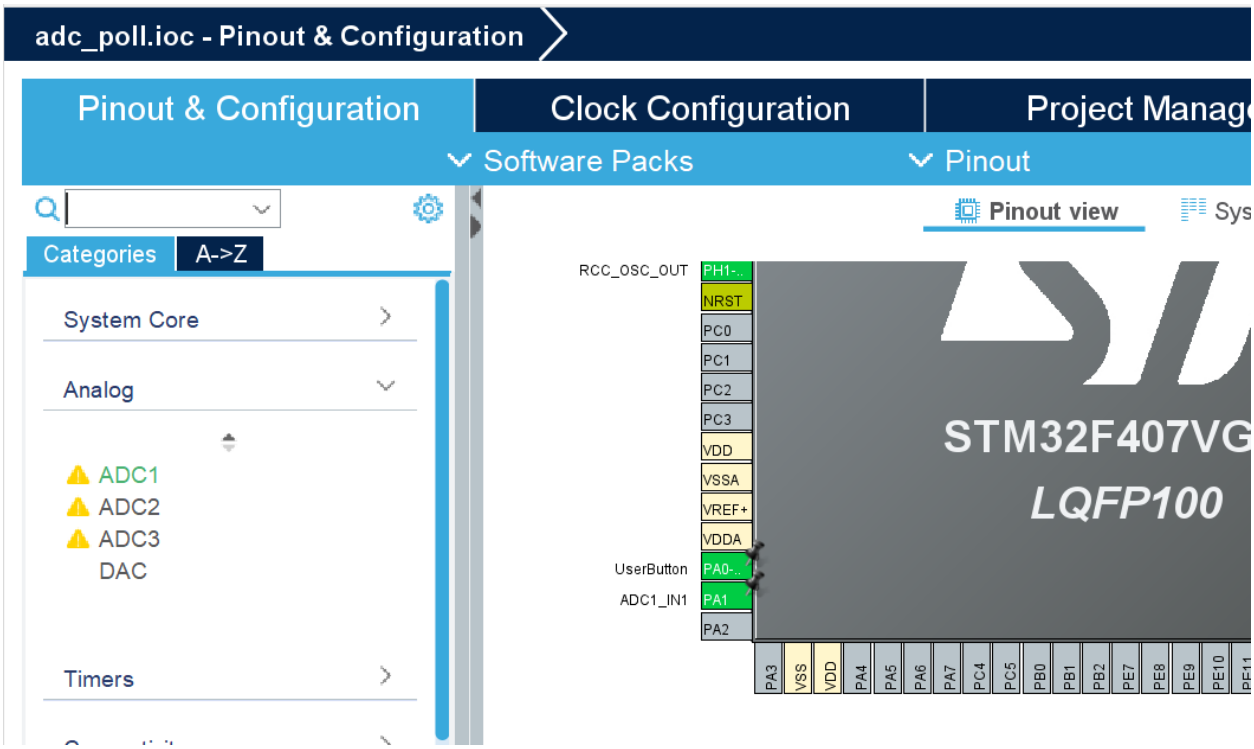
In this project you will generate a sine wave from a function generator, read voltage values from an ADC, print voltage levels, and light LEDs to signal voltage levels.

Open `adc_poll.ioc`.

Three ADCs are available on the STM32F407, each with 16 channels that can work at the same time. Click on pins to find where the ADC channels are available. For example, pin PC1 offers channel 10 of ADC1, ADC2, and ADC3.

For this project, you will use ADC1 channel 1 to read voltages from the function generator, but many other options would work equally well. Find the appropriate pin for channel 1 of ADC1 and select `ADC1_IN1`. If you wish you can enter a user label for this pin.

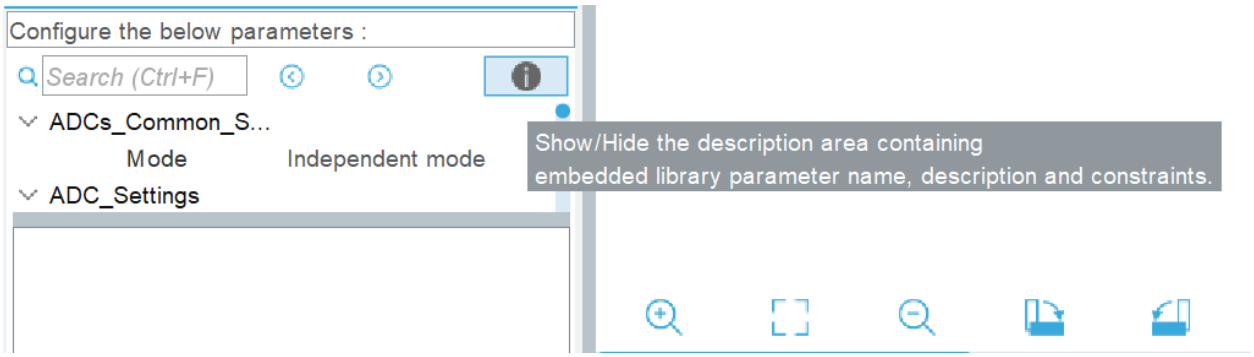
Click on the Analog category. Notice that ADC1 is highlighted in green. The yellow triangles alert you that some selections may conflict, but they are not errors.



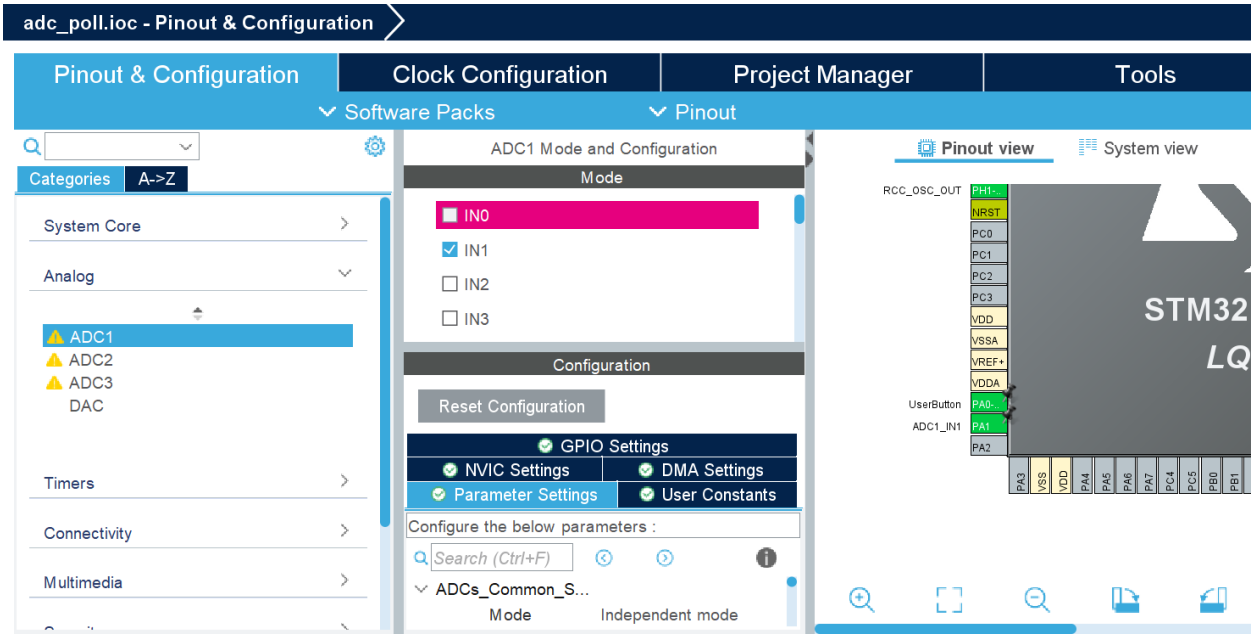
Click on ADC1. Under Mode, you will see IN1 checked off, to show channel 1 will be active. Below Mode is the Configuration area.

Tip: You can drag the Configuration area larger.

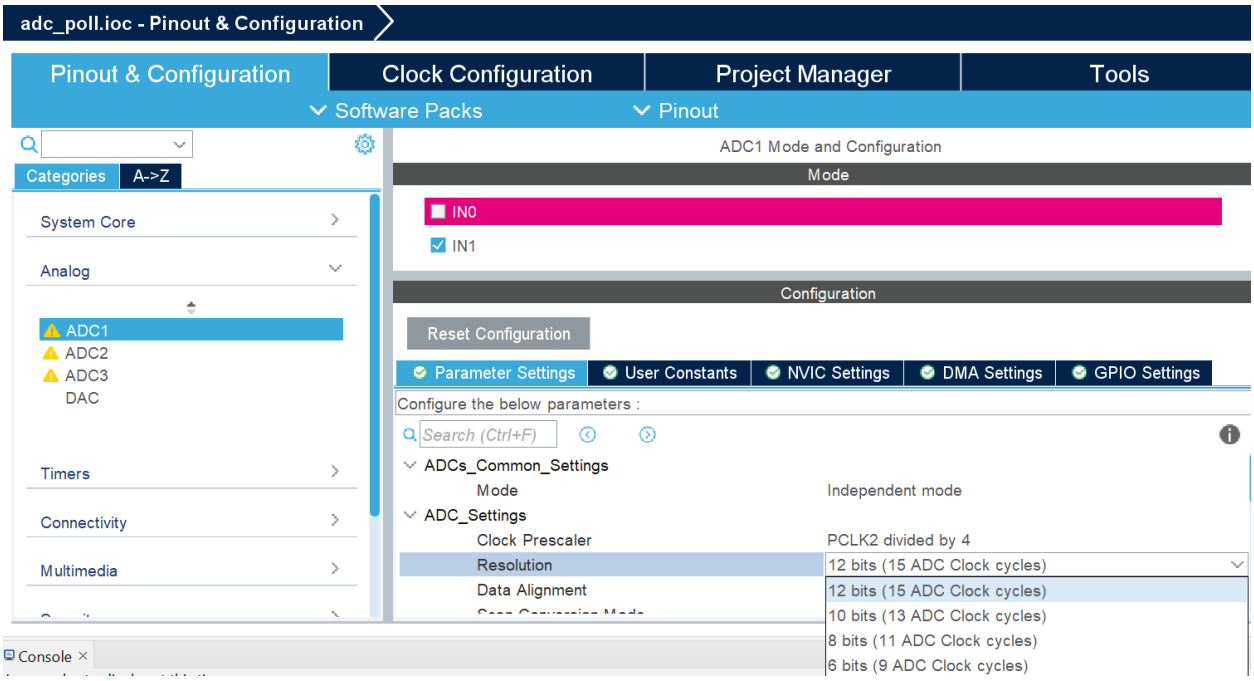
Tip: Unselect the i (information) button if it is selected so that the information window does not **block your view of the Configuration area.**



Tip: The small black arrows in the grey border on the right edge of the Mode and Configuration window can be used to expand and compress the window.



Click on the Parameter Settings tab. Note that Resolution can be set from 6 to 12 bits. Fewer bits mean lower accuracy but faster performance. In brackets you see the number of clock cycles need to perform each analog-to-digital conversion. You can use 12-bit resolution for this project.



Browse the other parameter settings and tabs. No other parameters need to be modified for this project.

Now that you have configured an ADC, generate code for your project.

Section 7.2.2 of the HAL User Manual describes the three modes of operation for ADC:

Execution of ADC conversions

1. ADC driver can be used among three modes: polling, interruption, transfer by DMA.

Polling mode IO operation

- Start the ADC peripheral using HAL_ADC_Start()
- Wait for end of conversion using HAL_ADC_PollForConversion(), at this stage user can specify the value of timeout according to his end application
- To read the ADC converted values, use the HAL_ADC_GetValue() function.
- Stop the ADC peripheral using HAL_ADC_Stop()

Interrupt mode IO operation

- Start the ADC peripheral using HAL_ADC_Start_IT()
- Use HAL_ADC_IRQHandler() called under ADC_IRQHandler() Interrupt subroutine
- At ADC end of conversion HAL_ADC_ConvCpltCallback() function is executed and user can add his own code by customization of function pointer HAL_ADC_ConvCpltCallback
- In case of ADC Error, HAL_ADC_ErrorCallback() function is executed and user can add his own code by customization of function pointer HAL_ADC_ErrorCallback
- Stop the ADC peripheral using HAL_ADC_Stop_IT()

DMA mode IO operation

- Start the ADC peripheral using HAL_ADC_Start_DMA(), at this stage the user specify the length of data to be transferred at each end of conversion
- At The end of data transfer by HAL_ADC_ConvCpltCallback() function is executed and user can add his own code by customization of function pointer HAL_ADC_ConvCpltCallback
- In case of transfer Error, HAL_ADC_ErrorCallback() function is executed and user can add his own code by customization of function pointer HAL_ADC_ErrorCallback
- Stop the ADC peripheral using HAL_ADC_Stop_DMA()

For this project, you will use polling mode.

In the Project Explorer, find `adc.c` in `Core/Src`. Near the top of the file you will see the declaration:

```
ADC_HandleTypeDef hadc1;
```

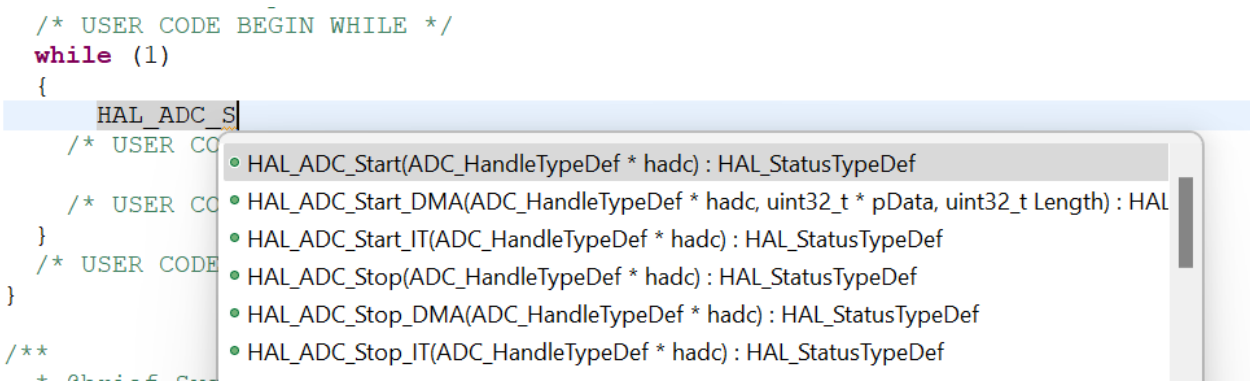
If you hover over `ADC_HandleTypeDef` (or right-click it and choose Open Declaration), you will see the parts of the structure, as well as the types of functions that can operate on this type of structure. The variable name `hadc1` refers to the "handle" for the ADC stream you have defined in your configuration file. In the `MX_ADC1_Init` function that follows you will see `ADC_RESOLUTION_12B` and all the other configuration parameters. A little further down you will see `ADC_CHANNEL_1`. All of this code has been created automatically from the `.ioc` configuration file.

Recall that pin `PA1` connects to `ADC1_IN1`. When you test your program, the function generator output will be connected between pin `PA1` and `GND` on the Discovery board.

Open `main.c` and locate the `while(1)` loop. As suggested in the HAL User Manual, four functions are required for ADC in polling mode: `HAL_ADC_Start()`, `HAL_ADC_PollForConversion()`, `HAL_ADC_GetValue()`, and `HAL_ADC_Stop()`.

In a legal `USER CODE` area in the `while(1)`, start typing `HAL_ADC_Start`. Part-way through, click Ctrl + Space to see functions that fit was you are typing. Double-click a suggestion to accept it. `HAL_ADC_Start` has just one argument: the handle for your ADC stream. Since a pointer to an `ADC_HandleTypeDef` is required, finish typing:

```
HAL_ADC_Start(&hadc1);
```



Next, type `HAL_ADC_PollforConversion`. This function requires two arguments: a pointer to `ADC_HandleTypeDef`, and a timeout value. The timeout value is an integer in msec. If no ADC is completed before the timeout elapses, the function returns. This ensures your program does not hang indefinitely. A timeout of about 200 msec is reasonable.

The next call is to the function `HAL_ADC_GetValue`. This function has just one argument, the pointer to the ADC handle, and returns a `uint32_t`. In the `USER CODE` PV (Private Variables) area, declare a global variable `data_in`.

```
uint32_t data_in;
```

Return the ADC value into your input variable.

```
data_in = HAL_ADC_GetValue(&hadc1);
```

Call `HAL_ADC_Stop` with an appropriate argument. Stopping the ADC means it will not waste energy between polls.

With the lines currently in your `while(1)` loop, the ADC will be called at the maximum core clock frequency. Add a `HAL_Delay` of 500 msec to prevent the ADC from being polled too frequently.

Compile your program.

Section 5.1.2 of the STM32F407 indicates that the power supply for the STM3F407 is nominally 3.3 V in a 1.8 V to 3.6 V range. This supply also powers the ADC, so input voltages should lie in the range 0 V to 3 V.

Challenge: Set up a function generator to produce a 3 Vpp, 100 Hz sine wave. View the waveform on an oscilloscope. Verify amplitude and frequency. Add a DC offset until the sine wave lies between 0 V and 3 V.

Add a `printf` statement to your `while(1)` to print `data_in`, recalling that integers can be printed with the print format `%d`. Using a BNC-to-red/black clips cable, connect the sine wave from the function generator between PA1 and ground on the Discovery board. Compile your program, Run – Debug, Play – Resume. Watch the output in the SWV ITM Data Console. Since the ADC has a resolution of 12 bits, the smallest value should be 0 and the largest should be about $2^{12} - 1 = 4095$. Verify that you see the numbers increasing and decreasing with the sine wave. Stop your program if your Console fills up.

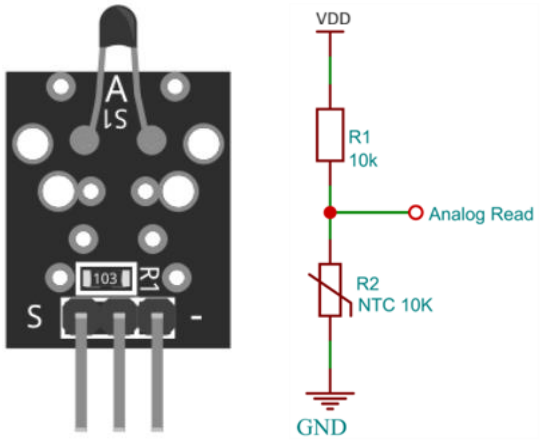
Try increasing or decreasing the frequency of the sine wave, and see the result for the ADC values.

Back at 100 Hz, set the DC offset of your function generator to 0 V. Your ADC values should show 0 for half of each period, because the negative-going portion of your sine wave is lost, and the maximum values should be about half of what they were. This exercise demonstrates the importance of ADC inputs being non-negative. Also, when the range of input values is close to 3 V, the ADC errors will be as small as possible.

Challenge: For 3 Vpp, 100 Hz sine wave input with 1.5 V offset, turn on the green LED when the ADC value exceeds 3200 and turn the green LED off when the ADC values drops below 800. The red LED should do the opposite.

Future options:

For sensor inputs, a conversion is needed between the integer value returned by the ADC and the physical quantity being measured by the sensor. For example, a KY-013 thermistor module can be used as a source for ADC input. This module incorporates a 10 kΩ resistor in series with a thermistor. Pin 1 is connected to the ADC input pin on the Discovery board, pin 2 is connected to 3 V, and pin 3 is connected to ground.



First the resistance of the thermistor R_T by solving the equation:

$$\frac{ADC}{4095} = \frac{R_T}{R_T + R_o}$$

where ADC is the integer reported by the ADC, 4095 is the maximum value returned from a 12-bit ADC, R_T is the resistance of the thermistor and $R_o = 10\ k\Omega$ is the value of the series resistor.

With the thermistor resistance, solve for temperature T in K with the Steinhart equation:

$$\frac{1}{T} = \frac{1}{T_o} + \frac{1}{B}\ln\left(\frac{R_T}{R_o}\right)$$

where T_o is room temperature 298 K and B is the thermistor coefficient with a value of 3950.

These equations may be used to convert an ADC integer into a temperature in program code.

Alternatively, an external microphone board such as the MAX9814 can be used as input to an ADC. Wiring details may be found at https://cdn-learn.adafruit.com/downloads/pdf/adafruit-agc-electret-microphone-amplifier-max9814.pdf.

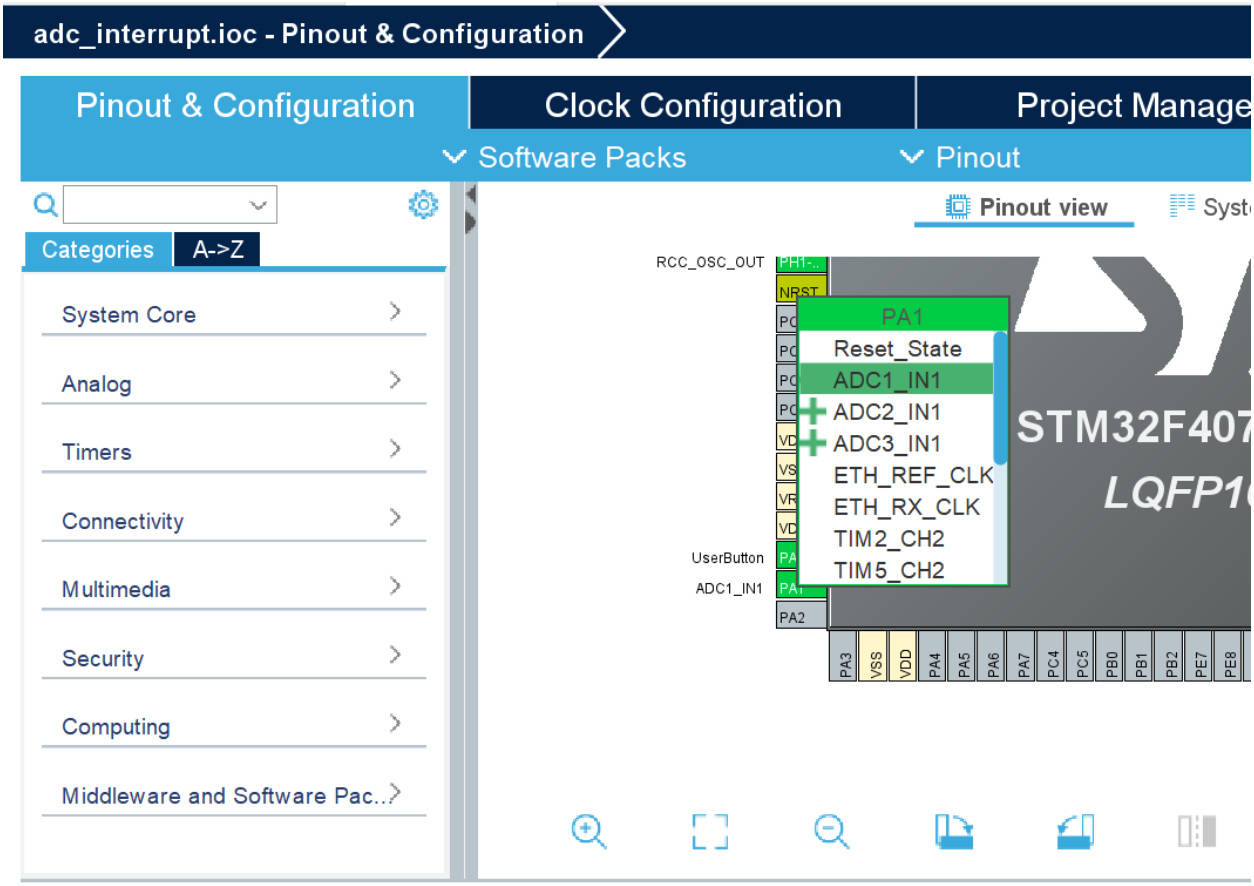
Fourth project: `adc_interrupt`

The STM32F407 chip has an internal temperature sensor, as may be seen in the block diagram in Section 3 of the STM32F407 datasheet. This sensor monitors the temperature of the chip. The absolute temperate it reports may not be consistent from chip to chip, but the change in chip temperature is very accurate. If the chip is working in an enclosure on a hot day, an increase in temperature might indicate a fan should be turned on, or the clock frequency should be reduced, for example.

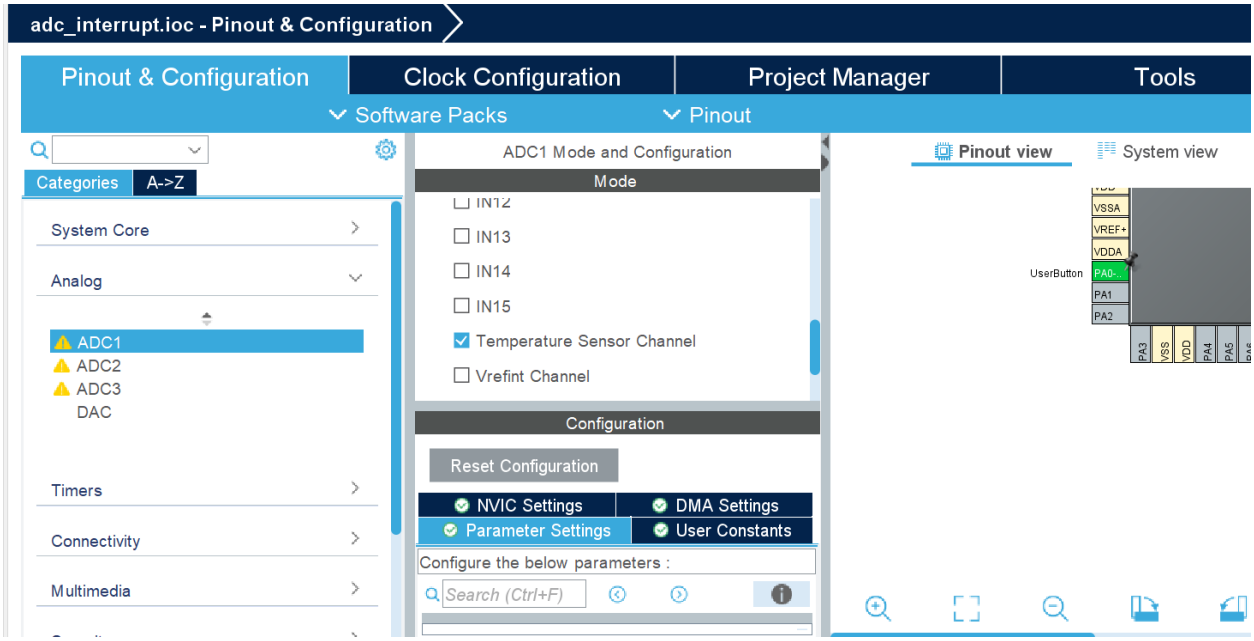
According to section 13.10 of the STM32F4 Reference Manual, the internal temperature sensor reading is available on ADC1, channel 16. This temperature sensor may be polled as in the `adc_poll` project, but in this project, interrupts will be used to provide the internal temperature sensor reading.

Make a new project based on `adc_poll.ioc`, and call it `adc_interrupt`.

In the configuration graphic, Reset_State for ADC1, channel 1, which will not be used in this project.



Under Analog, click on ADC1 and check off Temperature Sensor Channel. There is no pin on the graphic for this internal sensor.



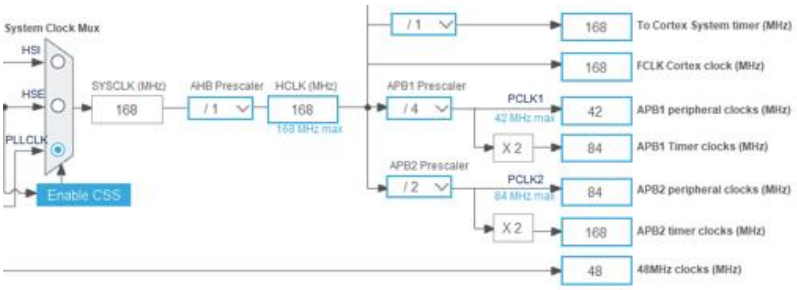
HAL interrupt-handling routines are not always fast enough to catch data coming in at a high rate. Notice that Resolution has 12 bits in 15 ADC clock cycles, where the extra 3 cycles represent the overhead needed for 12-bit ADC. The internal temperature sensor requires a minimum sampling time of 10 μsec to achieve $\pm 1°C$ accuracy. Since PCLK2 is 84 MHz, and since the ADC clock is PCLK2/4 = 21 MHz, the ADC sample time is 47.6 nsec. This means that at least 10 μsec/47.6 nsec = 210 ADC cycles are needed for reliable temperature sensing.

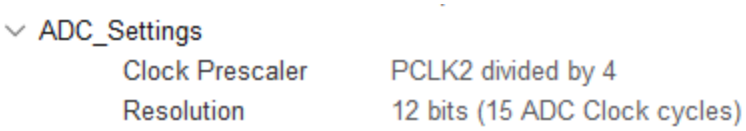
6.3.22 Temperature sensor characteristics

Table 80. Temperature sensor characteristics

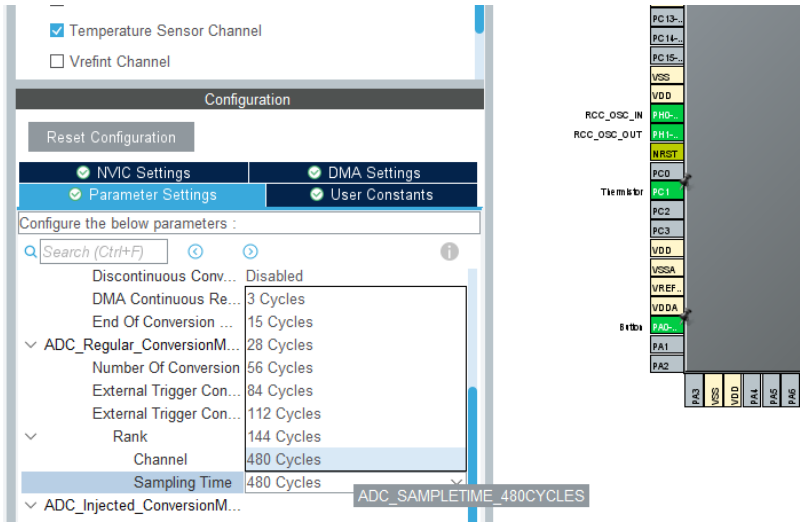
Symbol	Parameter	Min	Typ	Max	Unit
T_L (1)	V_{SENSE} linearity with temperature	-	±1	±2	°C
Avg_Slope (1)	Average slope	-	2.5	-	mV/°C
V_{25} (1)	Voltage at 25 °C	-	0.76	-	V
t_{START} (2)	Startup time	-	6	10	µs
T_{S_temp} (2)	ADC sampling time when reading the temperature (1 °C accuracy)	10	-	-	µs

1. Guaranteed based on test during characterization.
2. Guaranteed by design.

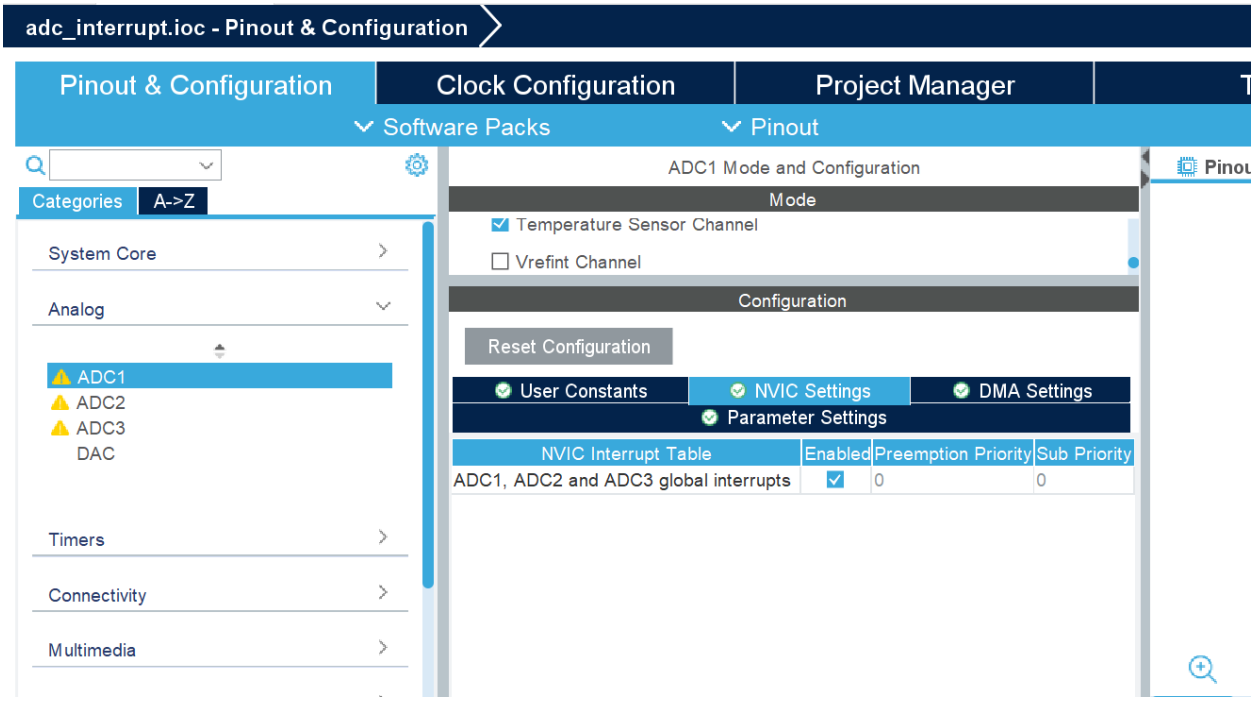




Select Rank – Sampling Rate – 480 Cycles to make this change, where 480 cycles is the lowest number of cycles higher than 210.



Under NVIC Settings, check off ADC1, ADC2 and ADC3 global interrupts. This will create interrupt handlers for your ADC.

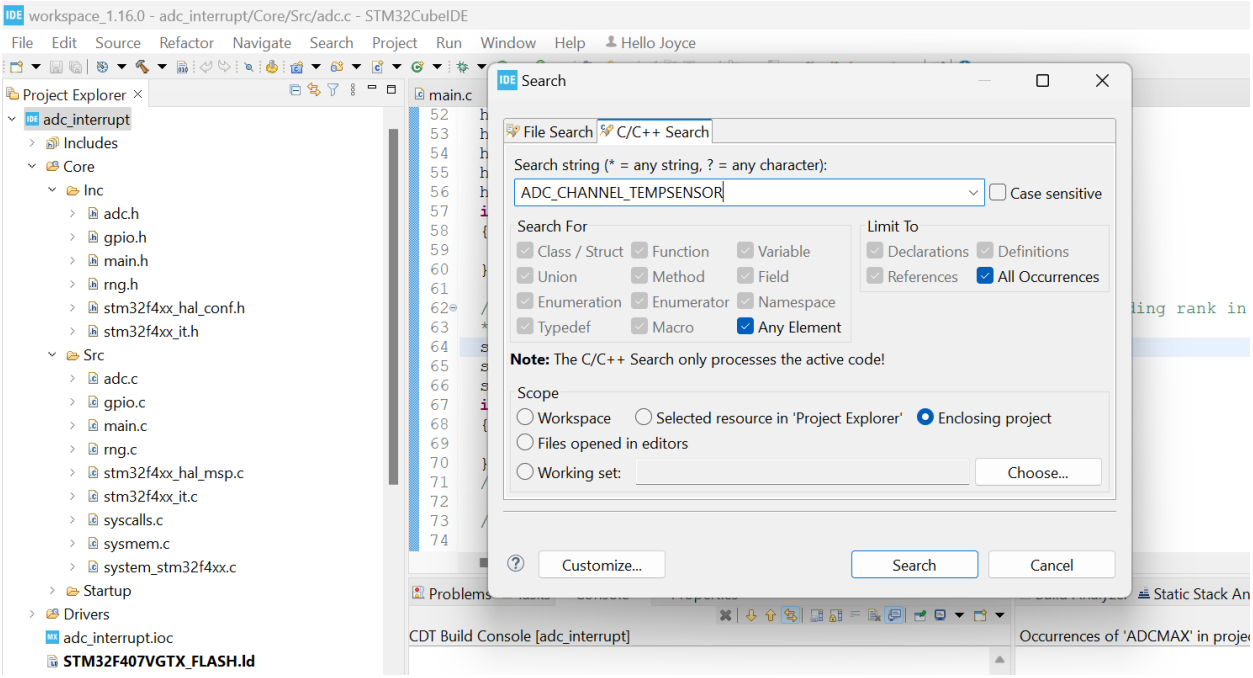


Generate project code.

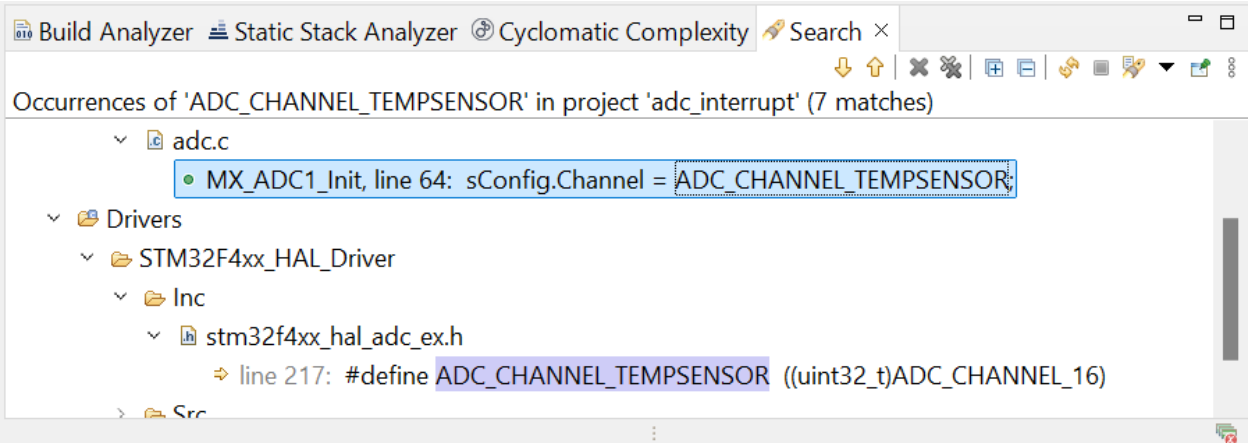
Tip: As a shortcut, you simply save the `.ioc` file, and the IDE will ask if you want to generate code. When you generate code using the menu, the `.ioc` file is saved automatically.

From `Core/Src`, open `adc.c`. In the `MX_ADC1_Init` function, notice the reference to `ADC_CHANNEL_TEMPSENSOR` and to 480 cycles of sampling time.

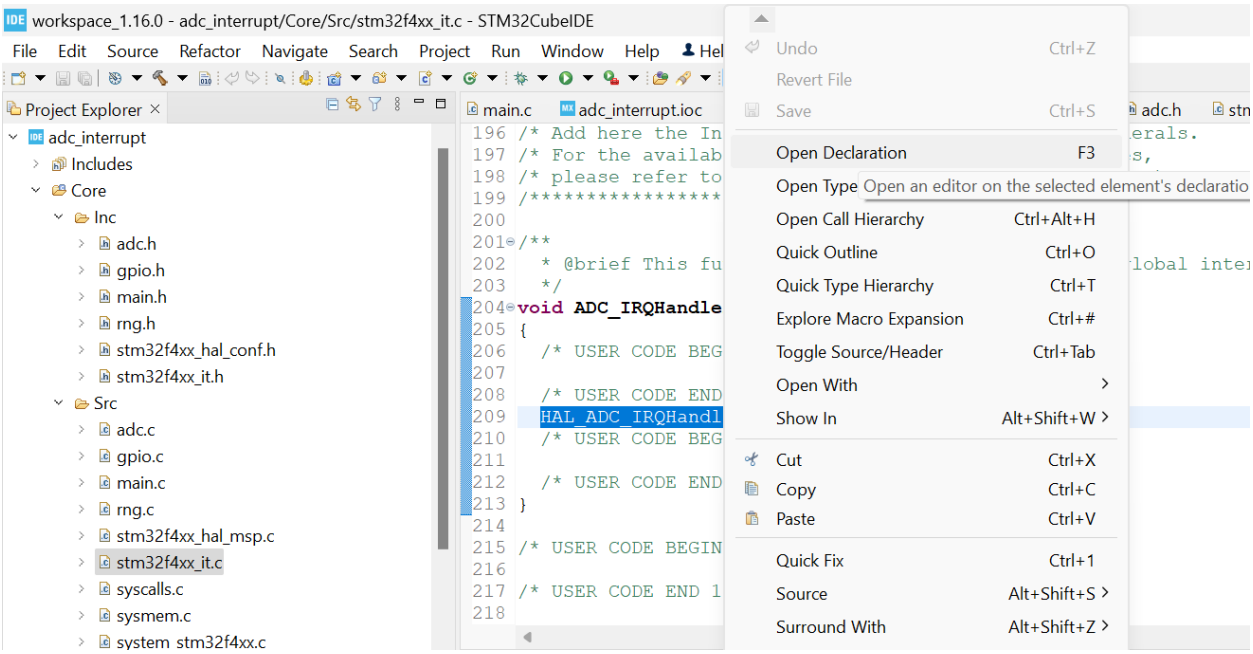
Click on the project name adc_interrupt and click Search – Search. Type in `ADC_CHANNEL_TEMPSENSOR` as the Search string. Click the Enclosing Project button and click Search.



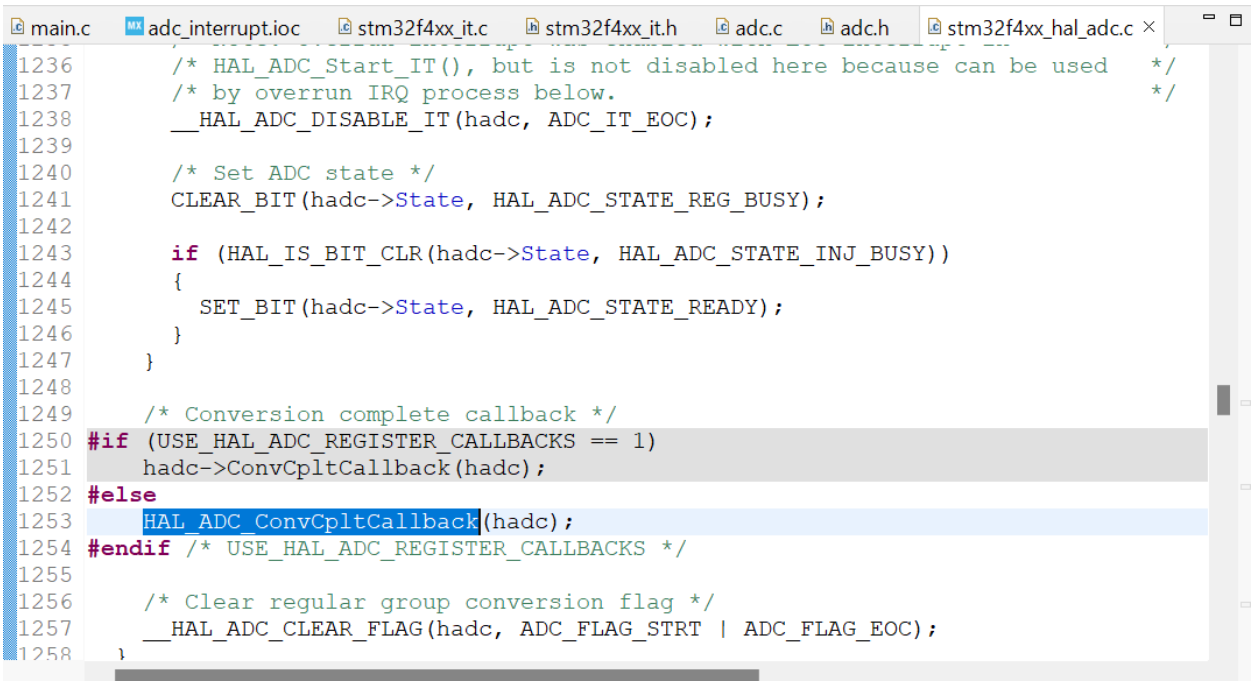
In the Search return window, you can see all the places where the phrase `ADC_CHANNEL_TEMPSENSOR` occurs. Open `Drivers – STM32F32F4xx_HAL_Driver – Inc – stm32f4xx_hal_adc_ex.h`, where you will see a confirmation that channel 16 is connected to the internal temperature sensor.



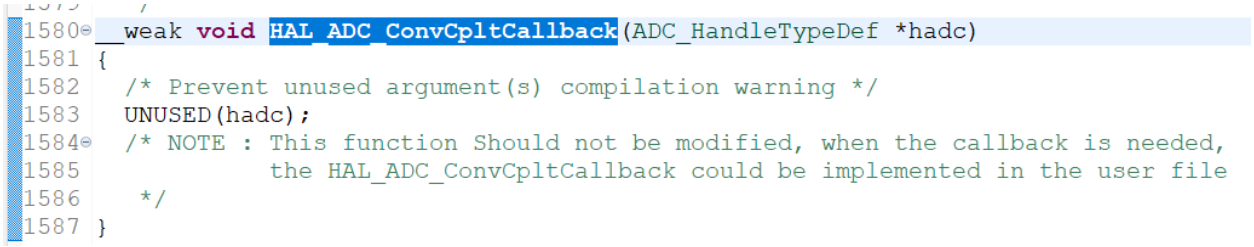
At the bottom of the file `stm32f4xx_it.c` in `Core/Src`, you will see the `ADC_IRQHandler` function that was created. The function `ADC_IRQHandler` calls `HAL_ADC_IRQHandler`. Right-click this function and Open Declaration.



In the `HAL_ADC_IRQHandler` function, you will find a call to `HAL_ADC_ConvCpltCallback`. This is the function that is called when the ADC raises an interrupt to indicate it has received a new sample value.



Right-click on `HAL_ADC_ConvCpltCallback` and Open Declaration.



As indicated in the comments, the `HAL_ADC_ConvCpltCallback` function must be implemented in the user code. The `__weak` version of the function is replaced by whatever the user code provides.

Tip: Lines of code that are greyed out are inactive.

Recall the steps from Section 7.2.2 of the HAL User Manual for ADC interrupt mode:

Interrupt mode IO operation

- Start the ADC peripheral using HAL_ADC_Start_IT()
- Use HAL_ADC_IRQHandler() called under ADC_IRQHandler() Interrupt subroutine
- At ADC end of conversion HAL_ADC_ConvCpltCallback() function is executed and user can add his own code by customization of function pointer HAL_ADC_ConvCpltCallback
- In case of ADC Error, HAL_ADC_ErrorCallback() function is executed and user can add his own code by customization of function pointer HAL_ADC_ErrorCallback
- Stop the ADC peripheral using HAL_ADC_Stop_IT()

It is not important to stop the ADC interrupt for this project. `HAL_ADC_Stop_IT()` might be important in some applications, where ADC interrupts can be stopped after some number of cycles, to save power.

Open `main.c` for the `adc_interrupt` project.

In a `USER CODE` area of your `while(1)` loop, enter the code:

```
HAL_ADC_Start_IT(&hadc1);
// compute sense voltage from ADC value
// compute temperature from sense voltage
// print temperature
HAL_Delay(200);
```

The `HAL_Delay` function prevents results from being printed too quickly.

In `USER CODE` area 0, add a function to read in an ADC value whenever an ADC interrupt is raised.

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
     if(hadc->Instance == ADC1) {
          adc_input = HAL_ADC_GetValue(&hadc1);
     }
}
```

Hover over `HAL_ADC_GetValue` to learn the type of the returned variable. Declare `adc_input` in the `USER CODE` private variable area with an appropriate type.

The ADC sample value `adc_input` is a number between 0 and $2^{12} - 1 = 4095$. This number must first be converted to a voltage V_{sense}.

Challenge: The 12-bit ADC maps voltages between 0 V and 3 V into numbers between 0 and 4095. Convert the ADC sample value `adc_input` into a voltage `V_sense` in your `while(1)` loop. Declare `V_sense` as a `float` value.

Section 5.3.22 of STM32F407 Datasheet provides the characteristics of the internal temperature sensor.

5.3.22 Temperature sensor characteristics

Table 69. Temperature sensor characteristics

Symbol	Parameter	Min	Typ	Max	Unit
T_L (1)	V_{SENSE} linearity with temperature	-	±1	±2	°C
Avg_Slope (1)	Average slope	-	2.5		mV/°C
V_{25} (1)	Voltage at 25 °C	-	0.76		V
t_{START} (2)	Startup time	-	6	10	µs
T_{S_temp} (2)	ADC sampling time when reading the temperature (1 °C accuracy)	10	-	-	µs

1. Guaranteed by characterization.
2. Guaranteed by design.

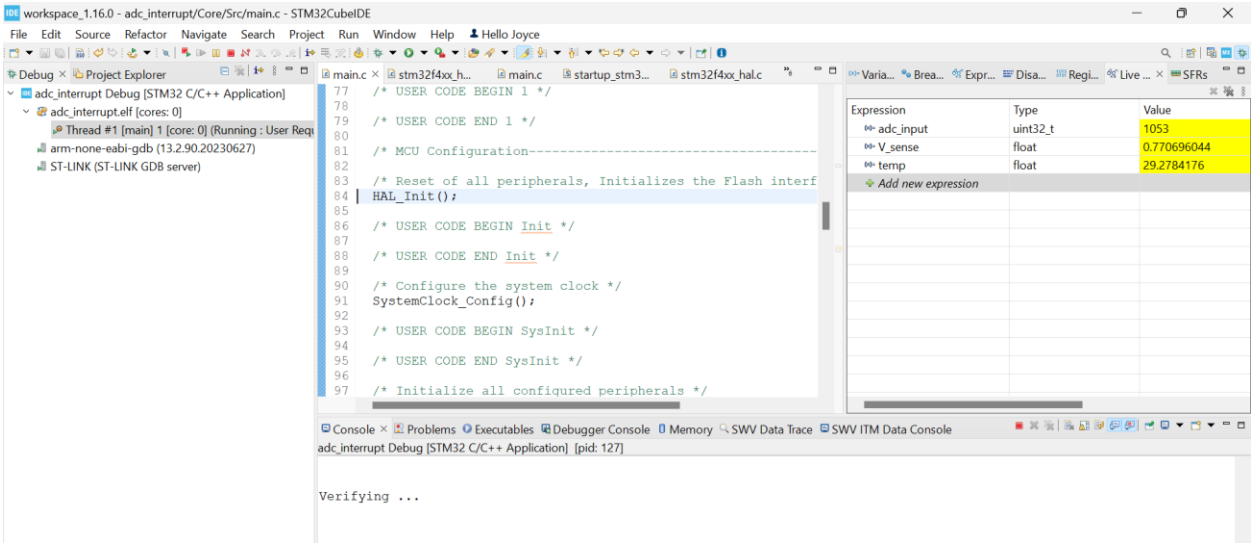
From the table:

$V_{sense} = \text{Avg\_Slope}\ (T - 25) + V_{25}$

where T is the chip temperature in Celsius

Challenge: Solve the voltage-temperature equation for temperature T and implement the equation in your `while(1)` loop, declaring and computing temperature as a `float` value. Follow the directions in Appendix 1 to add print capability to your project. Print the temperature in your `while(1)` loop, remembering that `%4.1f` is a suitable format for a `float` value in a `printf` statement. Warm the chip with your breath, or cool it with a fan, to verify that the temperature increases or decreases slightly.

Tip: While you are in Run – Debug mode, you can view the value of any global variables in your program. In the window at the top right, locate the Live Expressions tab. Type in the name of a global variable. The value of the variable will update in real time. Click x to remove a variable, or xx to remove all variables.

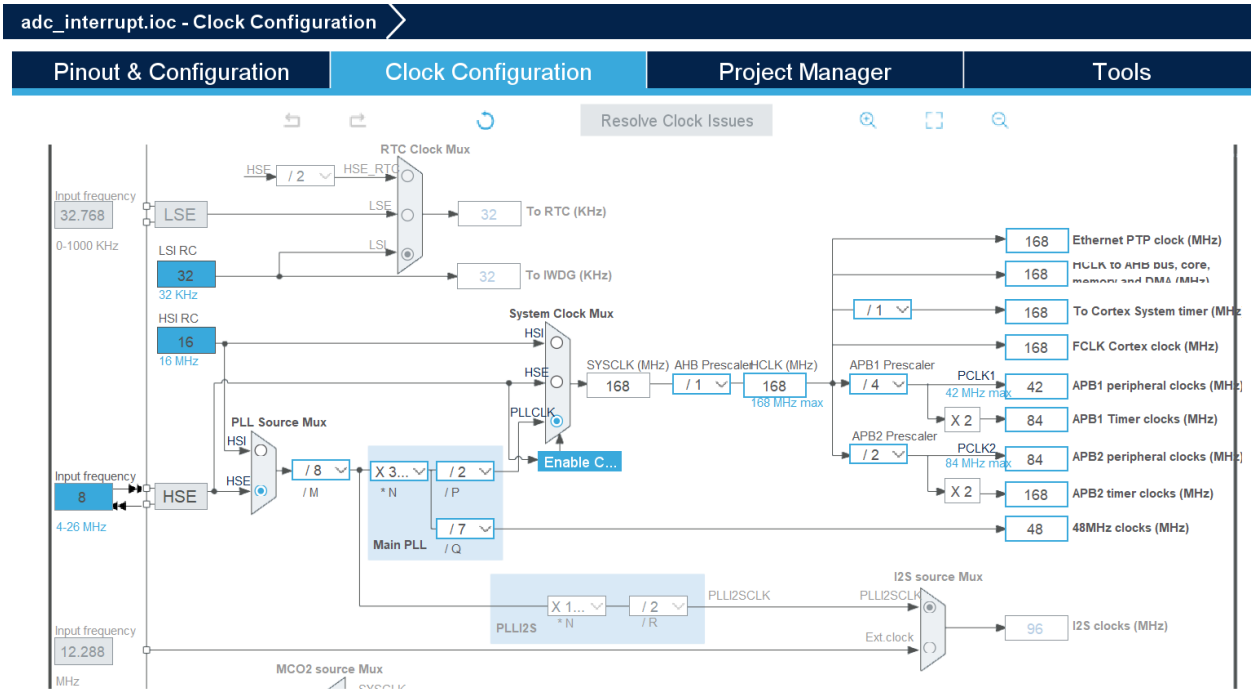


Tip: You can right-click to the left of a line of code, choose Add Breakpoint..., and add a Condition, for example, `temp > 35`.

In your project, the ADC interrupt occurs as soon as the ADC has acquired a new value. Alternatively, the ADC can be triggered by a timer. A timer-triggered ADC can be useful when the sensor does not need to be read very quickly, for example from a temperature sensor in a room.

Open the on `adc_interrupt.ioc` configuration file.

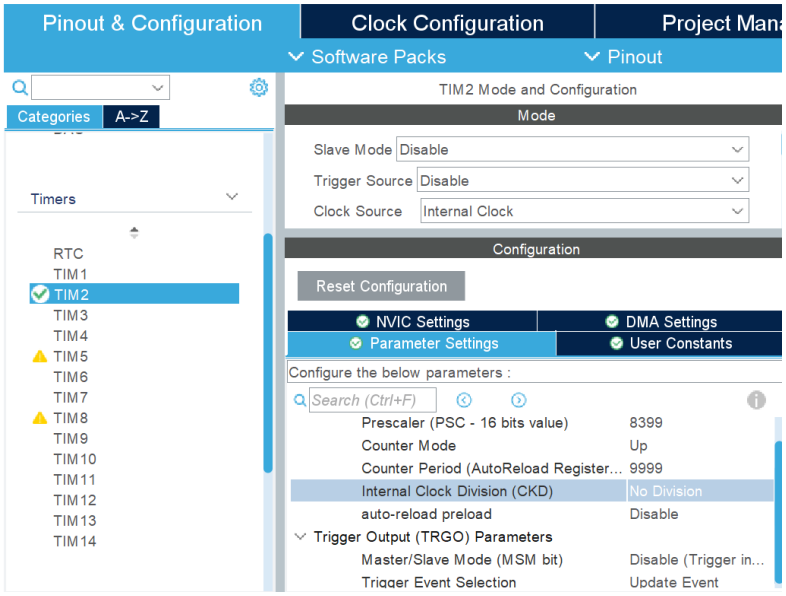
Click on the Clock Configuration tab. The APB2 peripheral clock controls the ADC and DAC timers. Note the 84 MHz frequency for this clock.



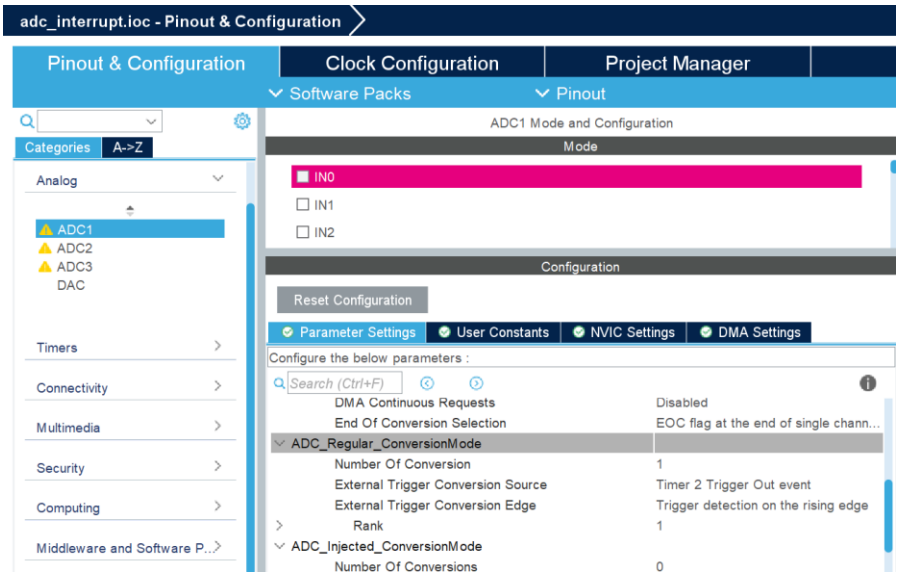
Go to the Pinout & Configuration tab. Set up a timer such as TIM2 with Internal Clock as the Clock Source. Under Parameter Settings, choose Update Event for the Trigger Event Selection. The Prescaler and the Counter Period determine the sampling frequency for the timer. Specifically:

$$\text{sampling frequency} = \frac{\text{APB2 peripheral clock frequency}}{(\text{Prescaler} + 1)(\text{Counter Period} + 1)}$$

Suppose it is sufficient to check a temperature sensor once per second, that is, a 1 Hz sampling frequency. One solution for obtaining 1 Hz sampling is to choose Prescaler = 8399 and Counter Period = 9999.



Go to ADC1. Under Parameter Settings, change the External Trigger Conversion Source to Timer 2 Trigger Out event. The timer will now trigger ADC1 interrupts.



Generate code.

It is helpful to verify your sampling frequency. We will do this by toggling an LED. In the `USER CODE` area 0 in `main.c`, you can modify your function as suggested below, where the reference to `OrangeLED` follows the naming from pin PD13 on your `.ioc` configuration.

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
     if(hadc->Instance == ADC1) {
          adc_input = HAL_ADC_GetValue(&hadc1);
```

```
    }
    HAL_GPIO_TogglePin(OrangeLED_GPIO_Port, OrangeLED_Pin);
}
```

One additional code change is needed. In `USER CODE` area 2, add a line that starts the timer:

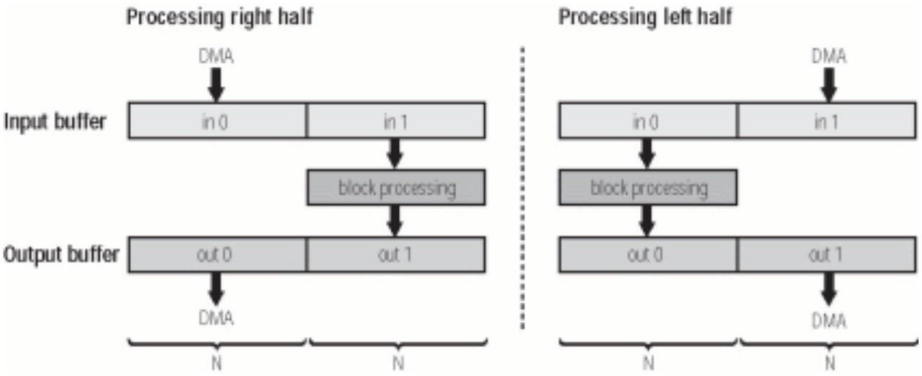
```
HAL_TIM_Base_Start(&htim2);
```

Run – Debug your project. You should observe the temperature printing and the orange LED toggling once per second.

Fifth project: `adc_dac_dma`

In this project, you will use DMA for both ADC and DAC. You will define two ADC inputs, one from an external source such as a function generator, and one from the internal temperature sensor used in the `adc_interrupt` project. The inputs from the external ADC source will be passed to a DAC output.

Using DMA means that accepting inputs and sending outputs is offloaded from the main processor and handled by a DMA controller. While one half of the input buffer is accepting data, the other half is being processed; while one half of the output buffer is filling up with processed values, the other half is being transmitted.



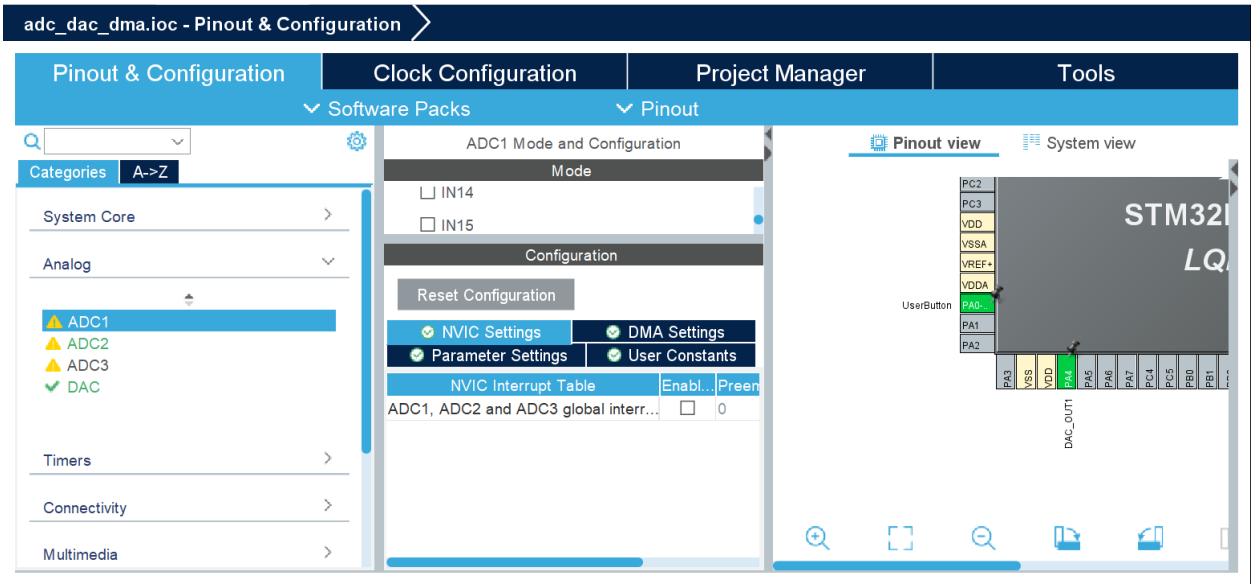
Create a new project based on `adc_interrupt.ioc` and call it `adc_dac_dma`.

Configure pin PC1 as an ADC2 input on channel 11. This ADC input will be connected to a function generator.

Configure PA4 as a DAC output. This DAC output will be connected to an oscilloscope.

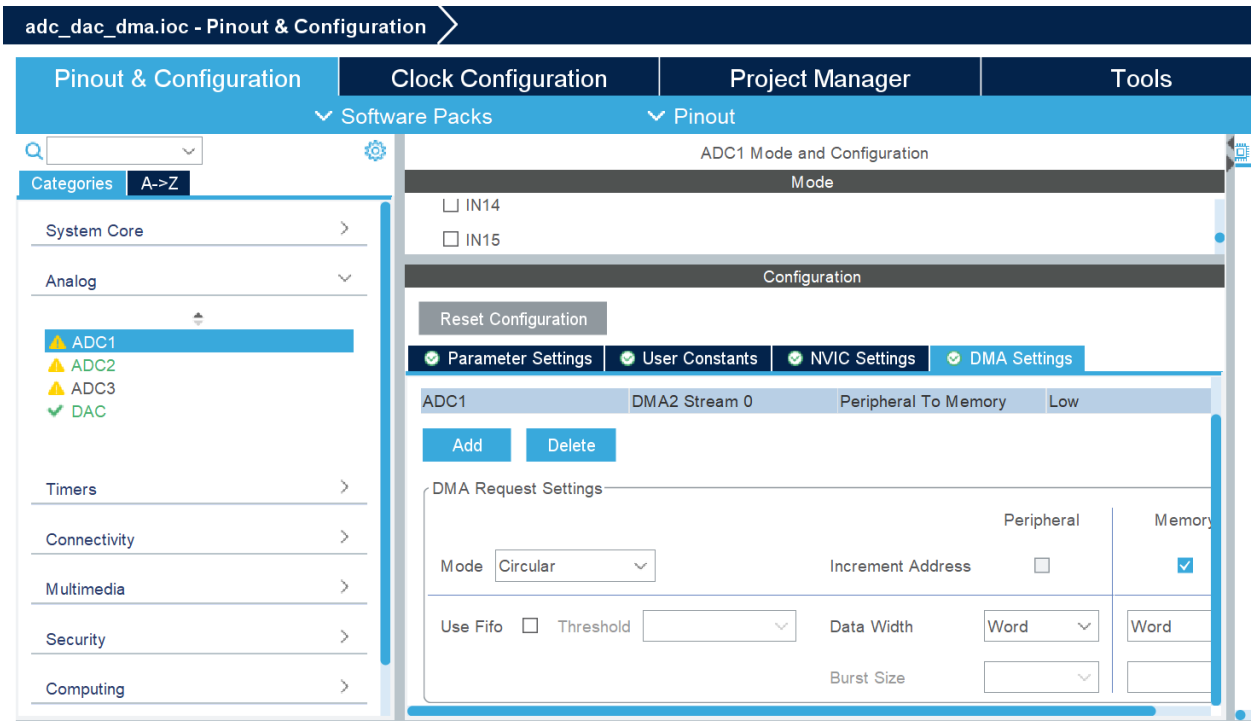
Click on the Analog category. Verify that ADC1 is still configured to access the internal temperature sensor on channel 16, and that Parameter Settings – Rank – Sampling Time is still 480 cycles, as in the previous project.

Under NVIC Settings for ADC1, uncheck global interrupts. DMA will be used instead of interrupts.

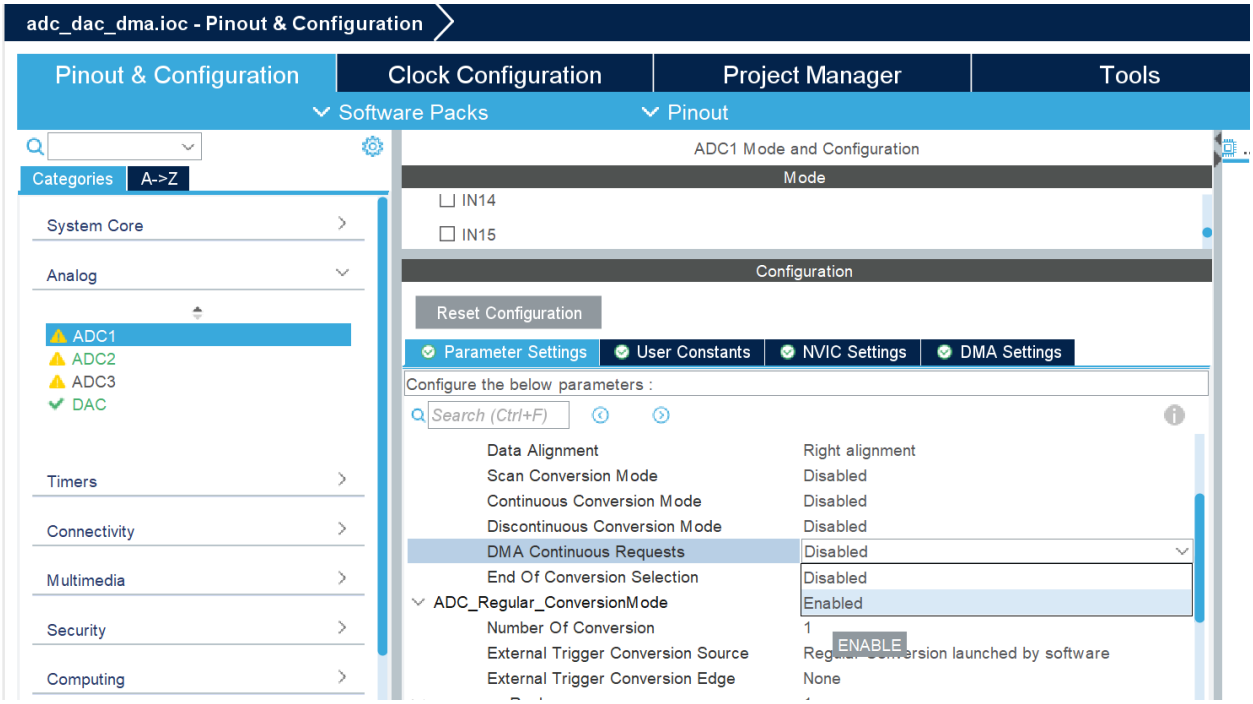


Click on DMA Settings – Add. Select ADC1 from the dropdown list. Choose circular mode for DMA so when a DMA buffer is full the DMA pointer will return to the start of the buffer. Data width should follow the declaration for the user-defined array that serves as the DMA buffer. If the array is declared `uint16_t`, choose Half Word; if the array is declared `uint32_t`, choose Word. HAL DMA functions operate with `uint32_t`. If a `uint16_t` DMA array is used, it can be cast to `uint32_t` in HAL functions.

It will be convenient to use a `uint32_t` DMA array in this project, so Data Width can be set to Word.



For Parameter Settings – ADC_Settings – DMA Continuous Requests, choose Enabled. Otherwise, the DMA buffer will fill only once.



For ADC2, set Parameter Settings – Rank – Sampling Time to 84 cycles. The minimum number of ADC cycles is 3, meaning that the number of cycles devoted to a 12-bit ADC is 12+3 cycles. This minimum overhead is not always sufficient to ensure that ADC voltages are stable enough to be read accurately. The ADC clock runs at PCLK2/4, or 21 MHz, corresponding to an ADC sample time of 47.6 nsec. The sampling time for this project will be 16 kHz, so the time between samples is 62.5 μsec. If an ADC contains N active channels, then each channel must be serviced within 62.5/N μsec. In this project there is only 1 active channel per ADC. Since (62.5 μsec/47.6 nsec) = 1313.0, up to 1313 ADC cycles can be assigned to each sample, so as many as 1313 – 12 = 1301 cycles can be selected for ADC overhead. The highest number of ADC cycles that can be chosen is 480 cycles. This is an acceptable choice, but might take too much time if other peripherals must also be serviced. A balanced choice would be 84 cycles for ADC sampling time, leading to 12+84 = 96 cycles for each conversion.

Set up DMA for ADC2 in the same way as for ADC1.

For the DAC, only the DMA Settings tab needs to be changed. Use the same settings as for the ADCs.

Note that the streams of DMA for each ADC and DAC are distinct. If you look at the NVIC Settings for ADC1, ADC2, or DAC, you will see that DMA interrupts have automatically been enabled.

Generate code for your project.

Open `adc.c` from `Core/Src`. You will see two ADCs: the handle for the internal temperature sensor ADC, `hadc1`, and the handle for the second ADC, `hadc2`.

The file `dac.c` refers to the handle for the DAC, `hdac`.

ADC1 will be used to monitor the internal junction temperature of the STM32F407. The inputs from ADC2 will be passed to the DAC.

It is good practice to operate a DMA buffer in two halves. While the first half is filling, the second half can be processed, and vice versa. If operating on the full buffer, values in the buffer may change while processing is taking place.

In `USER CODE` private defines area in `main.c`, enter:

```
#define HALFBUFFERLENGTH 100
```

In `USER CODE` private variables area in `main.c`, declare three `uint32_t` arrays,

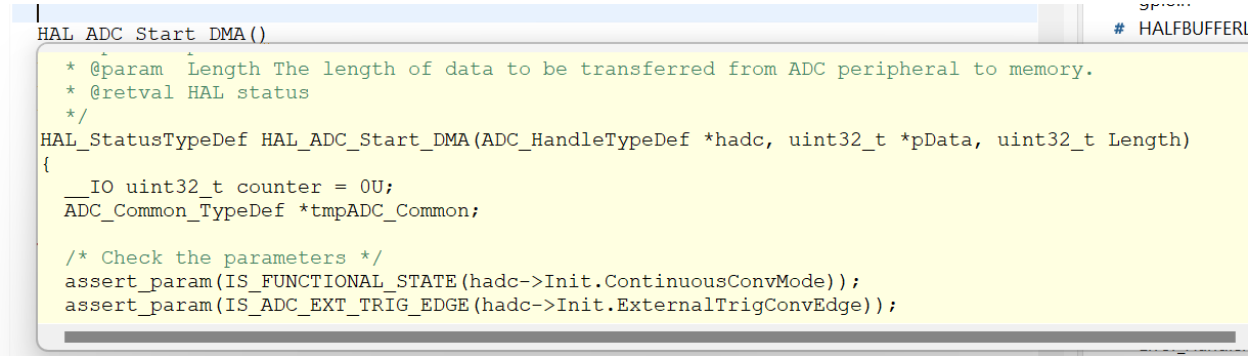
```
uint32_t adc_temp[2*HALFBUFFERLENGTH];
uint32_t adc_input[2*HALFBUFFERLENGTH];
uint32_t dac_output[2*HALFBUFFERLENGTH];
```

Recall the steps from Section 7.2.2 of the HAL User Manual for ADC DMA mode:

DMA mode IO operation

- Start the ADC peripheral using HAL_ADC_Start_DMA(), at this stage the user specify the length of data to be transferred at each end of conversion
- At The end of data transfer by HAL_ADC_ConvCpltCallback() function is executed and user can add his own code by customization of function pointer HAL_ADC_ConvCpltCallback
- In case of transfer Error, HAL_ADC_ErrorCallback() function is executed and user can add his own code by customization of function pointer HAL_ADC_ErrorCallback
- Stop the ADC peripheral using HAL_ADC_Stop_DMA()

You will not need to stop DMA in this project, as continuous DMA requests are being made. In the `USER CODE` area 2, the function `HAL_ADC_Start_DMA` must be added for each ADC and the function `HAL_DAC_Start_DMA` must be added for the DAC. DMA is started just once, before your `while(1)` loop, because it works in parallel with program execution. Type `HAL_ADC_Start_DMA` and hover over the function name to learn what arguments are needed. For example, `HAL_ADC_Start_DMA` requires a pointer to an ADC handle, a pointer to the user-defined DMA buffer array, and the length of the DMA buffer.



Thus, to start DMA for the internal temperature sensor ADC, type:

```
HAL_ADC_Start_DMA(&hadc1, (uint32_t *) adc_temp, 2*HALFBUFFERLENGTH);
```

Starting the DMA for the DAC requires five arguments, including the DAC channel number and the alignment of the data:

```
HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, (uint32_t *) dac_output,
2*HALFBUFFERLENGTH, DAC_ALIGN_12B_R);
```

Challenge: Add a `HAL_ADC_Start_DMA` line for the external ADC source. Be sure to use the correct handle for this ADC.

Compile your project to verify that it is error-free.

When a DMA buffer is half-full, the function the `HAL_ADC_ConvHalfCpltCallback` is called and when the DMA buffer if full, the function `HAL_ADC_ConvCpltCallback` is called.

In the `adc_interrupt` project, the `HAL_ADC_ConvCpltCallback` function was used to get a value from the ADC. The DMA controller continuously fills DMA buffers, so it is not necessary to get values. The DMA for ADC1 will automatically fill the `adc_temp` array, without user intervention.

Create two functions, `HAL_ADC_ConvHalfCpltCallback` and `HAL_ADC_ConvCpltCallback`, and place them in the `USER CODE` area 0. The first function should operate on the first half of a DMA array, and the second should operate on the second half of the DMA array.

```
void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef *hadc)
{
     if(hadc->Instance == ADC1) {

      }
}
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
     if(hadc->Instance == ADC1) {
```

```
        }
}
```

Challenge: Data from sensors is often noisy. It is common practice to average multiple sensor readings to obtain a more reliable estimate. In the function `HAL_ADC_ConvHalfCpltCallback`, add a `for` loop to add up the values from `adc_temp[0]` to `adc_temp[HALFBUFFERLENGTH-1]`, and divide the result by `HALFBUFFERLENGTH`. In the function `HAL_ADC_ConvCpltCallback`, add a `for` loop to add up the values from `adc_temp[HALFBUFFERLENGTH]` to `adc_temp[2*HALFBUFFERLENGTH-1]`, and divide the result by `HALFBUFFERLENGTH`. Declare global `float` variables as needed. The averaged ADC value can be converted to a temperature using the equations provided in the `adc_interrupt` project if desired.

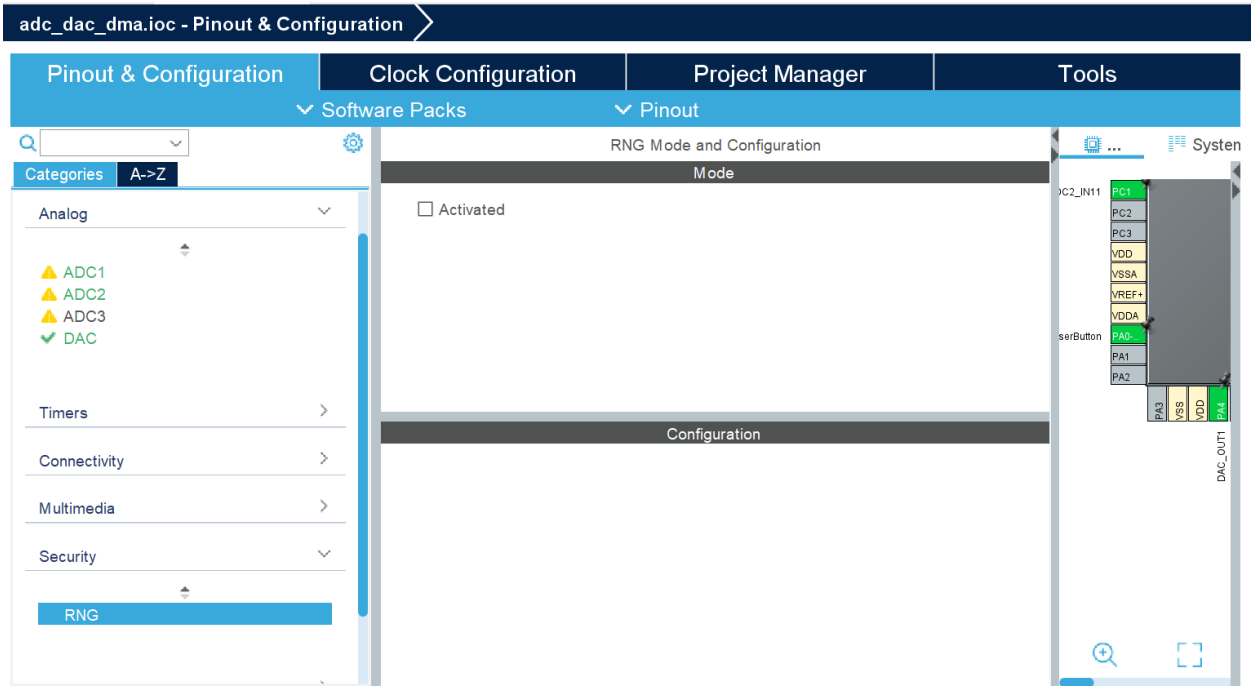
Challenge: Add an `else if(hadc->Instance == ADC2){ }` statement to the `HAL_ADC_ConvHalfCpltCallback` and `HAL_ADC_ConvCpltCallback` functions, to copy the values in the `adc_input` array, which was filled by the DMA for ADC2, into the `dac_output` array. In the `HAL_ADC_ConvHalfCpltCallback` function, the first half of the array should be copied; in the `HAL_ADC_ConvCpltCallback` function, the second half of the array should be copied.

Compile your program to confirm it is error-free.

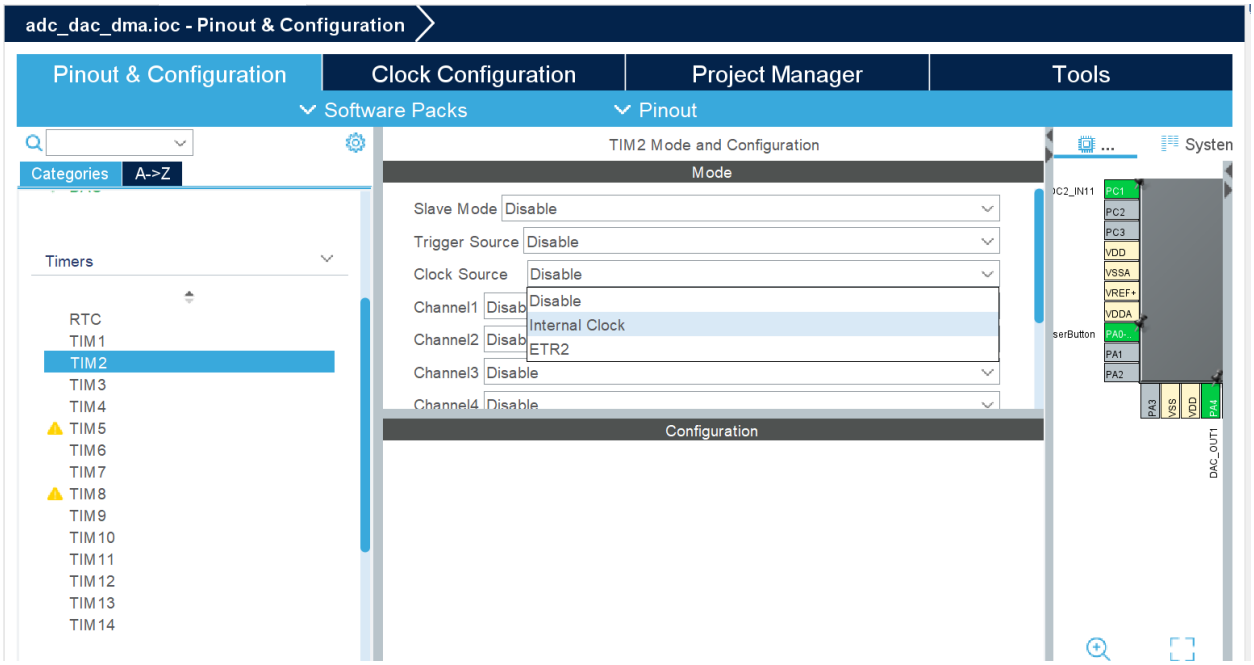
DMA can run at the core clock speed, without the use of a timer, but when DMA runs extremely quickly there is little time for processing, which can create glitches in the output. To avoid this problem, you will set up a timer to operate the ADCs and DAC. The timer will run at 16 kHz, so the filling of half a DMA buffer will take enough time to allow a decent amount of processing for the other half.

Recall that one input for this project comes from the internal temperature sensor, and the other from the function generator. 16 kHz sampling is far faster than is needed for monitoring the temperature sensor, but adequate for low frequency audio. For high quality audio input, a higher sampling frequency such as 48 kHz should be used (though at this higher sampling rate, there is not enough time to get an accurate temperature sensor reading).

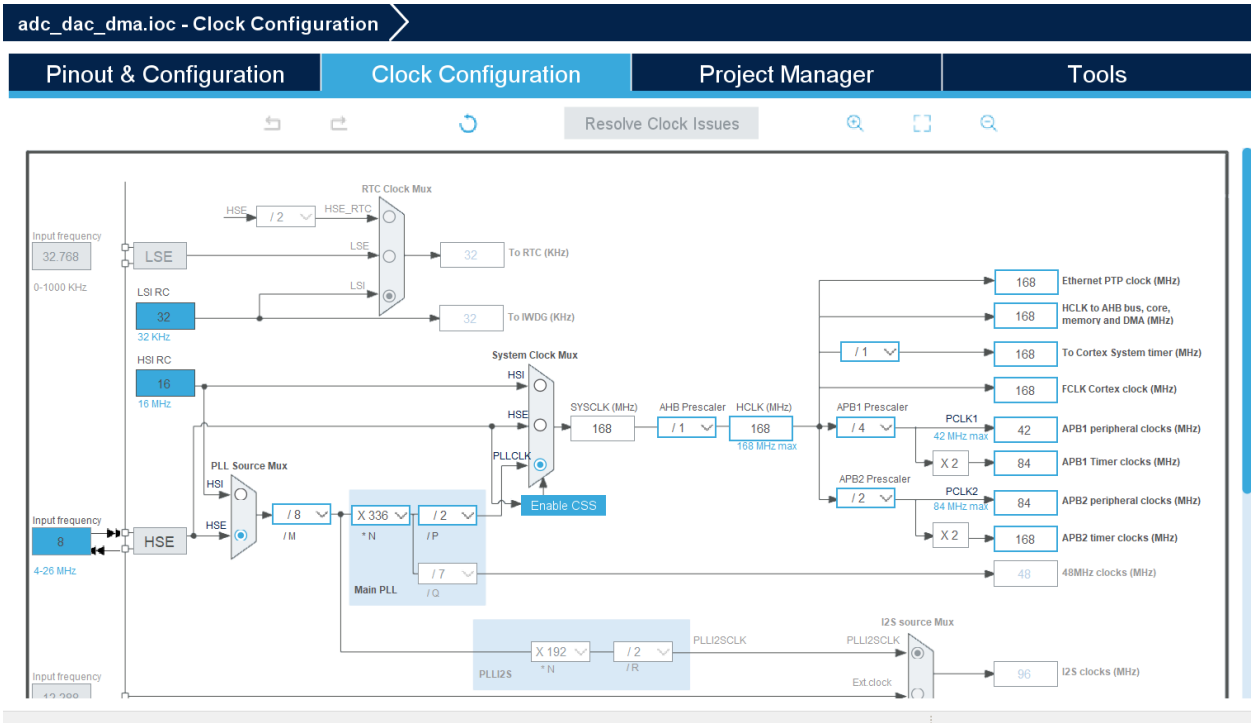
Open `adc_dac_dma.ioc`. Since a timer will be used to manage the clocking, RNG can be removed. Click on the Security category, and click on RNG. Remove the checkmark to deactivate.



In the Timers category, TIM2 should already be set up with Internal Clock as the Clock Source, as it was used in the `adc_interrupt` project, but the frequency of the timer for `adc_dac_dma` must be changed to 16 kHz.



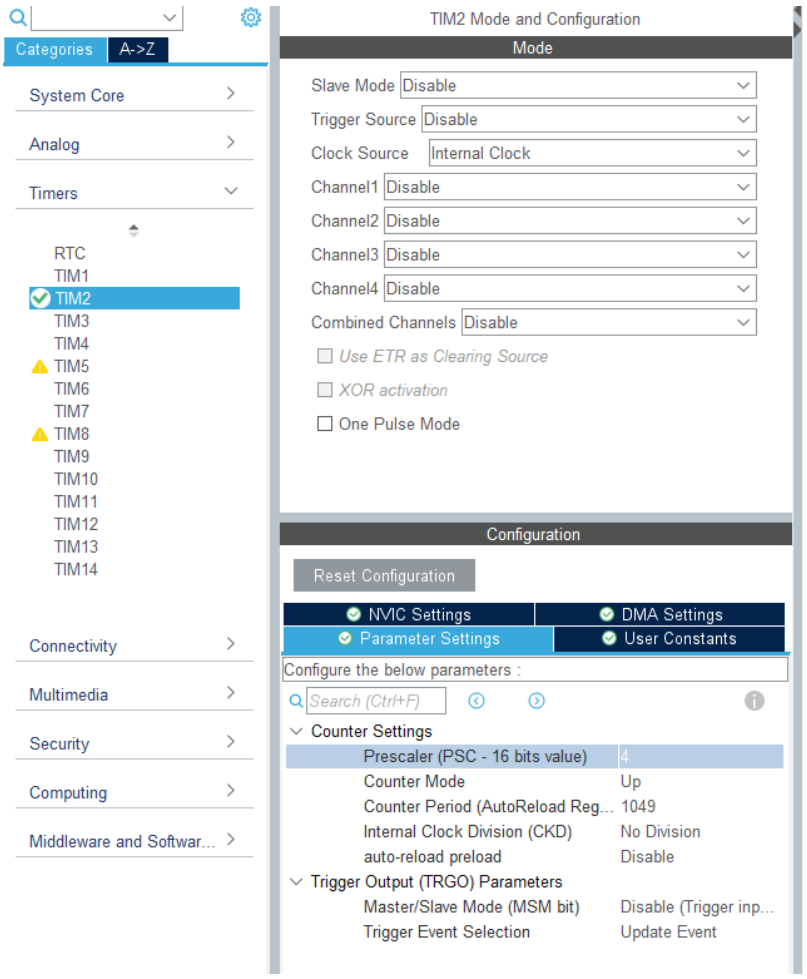
Recall that the APB2 peripheral clock that controls the ADC and DAC timers has a frequency of 84 MHz.



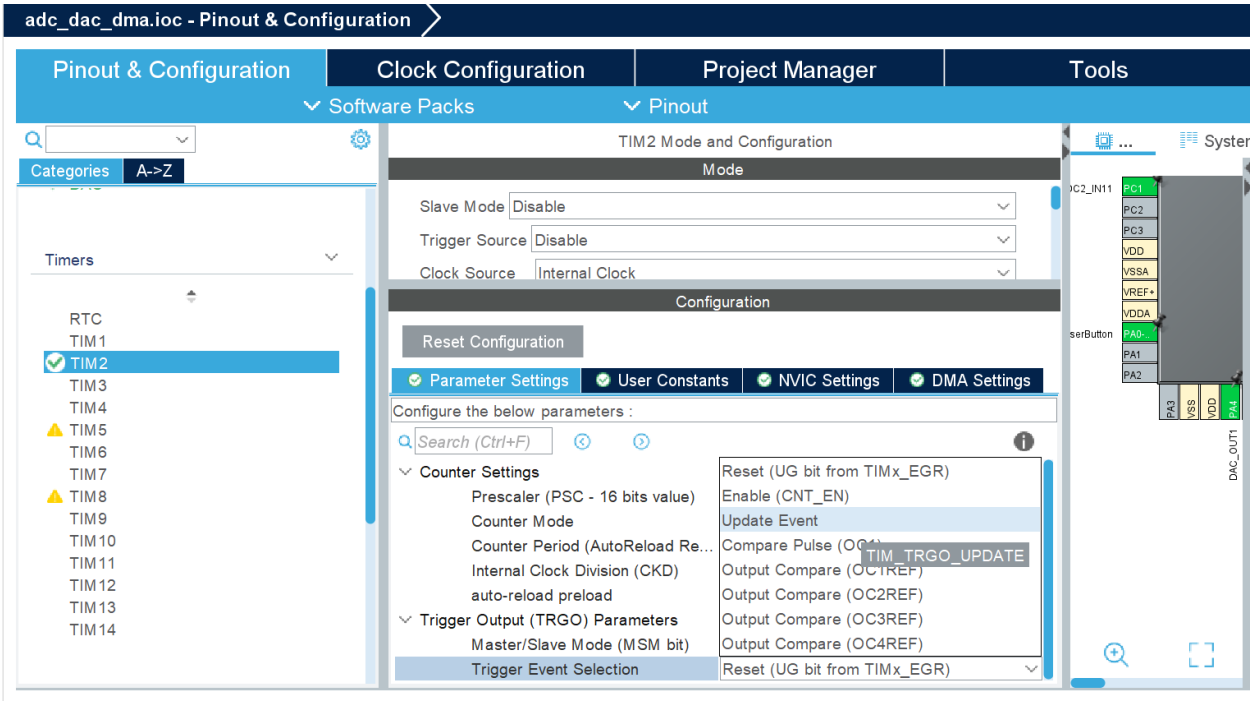
Return to TIM2. Under Parameter Settings – Counter Settings, the Prescaler and the Counter Period determine the sampling frequency for the timer. Specifically:

$$\text{sampling frequency} = \frac{\text{APB2 peripheral clock frequency}}{(\text{Prescaler} + 1)(\text{Counter Period} + 1)}$$

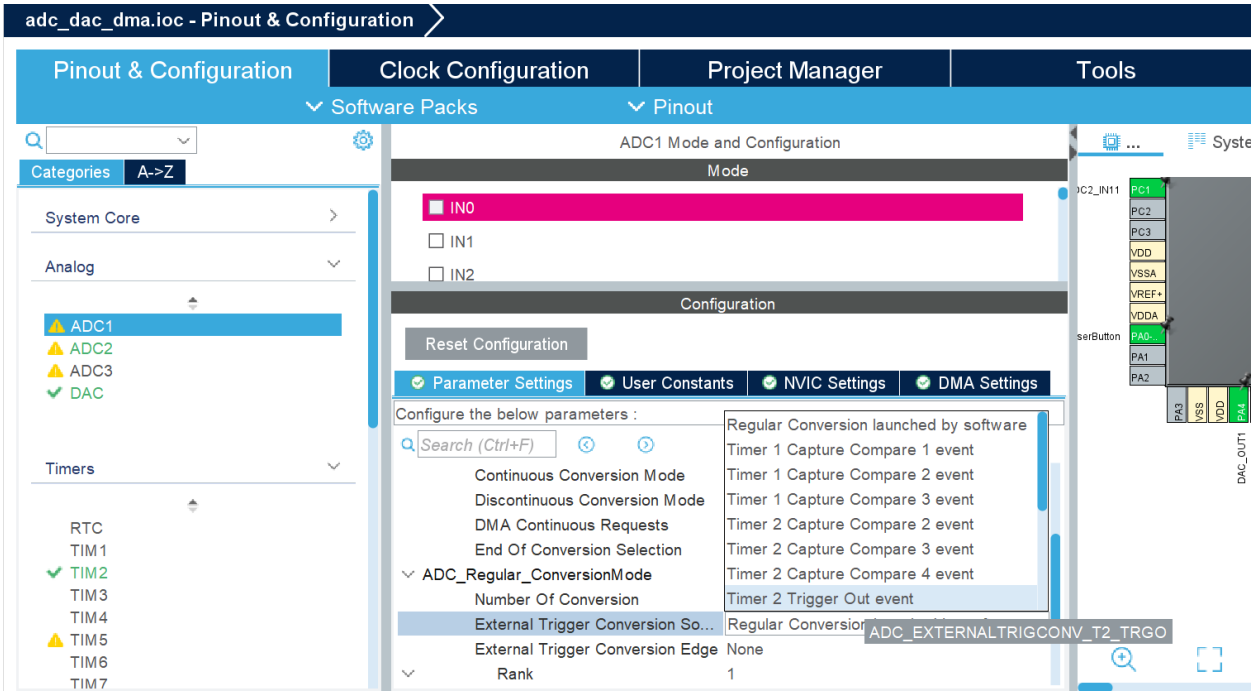
One solution for obtaining 16 kHz sampling is to choose Prescaler = 4 and Counter Period = 1049.



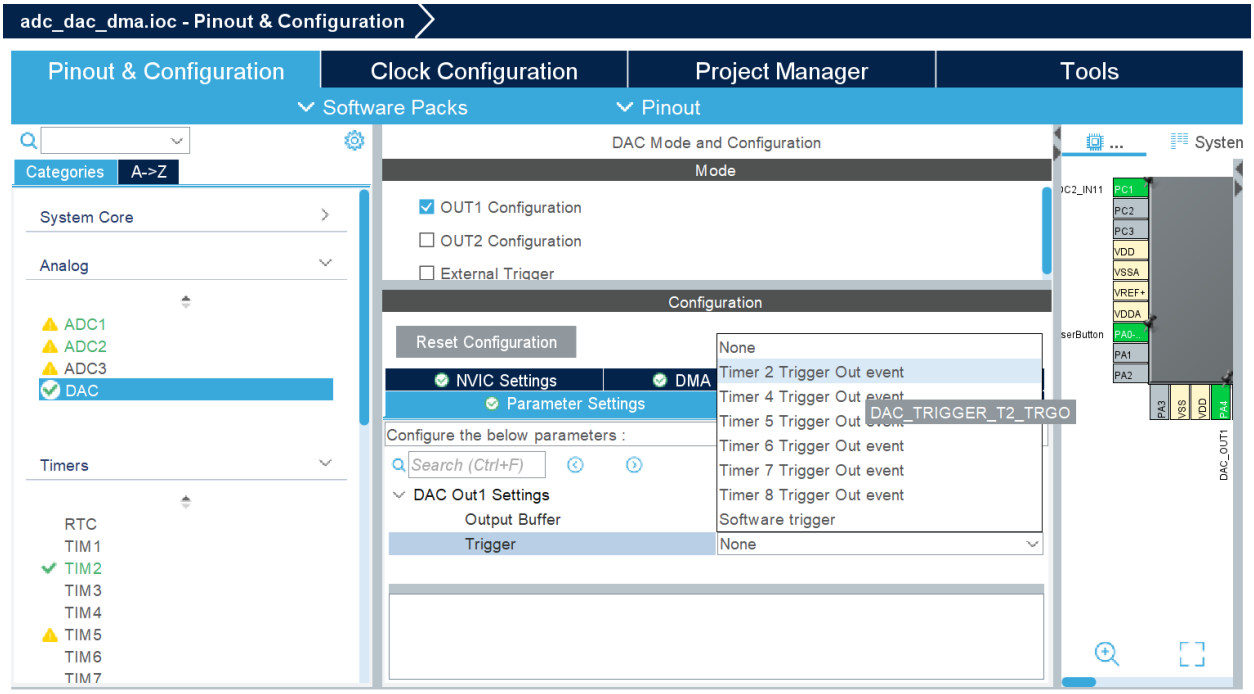
Under Parameter Settings – Trigger Output, change Trigger Event Selection to Update Event (as in the previous project):



Each ADC and DAC must now be linked to the timer. Go to ADC1. Under Parameter Settings – External Trigger Conversion Source, choose Timer 2 Trigger Out Event. Do the same for ADC2.



Go to DAC. Under Parameter Settings – DAC Out1 Settings, change Trigger to Timer 2 Trigger Out event.



Generate code for your project.

To start the timer, add in `USER CODE` area 2, the line:

```
HAL_TIM_Base_Start(&htim2);
```

Compile your program.

Set up a function generator to generate a signal that is at most 3 Vpp , with an offset of 1.5 V, so that the signal is suitable for an ADC input. Attach the function generator signal between pins PC1 and ground on the Discovery board.

Run – Debug.

Click Play – Resume.

Challenge: While in Debug, enter variable names in the Live Expressions window. Monitor your averaged ADC1 value. Open the arrays `adc_temp`, `adc_input`, and `dac_output`. Verify that the numbers in the array are reasonable: `adc_temp` values should be fairly consistent, as they follow temperature, `adc_input` should follow the shape of your function generator signal, `dac_output` should follow `adc_input`. Pause and restart execution. Insert breakpoints, experiment with step over and step into.

Connect an oscilloscope between pins PA4 and ground to monitor the DAC output. Verify that the signal from your function generator appears on the oscilloscope. Change signal frequency and see the oscilloscope follow.

In addition to Live Expressions, you can pause execution and hover over a variable to see its value(s).

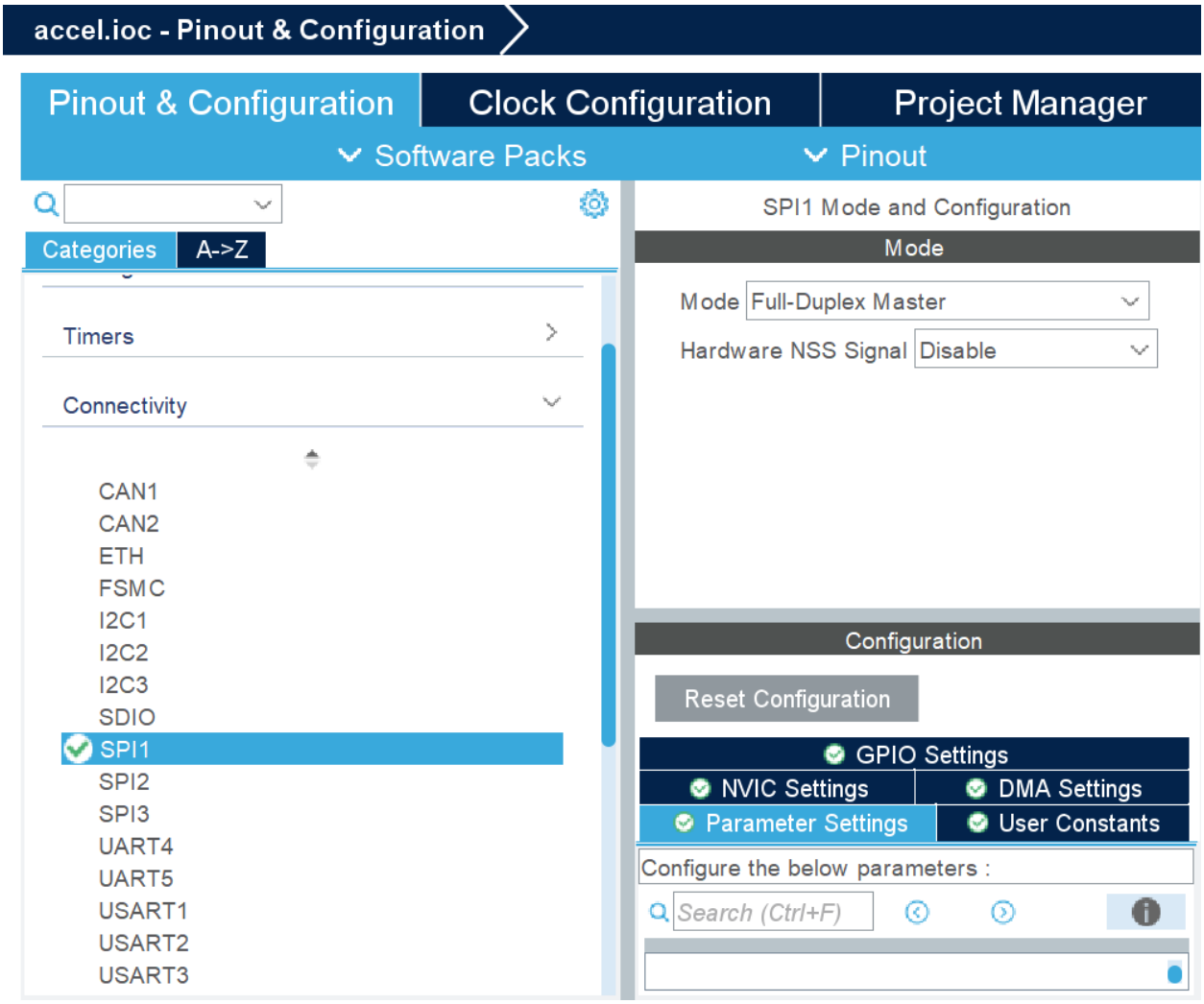
As noted in the `adc_poll` project, the function generator input to ADC can be replaced by a multitude of sources, such as thermistors or microphones.

Sixth project: `accel`

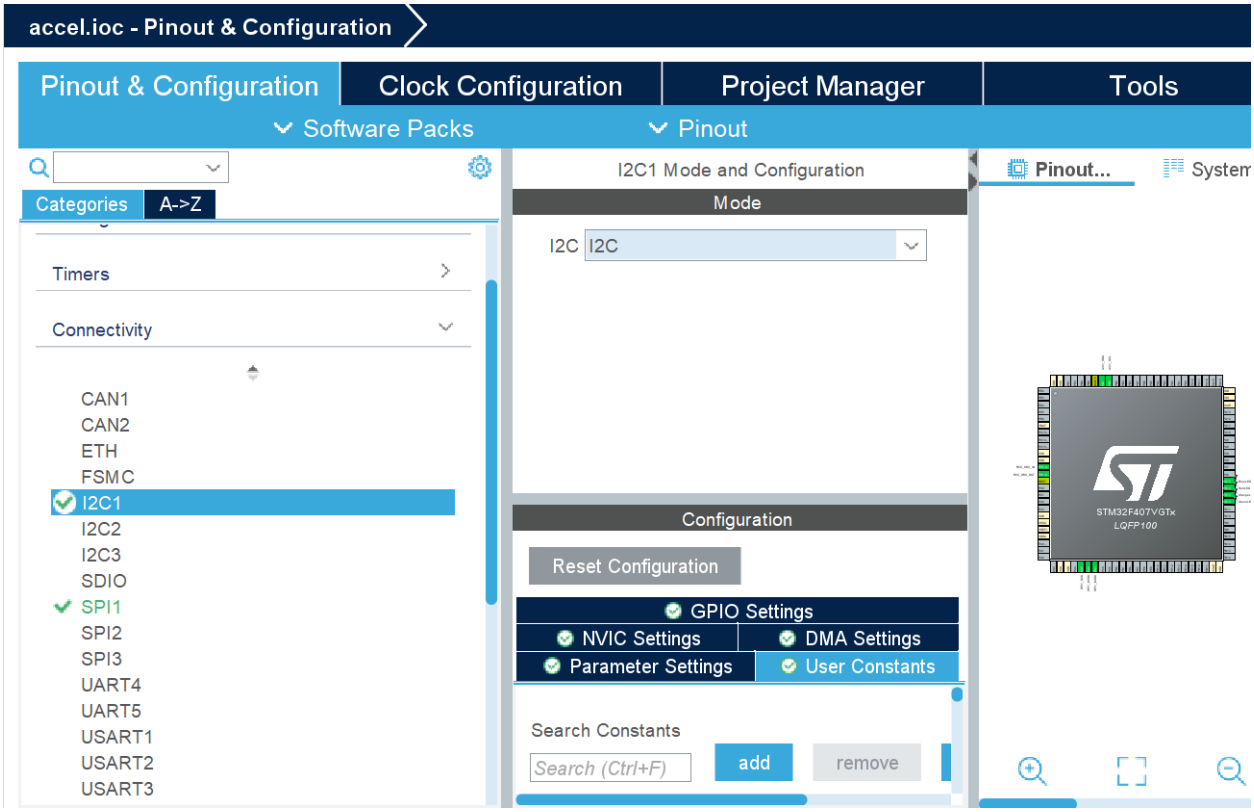
The STM32F407 has an onboard three-axis MEMS (micro-electromechanical system) accelerometer LIS3DSH. The datasheet https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/406/LIS3DSH.pdf lists example applications.

Create a new project called `accel`, based on `blink_LED.ioc`.

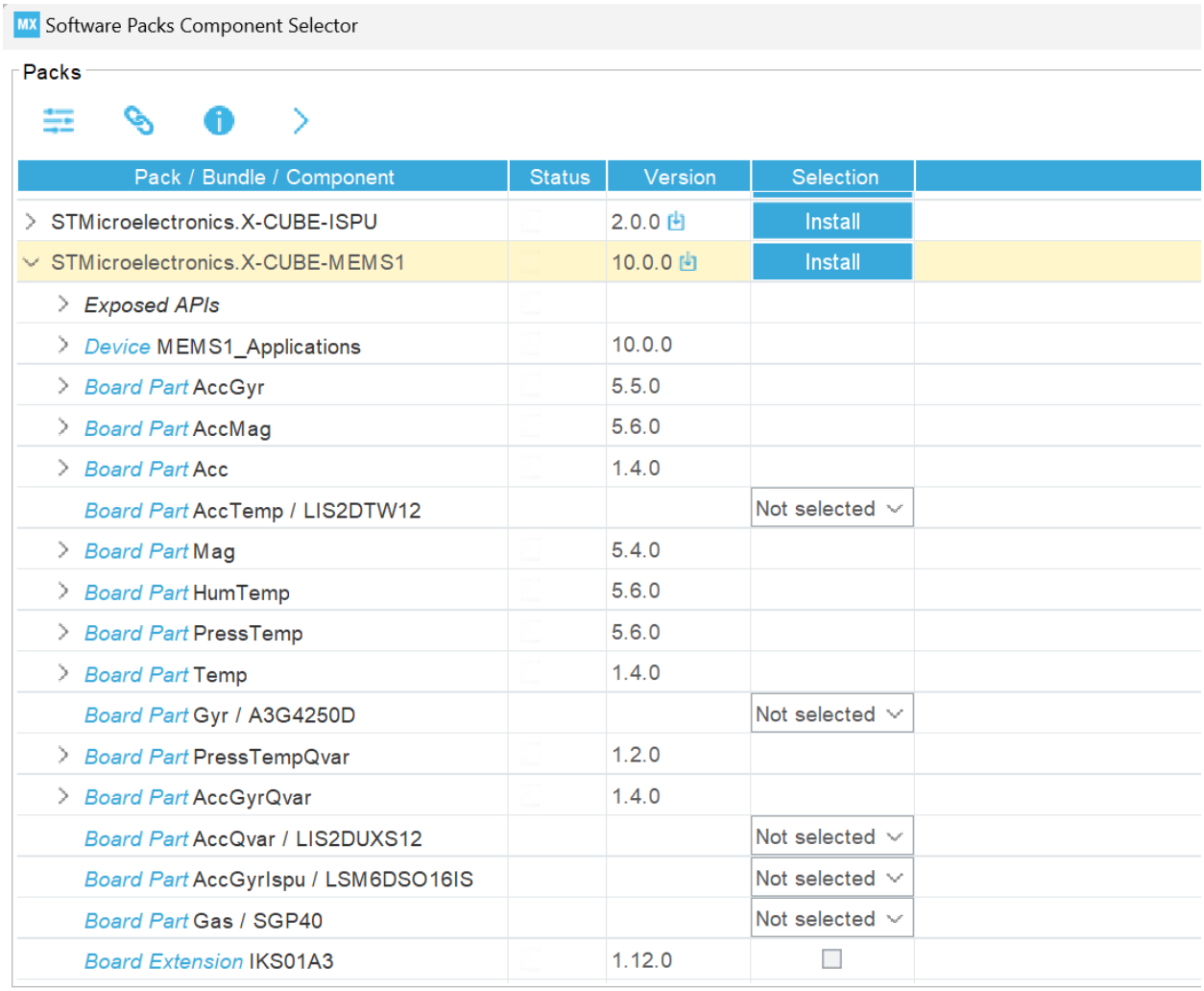
The accelerometer communicates using SPI and I²C communication protocols. To turn SPI on, go to the Connectivity category, select SPI1, and select Full Duplex – Master mode.



To turn I²C on, click on I2C1 and select I2C mode.



Still in `accel.ioc`, click on Software Packs – Select Components. Since the onboard is a MEMS device, open the MEMS1 pack. Unfortunately the LIS3DSH accelerometer is not among the options. If so, it would be possible to obtain the required software and install it from the Middlewares and Software Packs category. It is useful to be aware of the available devices for future reference.

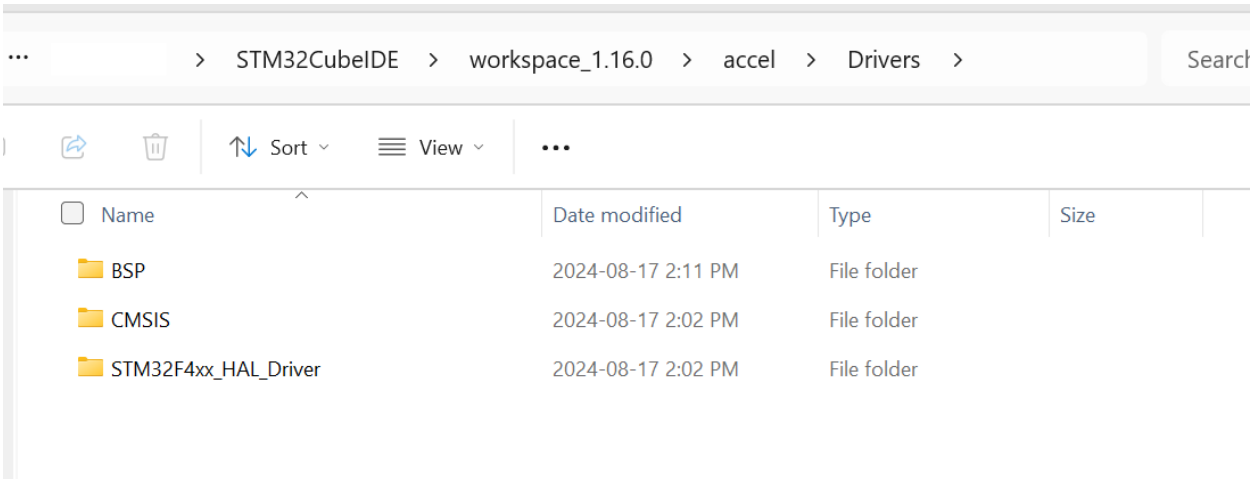


Generate code for your project.

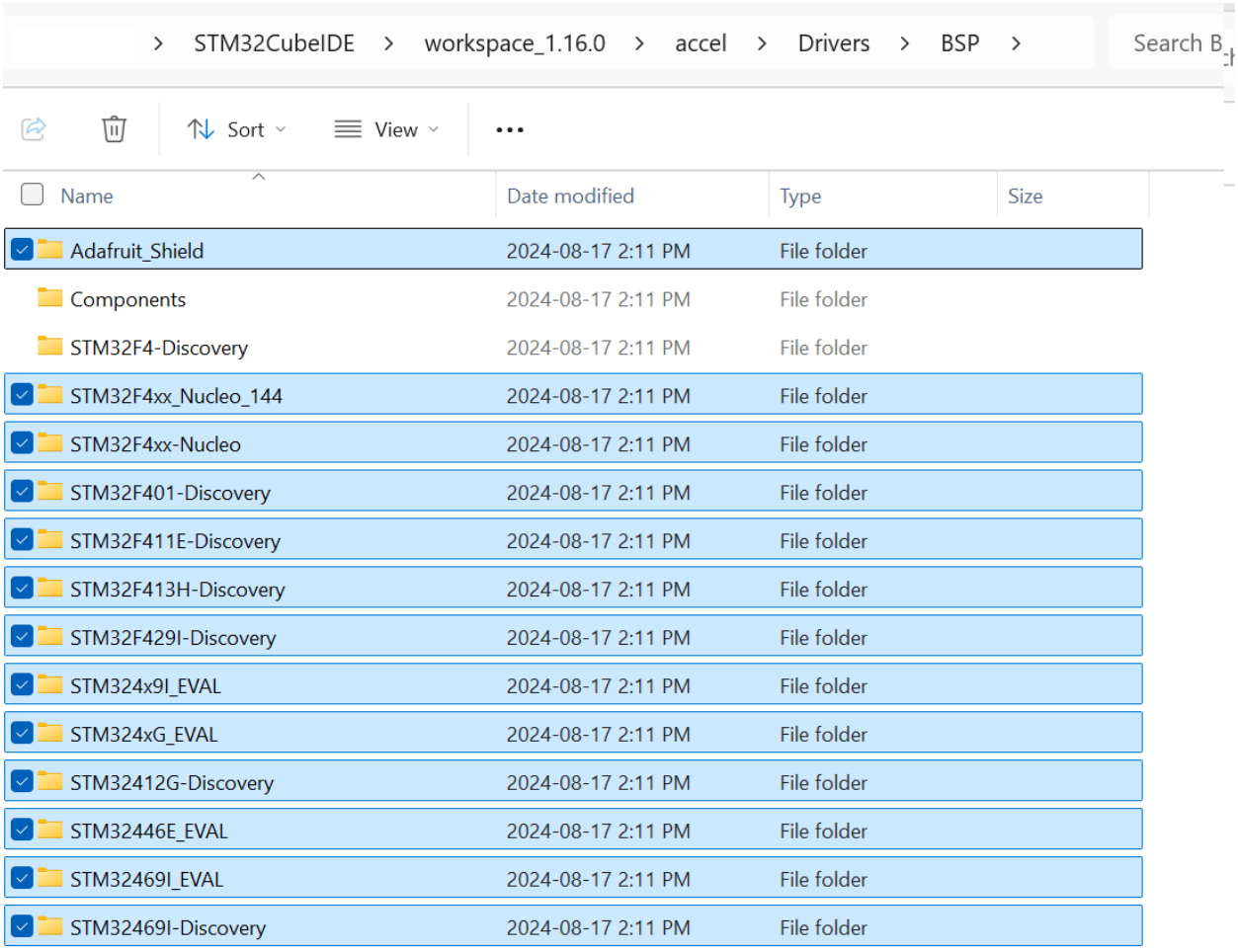
Although it is not among the software packs, the code needed for the accelerometer (and other devices as well) is part of the STM32CubeIDE installation. It may be found on your PC (possibly with a different version number) at:

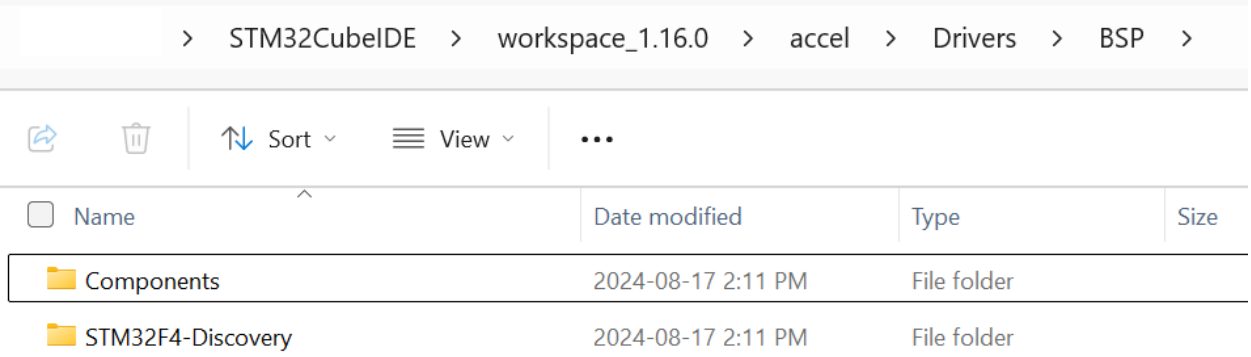
`C:\Users\`*`username`*`\STM32Cube\Repository\STM32Cube_FW_F4_V1.28.1\Drivers\BSP`

Copy the entire BSP directory from the Repository and paste it into the Drivers folder of your `accel` project.

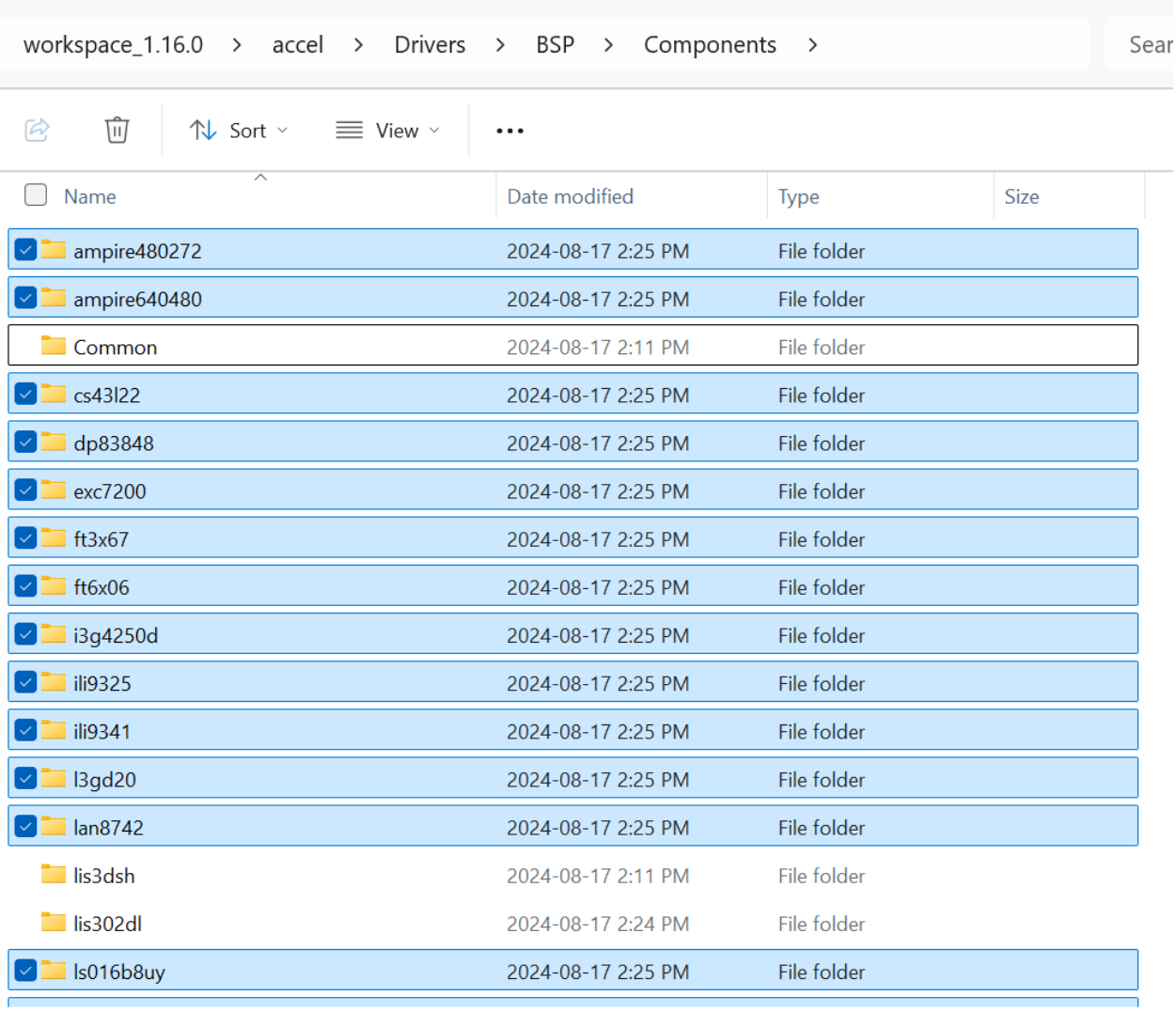


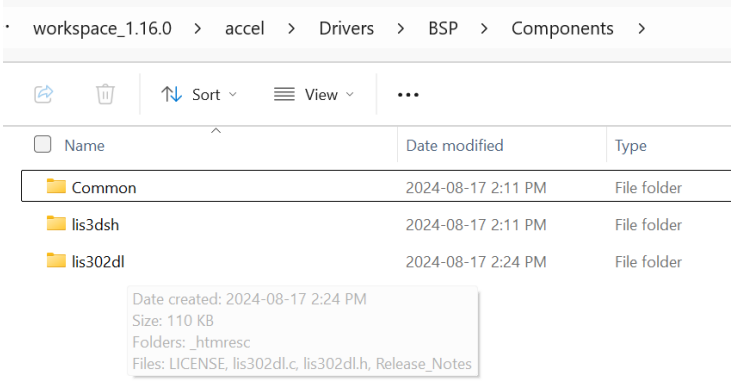
Open the BSP folder in your project. Delete all folders except Components and STM32F4-Discovery. All other folders serve different boards.



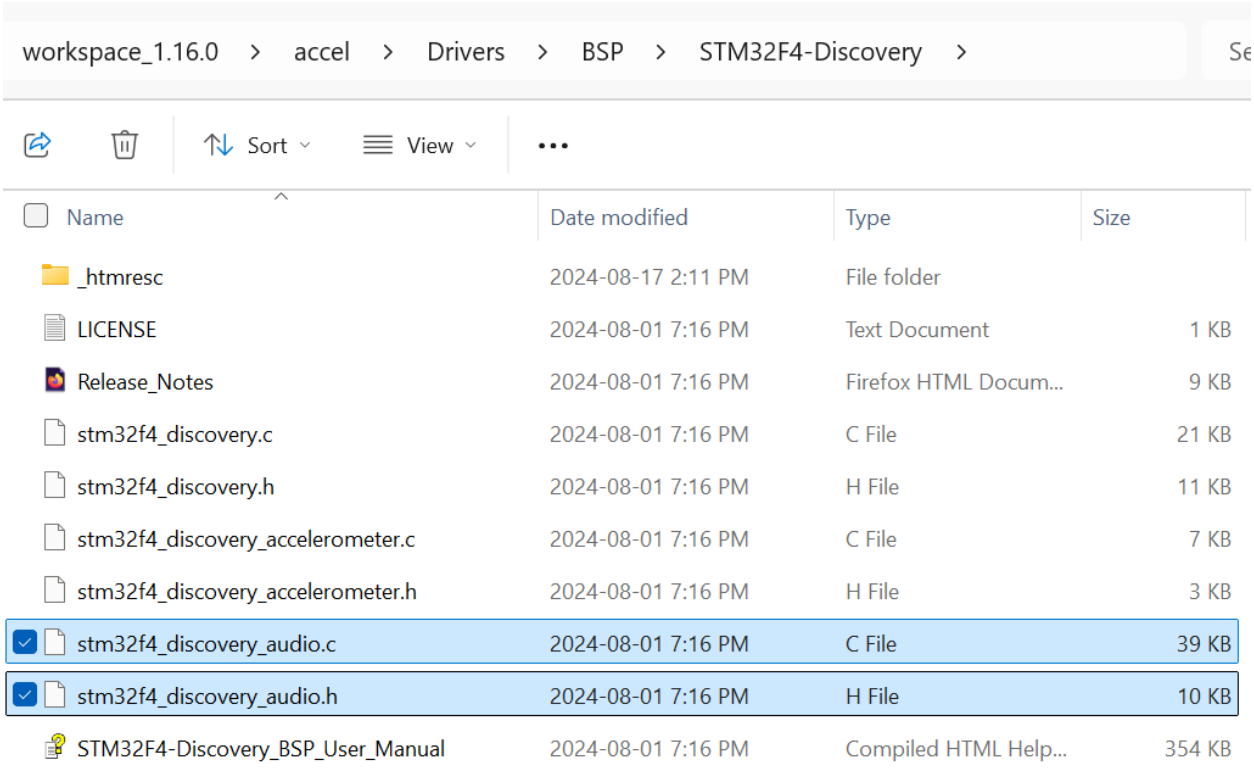


In the Components directory, remove all folders except `Common`, `lis3dsh`, and `lis302dl`. The `lis302dl` file is included because it is the accelerometer used by an older version of the board, also referred to in the include files.





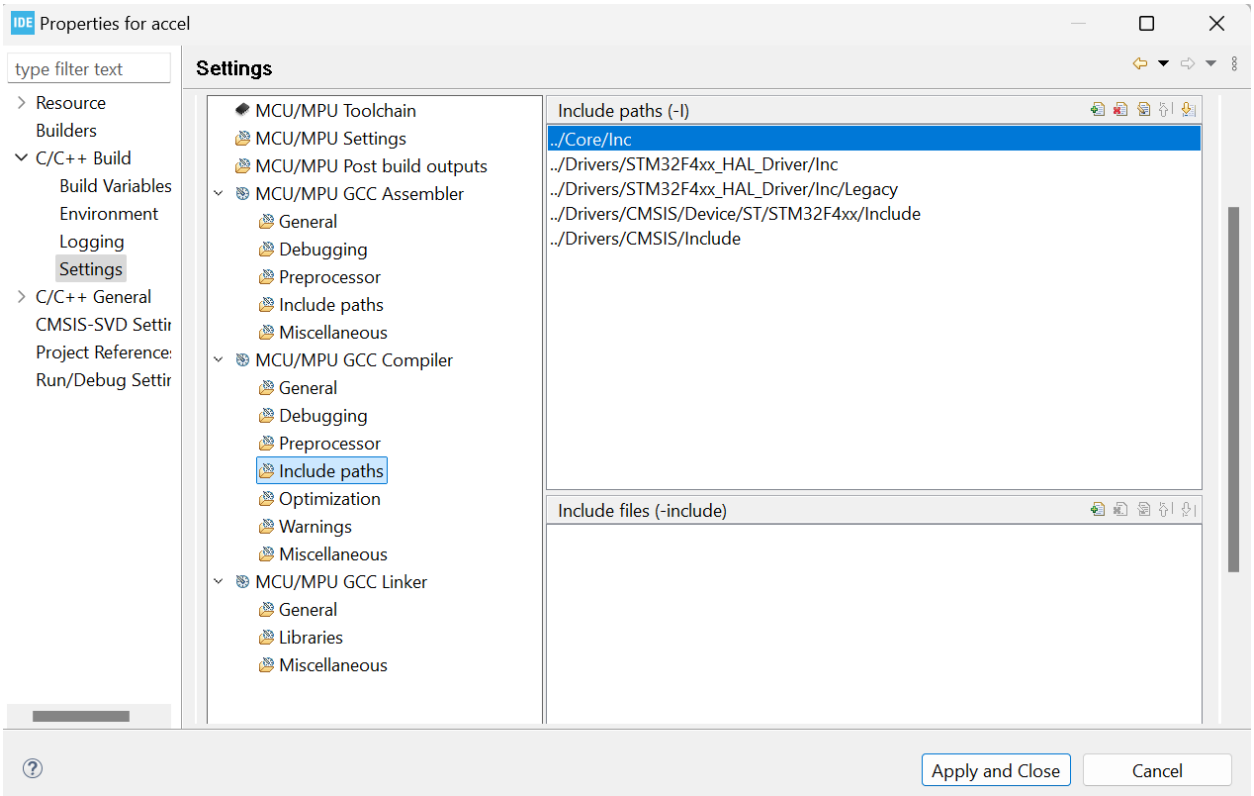
From the `STM32F4-Discovery` folder, you may remove two files related to audio, which are not used in this project.



Since the files in your project folder have changed, right-click on the project name `accel` in the Project Explorer in STM32CubeIDE and choose Refresh.

Tip: Whenever you add or delete files in your project folder outside of STM32CubeIDE, right-click your project name and refresh.

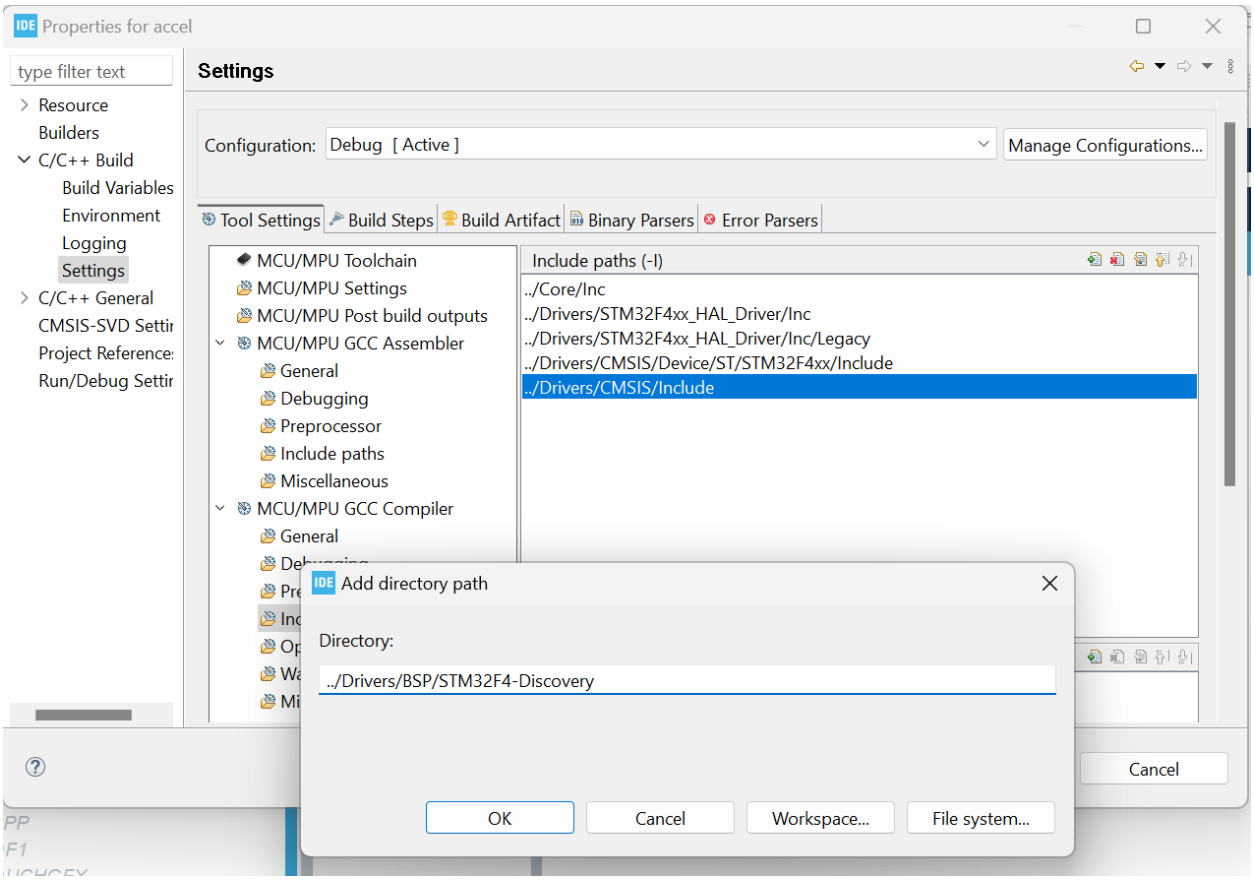
You must direct the compiler to the new `BSP` folder so that the code for the accelerometer can be used. Right-click on the project name `accel` and choose Properties. Go to C/C++ Build – Settings – MCU/MPU GCC Compiler – Include Paths.



All of the Include paths listed refer to folders needed for project compilation. The paths are relative to the Debug folder in your project file structure (as may be seen in C/C++ Build/Environment, where current working directory CWD is listed).

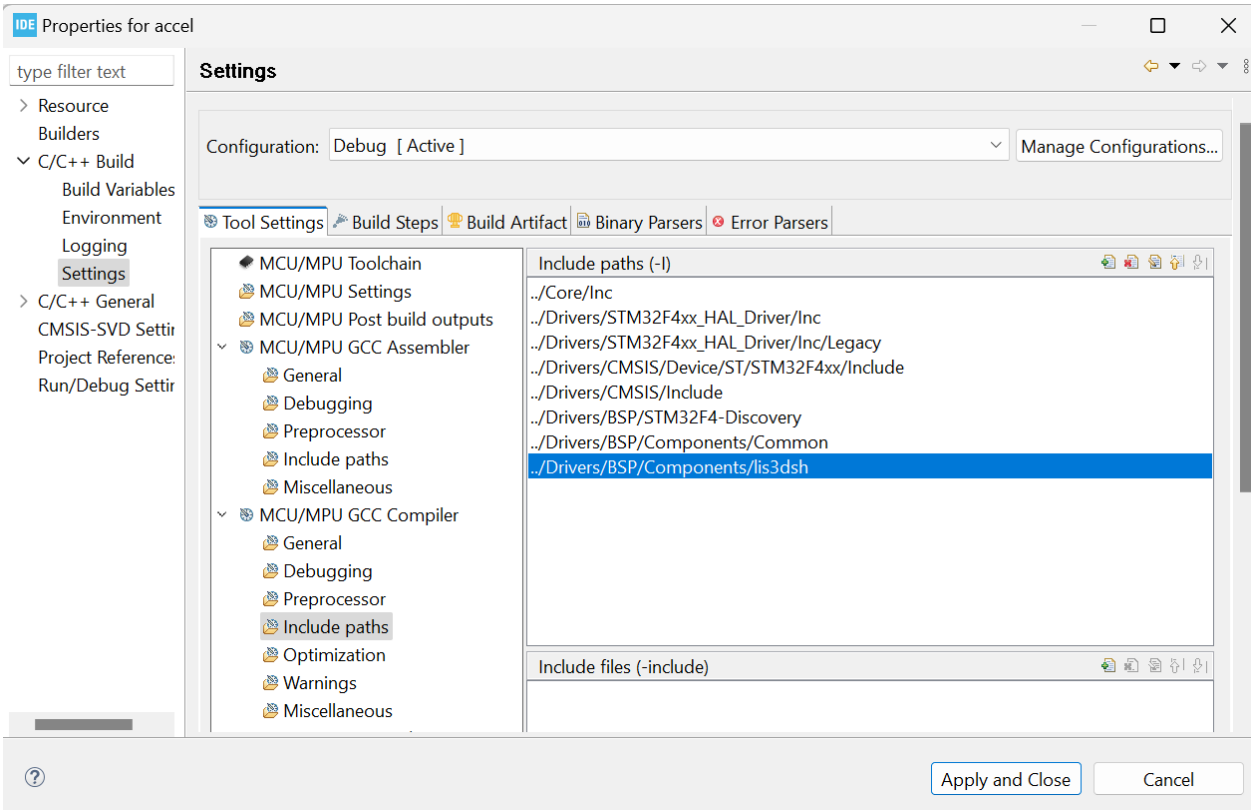
Click on the last include path. Click the green + above the Include paths window. Enter the include path:

```
../Drivers/BSP/STM32F4-Discovery
```



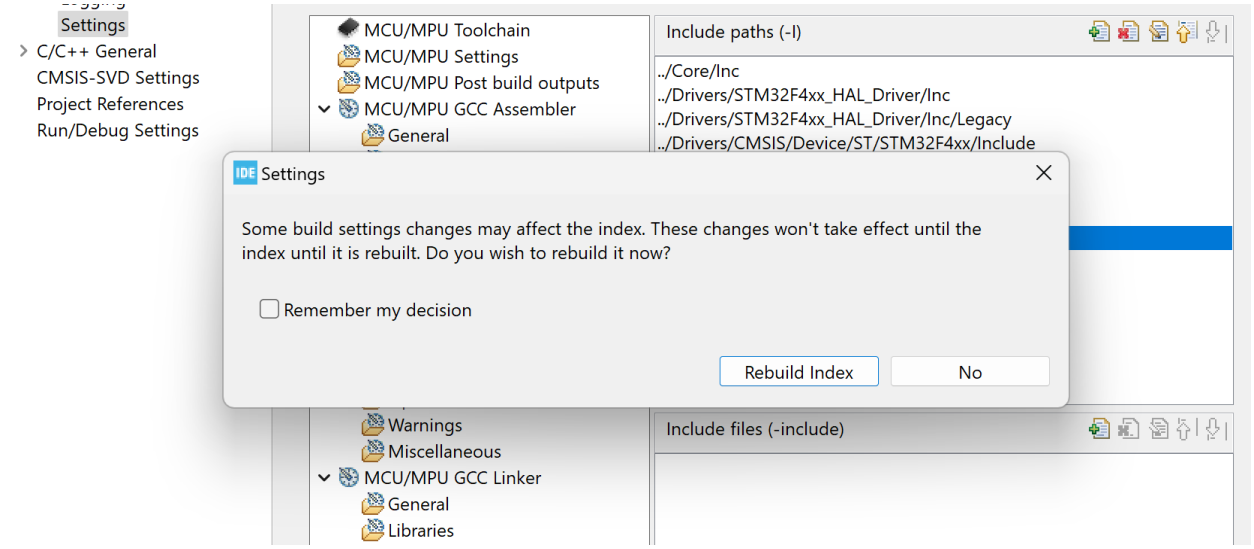
Also add the Include paths:

```
../Drivers/BSP/Components/Common
../Drivers/BSP/Components/lis3dsh
```



Click Apply and Close.

If asked, rebuild the index.



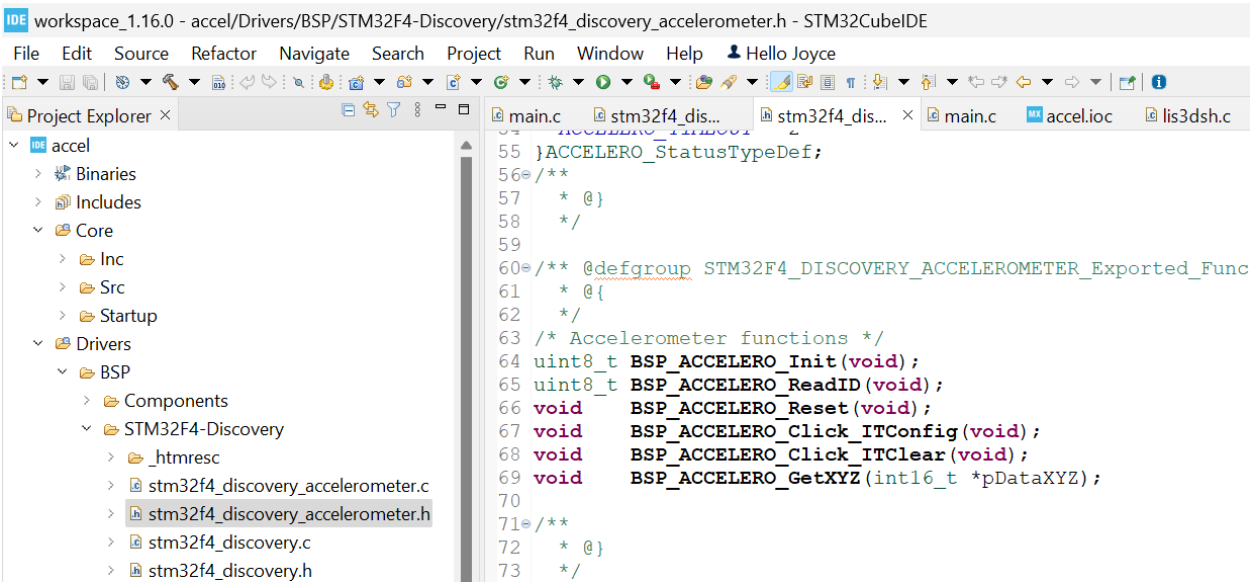
Compile your project to ensure it is error-free.

In `USER CODE` Private Includes area, add the line

```
#include "stm32f4_discovery_accelerometer.h"
```

```
23
24 /* Private includes ----------------------------------
25 /* USER CODE BEGIN Includes */
26 #include "stm32f4_discovery_accelerometer.h"
27 /* USER CODE END Includes */
28
```

Open the `stm32f4_discovery_accelerometer.h` file. The accelerometer functions are listed.



In `USER CODE` area 2, initialize the accelerometer:

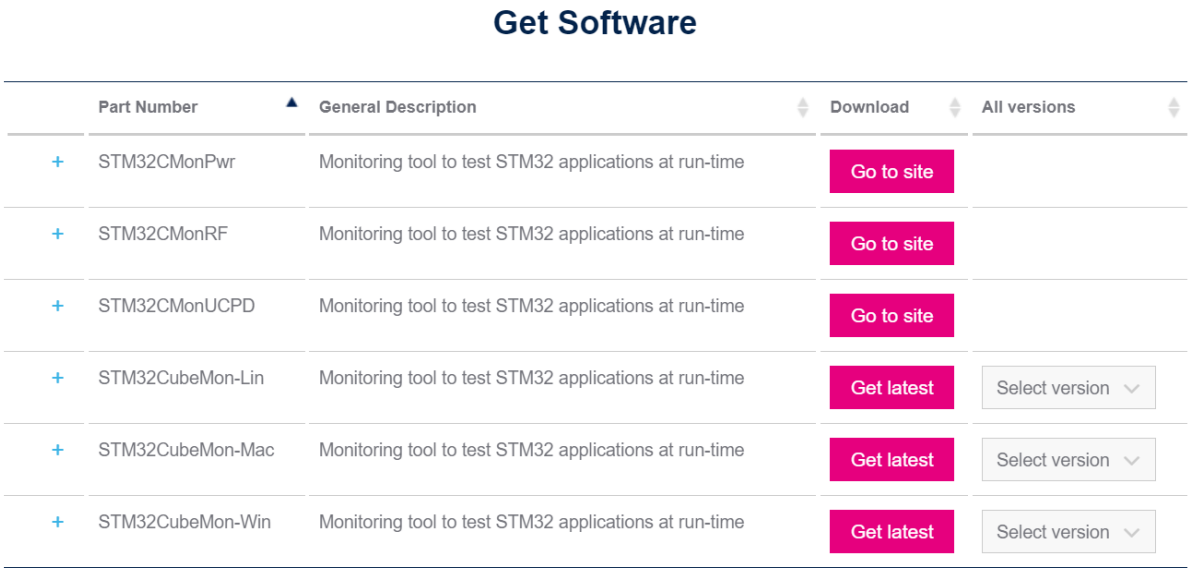
```
BSP_ACCELERO_Init();
```

Within your `while(1)` loop, begin typing `BSP_ACCELERO_GetXYZ()`. Hover over the function name to see that the argument to the function is an `int16_t` array called `xyz` with three elements, one for X, Y and Z. For convenience, add three `int16_t` variables:

```
myx = xyz[0];
myy = xyz[1];
myz = xyz[2];
```

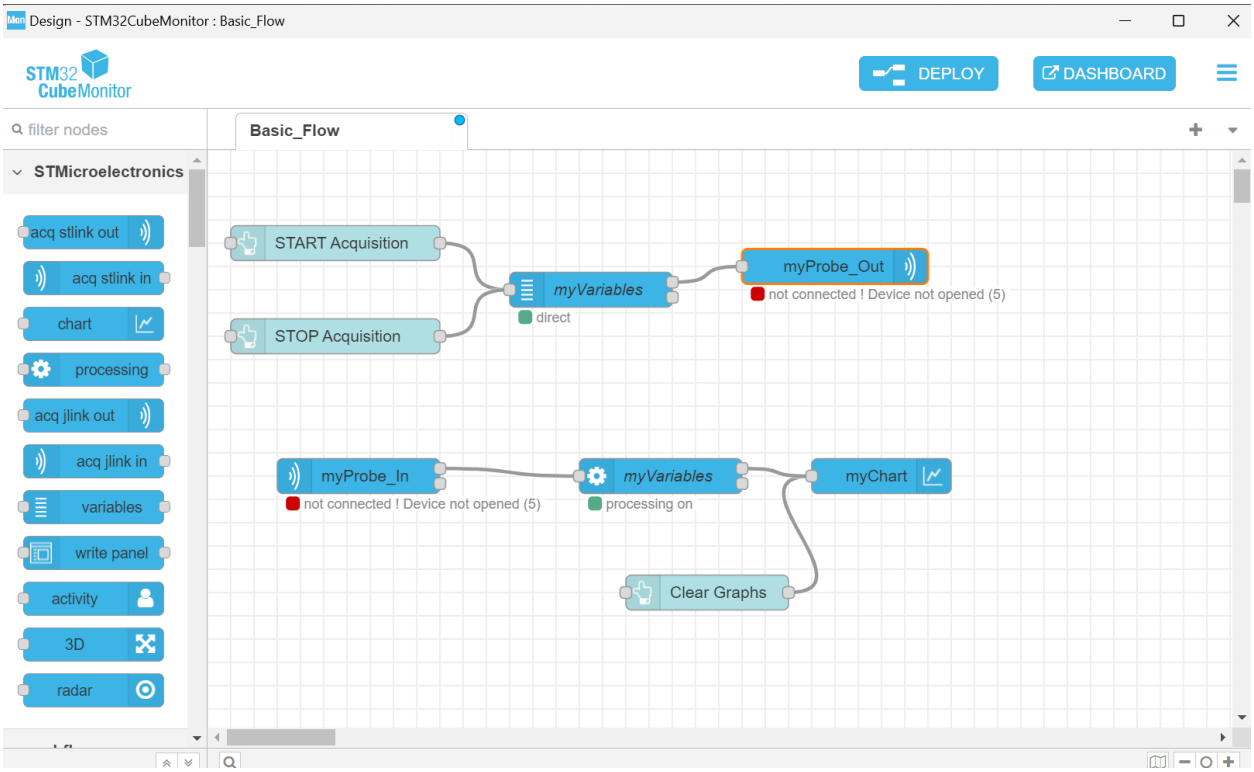
Challenge: Declare an array in the `USER CODE` private variables area, and pass the array name `xyz` to `BSP_ACCELERO_GetXYZ`. Compile and debug your project. Enter the name of the array `xyz` and/or the `myx`, `myy`, `myz` variables, in the Live Expressions window. Play – Resume. Verify that the X, Y and Z components of acceleration change as you rotate the board around different axes. An acceleration of 1g appears as an integer of about 1000. When an axis reports 1000, flipping the board should change the report to about -1000.

The tool STM32CubeMonitor is very useful for monitoring changing values. It may be downloaded from https://www.st.com/en/development-tools/stm32cubemonitor.html. Download and install the latest Windows version.

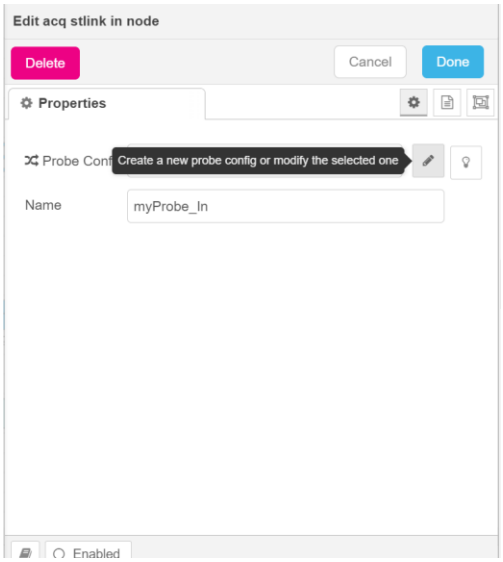


STM32CubeMonitor is an alternative to the ST-LINK Debugger that has been used in all the projects so far. Only one of these can run at a time.

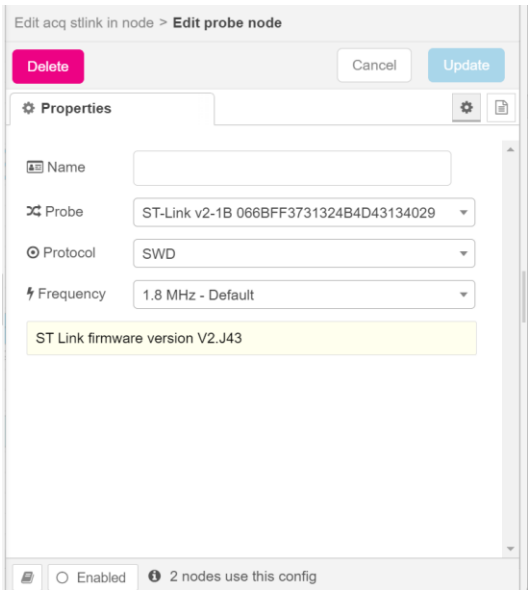
When STM32CubeMonitor opens, it will not be connected to your board. (If you see a red triangle, see comments below.)

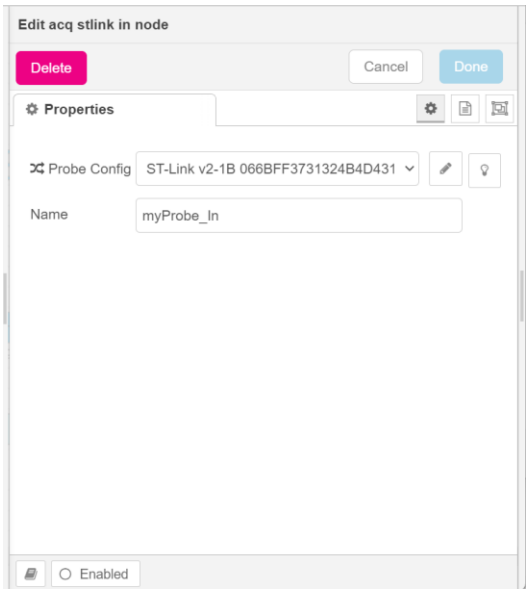


Double-click on myProbe_Out and then double-click on the edit button.

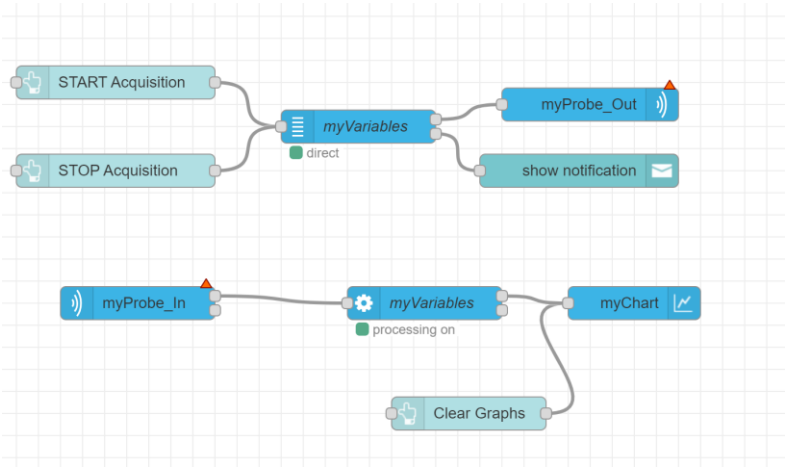


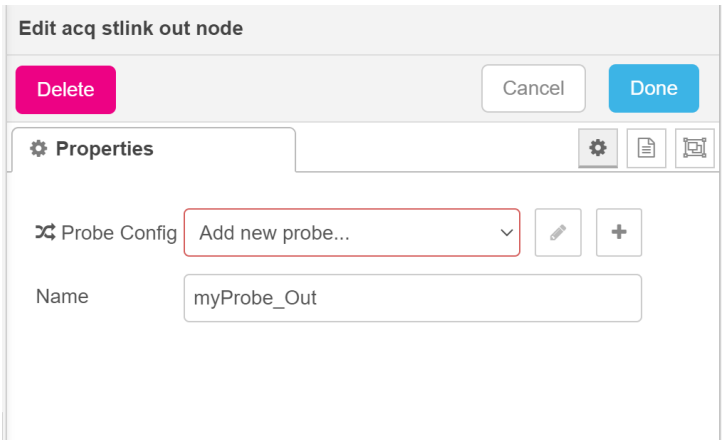
Click Update and Done.

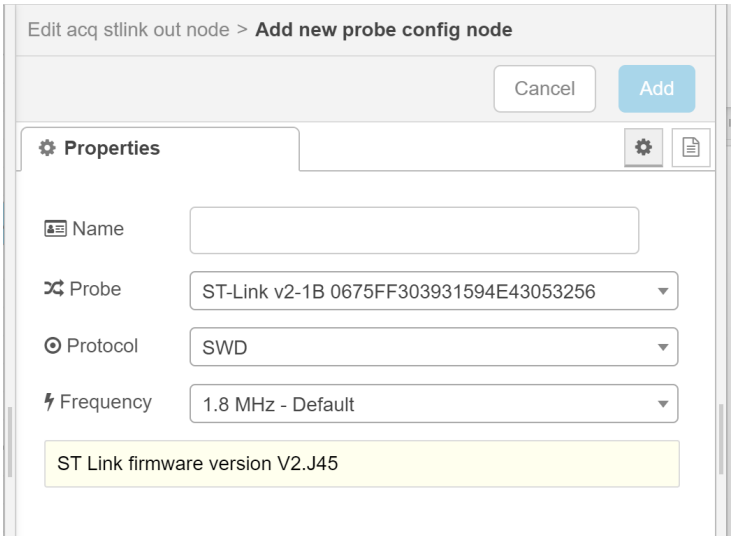




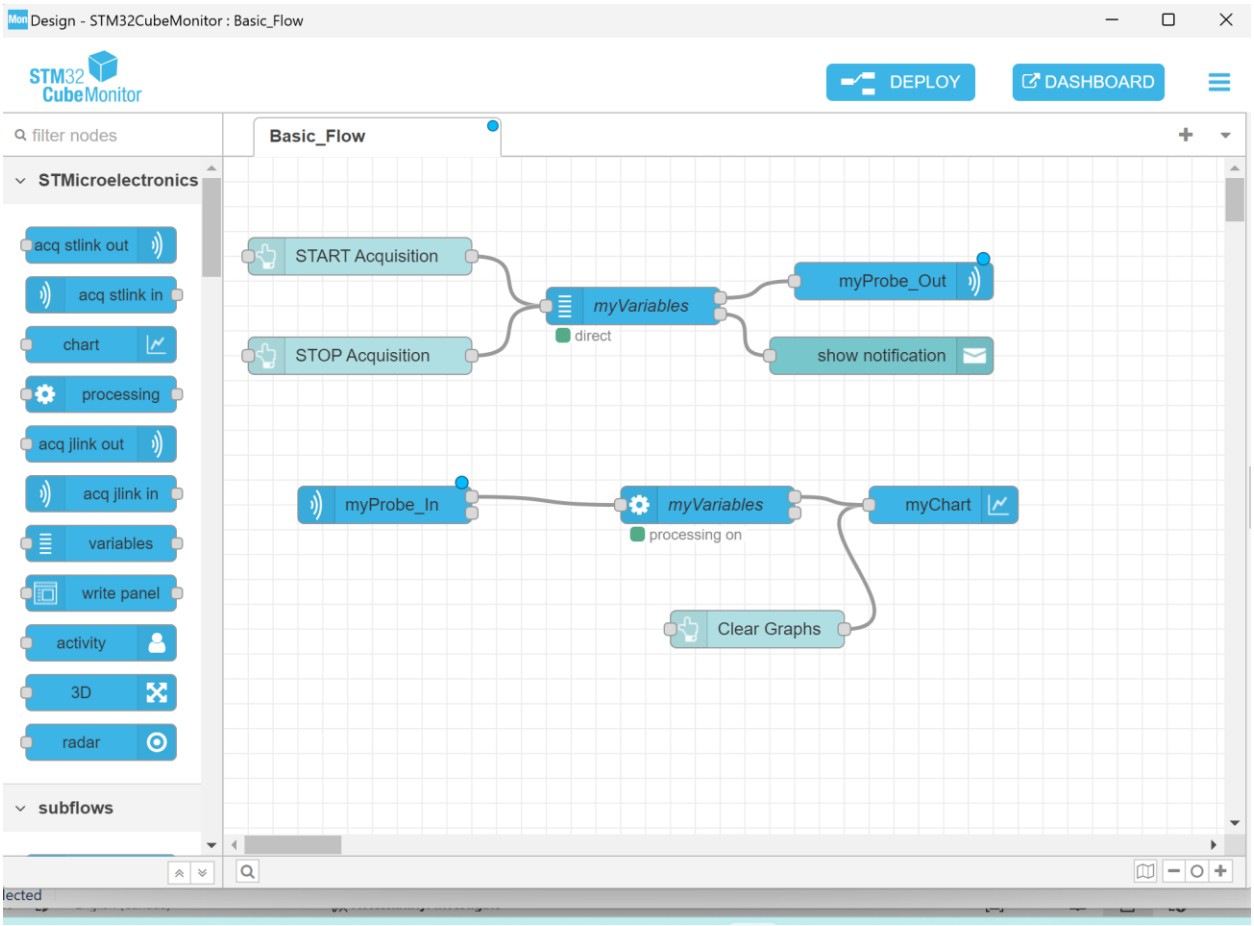
If when you open STM32CubeMonitor, myProbe_Out shows a red triangle, double-click on it, click + sign to add a probe, select the probe, and click Add.



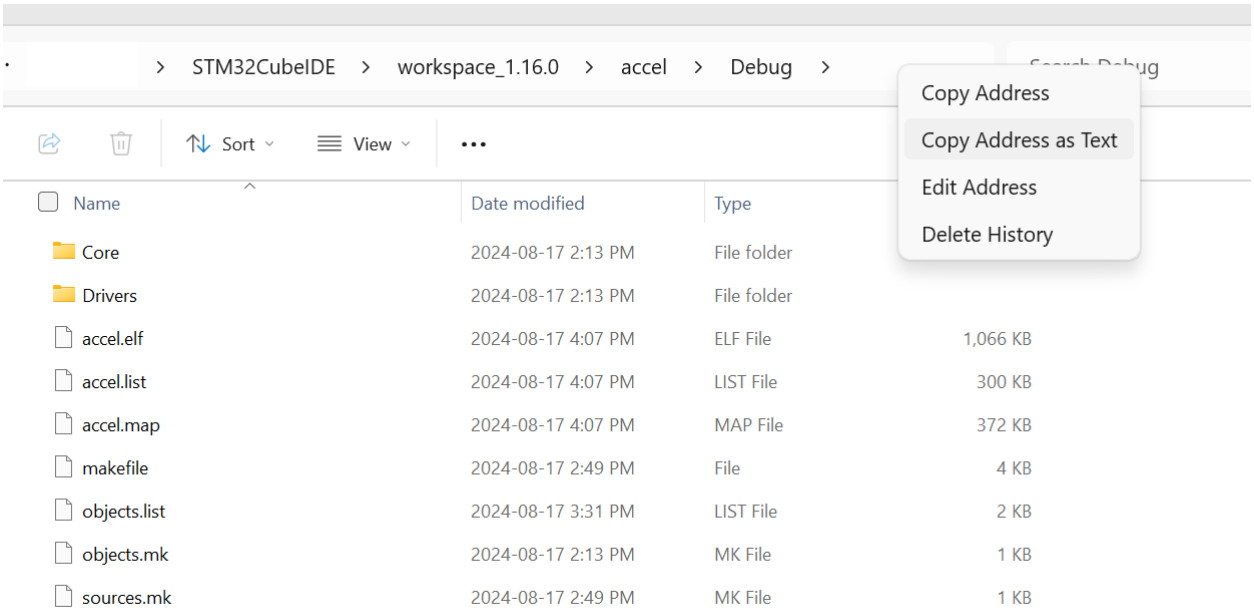




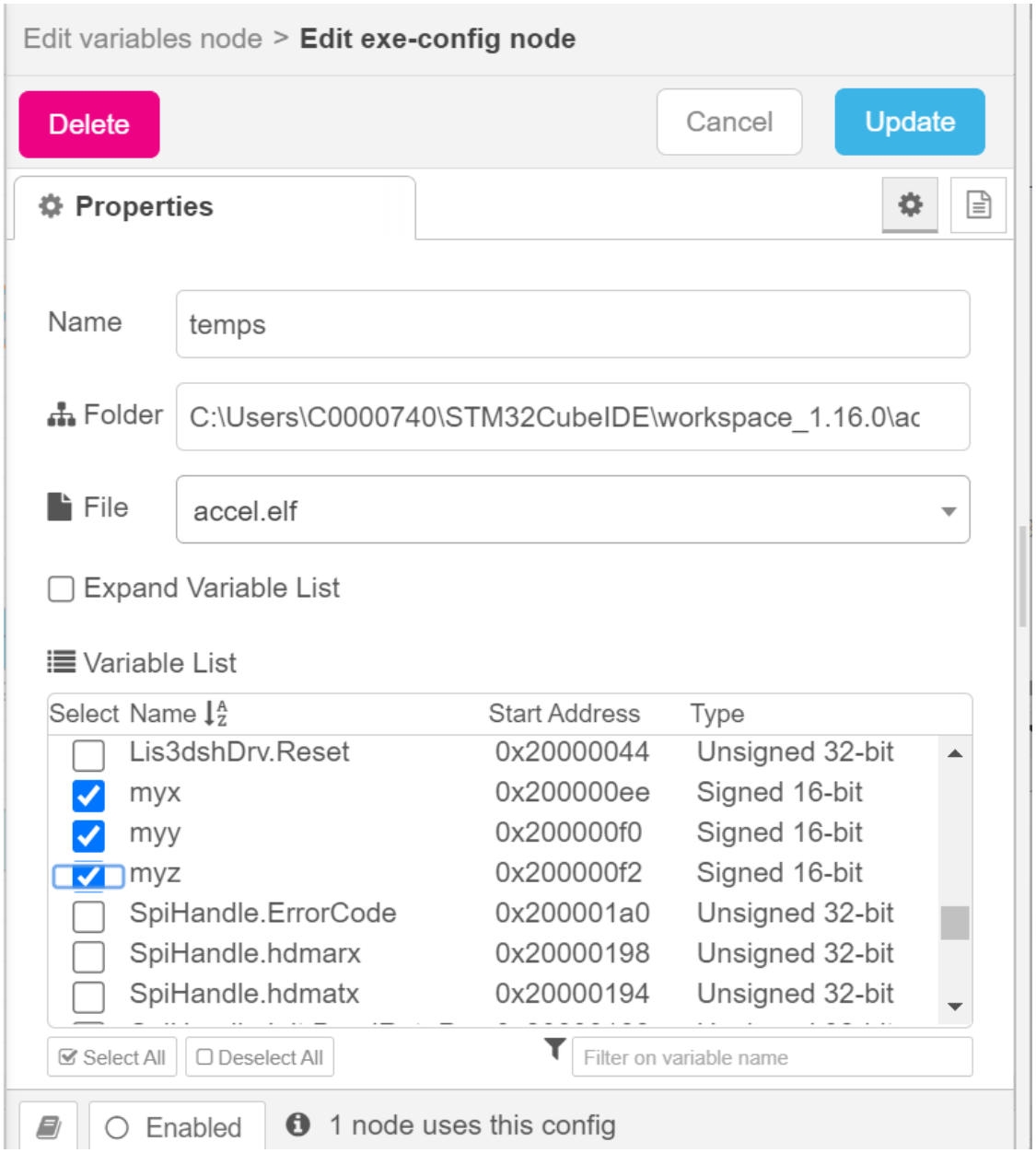
If necessary select the same probe for MyProbe_In. The red alerts should be gone.



Go to your File Explorer and navigate to the Debug folder of your `accel` project. Copy Address as Text.

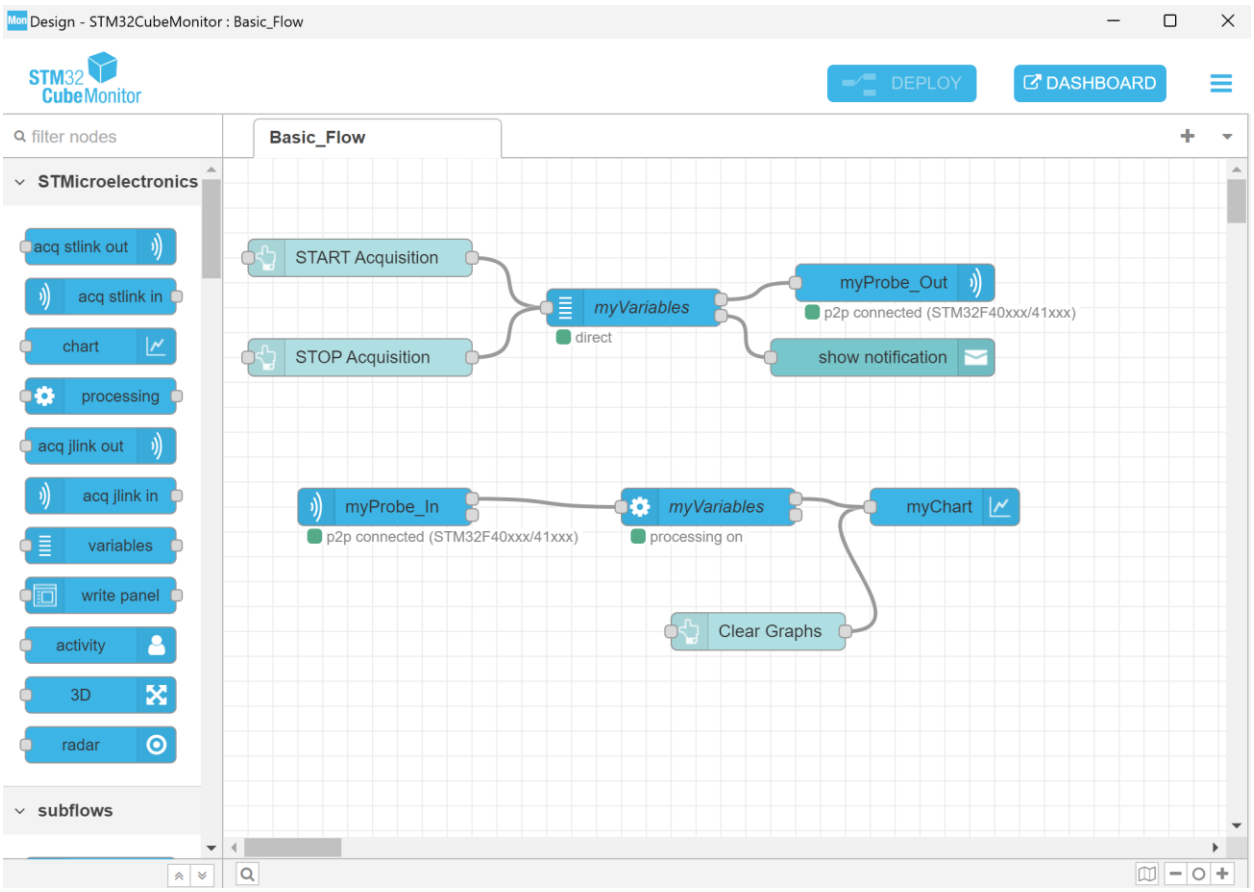


Go back to STM32CubeMonitor. Double-click on myVariables in the top line. STM32CubeMonitor requires a `.elf` file as an executable. Click edit beside the Executable box. If edit is greyed out, click + instead. Enter a name. Paste the path to your Debug folder in the Folder box. Select `accel.elf` for your File box. Once STM32CubeMonitor recognizes your project, all of the project variables will be listed. Check off `myx`, `myy`, and `myz` variables. Click Update (or Add) and Done.



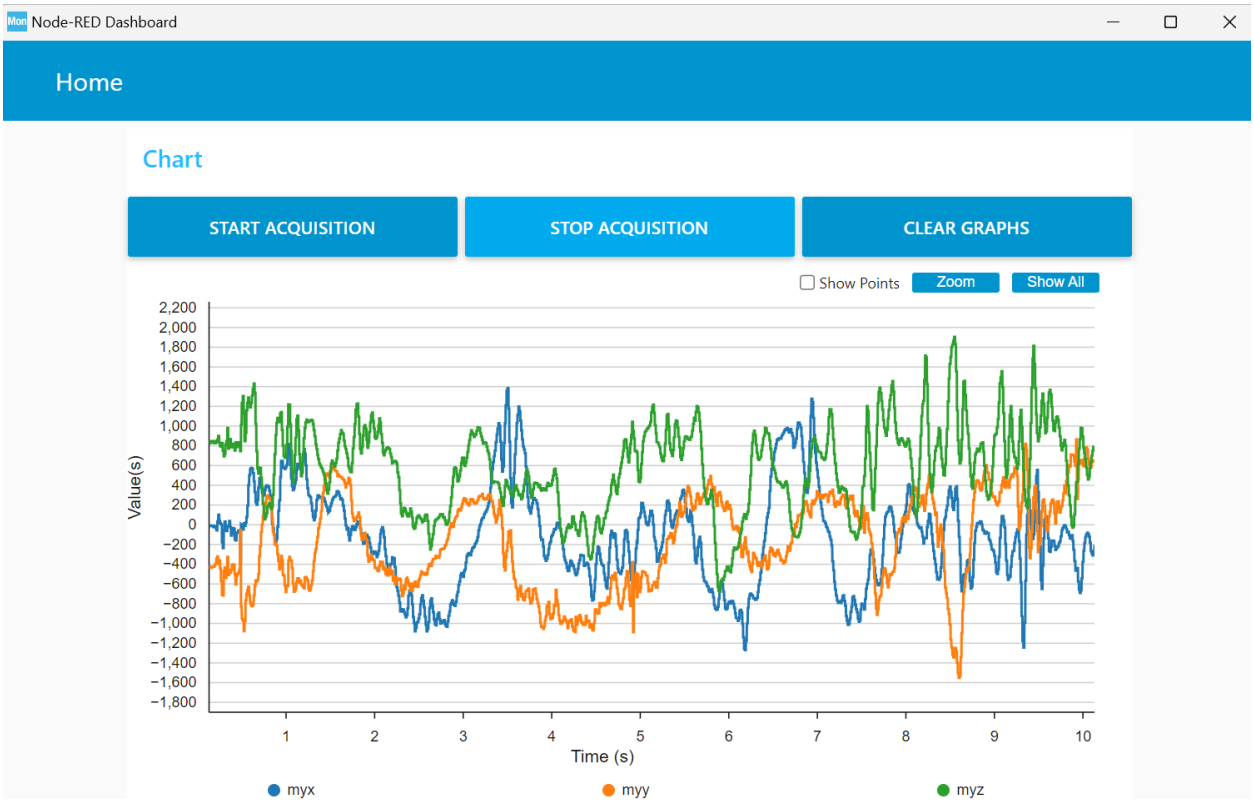
Click Deploy.

myProbe_In and myProbe_Out should now show connected. If they do not, unplug your board from your PC and plug it back in.



Click Dashboard. You will see buttons for Start Acquisition, Stop Acquisition, and Clear Graphs, which were shown in the flow diagram.

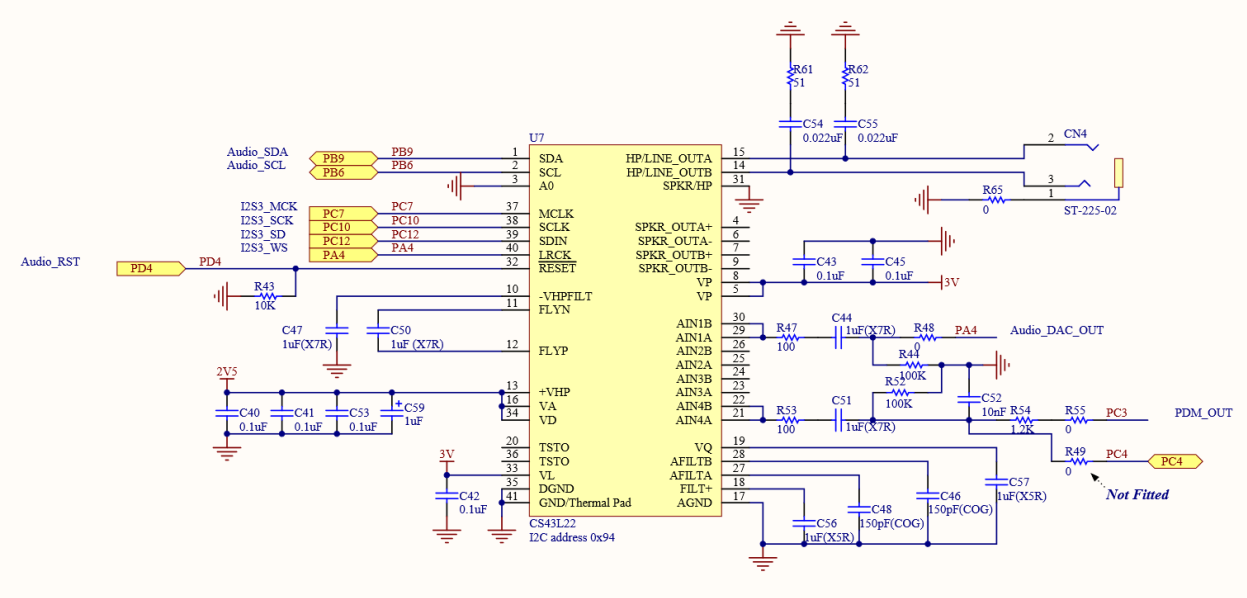
Click Start Acquisition. Rotating the board will produce variations in acceleration for each dimension.


Node-RED Dashboard
Home
Chart
START ACQUISITION
STOP ACQUISITION
CLEAR GRAPHS
Show Points
Zoom
Show All
Value(s)
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
−200
−400
−600
−800
−1,000
−1,200
−1,400
−1,600
−1,800
1
2
3
4
5
6
7
8
9
10
Time (s)
myx
myy
myz

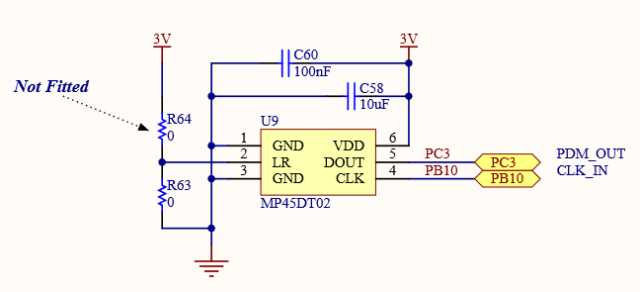
Seventh project: `mic_headphone`

In this project, you will set up the onboard MEMS mic as input and route the signal to the onboard CS43L22 DAC output.

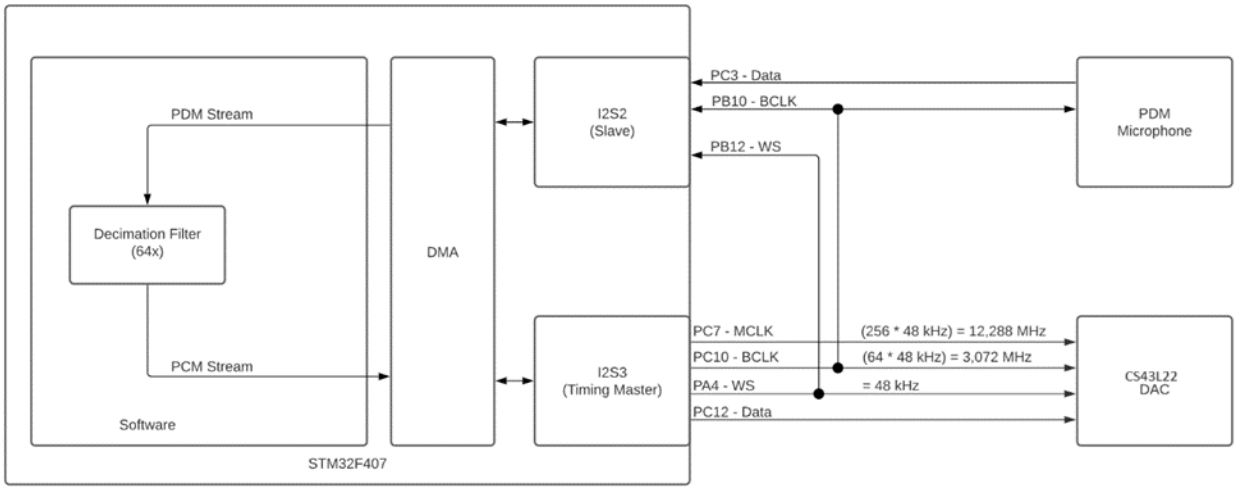
In the schematic for the STM32F407 Discovery board, the important input connections for the CS43L22 DAC are shown. These include Audio_RST on pin PD4, I^2C connections Audio_SDA (pin PB9) and Audio_SCL (pin PB6), and I^2S connections on pins PC7, PC10, PC12, and PA4.



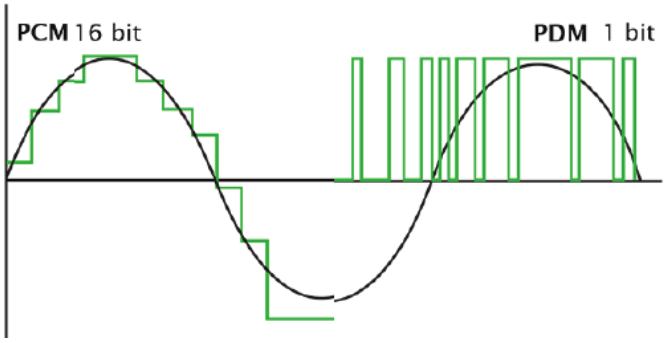
The schematic also shows the important output connections for the MEMS mic, data on PC3 and clock on PB10.



The connections required for the project are shown in the block diagram. The CS43L22 DAC acts as master and the PDM microphone acts as slave. Note that two external wire connections will be needed on the board: between PA4 and PB12, and between PC10 and PB10. These wires will be added later.



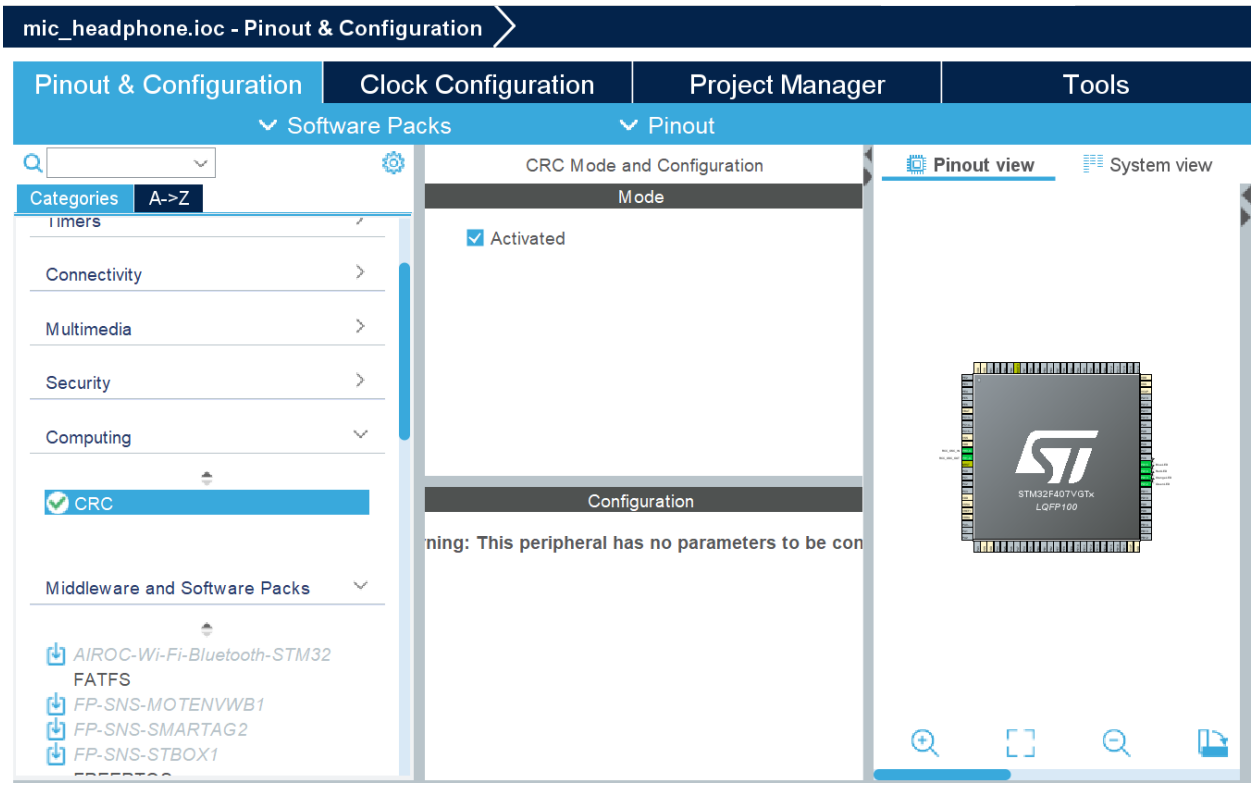
The MEMS mic produces a PDM (pulse density modulation) signal. PDM uses 1-bit sigma-delta modulation. The PDM signal must be converted to PCM (pulse code modulation) for processing. This requires a decimation filter, functionality that is available in the STM32 environment. More details are available at https://yatian-liu.github.io/public/PDM_PCM_Signal_Conversion_FPGA.pdf and https://www.st.com/resource/en/user_manual/um2372-stm32cube-pdm2pcm-software-library-for-the-stm32f4f7h7-series-stmicroelectronics.pdf.



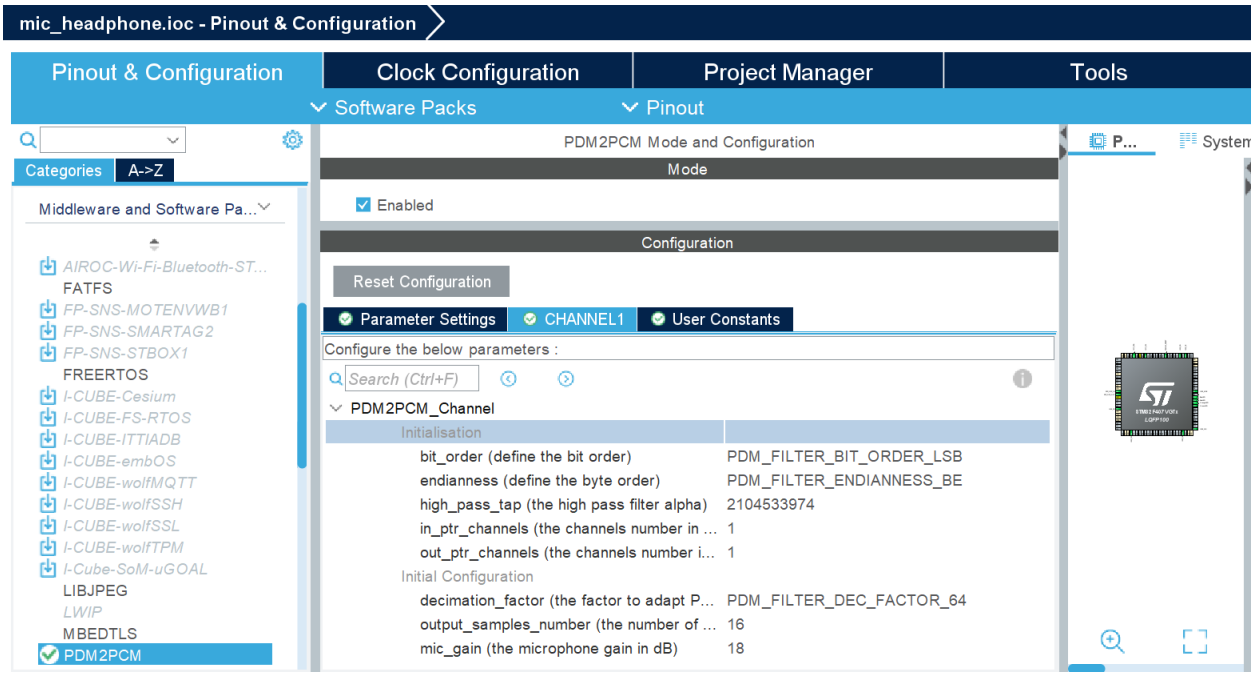
Create a new STM32 project based on `blink_LED.ioc`, and call the new project `mic_headphone`.

Under the Security category, deactivate RNG, as it will not be needed in this project.

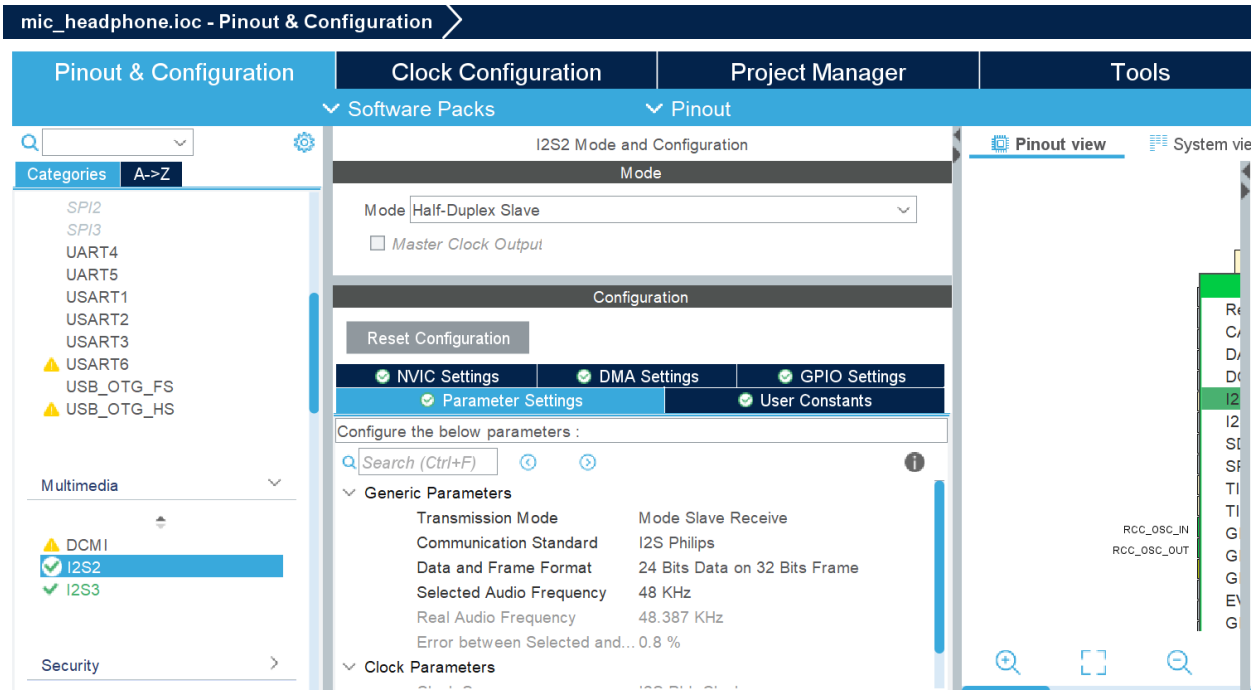
In the Computing category, click CRC, and activate. CRC is used by PDM2PCM.



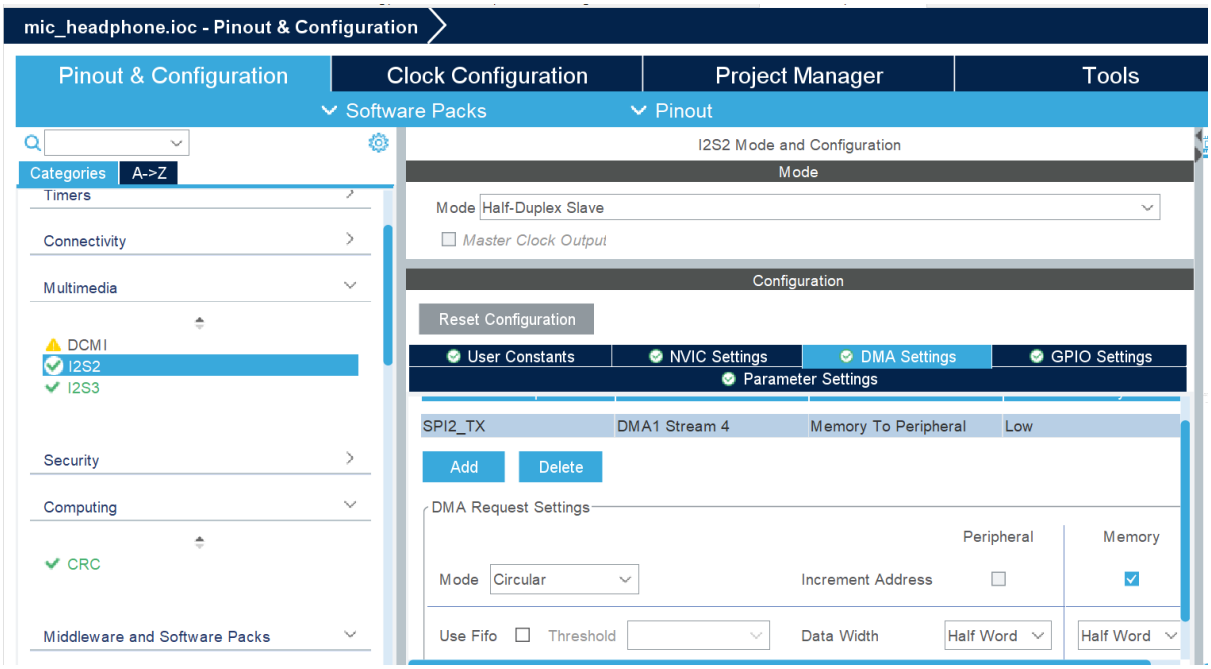
In the Middleware and Software Packs category, click PDM2PCM, and enable. Change the number of input and output channels to 1. Note the default 64x decimation factor, and the number of output samples produced by each call to the PDM filter, which is set to 16. Change the microphone gain to 18.



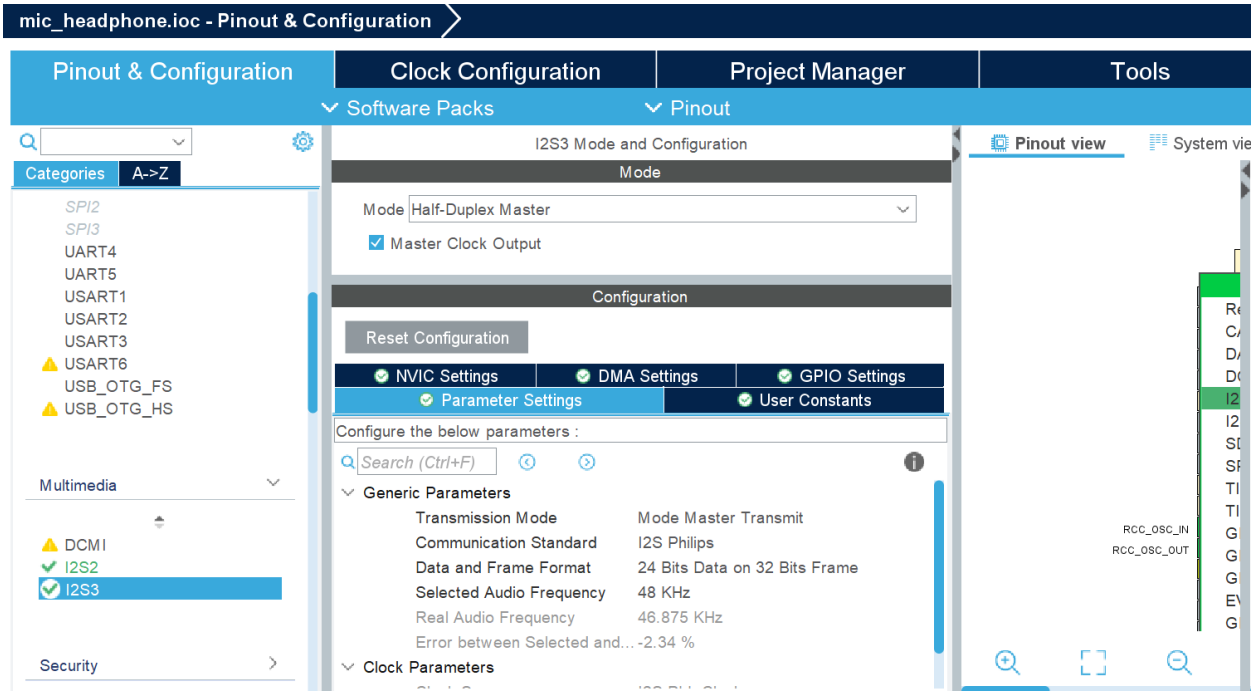
In the Multimedia category, choose I2S2. This connection will feed PDM data from the MEMS mic to the microphone. Choose Half Duplex – Slave mode. Choose Mode Slave Receive, Communication Standard I2S Philips, Data and Frame Format 24 Bits Data on 32 Bits Frame (to serve the resolution of the CS43L22 DAC), and Selected Audio Frequency 48 kHz. Note the Real Audio Frequency that is slightly different from the Selected Audio Frequency. It is not possible to obtain a desired frequency with perfect accuracy, but it is possible to make changes to the Clock Configuration to make the two frequencies closer, if required. The decimation factor of 64 means that the clock for the PDM MEMS mic will be running at (64)(48 kHz) = 3.072 MHz.



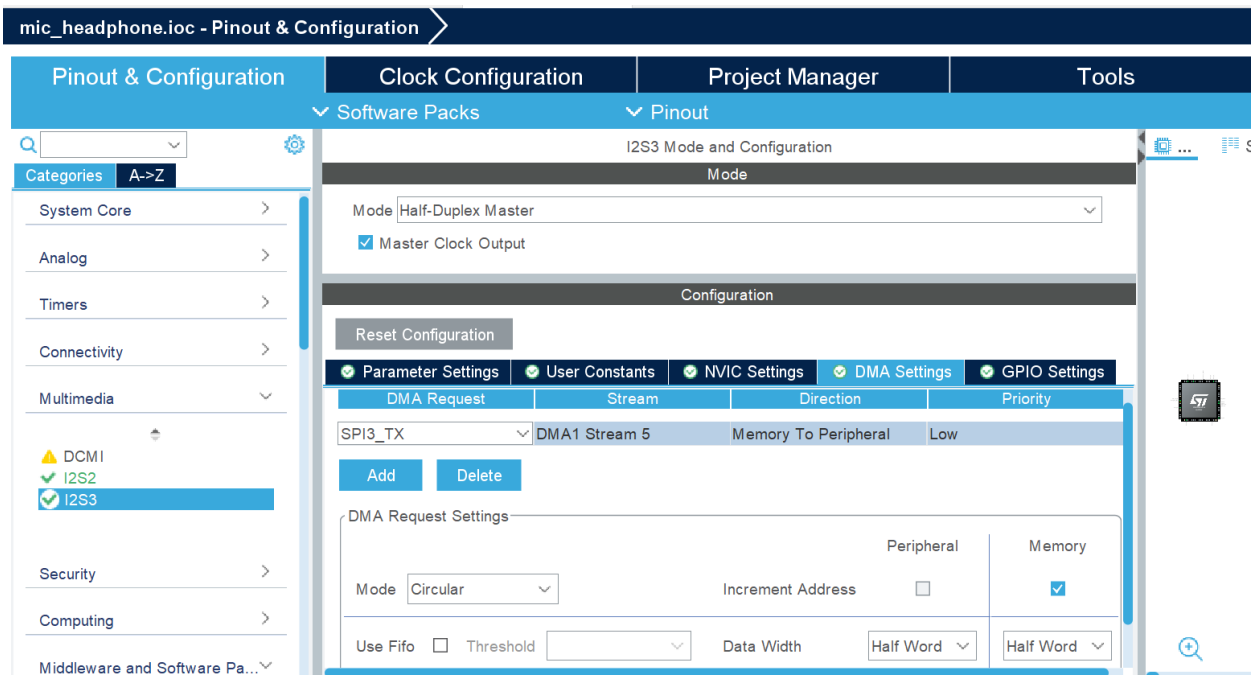
For I2S2, set up DMA in Circular Mode, with Half-Word Data Width. This will match the size of the data declared in `main.c`.



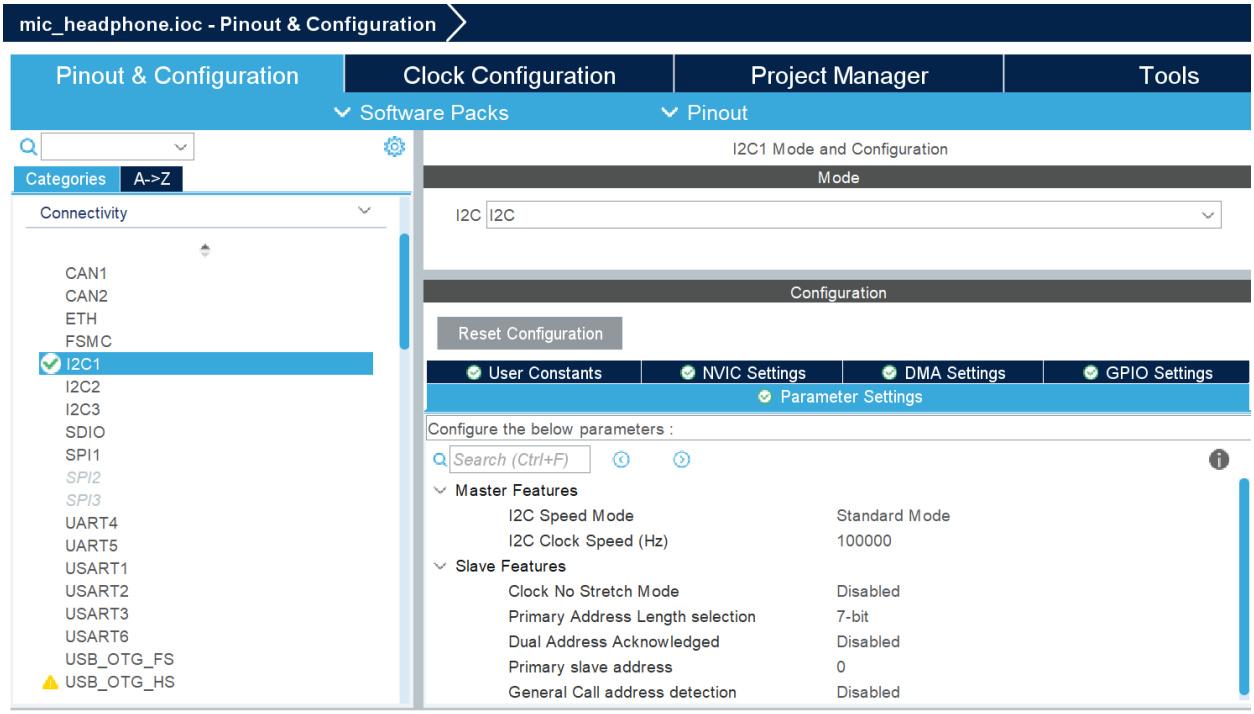
Still in the Multimedia category, choose I2S3. This connection will feed data from the microcontroller to the DAC. Choose Half Duplex – Master mode. Check off Master Clock Output. Choose Communication Standard I2S Philips, Data and Frame Format 24 Bits Data on 32 Bits Frame, and Audio Frequency 48 kHz.



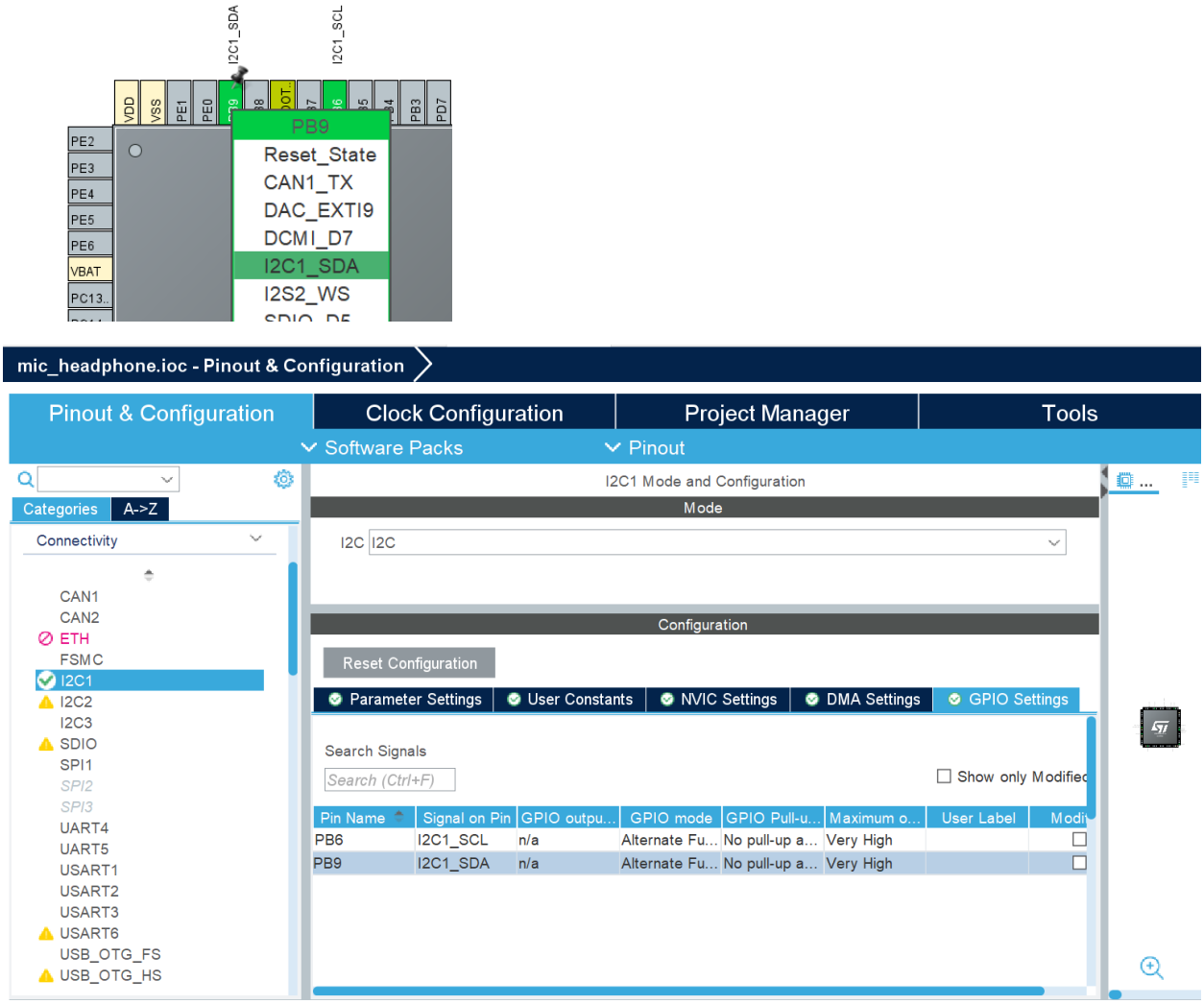
For I2S3, set up DMA in Circular Mode, with Half-Word Data Width.



In the Connectivity category, choose I2C1. Set I2C Mode.

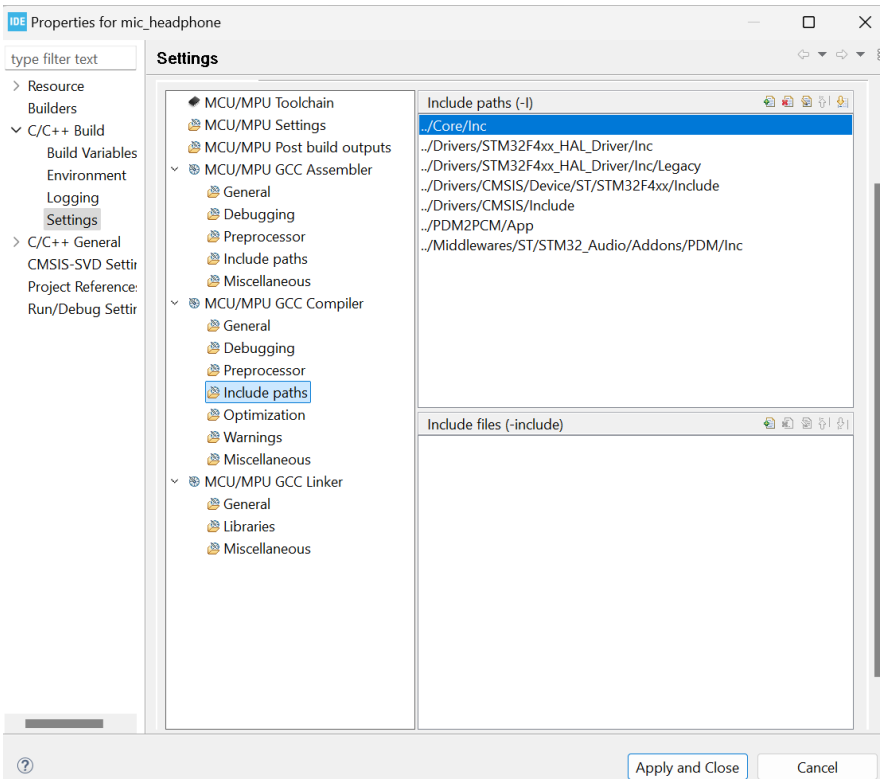


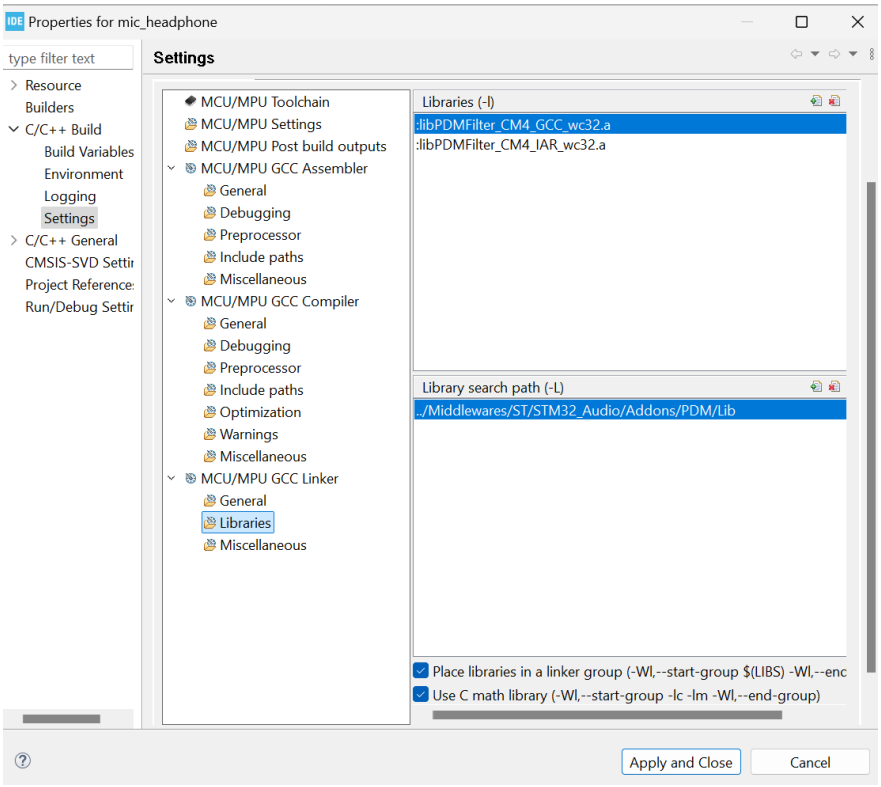
Click on the GPIO Settings for I2C1. The two pins assigned to I2C1 should be PB6 and PB9, as shown in the schematic. The configuration file does not always seem to assign PB9 correctly. If this happens, open the Pinout View, click on PB9, and choose I2C1_SDA.



Finally, click on pin PD4 in the Pinout View. Choose GPIO Output. Enter the user label CS43L22_RESET.

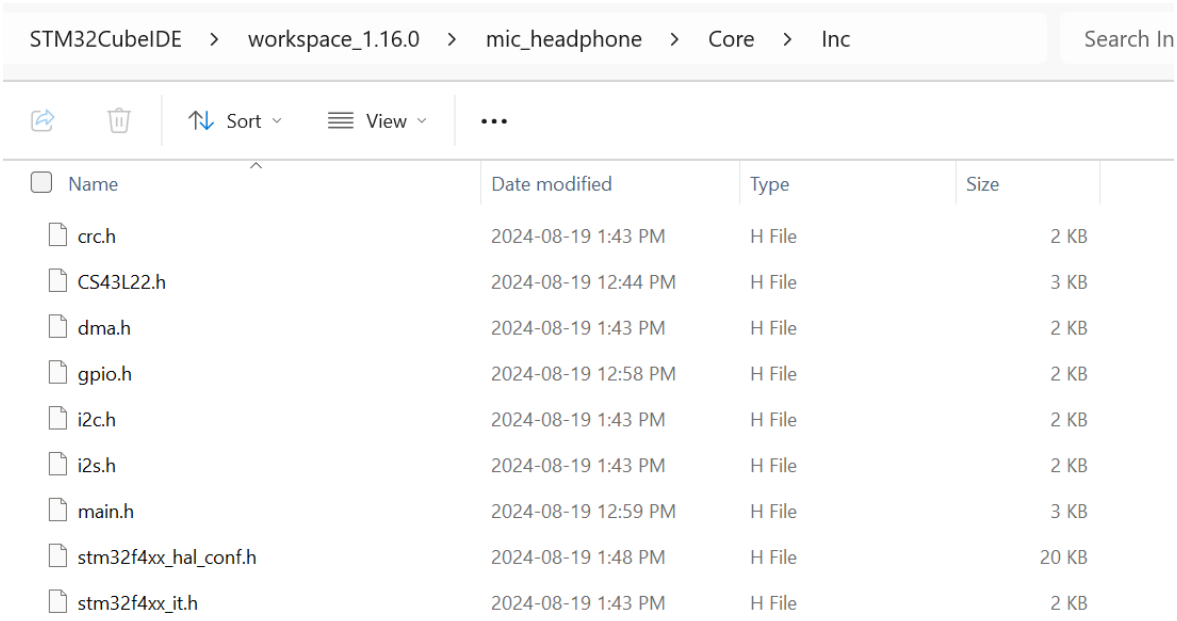
Generate code for your project. Right-click on the project name `mic_headphone` in the Project Explorer and choose Properties. The selection of PDM2PCM in the `.ioc` file created new entries among the Include paths, and also new libraries.



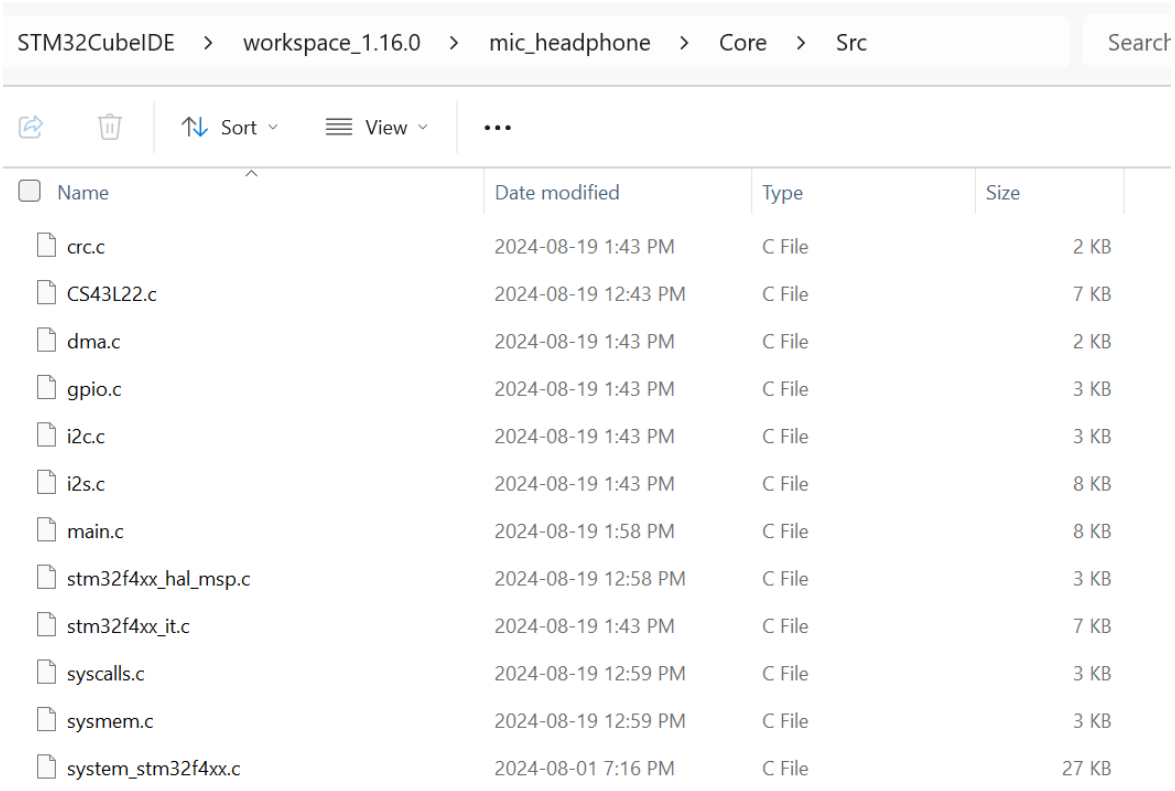


Two files are needed to interface with the CS43L22 DAC: `CS43L22.h` and `CS43L22.c`. These files are provided in Appendix 2. Copy the contents of each file into a text editor and save.

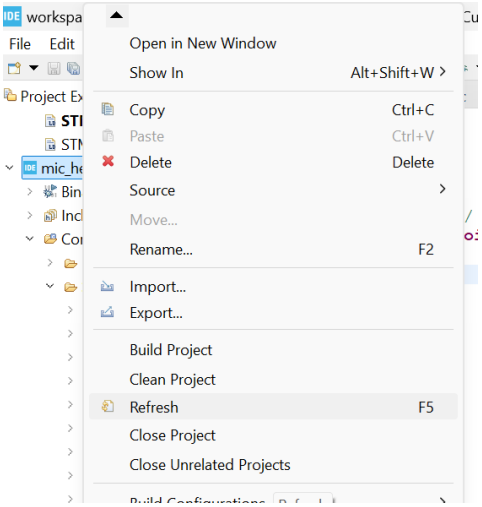
In your File Explorer, place `CS32L22.h` in the `Core/Inc` directory of your project.



In your File Explorer, place `CS43L22.c` in the `Core/Src` directory of your project.



Go to STM32CubeIDE. Right-click on your project name `mic_headphone` and click Refresh, so that the two new files are incorporated into the project listing.



In the `USER CODE` includes area, add the line:

```
#include "CS43L22.h"
```

In the USER CODE Private Defines area, add the line:

```
#define N 32
```

In the `USER CODE` Private Variables area, add the lines:

```
uint16_t txBuf[4*N];
uint16_t pdmRxBuf[4*N];
uint16_t MidBuffer[16];
uint8_t txstate = 0;
uint8_t rxstate = 0;

uint16_t fifobuf[8*N];
uint8_t fifo_w_ptr = 0;
uint8_t fifo_r_ptr = 0;
uint8_t fifo_read_enabled = 0;
```

In `USER CODE` area 0, add the lines:

```
void FifoWrite(uint16_t data) {
      fifobuf[fifo_w_ptr] = data;
      fifo_w_ptr++;
}

uint16_t FifoRead() {
      uint16_t val = fifobuf[fifo_r_ptr];
```

```
      fifo_r_ptr++;
      return val;
}

void HAL_I2S_TxHalfCpltCallback(I2S_HandleTypeDef *hi2s) {
      txstate = 1;
}

void HAL_I2S_TxCpltCallback(I2S_HandleTypeDef *hi2s) {
      txstate = 2;
}

void HAL_I2S_RxHalfCpltCallback(I2S_HandleTypeDef *hi2s) {
      rxstate = 1;
}

void HAL_I2S_RxCpltCallback(I2S_HandleTypeDef *hi2s) {
      rxstate = 2;
}
```

The callback functions are executed every time a half or full DMA buffer is complete, either for the receiving end (from the microphone) or for the transmitting end (to the DAC).

In `USER CODE` area 2, add the lines:

```
  CS43_Pin_RST_Init();
  CS43_Init(hi2c1, MODE_I2S);
  CS43_SetVolume(40); // 0 - 40
  CS43_Enable_RightLeft(CS43_RIGHT_LEFT);
  CS43_Start();

  HAL_I2S_Transmit_DMA(&hi2s3, (uint16_t *) txBuf, 2*N);
  HAL_I2S_Receive_DMA(&hi2s2, (uint16_t *) pdmRxBuf, 2*N);
```

These lines initialize the CS43L22 DAC and also initialize the DMAs. It is important that the master, the DAC DMA, be initialized before the slave, the mic DMA.

Inside your `while(1)` loop, in the `USER CODE` area 3, enter the lines:

```
        if (rxstate==1) {

              PDM_Filter(&pdmRxBuf[0],&MidBuffer[0], &PDM1_filter_handler);
              for (int i=0; i<16;i++) { FifoWrite(MidBuffer[i]); }
              if (fifo_w_ptr-fifo_r_ptr > 4*N) fifo_read_enabled=1;
              rxstate=0;
        }
        if (rxstate==2) {
              PDM_Filter(&pdmRxBuf[2*N],&MidBuffer[0], &PDM1_filter_handler);
              for (int i=0; i<16;i++) { FifoWrite(MidBuffer[i]); }
```

```
      rxstate=0;
}
if (txstate==1) {
      if (fifo_read_enabled==1) {
         for (int i=0; i<2*N;i=i+4) {
           uint16_t data = FifoRead();
           txBuf[i] = data;
           txBuf[i+2] = data;
         }
      }
      txstate=0;
}
if (txstate==2) {
      if (fifo_read_enabled==1) {
            for (int i=2*N; i<4*N;i=i+4) {
                 uint16_t data = FifoRead();
                 txBuf[i] = data;
                 txBuf[i+2] = data;
            }
      }
      txstate=0;
}
```

Some explanation: When the DMA buffer receiving values from the mic, `pdmRxBuf`, is half full, `rxstate` is set to `1`, the PDM values are converted to PCM by `PDM_Filter`, and the results are stored in `Midbuffer`. A FIFO buffer is used because, though the clocks are DAC DMA and mic DMA are synchronous, the mic DMA is started just after the DAC DMA. The FIFO buffer ensures a decent amount of data from the mic is written into the FIFO buffer before any attempt is made to transmit data to the DAC.

When the first half of DAC DMA buffer has been transmitted to the DAC, `txstate` is set to 1. Providing enough data from the mic is present in the FIFO buffer, the data is copied into the first half of the DAC DMA buffer `txBuf`. The same data is used for left and right channels of the stereo DAC, since only one channel of data is available from the single mic.

The second half of the DMA buffers are handled in a similar way, with `rxstate` equal to `2` indicating that new data is available from the mic in the `pdmRxBuf` buffer, and `txstate` equal to `2` indicating that data has been transmitted to the DAC and new data can be copied into the `txBuf` buffer.

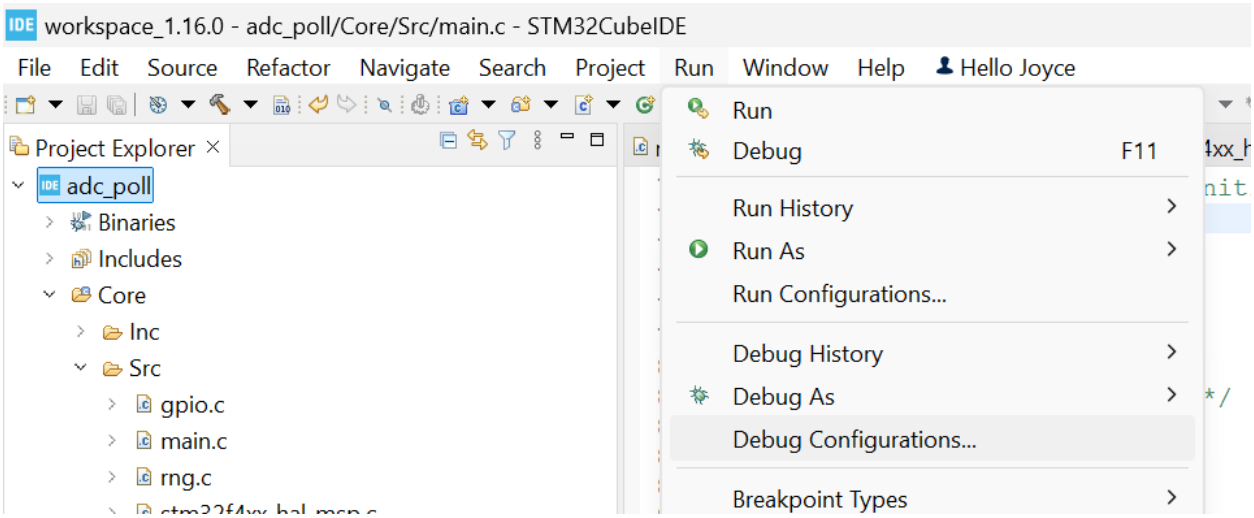
Challenge: Use female-to-female jumper wires to connect PA4 to PB12, and PC10 to PB10. Plug a headset into the headphone jack of the Discovery board. Compile your project. Run – Debug. Play – Resume. Tap on the mic on the board. The MEMS mic is the small silver or black square marked U9, near the micro USB connector. Do you hear the clicks in your headphones? Move as far from the board as your headphones will allow and ask a friend to speak into the mic, or play music from your phone near the mic. Can you hear the signal clearly in your headphones? This project is talk-through; processing of the microphone signal can be added to the program if desired.

Appendix 1 Modifications in STM32CubeIDE to permit use of `printf()`

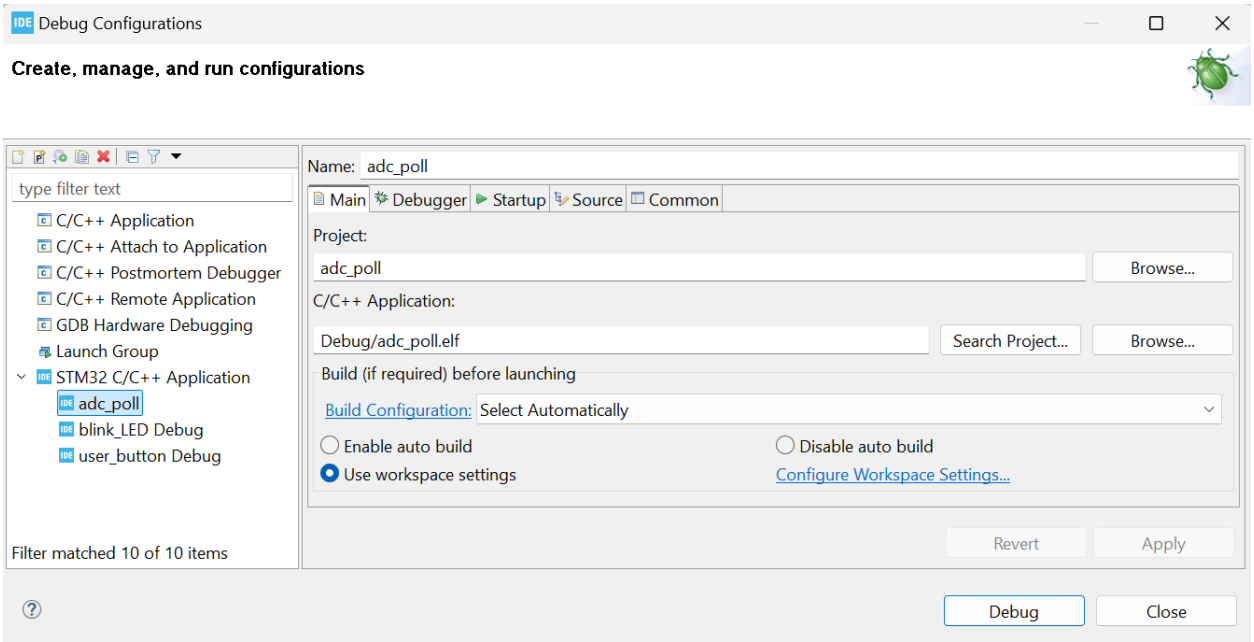
Open your project in STM32CubeIDE.

Run – Debug your project at least once. Stop the debugger.

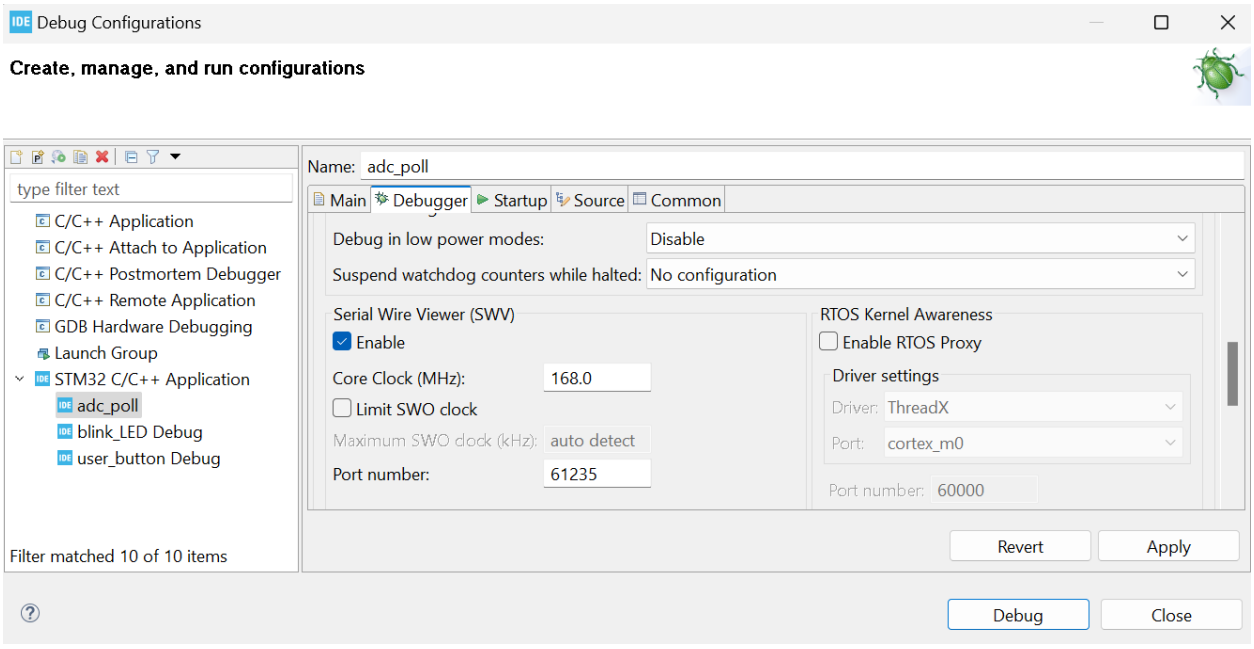
Choose Run – Debug Configurations.....



Make sure your project is selected.

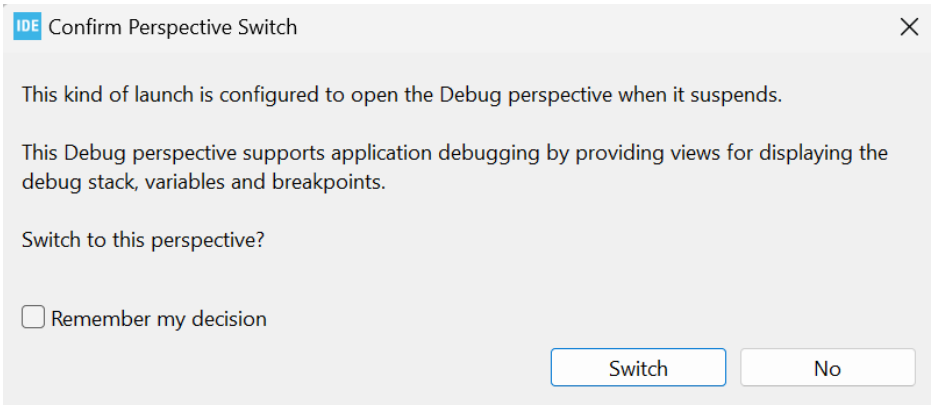


Click on the Debugger tab and scroll down to enable Serial Wire Viewer. Enter the Core Clock (HCLK) from your clock configuration. If you are using the maximum core Core Clock frequency for the STM32F407 Discovery board, this frequency will be 168 MHz.

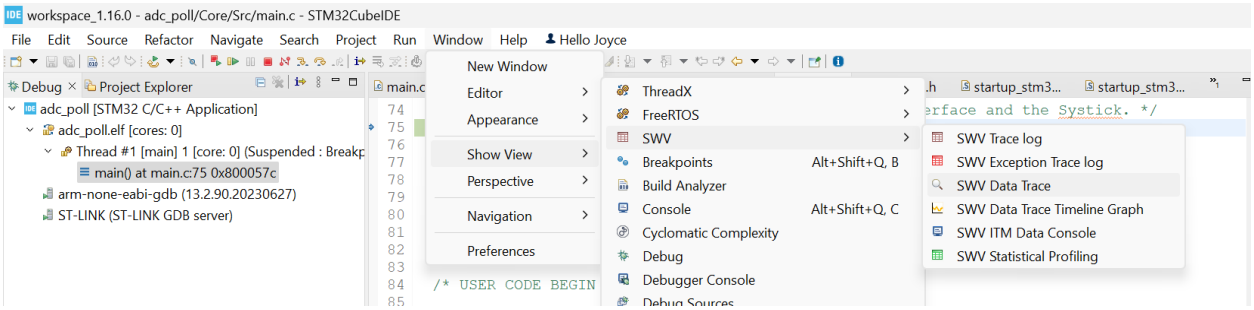


Click Apply and Debug.

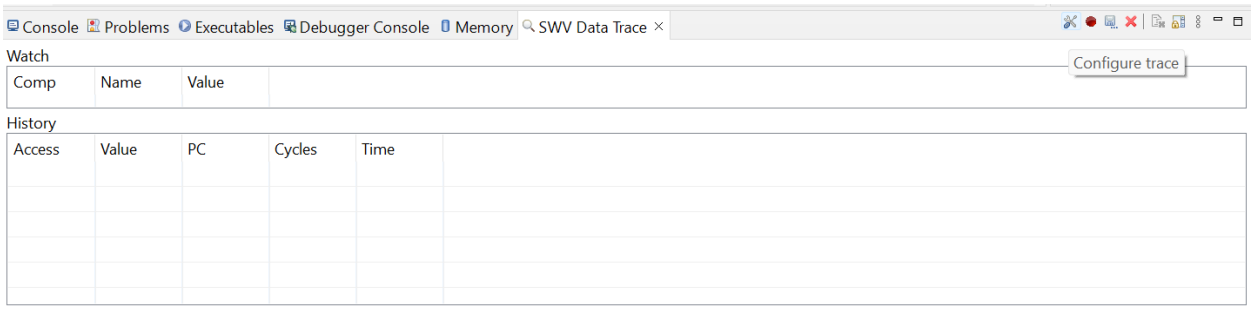
Confirm perspective switch when prompted.



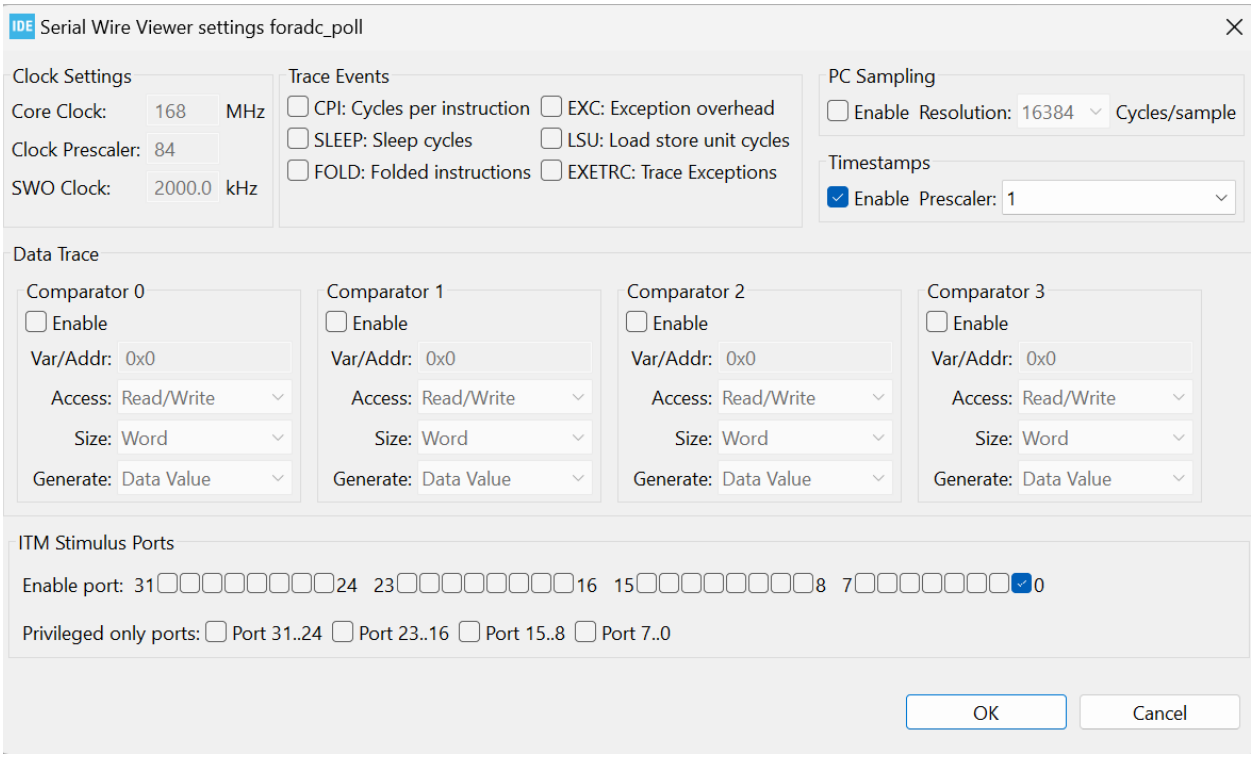
Open Window – Show View – SWV – SWV Data Trace.



Go to SWV Data Trace tab and click on the wrench/screwdriver icon to Configure Trace.

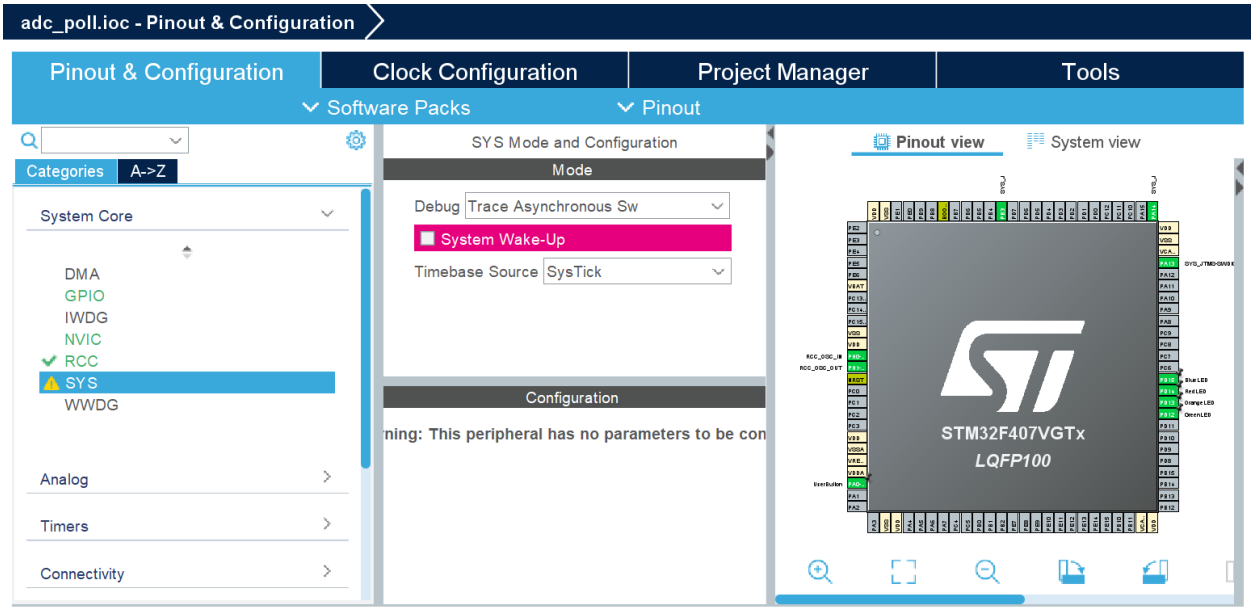


Enable the ITM Stimulus Port 0, which redirects printf() output to the SWV Console. Click OK.



Terminate the debugger session by clicking the red stop square.

Open the `.ioc` configuration file. Under System Core, select SYS and choose Debug mode Trace Asynchronous Sw. This step will activate three pins on the STM32F407. Generate code.



In the `main.c` file, in the Includes section, add `#include <stdio.h>`:

```
20 #include "main.h"
21 #include "rng.h"
22 #include "gpio.h"
23
24 /* Private includes ----------------
25 /* USER CODE BEGIN Includes */
26 #include <stdio.h>
27 /* USER CODE END Includes */
28
```

Add the following prototype in the PFP section:

```
50 /* Private function prototypes ----------
51 void SystemClock_Config(void);
52 /* USER CODE BEGIN PFP */
53 int __io_putchar(int ch);
54 /* USER CODE END PFP */
```

In the `USER CODE` area 4, put the following function:

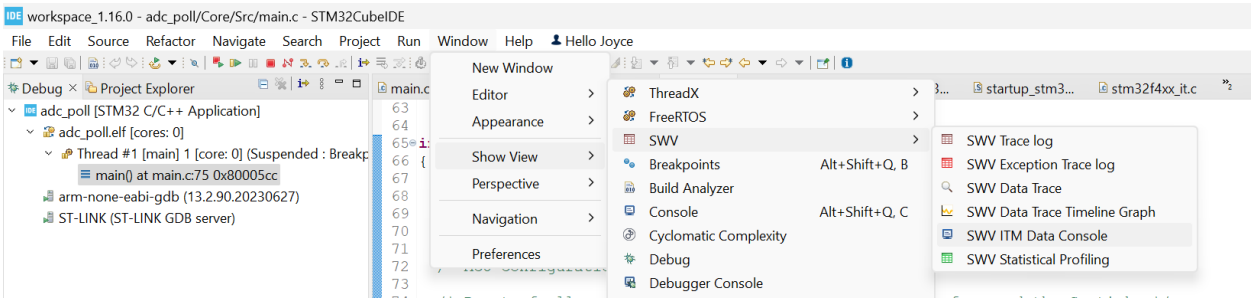
```
151 /* USER CODE BEGIN 4 */
152 int __io_putchar(int ch)
153 {
154     ITM_SendChar(ch);
155     return 0;
156 }
157 /* USER CODE END 4 */
```

Add `printf()` statements anywhere you like in your project. Don't forget `\n`, e.g.

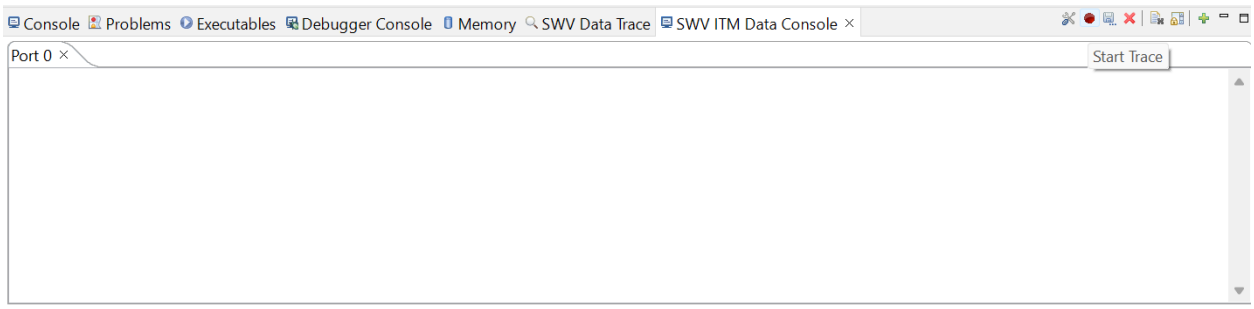
```
printf("Row is %d\n", row);
```

Build your project. Run – Debug.

Open Window – Show View – SWV – SWV ITM Data Console.



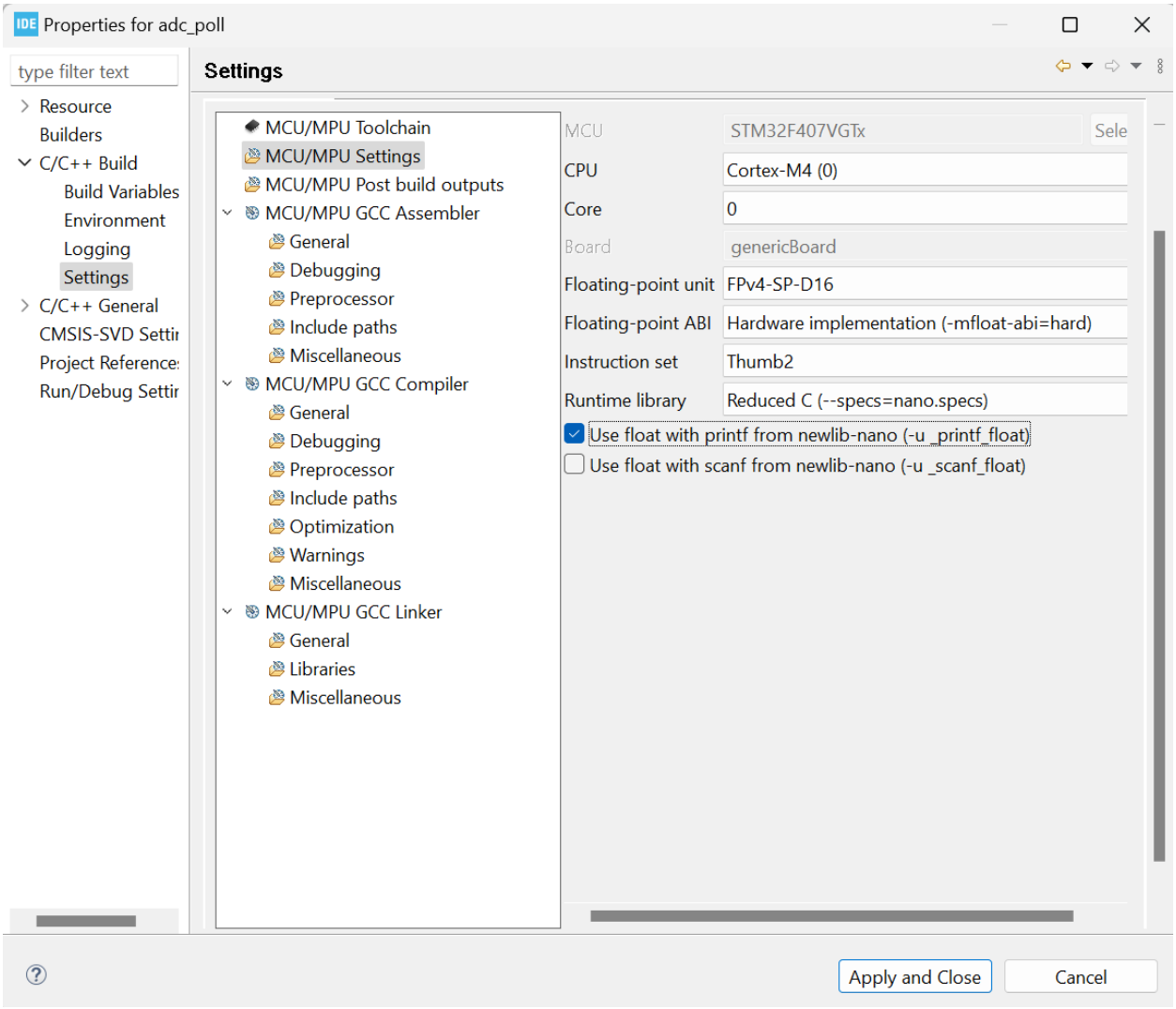
Click on the SWC ITM Data Console tab. Click the red Start Trace button.



Click the Play/Resume button.

Your `printf()` output will appear up in the SWV console. Note that, if the data rate is too great, some data may be missed in the display. While you are debugging, it may be helpful to make the variables you are printing global.

If you wish to print `float` values, there is one more step. Right-click on your project name in the Project Explorer. Scroll down the options to Properties. In C/C++ Build – Settings – MCU/MPU Settings, check off “Use float with printf from newlib -nano (-u _printf_float).”



Appendix 2 CS43L22.h and CS43L22.c

CS43L22.h:

```
//Header file
#include "stm32f4xx_hal.h"

//List of all the defines
#define POWER_CONTROL1                0x02
#define POWER_CONTROL2                0x04
#define CLOCKING_CONTROL                      0x05
#define INTERFACE_CONTROL1                    0x06
#define INTERFACE_CONTROL2                    0x07
#define PASSTHROUGH_A                         0x08
#define PASSTHROUGH_B                         0x09
#define MISCELLANEOUS_CONTRLS         0x0E
#define PLAYBACK_CONTROL                      0x0F
#define PASSTHROUGH_VOLUME_A          0x14
#define PASSTHROUGH_VOLUME_B          0x15
#define PCM_VOLUME_A                          0x1A
#define PCM_VOLUME_B                          0x1B
#define CONFIG_00                             0x00
#define CONFIG_47                             0x47
#define CONFIG_32                             0x32

#define   CS43L22_REG_MASTER_A_VOL    0x20
#define   CS43L22_REG_MASTER_B_VOL    0x21

#define DAC_I2C_ADDR                          0x94

#define CS43_MUTE                             0x00

#define CS43_RIGHT                            0x01
#define CS43_LEFT                             0x02
#define CS43_RIGHT_LEFT                       0x03

#define VOLUME_CONVERT_A(Volume)    (((Volume) > 100)? 255:((uint8_t)(((Volume) * 255) / 100)))
#define VOLUME_CONVERT_D(Volume)    (((Volume) > 100)? 24:((uint8_t)((((Volume) * 48) / 100) -
24)))

//Mode Select Enum
typedef enum
{
       MODE_I2S = 0,
       MODE_ANLG = 1,
} CS43_MODE;


//List of the functions prototypes

void CS43_Pin_RST_Init(void);
void CS43_Init(I2C_HandleTypeDef i2c_handle, CS43_MODE outputMode);
void CS43_Enable_RightLeft(uint8_t side);
void CS43_SetVolume(uint8_t volume);
void CS43_Start(void);
void CS43_Stop(void);
```

CS43L22.c:

```
#include "stm32f4xx_hal.h"
#include "CS43L22.h"

static void write_register(uint8_t reg, uint8_t *data);
static void read_register(uint8_t reg, uint8_t *data);

static uint8_t iData[2];
static I2C_HandleTypeDef i2cx;
extern I2S_HandleTypeDef hi2s3;


// Write to register
static void write_register(uint8_t reg, uint8_t *data)
{
	iData[0] = reg;
	iData[1] = data[0];
	HAL_I2C_Master_Transmit(&i2cx, DAC_I2C_ADDR, iData, 2, 100);
	//HAL_I2C_Master_Transmit(&i2cx, DAC_I2C_ADDR, data, size, 100);
}

// Read from register
static void read_register(uint8_t reg, uint8_t *data)
{
	iData[0] = reg;
	HAL_I2C_Master_Transmit(&i2cx, DAC_I2C_ADDR, iData, 1, 100);
	HAL_I2C_Master_Receive(&i2cx, DAC_I2C_ADDR, data, 1, 100);
}

// Reset pin initialization
void CS43_Pin_RST_Init(void)
{
	RCC->AHB1ENR|=RCC_AHB1ENR_GPIODEN;

	GPIOD->MODER|=GPIO_MODER_MODE4_0;
	GPIOD->MODER&=~GPIO_MODER_MODE4_1;
}

// Initialization
void CS43_Init(I2C_HandleTypeDef i2c_handle, CS43_MODE outputMode)
{
	__HAL_UNLOCK(&hi2s3);     // THIS IS EXTREMELY IMPORTANT FOR I2S3 TO WORK!!
	__HAL_I2S_ENABLE(&hi2s3); // THIS IS EXTREMELY IMPORTANT FOR I2S3 TO WORK!!
	HAL_GPIO_WritePin(GPIOD, GPIO_PIN_4, GPIO_PIN_SET);
	//(1): Get the I2C handle
	i2cx = i2c_handle;
	//(2): Power down
	iData[1] = 0x01;
	write_register(POWER_CONTROL1,iData);
	//(3): Enable Right and Left headphones
	iData[1] =  (2 << 6);  // PDN_HPB[0:1]  = 10 (HP-B always onCon)
	iData[1] |= (2 << 4);  // PDN_HPA[0:1]  = 10 (HP-A always on)
	iData[1] |= (3 << 2);  // PDN_SPKB[0:1] = 11 (Speaker B always off)
	iData[1] |= (3 << 0);  // PDN_SPKA[0:1] = 11 (Speaker A always off)
	write_register(POWER_CONTROL2,&iData[1]);
	//(4): Automatic clock detection
	iData[1] = (1 << 7);
	write_register(CLOCKING_CONTROL,&iData[1]);
	//(5): Interface control 1
	read_register(INTERFACE_CONTROL1, iData);
	iData[1] &= (1 << 5); // Clear all bits except bit 5 which is reserved
	iData[1] &= ~(1 << 7);  // Slave
	iData[1] &= ~(1 << 6);  // Clock polarity: Not inverted
	iData[1] &= ~(1 << 4);  // No DSP mode
	iData[1] &= ~(1 << 2);  // Left justified, up to 24 bit (default)
```

```
        iData[1] |= (1 << 2);
        iData[1] |=  (3 << 0);  // 16-bit audio word length for I2S interface
        write_register(INTERFACE_CONTROL1,&iData[1]);
        //(6): Passthrough A settings
        read_register(PASSTHROUGH_A, &iData[1]);
        iData[1] &= 0xF0;      // Bits [4-7] are reserved
        iData[1] |=  (1 << 0); // Use AIN1A as source for passthrough
        write_register(PASSTHROUGH_A,&iData[1]);
        //(7): Passthrough B settings
        read_register(PASSTHROUGH_B, &iData[1]);
        iData[1] &= 0xF0;      // Bits [4-7] are reserved
        iData[1] |=  (1 << 0); // Use AIN1B as source for passthrough
        write_register(PASSTHROUGH_B,&iData[1]);
        //(8): Miscellaneous register settings
        read_register(MISCELLANEOUS_CONTRLS, &iData[1]);
        if(outputMode == MODE_ANLG)
        {
                iData[1] |=  (1 << 7);   // Enable passthrough for AIN-A
                iData[1] |=  (1 << 6);   // Enable passthrough for AIN-B
                iData[1] &= ~(1 << 5);   // Unmute passthrough on AIN-A
                iData[1] &= ~(1 << 4);   // Unmute passthrough on AIN-B
                iData[1] &= ~(1 << 3);   // Changed settings take effect immediately
        }
        else if(outputMode == MODE_I2S)
        {
                iData[1] = 0x02;
        }
        write_register(MISCELLANEOUS_CONTRLS,&iData[1]);
        //(9): Unmute headphone and speaker
        read_register(PLAYBACK_CONTROL, &iData[1]);
        iData[1] = 0x00;
        write_register(PLAYBACK_CONTROL,&iData[1]);
        //(10): Set volume to default (0dB)
        iData[1] = 0x00;
        write_register(PASSTHROUGH_VOLUME_A,&iData[1]);
        write_register(PASSTHROUGH_VOLUME_B,&iData[1]);
        write_register(PCM_VOLUME_A,&iData[1]);
        write_register(PCM_VOLUME_B,&iData[1]);
}

// Enable Right and Left headphones
void CS43_Enable_RightLeft(uint8_t side)
{
        switch (side)
        {
                case 0:
                        iData[1] =  (3 << 6);  // PDN_HPB[0:1]  = 10 (HP-B always onCon)
                        iData[1] |= (3 << 4);  // PDN_HPA[0:1]  = 10 (HP-A always on)
                        break;
                case 1:
                        iData[1] =  (2 << 6);  // PDN_HPB[0:1]  = 10 (HP-B always onCon)
                        iData[1] |= (3 << 4);  // PDN_HPA[0:1]  = 10 (HP-A always on)
                        break;
                case 2:
                        iData[1] =  (3 << 6);  // PDN_HPB[0:1]  = 10 (HP-B always onCon)
                        iData[1] |= (2 << 4);  // PDN_HPA[0:1]  = 10 (HP-A always on)
                        break;
                case 3:
                        iData[1] =  (2 << 6);  // PDN_HPB[0:1]  = 10 (HP-B always onCon)
                        iData[1] |= (2 << 4);  // PDN_HPA[0:1]  = 10 (HP-A always on)
                        break;
                default:
                        break;
        }
        iData[1] |= (3 << 2);  // PDN_SPKB[0:1] = 11 (Speaker B always off)
        iData[1] |= (3 << 0);  // PDN_SPKA[0:1] = 11 (Speaker A always off)
```

```
        write_register(POWER_CONTROL2,&iData[1]);
}

// Set Volume Level
void CS43_SetVolume(uint8_t volume)
{
        int8_t tempVol = volume - 50;
        tempVol = tempVol*(127/50);
        uint8_t myVolume =  (uint8_t )tempVol;
        iData[1] = myVolume;
        write_register(PASSTHROUGH_VOLUME_A,&iData[1]);
        write_register(PASSTHROUGH_VOLUME_B,&iData[1]);

        iData[1] = VOLUME_CONVERT_D(volume);

        /* Set the Master volume */
        write_register(CS43L22_REG_MASTER_A_VOL,&iData[1]);
        write_register(CS43L22_REG_MASTER_B_VOL,&iData[1]);
}

// Start the Audio DAC
void CS43_Start(void)
{
        // Write 0x99 to register 0x00.
        iData[1] = 0x99;
        write_register(CONFIG_00,&iData[1]);
        // Write 0x80 to register 0x47.
        iData[1] = 0x80;
        write_register(CONFIG_47,&iData[1]);
        // Write '1'b to bit 7 in register 0x32.
        read_register(CONFIG_32, &iData[1]);
        iData[1] |= 0x80;
        write_register(CONFIG_32,&iData[1]);
        // Write '0'b to bit 7 in register 0x32.
        read_register(CONFIG_32, &iData[1]);
        iData[1] &= ~(0x80);
        write_register(CONFIG_32,&iData[1]);
        // Write 0x00 to register 0x00.
        iData[1] = 0x00;
        write_register(CONFIG_00,&iData[1]);
        //Set the "Power Ctl 1" register (0x02) to 0x9E
        iData[1] = 0x9E;
        write_register(POWER_CONTROL1,&iData[1]);
}

void CS43_Stop(void)
{
        iData[1] = 0x01;
        write_register(POWER_CONTROL1,&iData[1]);
}
```

Appendix 3 Additional Resources

https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/STM32F4-Discovery
Examples and demonstrations for STM32F407 Discovery board authored by STMicroelectronics.

https://www.youtube.com/playlist?list=PLfExI9i0v1sn_lQjCFJHrDSpvZ8F2CpkA
A collection of many videos for the STM32F407 Discovery board.

https://www.hackster.io/Labirenti/air-quality-monitoring-device-with-stm32f407-discovery-board-cfba98
Air quality monitor with STM32F407 Discovery board.

https://ruturajn.hashnode.dev/dc-motor-control-using-the-l298n-motor-driver-interfaced-with-the-stm32f407-discovery-kit
DC motor control.

https://www.instructables.com/Basic-Mobile-Phone-Using-STM32F407-Discovery-Kit-a/
Build a basic mobile phone with the STM32F407 Discovery board.

https://github.com/iwatake2222/DigitalCamera_STM32
Make a digital camera with the STM32F407 Discovery board.

https://github.com/SharathN25/STM32F407-Discovery
Useful documentation about details of STM32F407 Discovery board.

https://www.digikey.ca/en/maker/projects/getting-started-with-stm32-introduction-to-freertos/ad275395687e4d85935351e16ec575b1
RTOS (real time operating system) on STM32

https://www.st.com/content/st_com/en/campaigns/educationalplatforms.html
Educational Platforms: A set of open-source curricula to inspire tomorrow's engineers.

https://www.st.com/content/st_com/en/stm32-mcu-developer-zone.html
STM32 MCU Developer Zone.