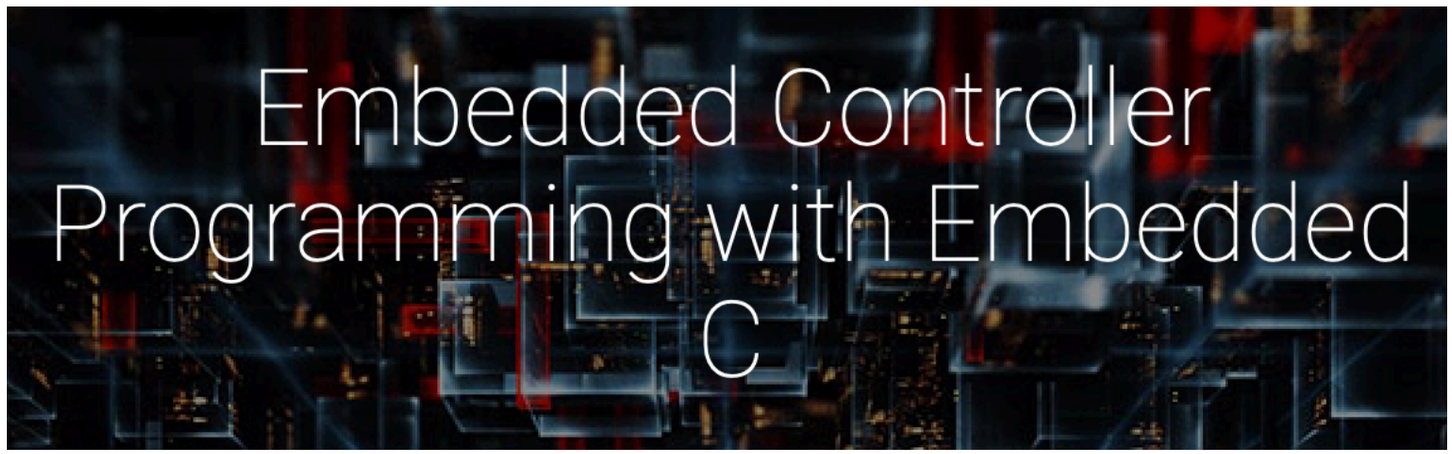


Lesson 1

Norman McEntire
norman.mcentire@gmail.com

Contents

- Introduction to Embedded C
- STM32CubeMX
- Tour of Generated Project
- TrueStudio
- Hello Blinking LED

Introduction to Embedded C

Introduction to Embedded C

- Good News: Embedded C is based on Standard C
  - More alike than different
  - Differences relate to lower-level hardware demands on Embedded C
- Classic Example: Hello World
  - Standard C: Display “Hello World” on console
  - Embedded C: Blink an LED

Standard C and Embedded C How They Are Alike

- Pre-processor
  - #include, #define, #if, etc.
- main() Function as starting point for code
- Primitive Data Types: char, short, int, long, float, double
- User Data Types: typedef, struct, enum
- Operators: +, -, *, /, %, ++, +=, -=, *=, /+, ||, &&, |, &, !, etc.
- Control Structures: if, if-else, switch, break
- Looping: for, while, do, continue, break

Standard C vs Embedded C main()

- Standard C
  - int main(int argc, char *argv[])
  - Command-line arguments
    - Returns when done
- Embedded C
  - int main(void)
  - No command-line arguments
  - Never returns (runs forever)

Standard C vs Embedded C printf() and scanf()

- Standard C
  - Use printf() to send output to console
  - Use scanf() to read from console
- Embedded C
  - printf() may or may not be supported
    - And if supported, goes to serial port
  - scanf() not typically supported

Standard C vs Embedded C Primitive Data Types

- Standard C
  - Use of char, short, int, long, float, double
  - Issue: Other than char/float/double, size of primitive data types not constant
- Embedded C
  - Use of ISO C99 Data Types
    - int8_t, uint8_t - 8 bits
    - int16_t, uint16_t - 16 bits
    - int32_t, uint32_t - 32 bits
    - int64_t, uint64_t - 64 bits

ISO C99 Data Types

Data type	Size	Range
int8_t	1 byte	-128 to 127
uint8_t	1 byte	0 to 255
int16_t	2 bytes	-32,768 to 32,767
uint16_t	2 bytes	0 to 65,535
int32_t	4 bytes	-2,147,483,648 to 2,147,483,647
uint32_t	4 bytes	0 to 4,294,967,295
int64_t	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	8 bytes	0 to 18,446,744,073,709,551,615

Standard C vs Embedded C Memory Addressing

- Standard C
  - Virtual Memory Address Space
  - The address your program uses is not the same as the physical memory address
    - If program does not “fit” into memory, it can be swapped in/out as needed
- Embedded C
  - No virtual memory
  - The addresses your program uses is the same as physical memory space
  - Your code must fit into memory - no swapping in/out of memory

Standard C vs Embedded C Memory Allocation

- Standard C
  - All memory the same - just one large block of virtual memory
  - Use of malloc(), calloc() to allocate memory
- Embedded C
  - May have different memory banks
    - Flash Memory
    - RAM Bank #1
    - RAM Bank #2
    - etc.

Standard C vs Embedded C Hardware Initialization

- Standard C
  - Hardware initialize the operating system
- Embedded C
  - Embedded C code initializes the hardware
    - Hardware Clocks
    - Interrupt Vectors

Standard C vs Embedded C Interrupt Handling

- Standard C
  - No concern for low-level interrupt handling
- Embedded C
  - Interrupt handling required for hardware devices that are interrupt driven

Standard C vs Embedded C Use of Pointers

- Standard C
  - Use of pointers to point to data structures, etc.
- Embedded C
  - Same as standard C use of pointers
  - PLUS….heavy use of pointers to access physical addresses
    - Example: Pointer to base of GPIO Register

Standard C vs Embedded C Use of Bit Operations

- Standard C
  - Some use of bit operations
    - Bitwise OR: |
    - Bitwise AND: &
- Embedded C
  - Heavy use of bit operations for I/O register operations
    - Set bits in registers
    - Test bits in registers

Standard C vs Embedded C Libraries

- Standard C
  - Uses “high-level” libraries that are “hardware independent”
- Embedded C
  - Uses “low-level” libraries that are close to the hardware
    - HAL - Hardware Abstraction Layer
    - LL - Lower Layer

Summary so far…

- Embedded C builds upon Standard C
  - Language Syntax the same: if, if-else, switch, for, while, do, continue, break, etc.
- Embedded C differences are due to need for lower-level hardware access
  - C99 Data Types (int8_t, uint8_t, etc.)
  - Use of physical addresses for memory and hardware access
    - Heavy use of pointers to hardware addresses
    - Heavy use of bit operations to set/clear bits and test for bits

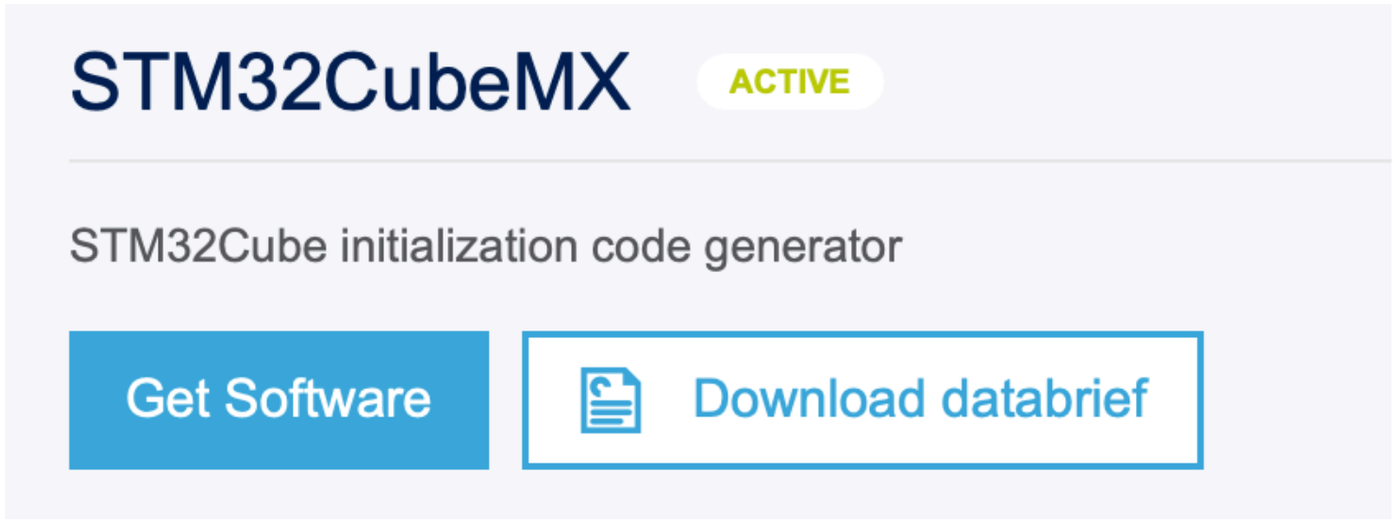
STM32CubeMX

STM32CubeMX

- STM32CubeMX is an Embedded C code generation tool
- We will use the tool to generate our first Embedded C code, then read/study/run the code
- Our approach
  - Install the tool
  - Generate a “Hello Embedded C” project
  - Build/Run on our STM32 Discovery Board

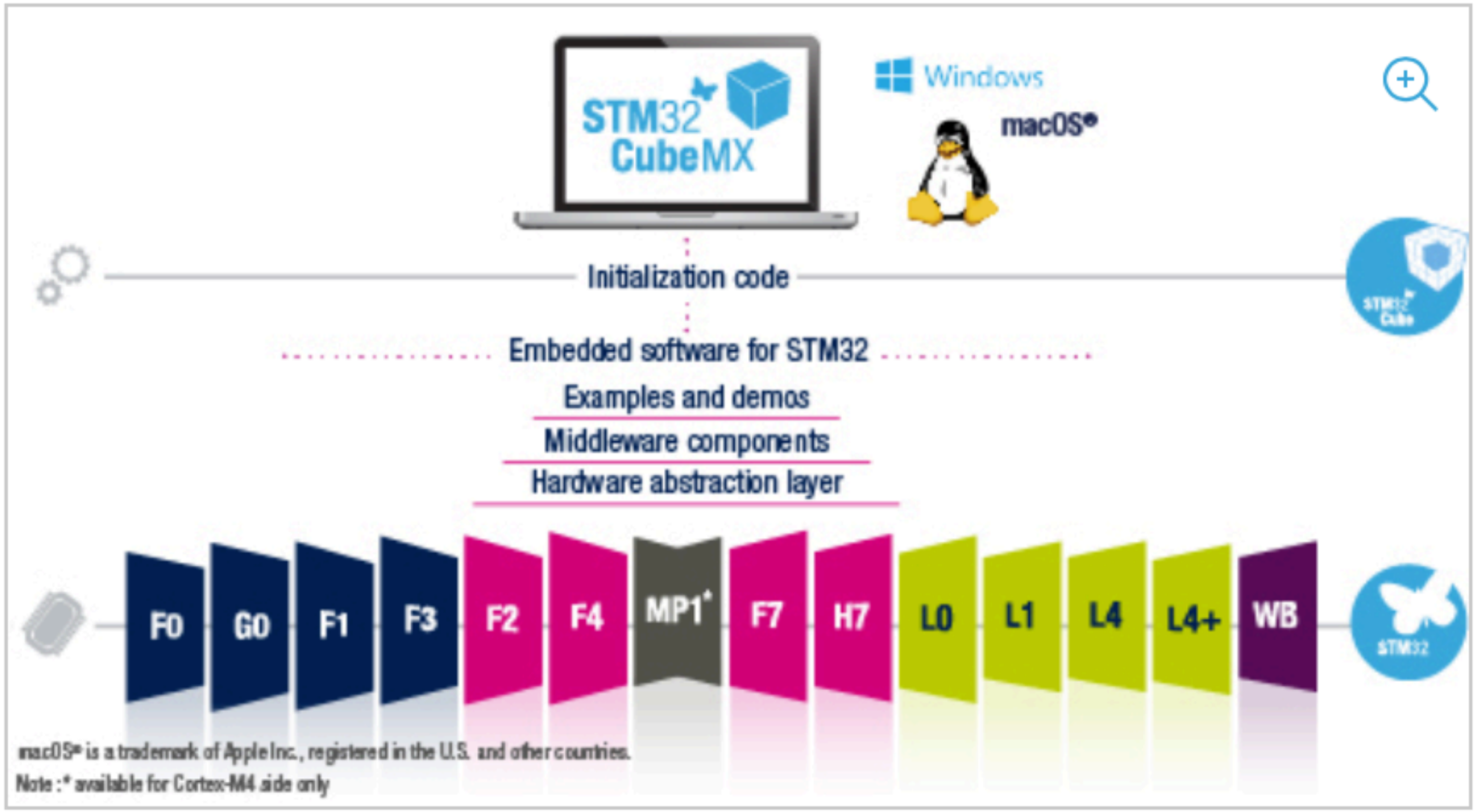
Step: Download and install STM32CubeMX

- https://www.st.com/en/development-tools/stm32cubemx.html

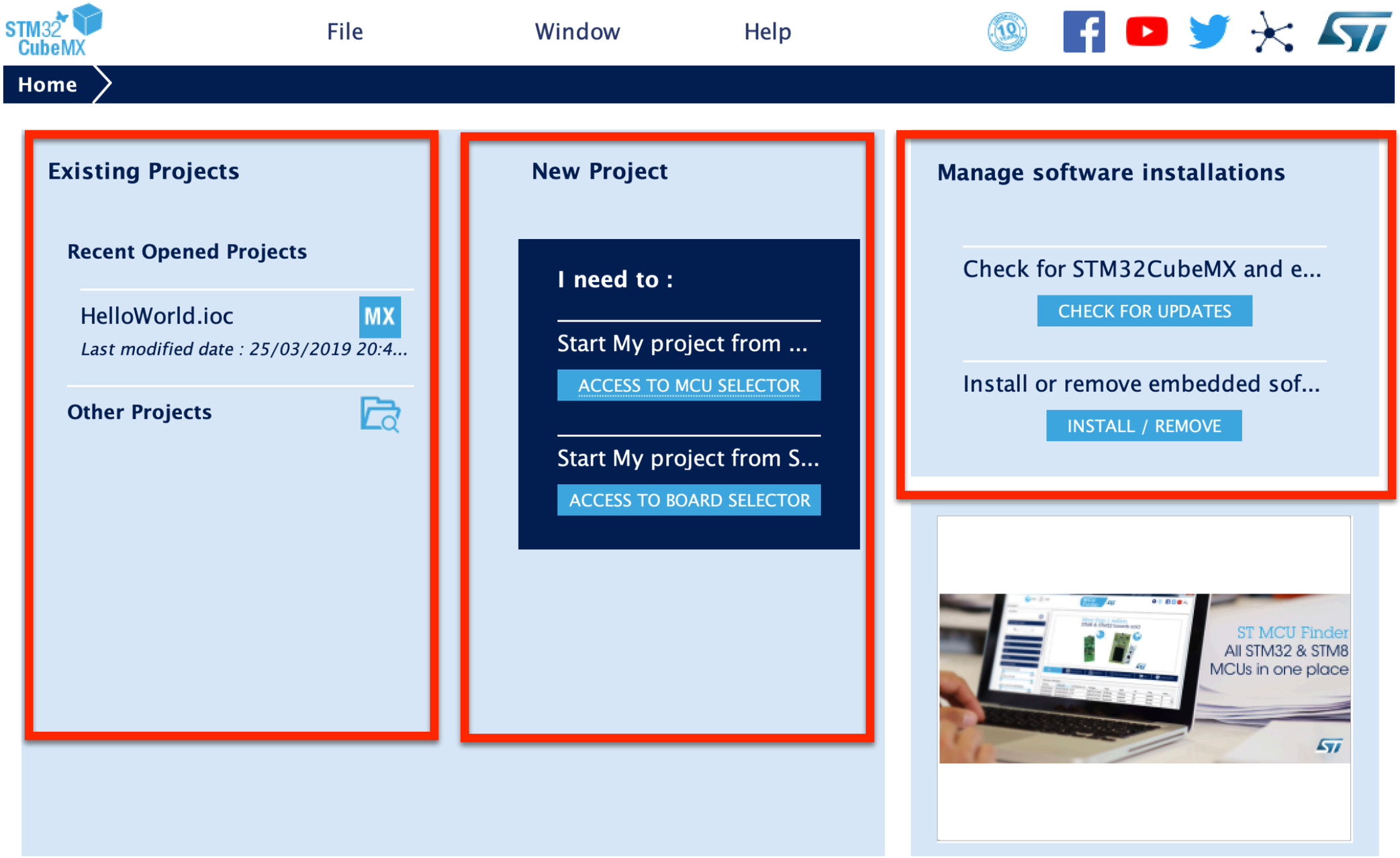


Note: STM32CubeMX works on:
Linux,
Mac,
and Windows
(Java-Based App)

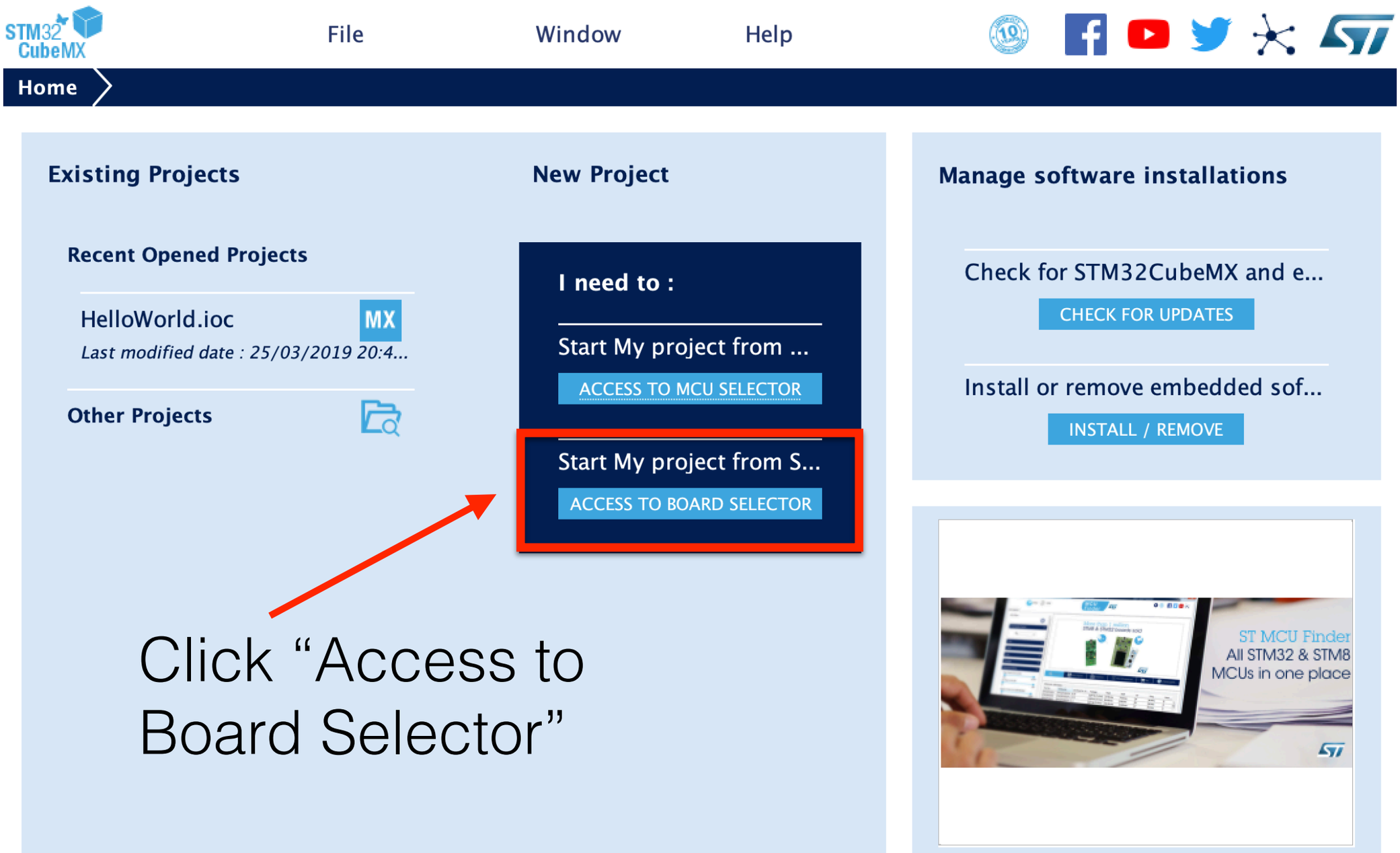
STM32CubeMX



Step: Startup STM32CubeMX

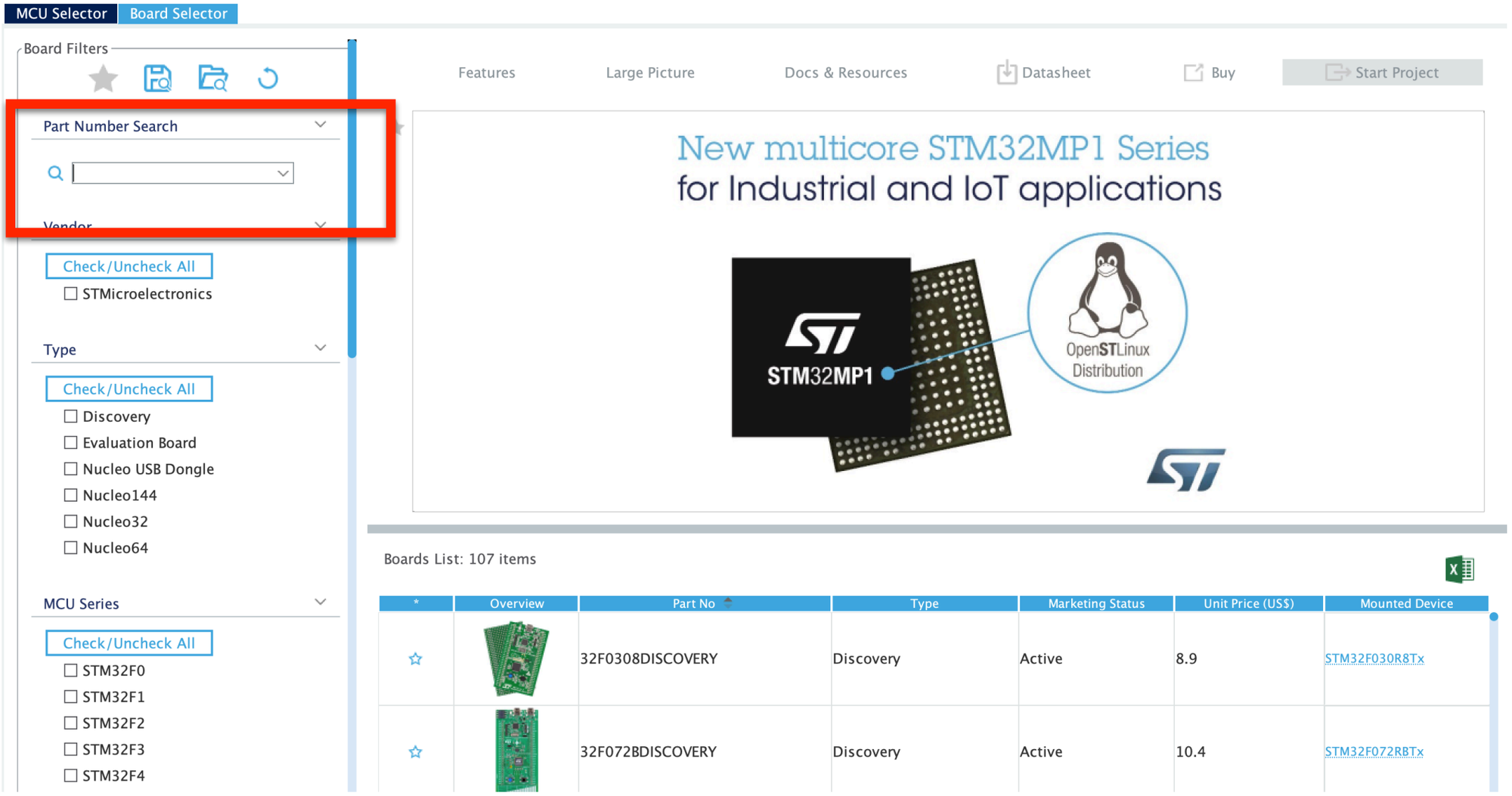


Step: Click on “Access to Board Selector”

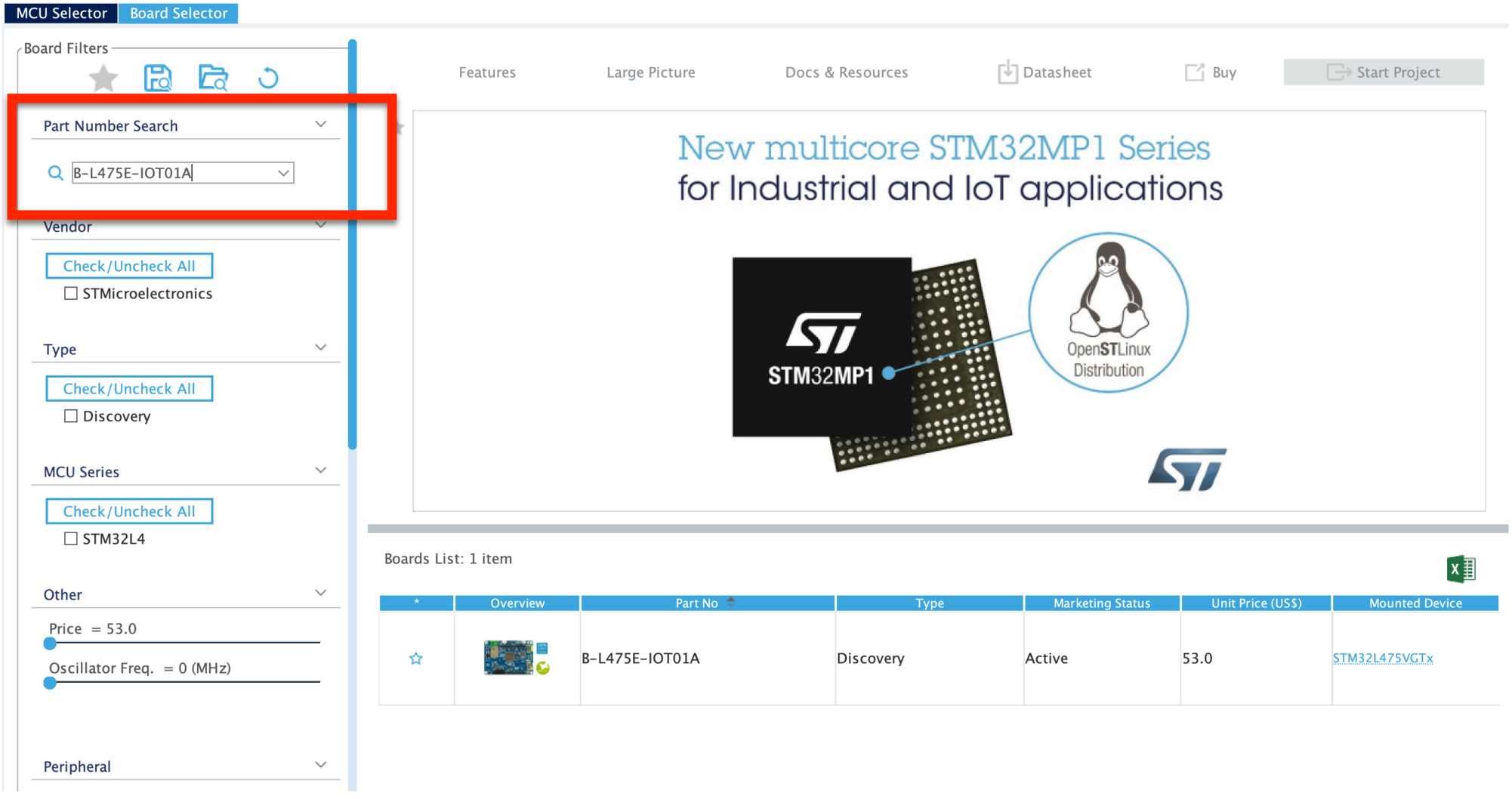


Click “Access to Board Selector”

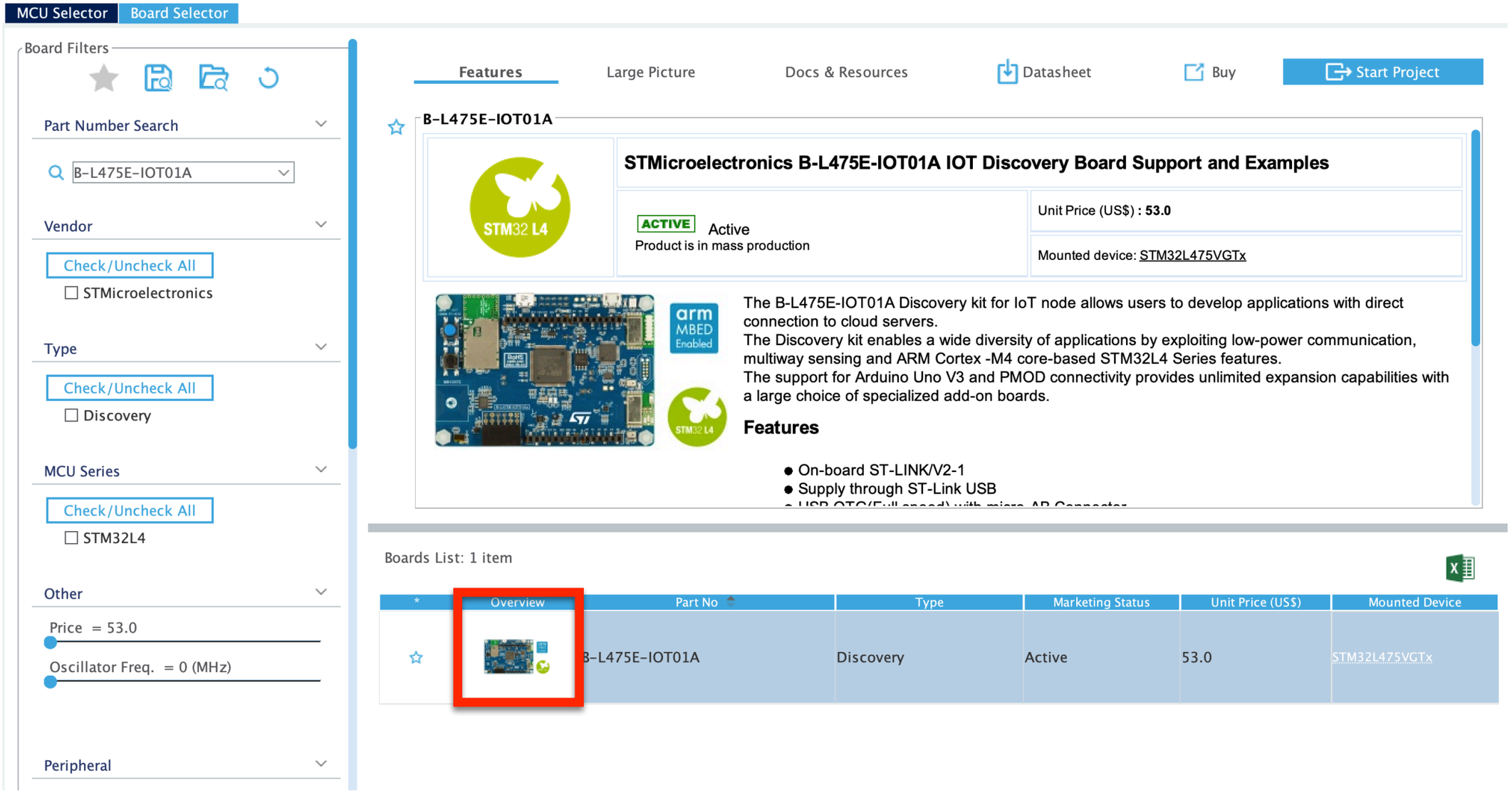
Step: Observe “Part Number Search”



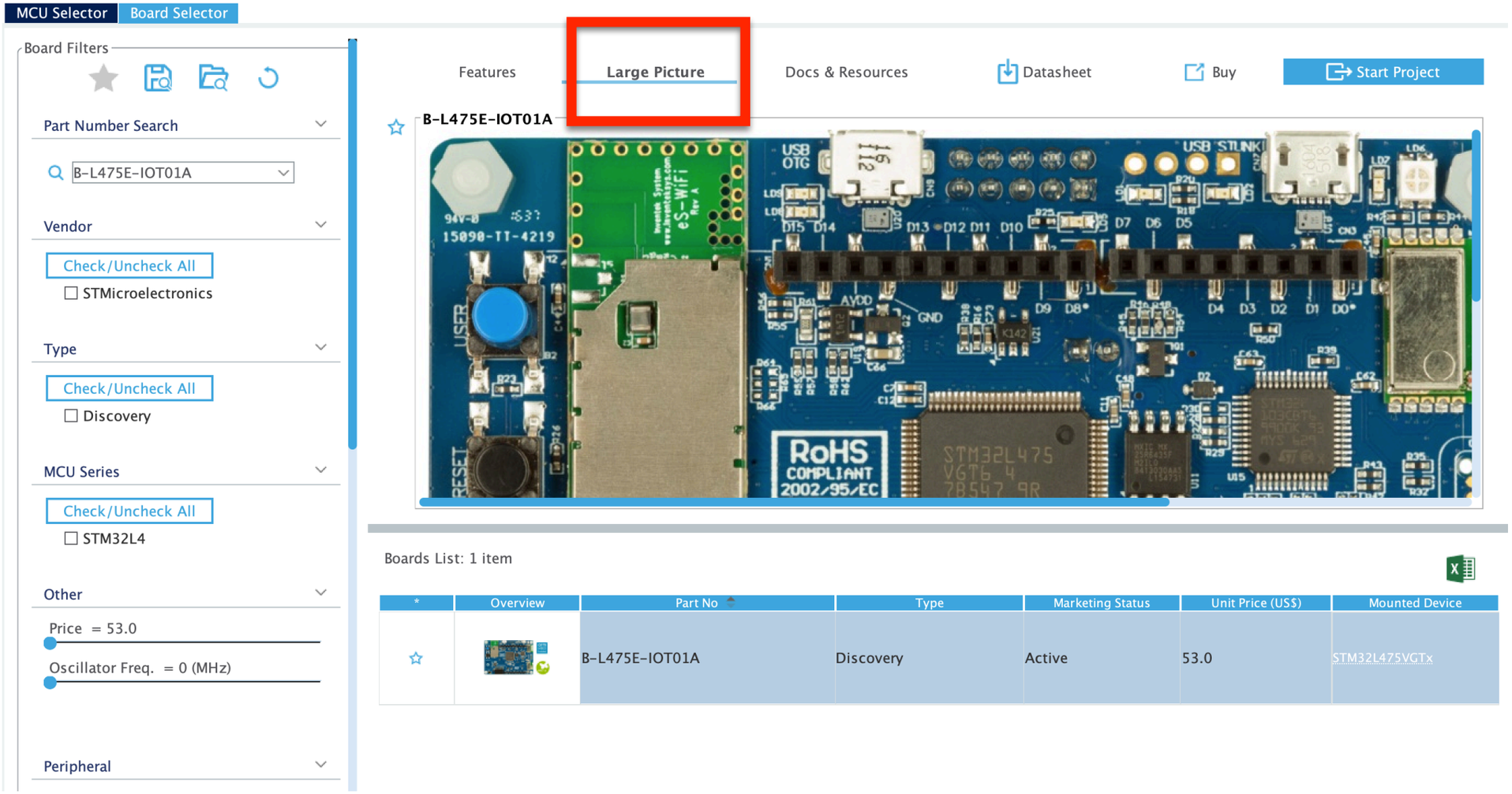
Step: Enter ”B-L475E-IOT01A”



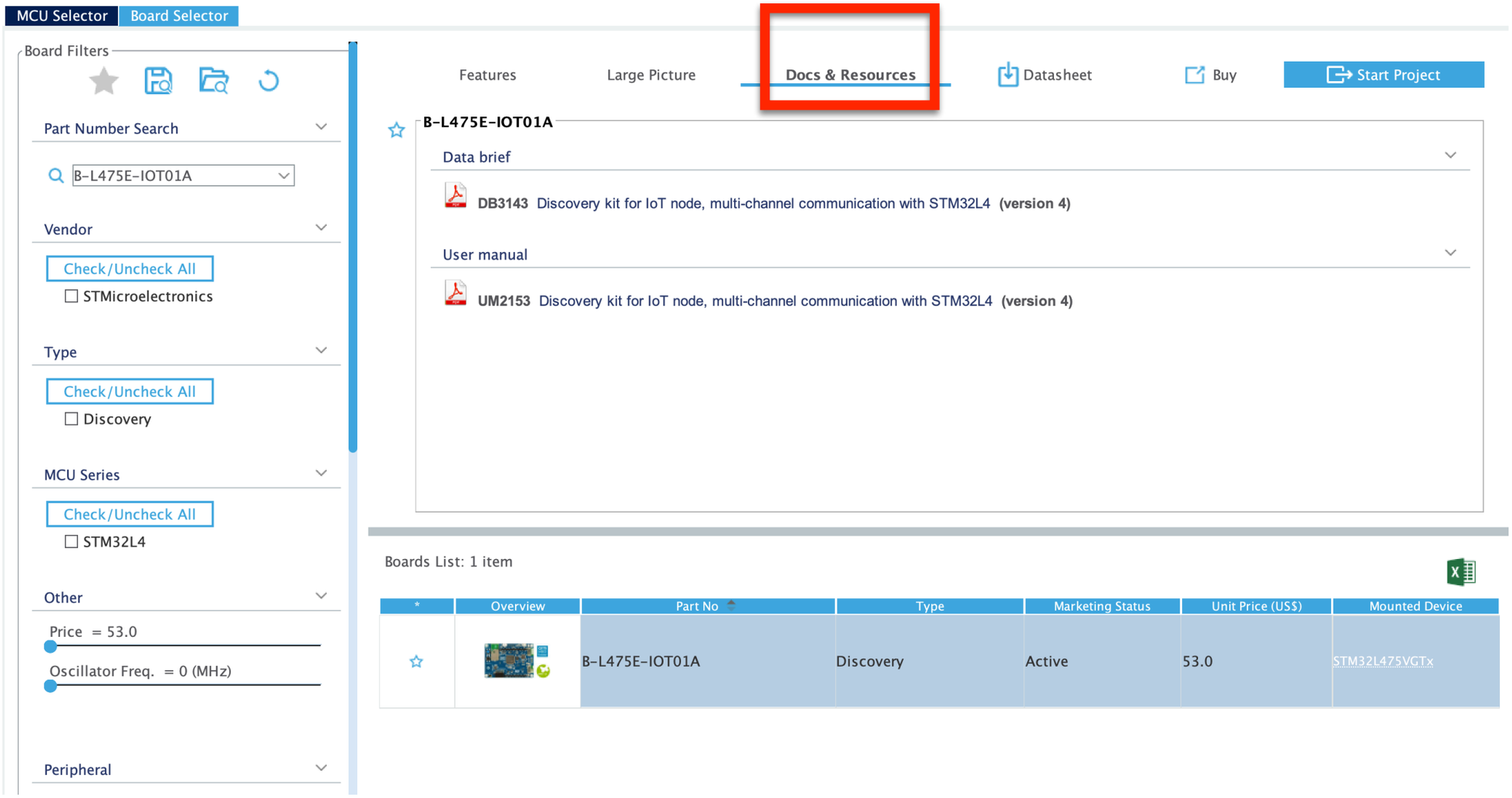
Step: Click on Image, and observe “Features”



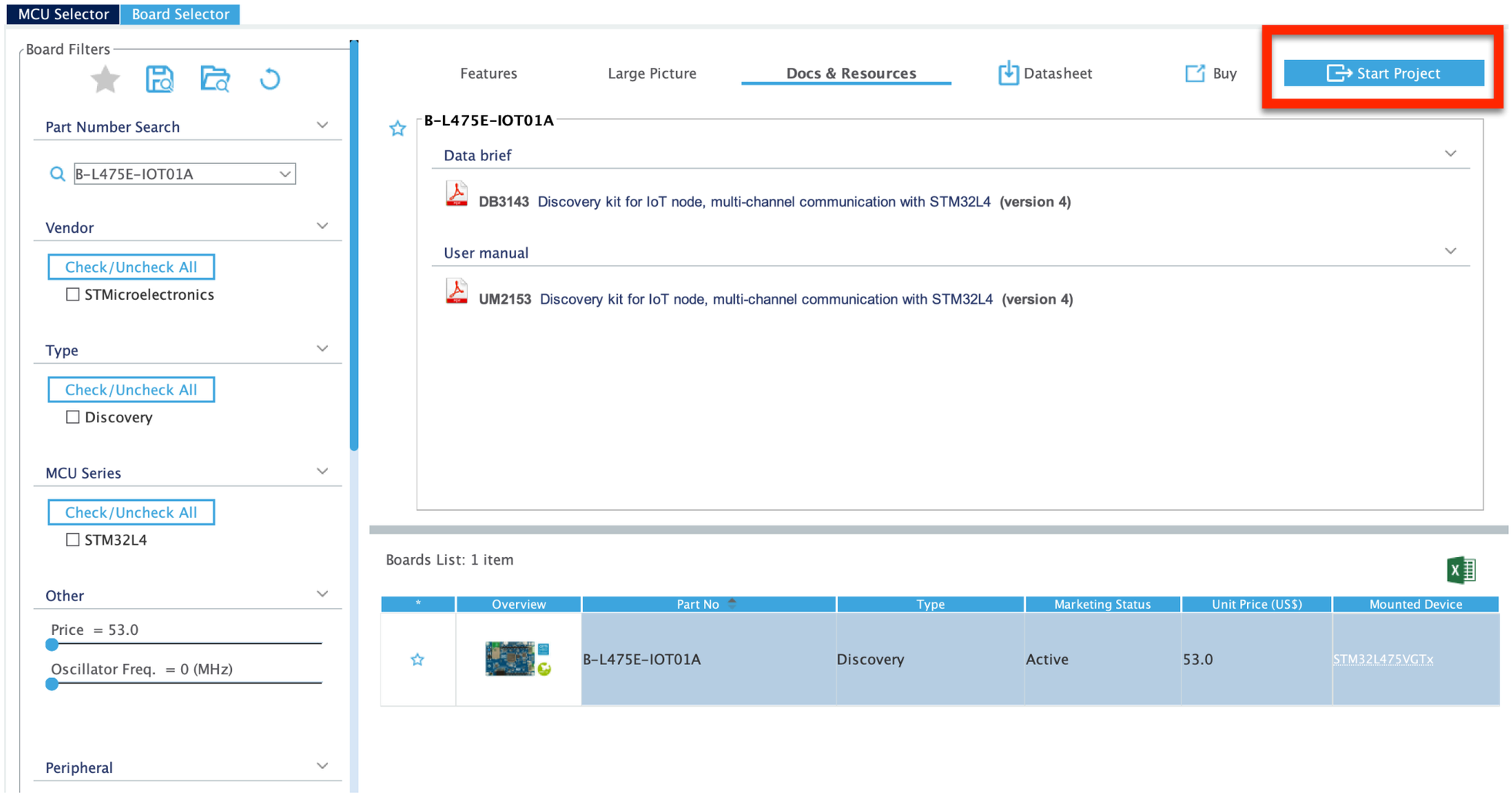
Step: Click on “Large Picture”



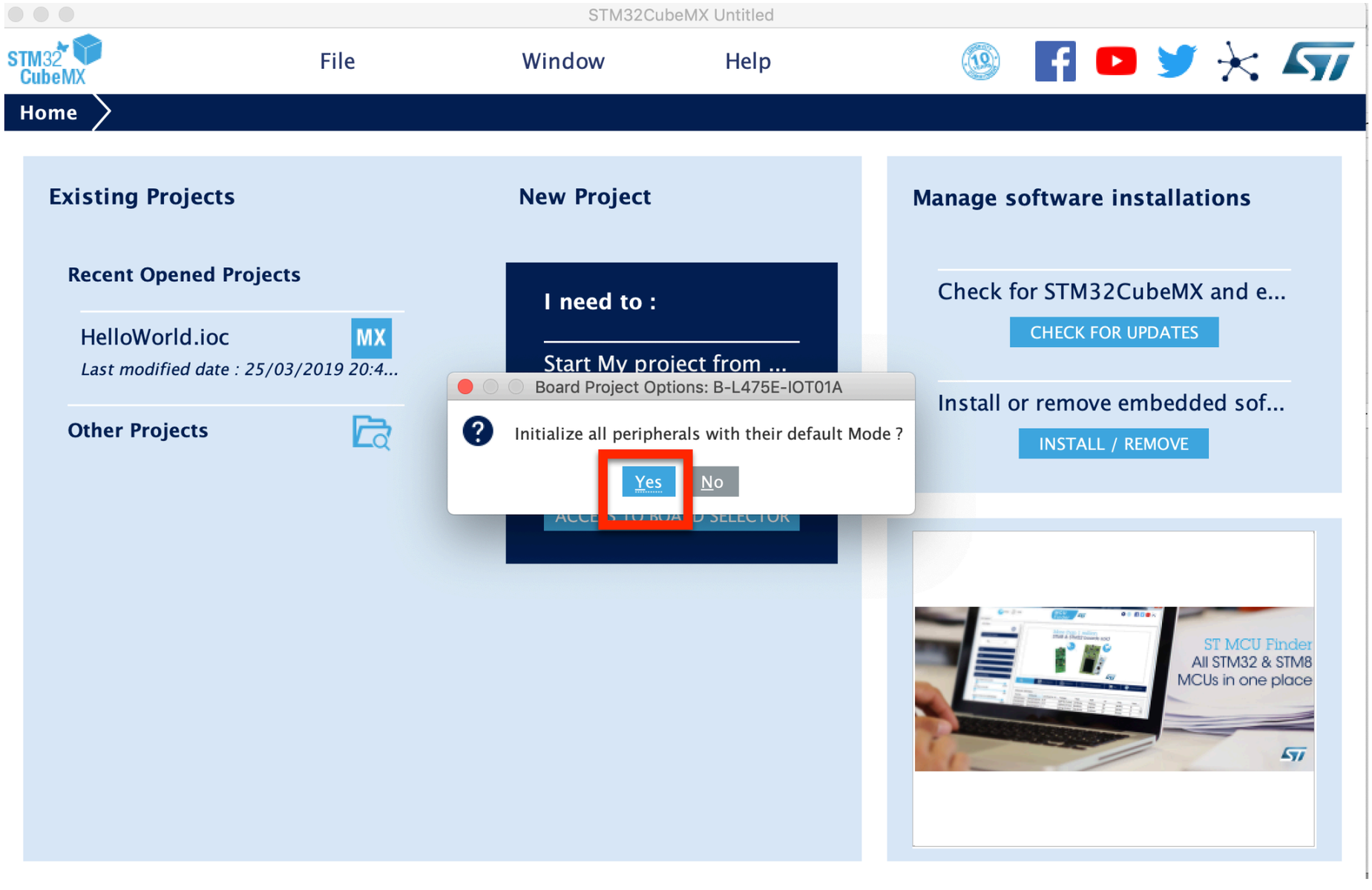
Step: Click on “Docs and Resources”



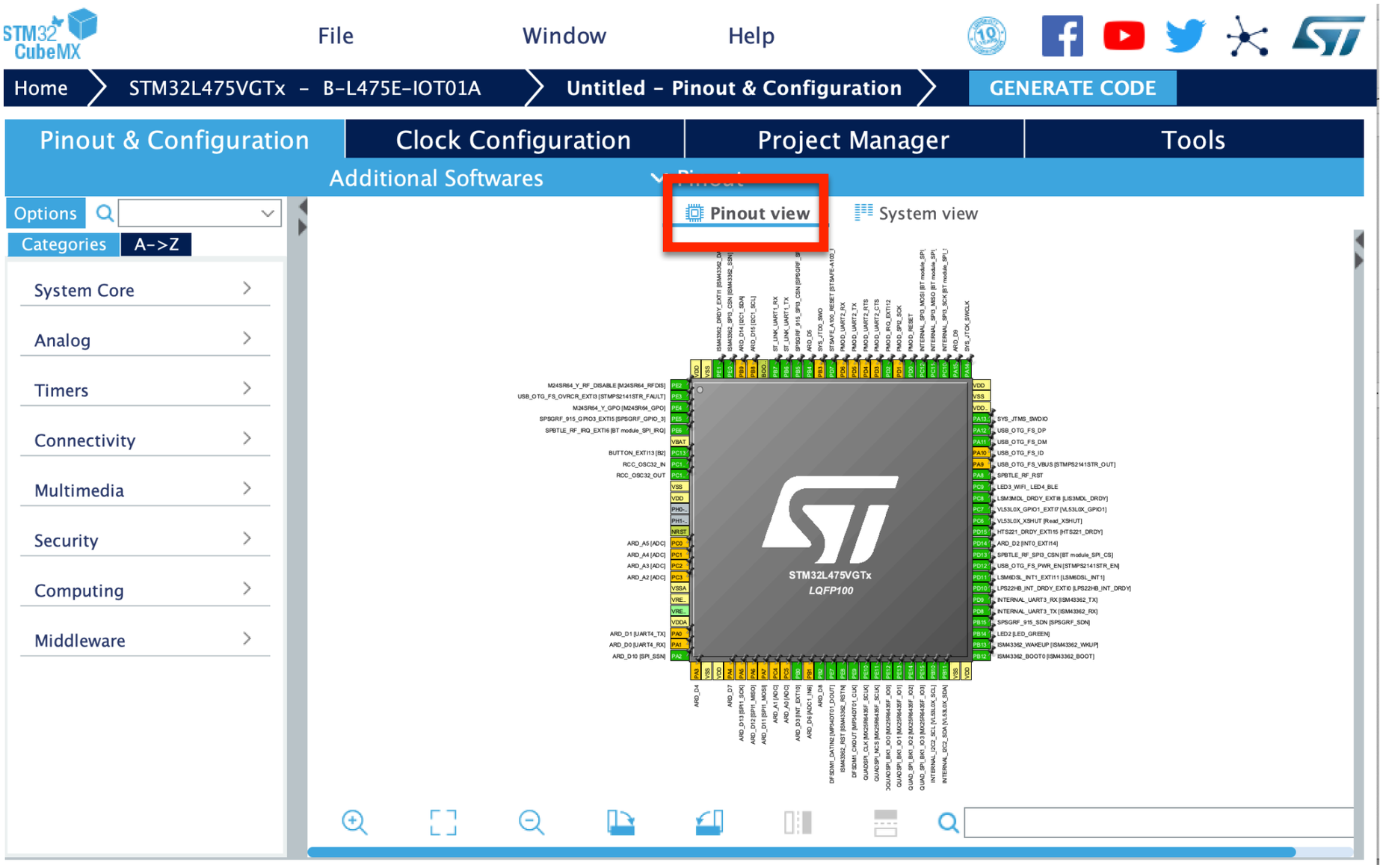
Step: Click on “Start Project”



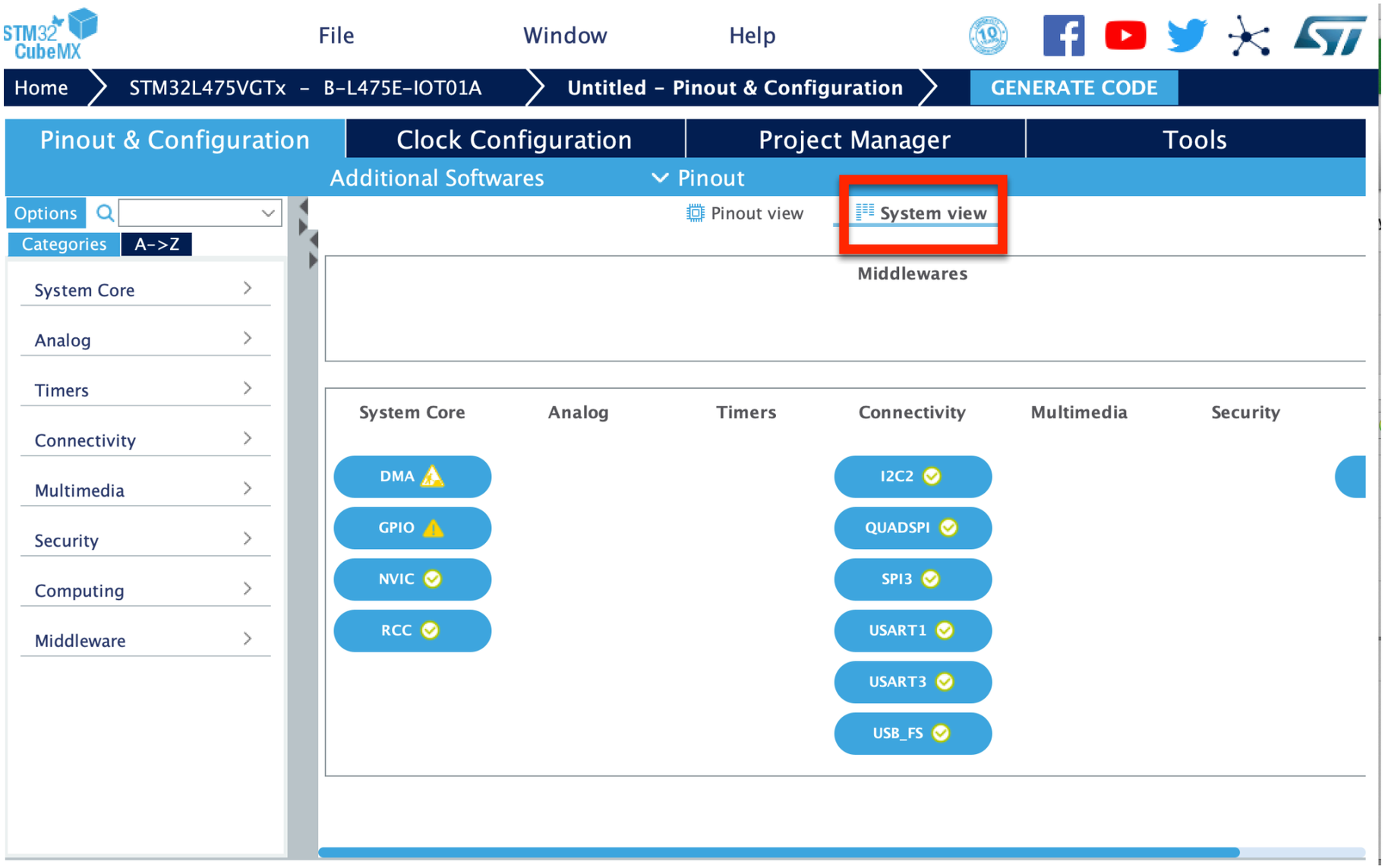
Step: Click on “Yes” (Initialize all …with default mode)



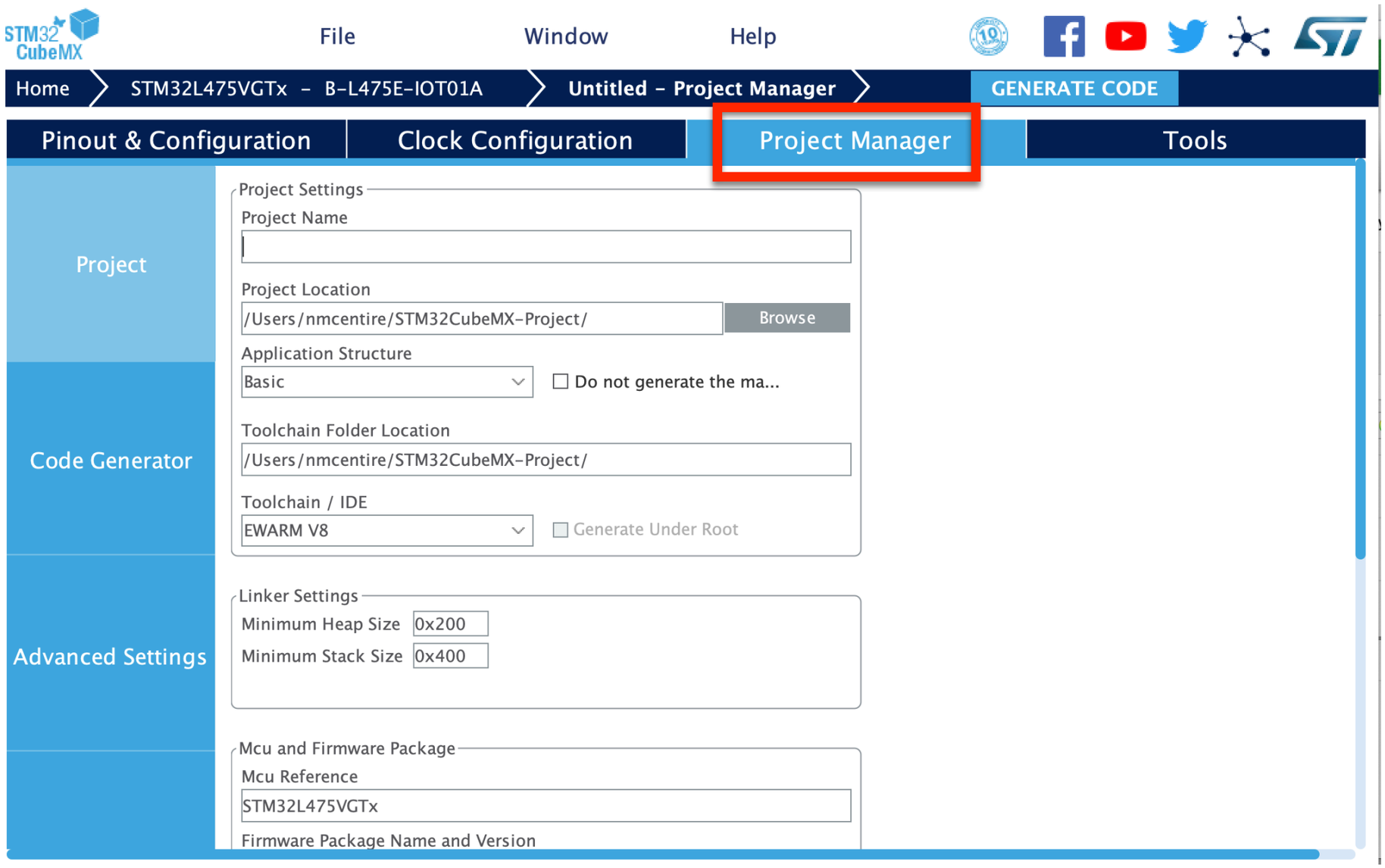
Step: Observe “Pinout View”



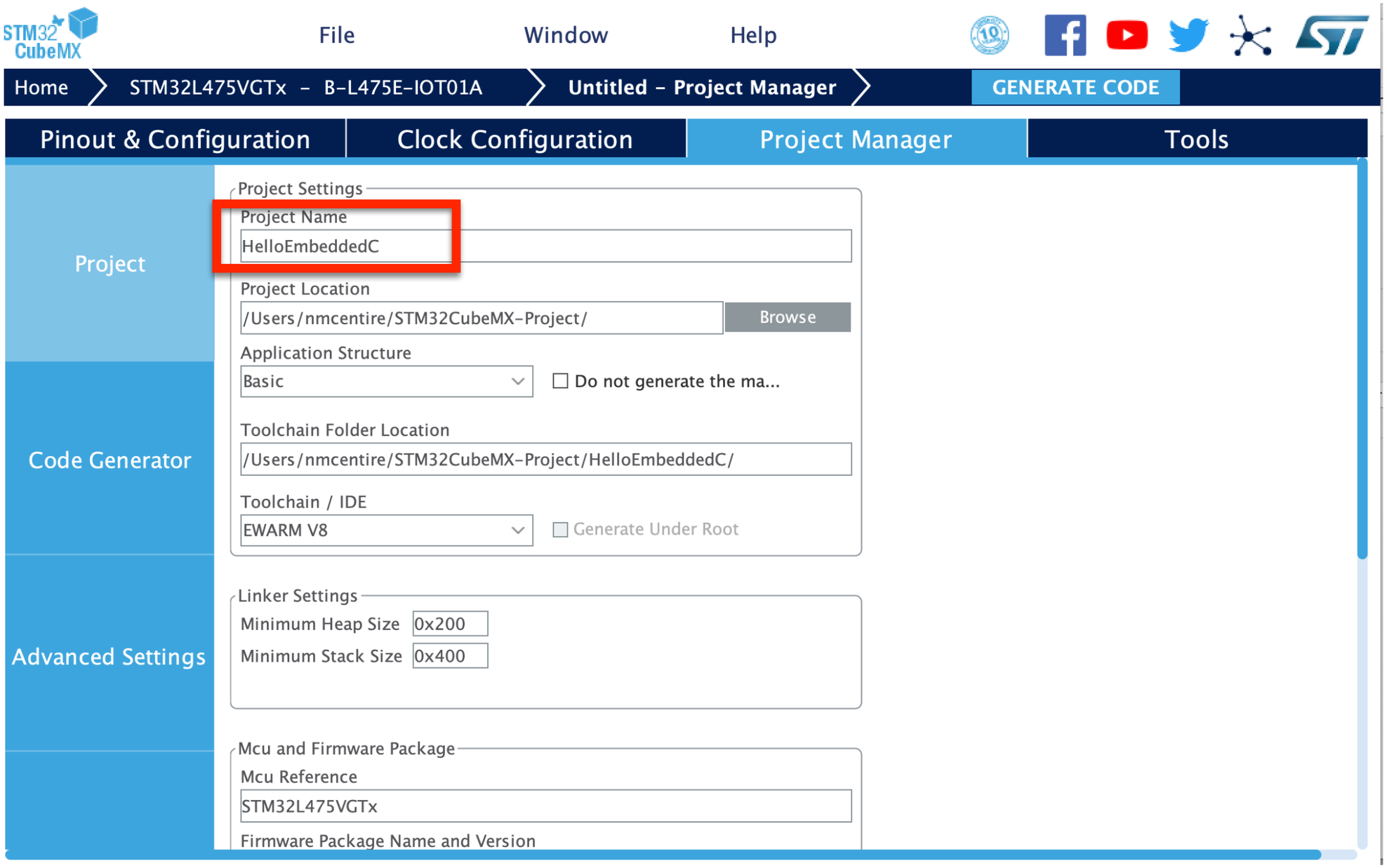
Step: Observe “System View”



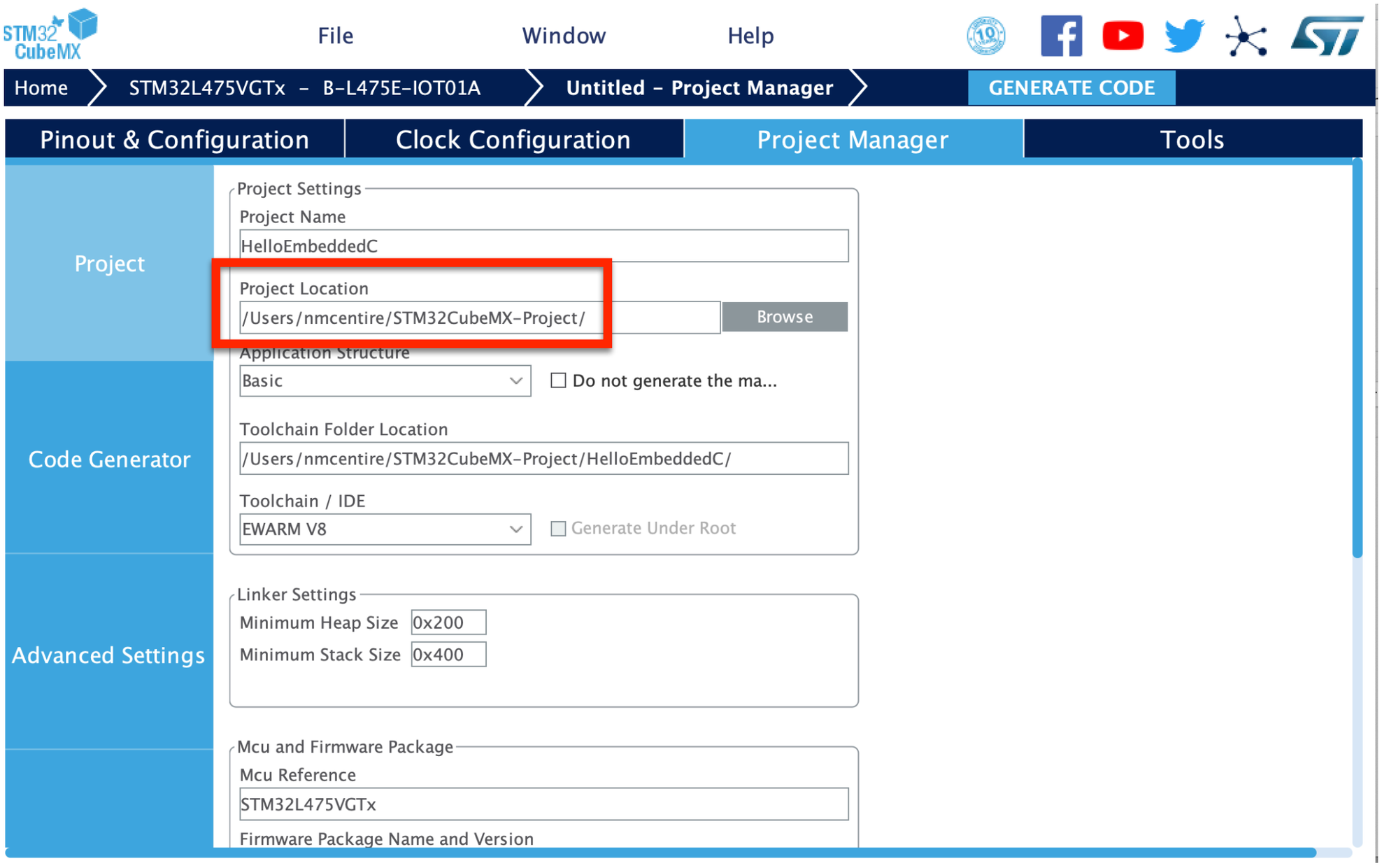
Step: Observe “Project Manager”



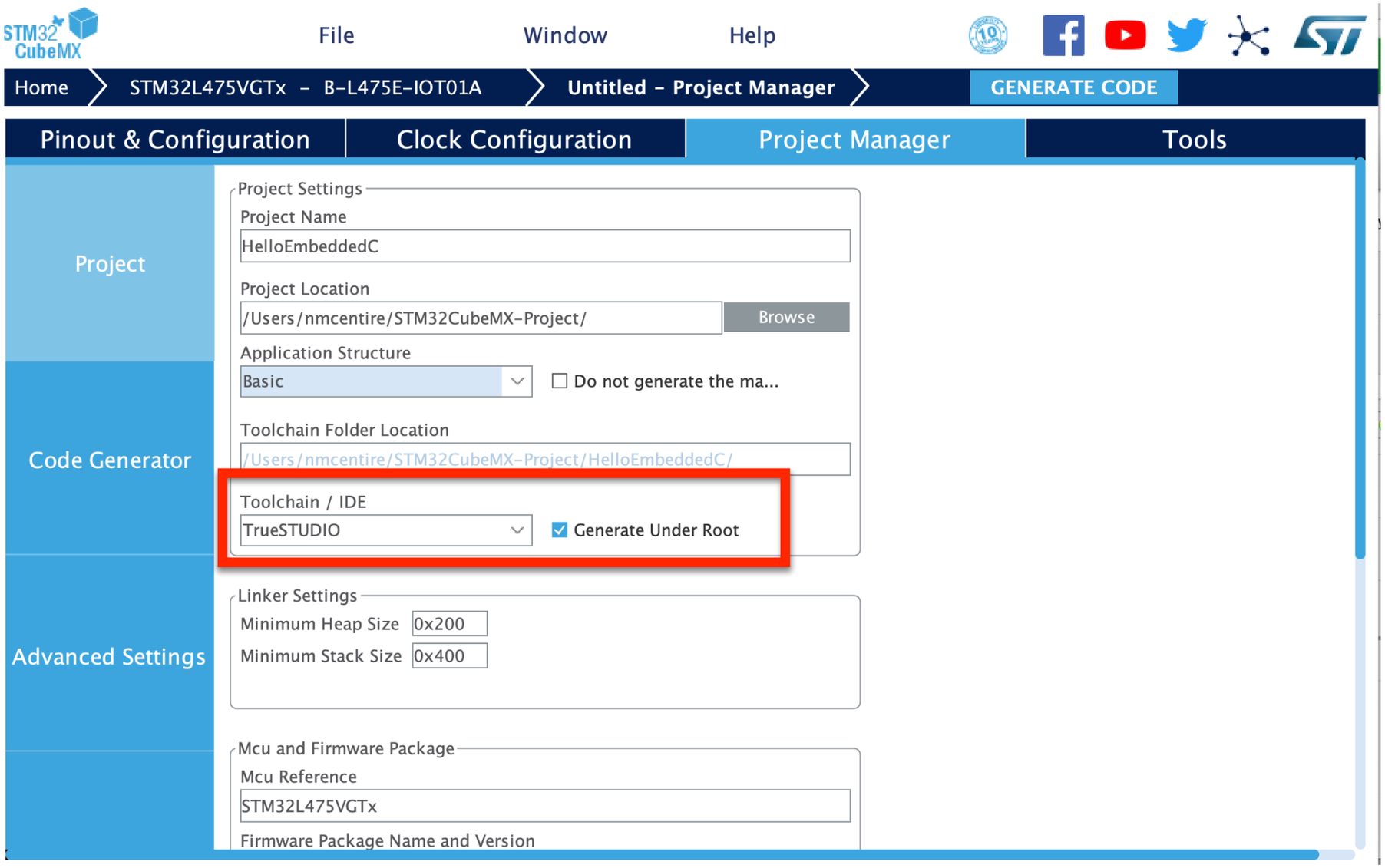
Step: Enter “HelloEmbeddedC” for Project Name



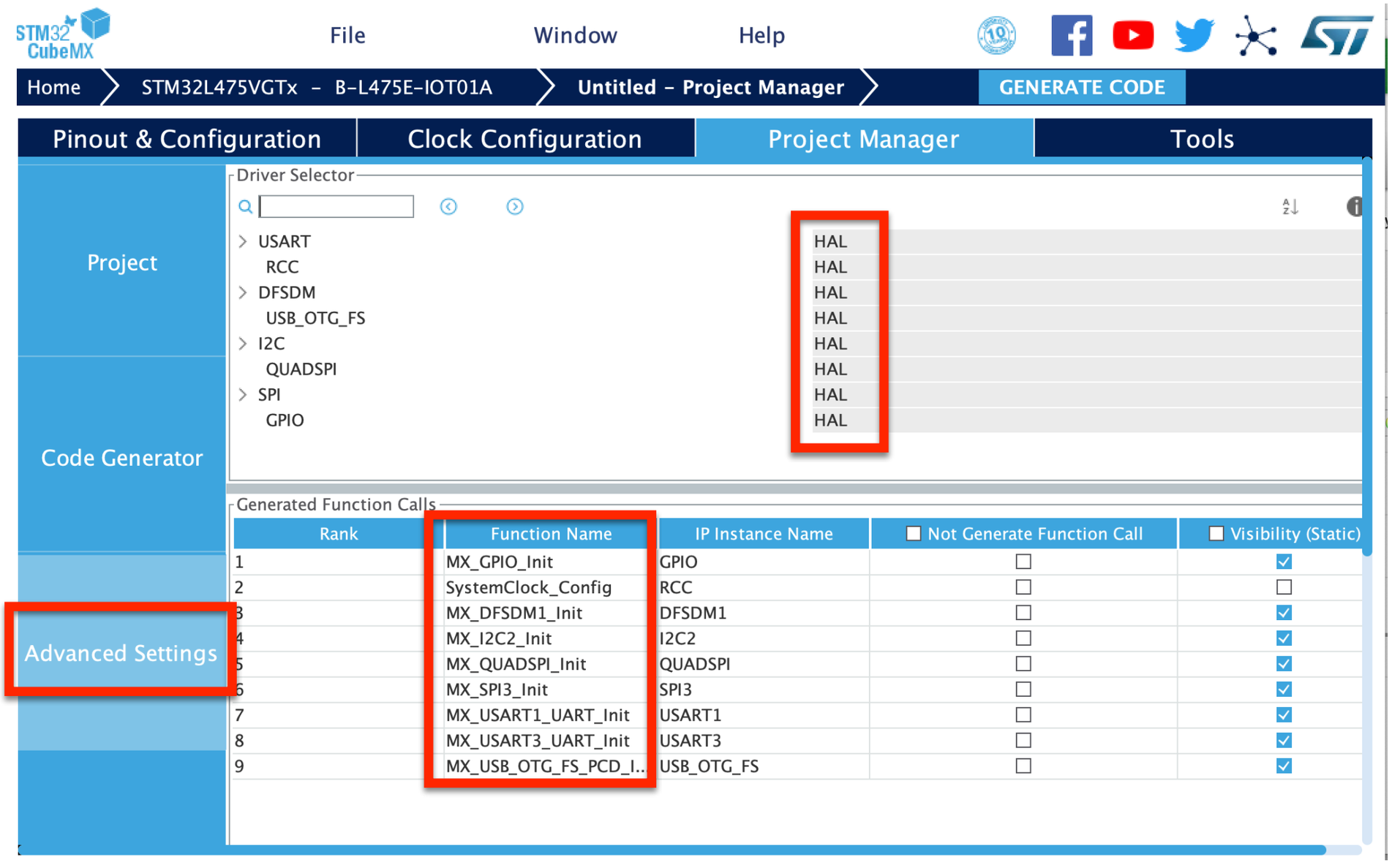
Step: Observe location where project will be stored



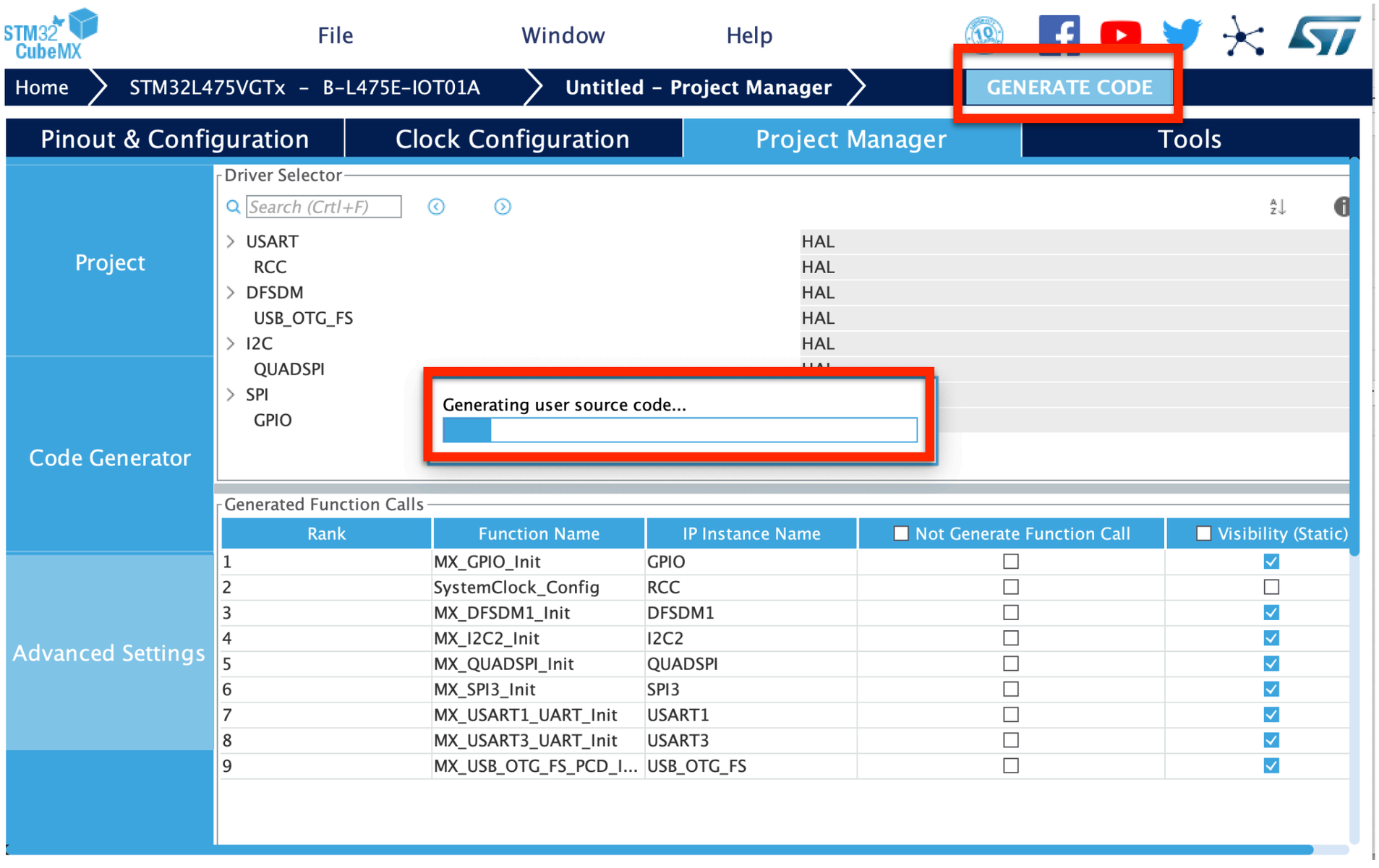
Step: Select “TrueStudio” for Toolchain / IDE



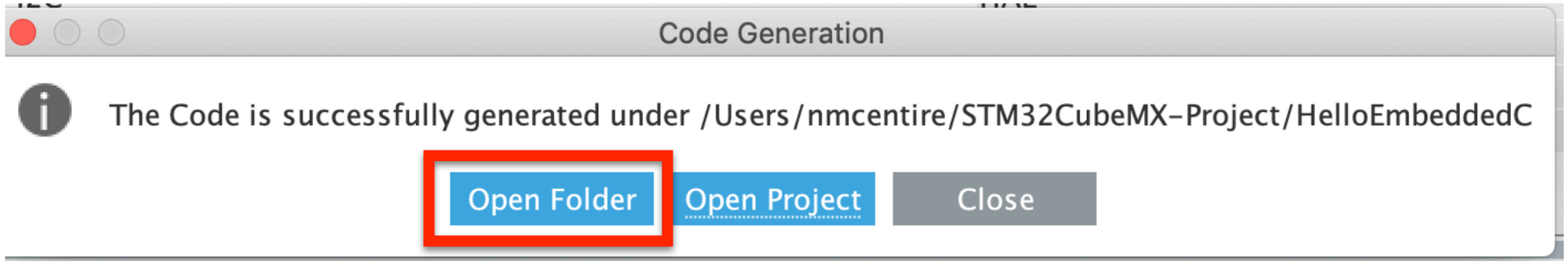
Step: Click on “Advanced Settings”, observing HAL and “Function Name”



Step: Click on “Generate Code”

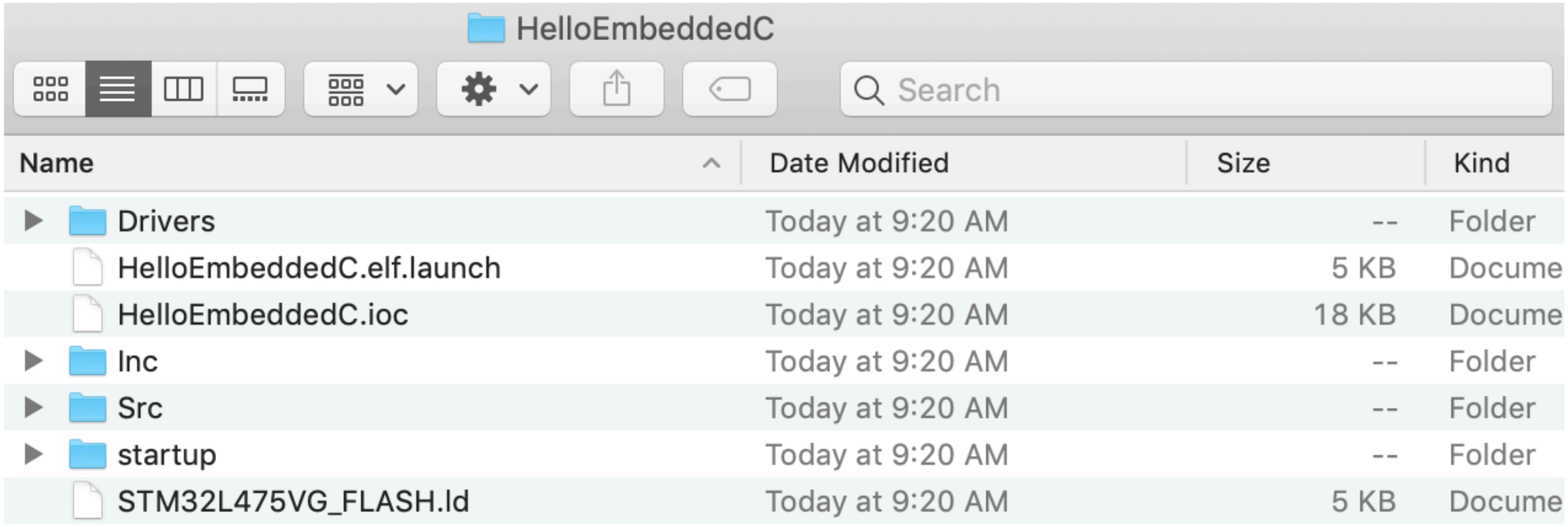


Step: Select “Open Folder”



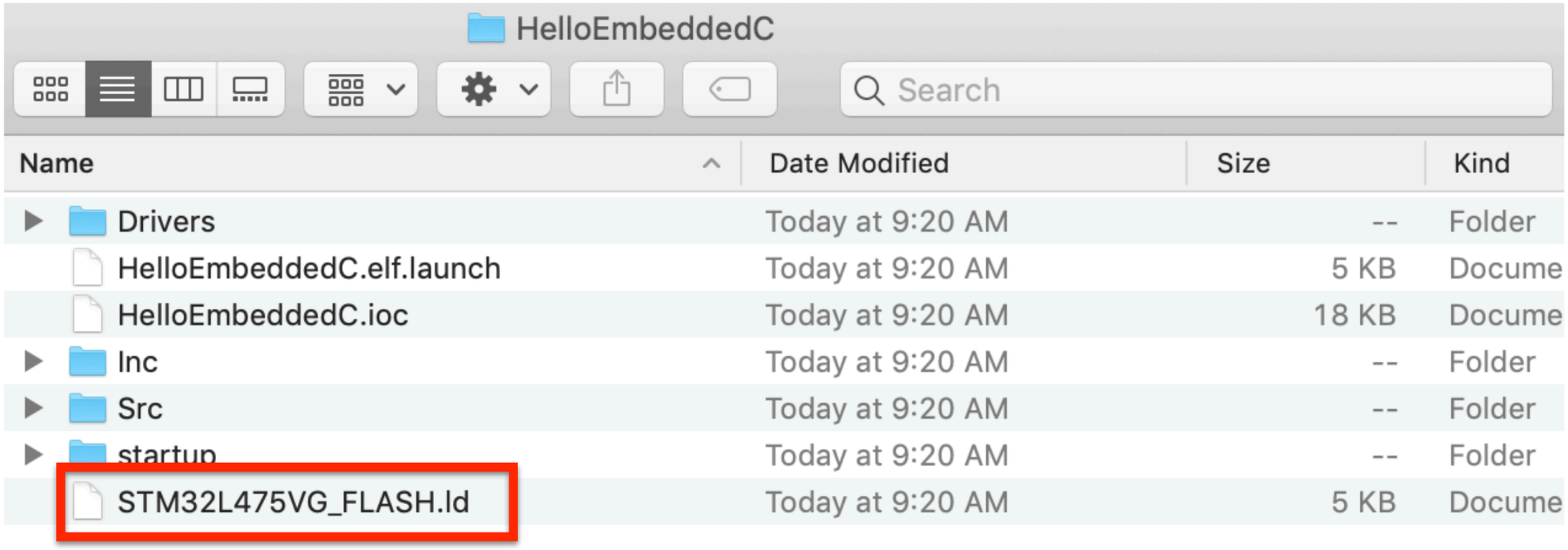
Tour of Generated Project

Step: View Open Folder



Note: The following slides will give a brief tour of key files/directories in the project.

Step: View STM32L475VG_FLASH.ld File (view as text file)



This is the “linker” file. See next slide.

Step: Linker Script - Part 1

```
**  File        : stm32_flash.ld
**
**  Abstract    : Linker script for STM32L475VG Device with
**                1024KByte FLASH, 128KByte RAM
**
**                Set heap size, stack size and stack location according
**                to application requirements.
**
**                Set memory bank area and size if external memory is used.
**
**  Target      : STMicroelectronics STM32
```

Key Points:

=1= Linker script tied to specific STM32 device.

=2= Heap Size (where memory allocated from)

=3= Stack Size (for local variables in functions)

=4= Notice comment about “Memory Bank”

Step: Linker Script - Part 2

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = 0x20018000;    /* end of RAM */
/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0x200;      /* required amount of heap  */
_Min_Stack_Size = 0x400; /* required amount of stack */

/* Specify the memory areas */
MEMORY
{
RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 96K
RAM2 (xrw)       : ORIGIN = 0x10000000, LENGTH = 32K
FLASH (rx)       : ORIGIN = 0x8000000, LENGTH = 1024K
}
```

Key Points:

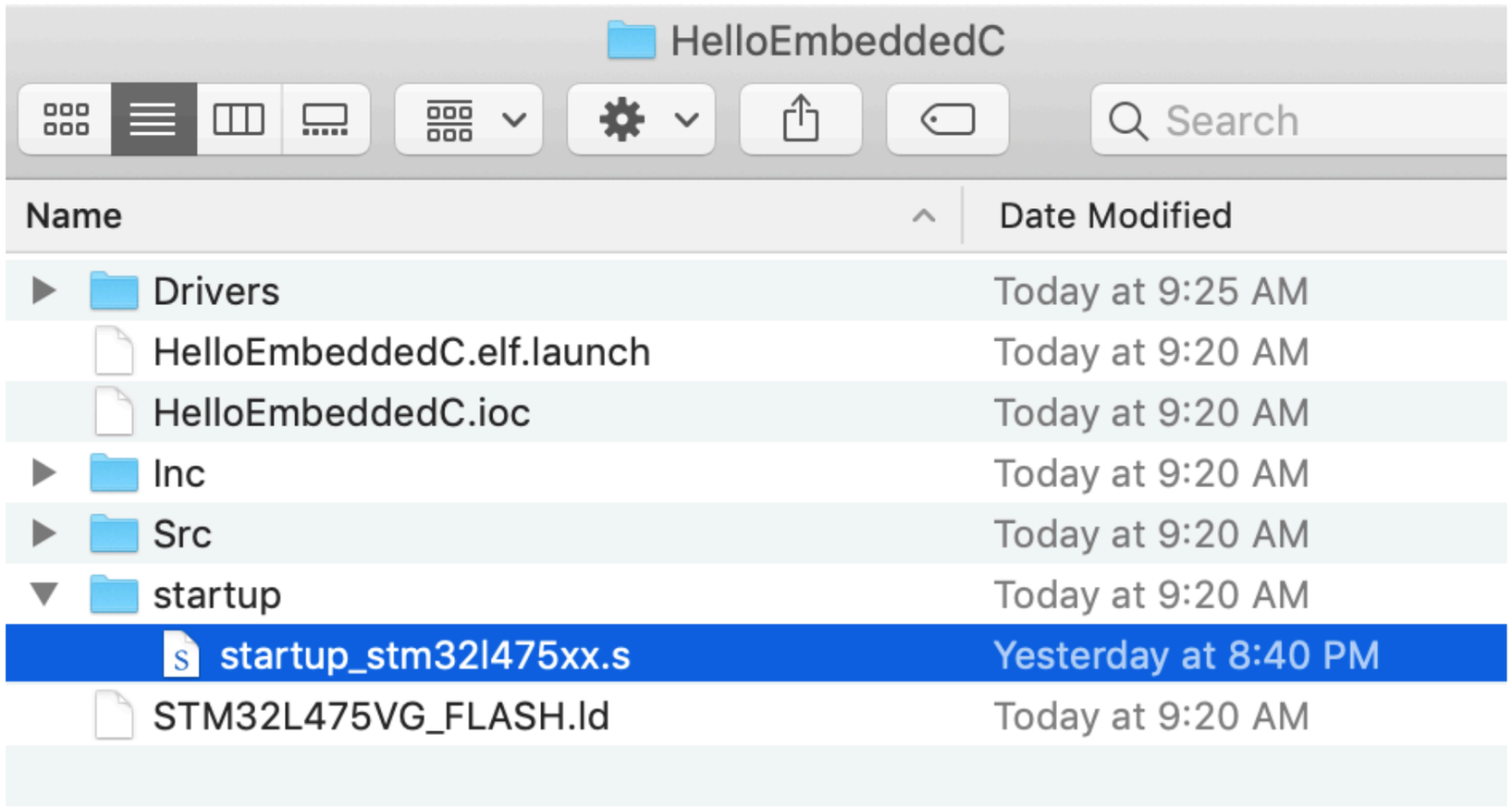
=1= At powerup/reset, ENTRY is Reset_Handler

=2= 0x2001_8000 as end of RAM for top of stack

=3= Min Heap and Stack Sizes

=4= Memory Addresses, RAM, RAM2, FLASH

Step: View “startup_stm32l475xx.s”



Note: .s is assembly language code!
See next slide

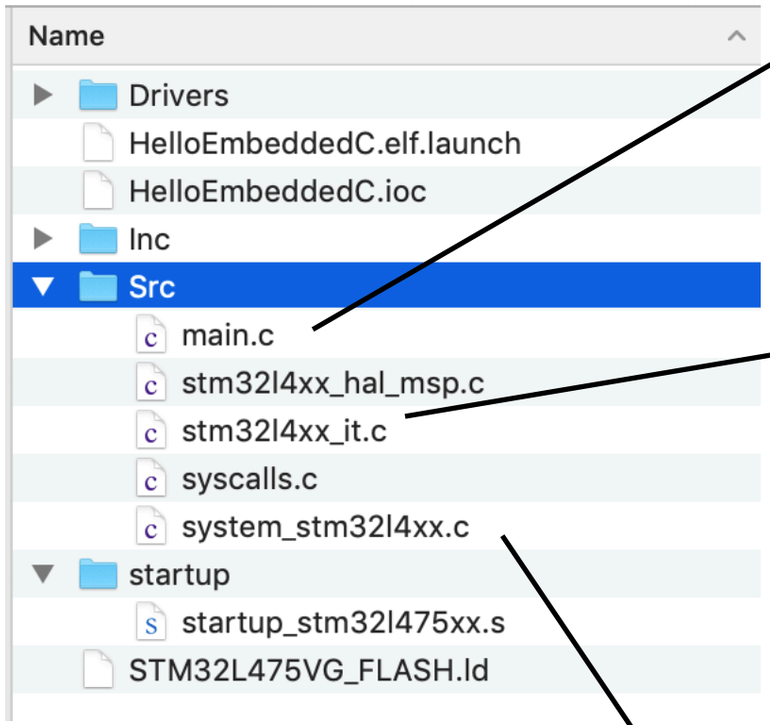
Step: View “startup_stm32l475xx.s”

```
Reset_Handler:
  ldr   sp, =_estack    /* Atollic update: set stack pointer */

/* Copy the data segment initializers from flash to SRAM */
  movs  r1, #0
  b     LoopCopyDataInit

CopyDataInit:
        ldr     r3, =_sidata
        ldr     r3, [r3, r1]
        str     r3, [r0, r1]
        adds    r1, r1, #4

LoopCopyDataInit:
        ldr     r0, =_sdata
        ldr     r3, =_edata
        adds    r2, r0, r1
        cmp     r2, r3
        bcc     CopyDataInit
        ldr     r2, =_sbss
        b       LoopFillZerobss
/* Zero fill the bss segment. */
FillZerobss:
        movs    r3, #0
        str     r3, [r2], #4

LoopFillZerobss:
        ldr     r3, = _ebss
        cmp     r2, r3
        bcc     FillZerobss

/* Call the clock system intitialization function.*/
    bl  SystemInit
/* Call static constructors */
    bl __libc_init_array
/* Call the application's entry point.*/
        bl      main
```

Code starts executing here

Code jumps to main() Embedded C code

Step: View “Src” directory

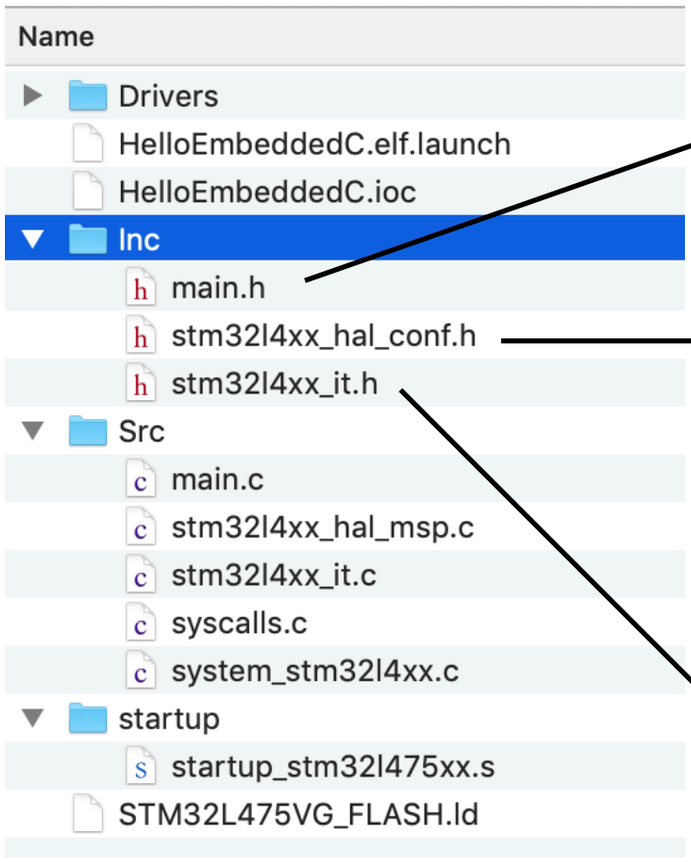



```
/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
```

```
/**
  ******************************************
  * @file    stm32l4xx_it.c
  * @brief   Interrupt Service Routines.
  ******************************************
```

```
/**
  ******************************************************************************
  * @file    system_stm32l4xx.c
  * @author  MCD Application Team
  * @brief   CMSIS Cortex-M4 Device Peripheral Access Layer System Source File
```

Step: View “Inc” directory




```
******************************************************************************
* @file           : main.h
* @brief          : Header for main.c file.
*                   This file contains the common defines of the application.
******************************************************************************
```

```
******************************************************************************
* @file    stm32l4xx_hal_conf.h
* @brief   HAL configuration file.
******************************************************************************
```

```
******************************************************************************
* @file    stm32l4xx_it.h
* @brief   This file contains the headers of the interrupt handlers.
******************************************************************************
```

Step: View “Inc/main.h”

```
/* Private defines -----------------------------------------
#define M24SR64_Y_RF_DISABLE_Pin GPIO_PIN_2
#define M24SR64_Y_RF_DISABLE_GPIO_Port GPIOE
#define USB_OTG_FS_OVRCR_EXTI3_Pin GPIO_PIN_3
#define USB_OTG_FS_OVRCR_EXTI3_GPIO_Port GPIOE
#define M24SR64_Y_GPO_Pin GPIO_PIN_4
#define M24SR64_Y_GPO_GPIO_Port GPIOE
#define SPSGRF_915_GPIO3_EXTI5_Pin GPIO_PIN_5
#define SPSGRF_915_GPIO3_EXTI5_GPIO_Port GPIOE
#define SPSGRF_915_GPIO3_EXTI5_EXTI_IRQn EXTI9_5_IRQn
#define SPBTLE_RF_IRQ_EXTI6_Pin GPIO_PIN_6
#define SPBTLE_RF_IRQ_EXTI6_GPIO_Port GPIOE
#define SPBTLE_RF_IRQ_EXTI6_EXTI_IRQn EXTI9_5_IRQn
#define BUTTON_EXTI13_Pin GPIO_PIN_13
#define BUTTON_EXTI13_GPIO_Port GPIOC
#define BUTTON_EXTI13_EXTI_IRQn EXTI15_10_IRQn
#define ARD_A5_Pin GPIO_PIN_0
#define ARD_A5_GPIO_Port GPIOC
#define ARD_A4_Pin GPIO_PIN_1
#define ARD_A4_GPIO_Port GPIOC
#define ARD_A3_Pin GPIO_PIN_2
#define ARD_A3_GPIO_Port GPIOC
#define ARD_A2_Pin GPIO_PIN_3
#define ARD_A2_GPIO_Port GPIOC
#define ARD_D1_Pin GPIO_PIN_0
#define ARD_D1_GPIO_Port GPIOA
#define ARD_D0_Pin GPIO_PIN_1
```

Note: These are The GPIO pin definitions

Step: View “Inc/ stm32l4xx_hal_conf.h”

```
/* ########################## Module Selection ##########################
/**
  * @brief This is the list of modules to be used in the HAL driver
  */

#define HAL_MODULE_ENABLED
/*#define HAL_ADC_MODULE_ENABLED   */
/*#define HAL_CRYP_MODULE_ENABLED   */
/*#define HAL_CAN_MODULE_ENABLED   */
/*#define HAL_COMP_MODULE_ENABLED   */
/*#define HAL_CRC_MODULE_ENABLED   */
/*#define HAL_CRYP_MODULE_ENABLED   */
/*#define HAL_DAC_MODULE_ENABLED   */
/*#define HAL_DCMI_MODULE_ENABLED   */
/*#define HAL_DMA2D_MODULE_ENABLED   */
#define HAL_DFSDM_MODULE_ENABLED
/*#define HAL_DSI_MODULE_ENABLED   */
/*#define HAL FIREWALL MODULE ENABLED   */
```

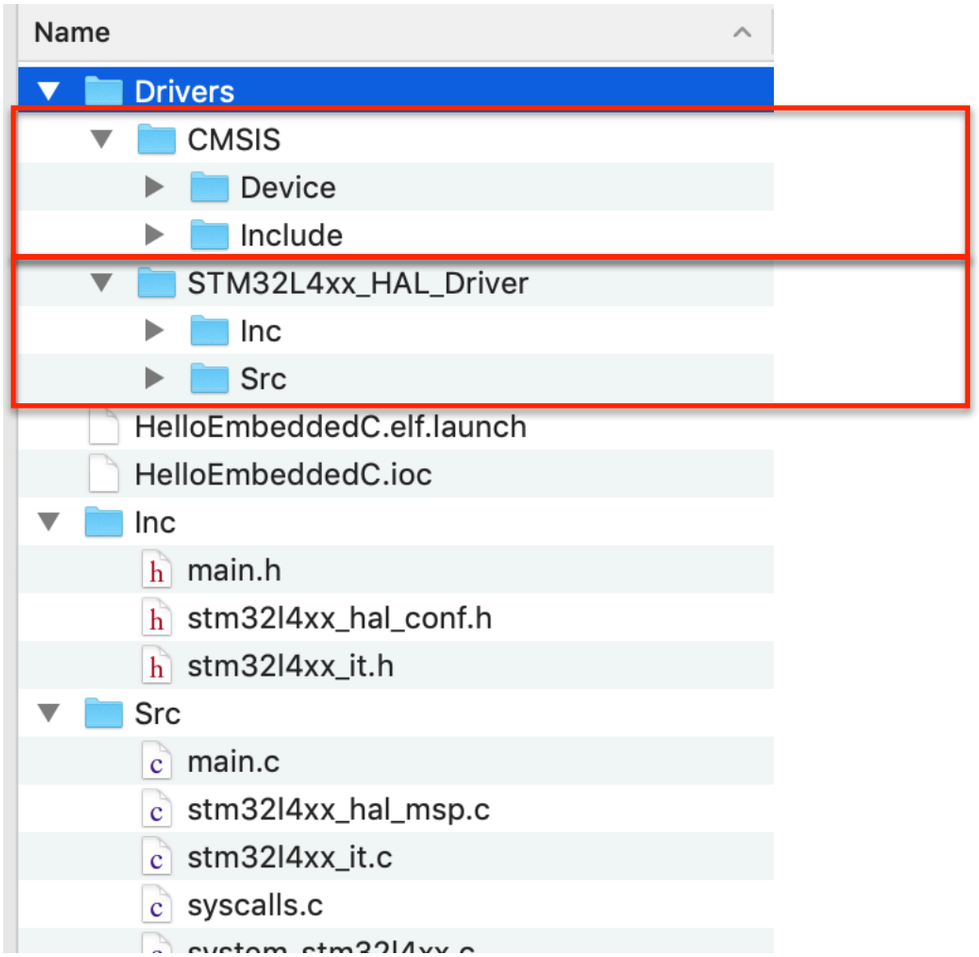
Note: These #defines determine what HAL features are include in build

Step: View “Inc/ stm32l4xx_it.h”

```
/* Exported functions prototypes --------
void NMI_Handler(void);
void HardFault_Handler(void);
void MemManage_Handler(void);
void BusFault_Handler(void);
void UsageFault_Handler(void);
void SVC_Handler(void);
void DebugMon_Handler(void);
void PendSV_Handler(void);
void SysTick_Handler(void);
void EXTI9_5_IRQHandler(void);
void EXTI15_10_IRQHandler(void);
/* USER CODE BEGIN EFP */
```

Note: These are function prototypes for the interrupt handlers

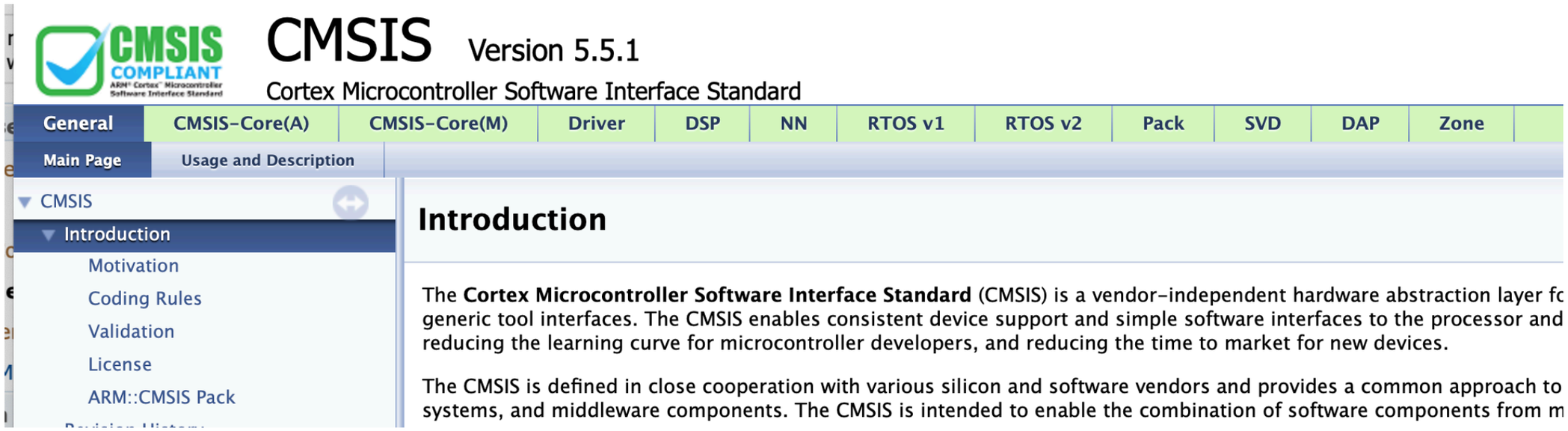
Step: View “Drivers” directory



CMSIS
Cortex M Software Interface Standard
(Vendor Independent HAL)

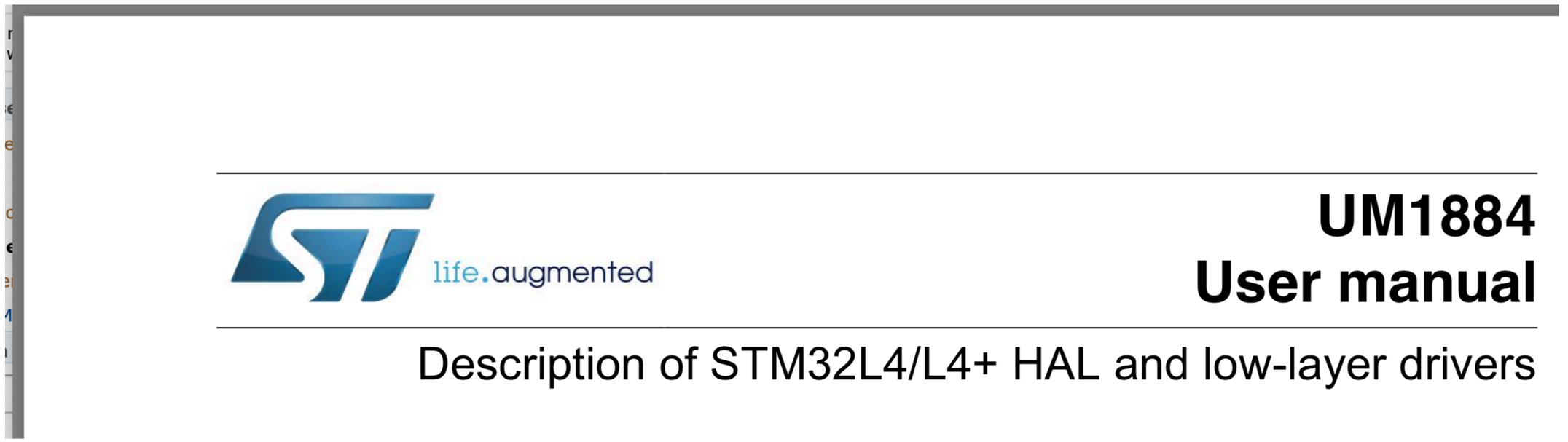
HAL
Hardware Abstraction Layer
(Vendor Dependent HAL)

http://www.keil.com/pack/doc/CMSIS/General/html/index.html



Vendor-Independent Hardware Abstraction Layer

https://www.st.com/content/ccc/resource/technical/document/user_manual/63/a8/8f/e3/ca/a1/4c/84/DM00173145.pdf/files/DM00173145.pdf/jcr:content/translations/en.DM00173145.pdf

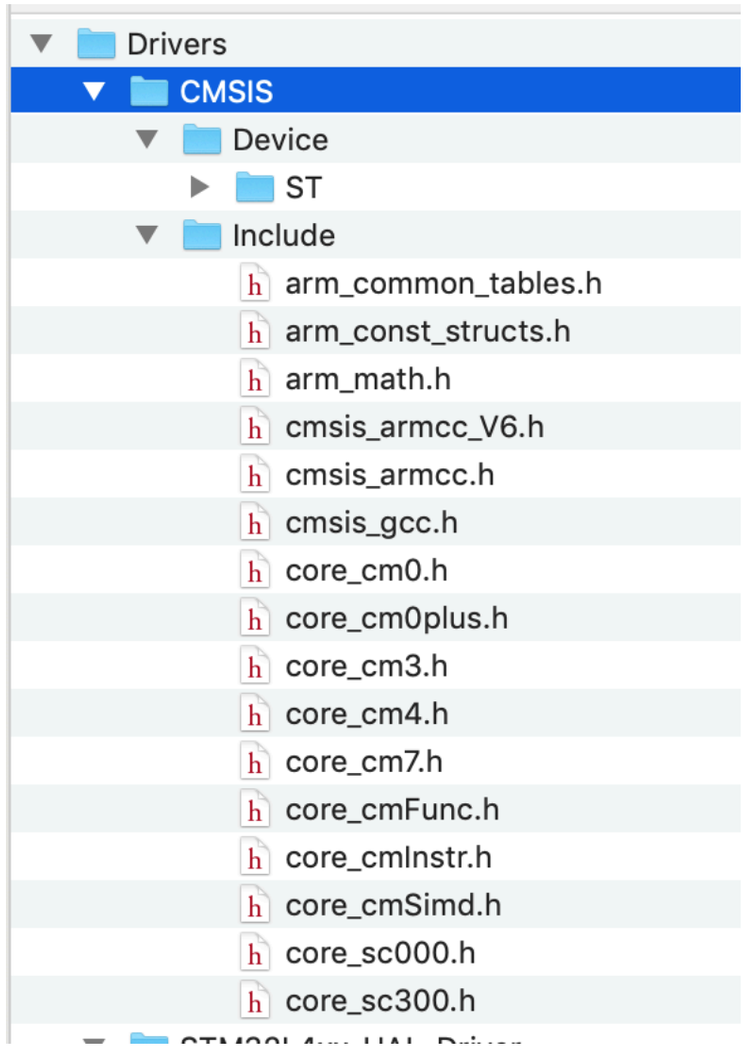


life.augmented

UM1884
User manual

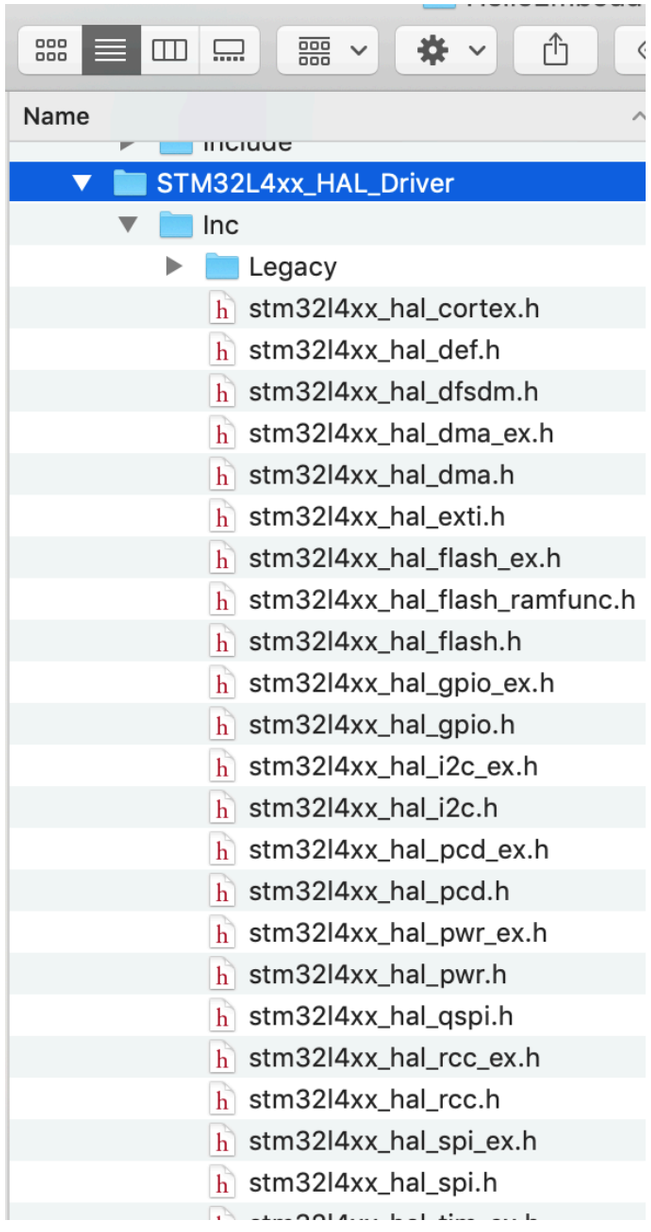
Description of STM32L4/L4+ HAL and low-layer drivers

Step: View “CMSIS API”



Notes:
These are focused on the Cortex M processor and not I/O devices.

Step: View “HAL API”



Notes:

=1= stm32l4xx_hal_...

=2= stm32l4xx_gpio.h

=3= stm32l4xx_gpio_ex.h

Summary so far…

- STM32CubeMX generates Embedded C Code
- Code Well organized
  - Drivers - Pre-Written CMSIS and HAL Code
  - Inc - Config Settings
  - Src - You'll begin your coding with main.c
  - startup - The assembly language startup code

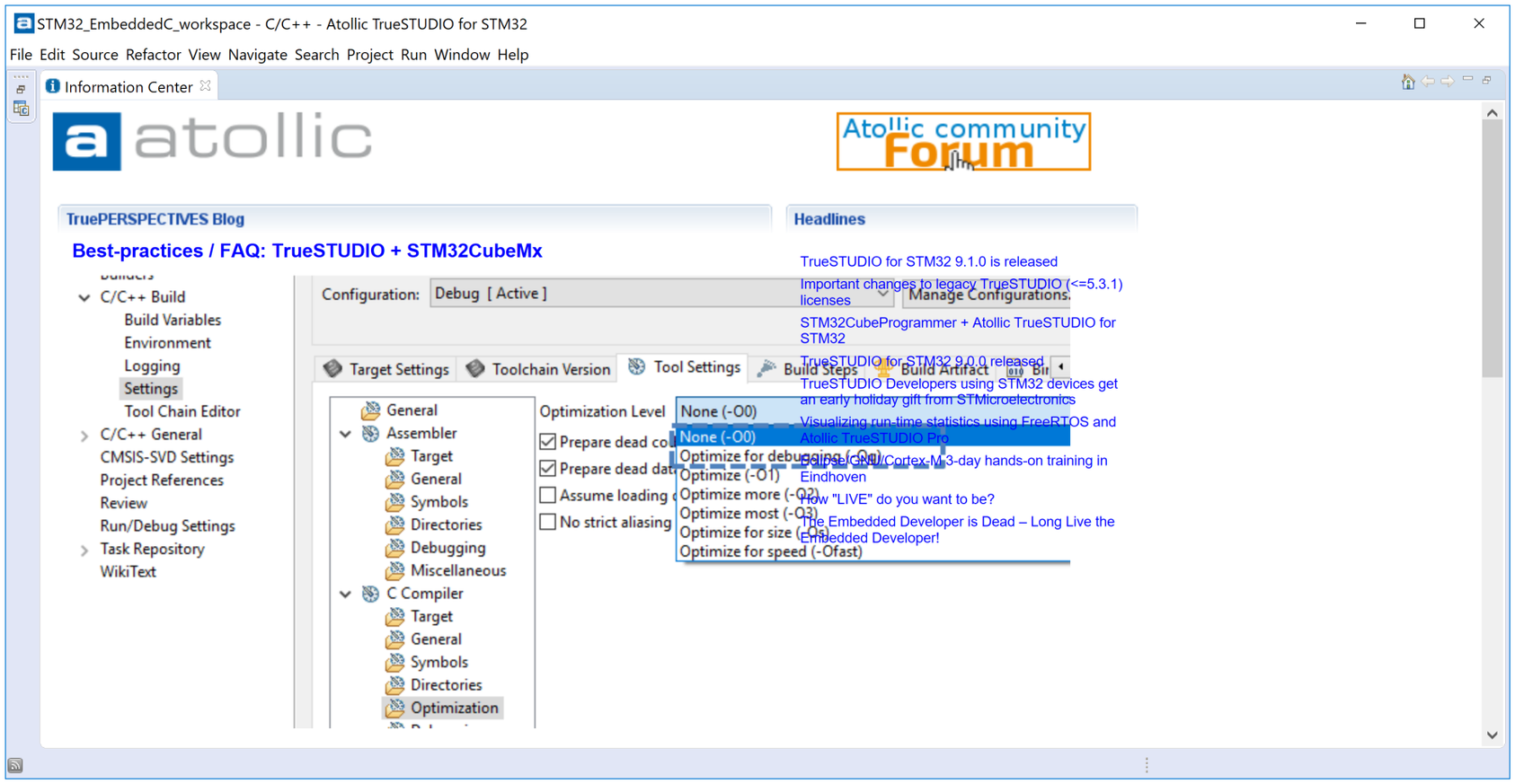
TrueStudio

Step: Install TrueStudio

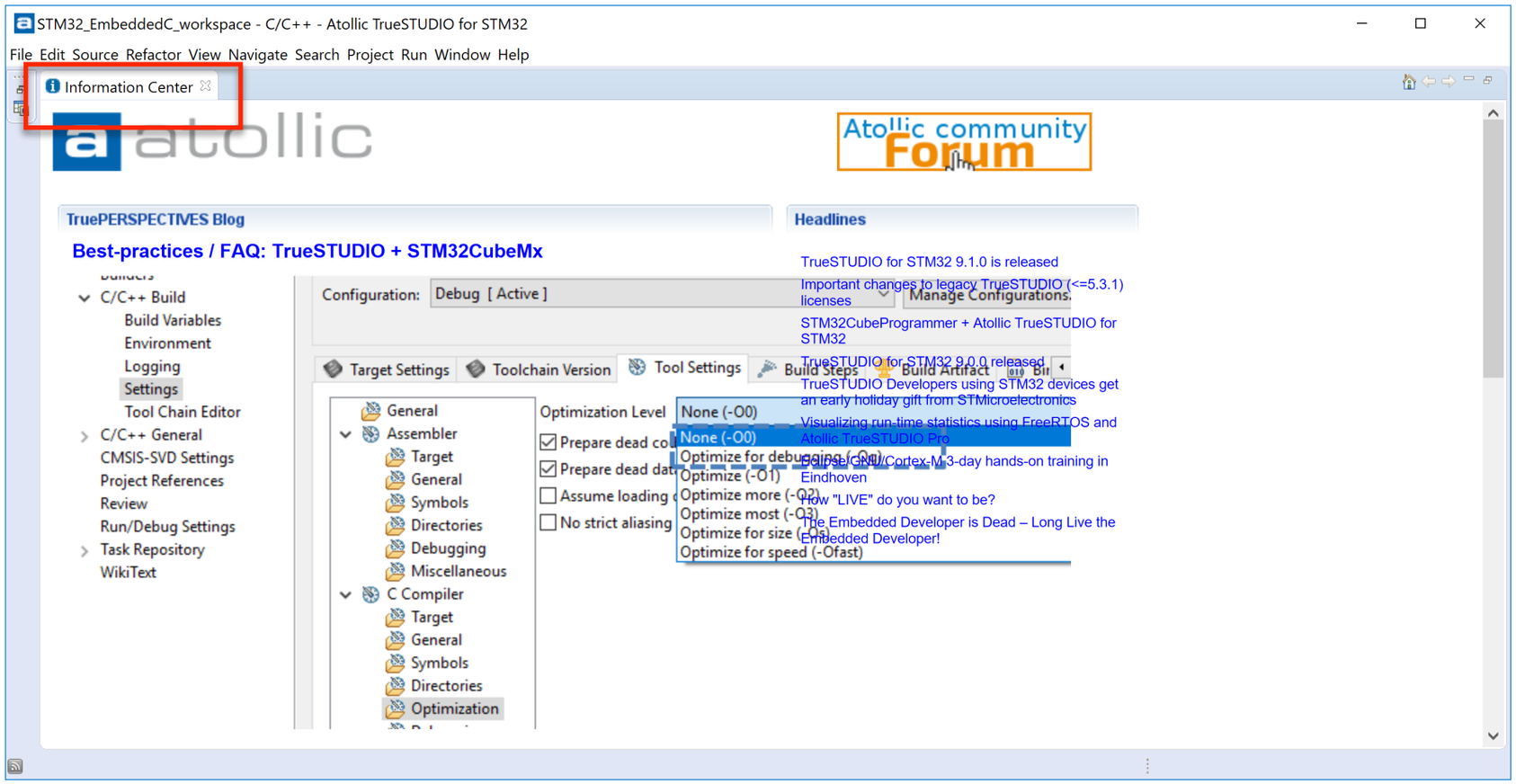
- https://atollic.com/truestudio/

Note: TrueStudio works on Windows and Linux but not Mac

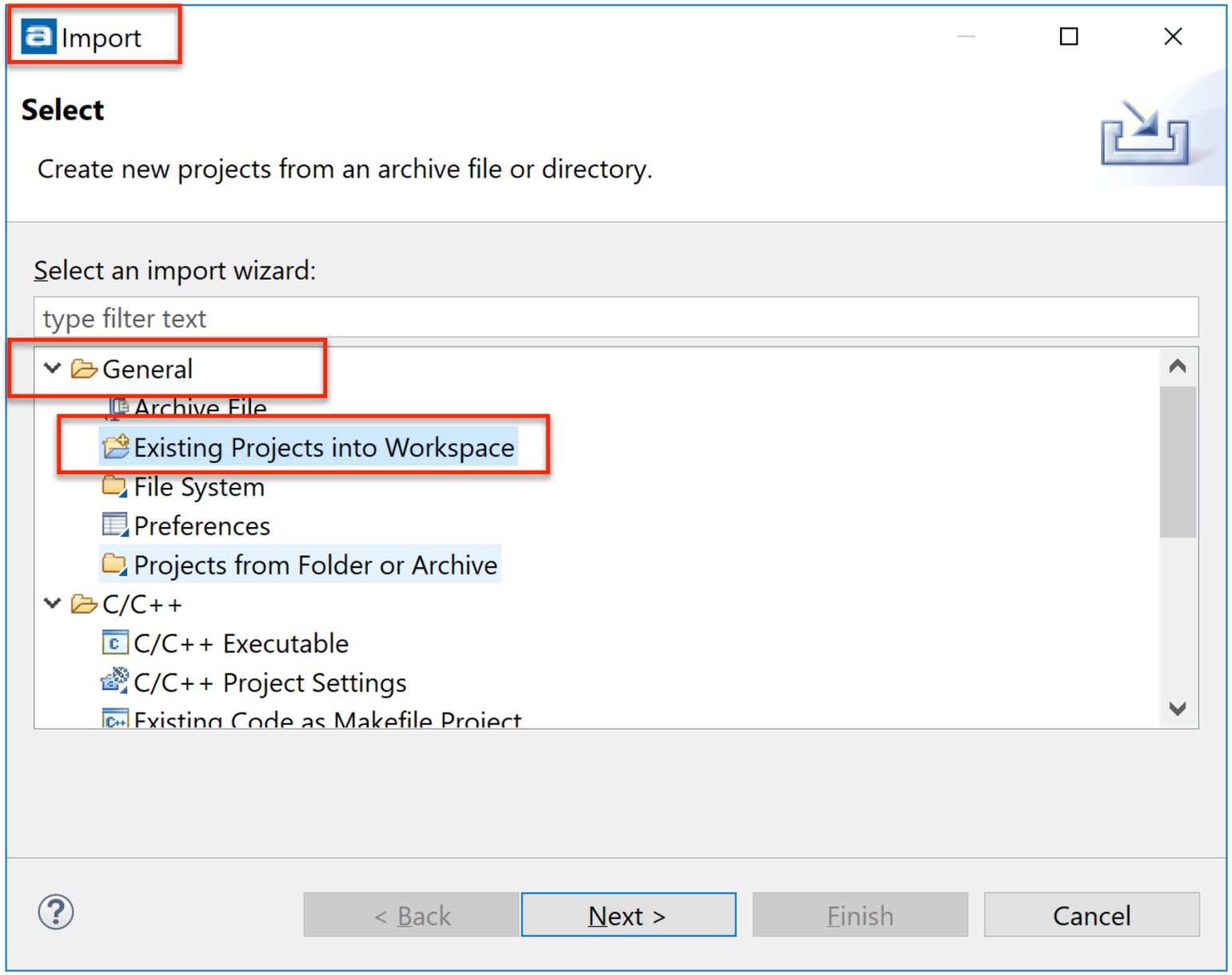
Step: Startup TrueStudio



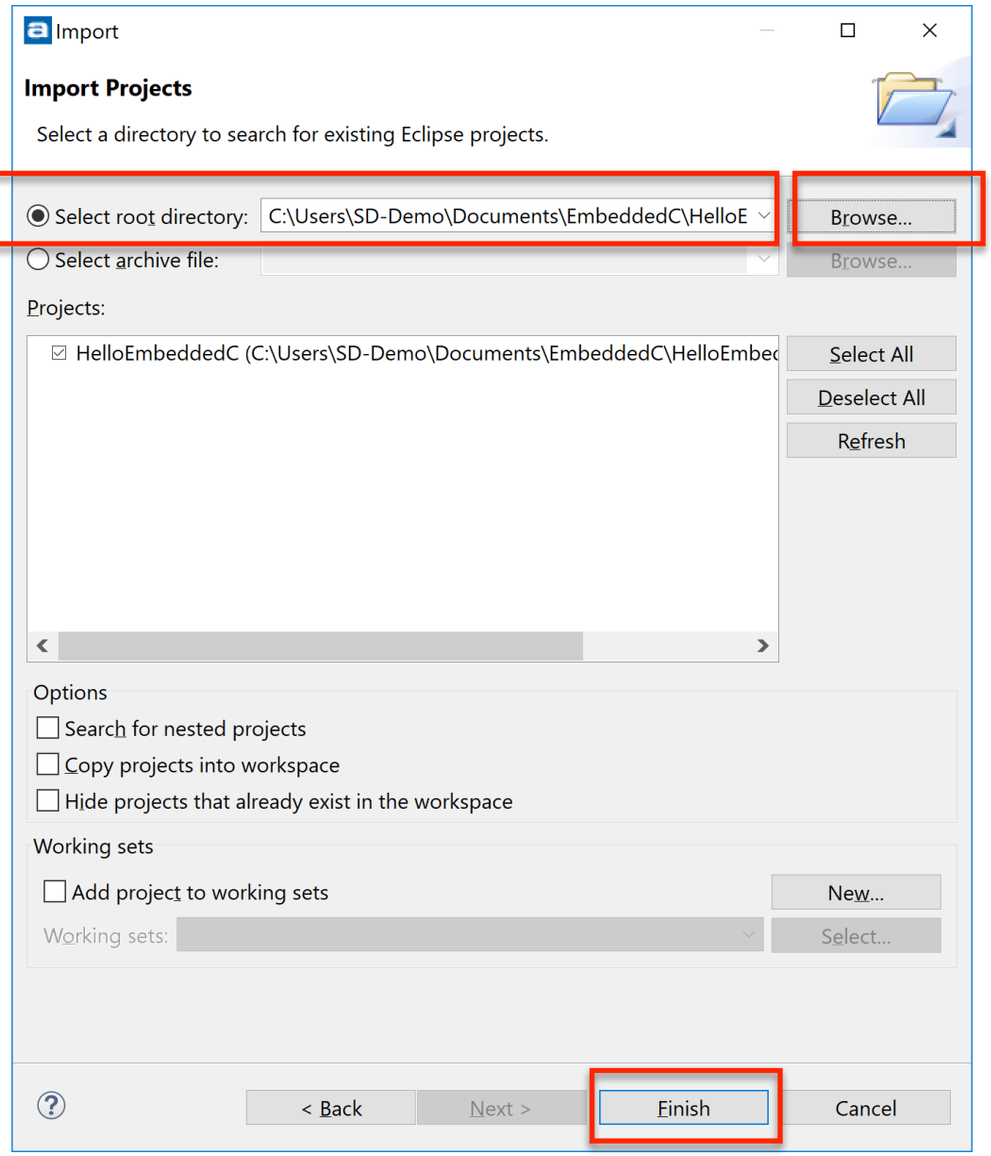
Step: Click to Close “Information Center”



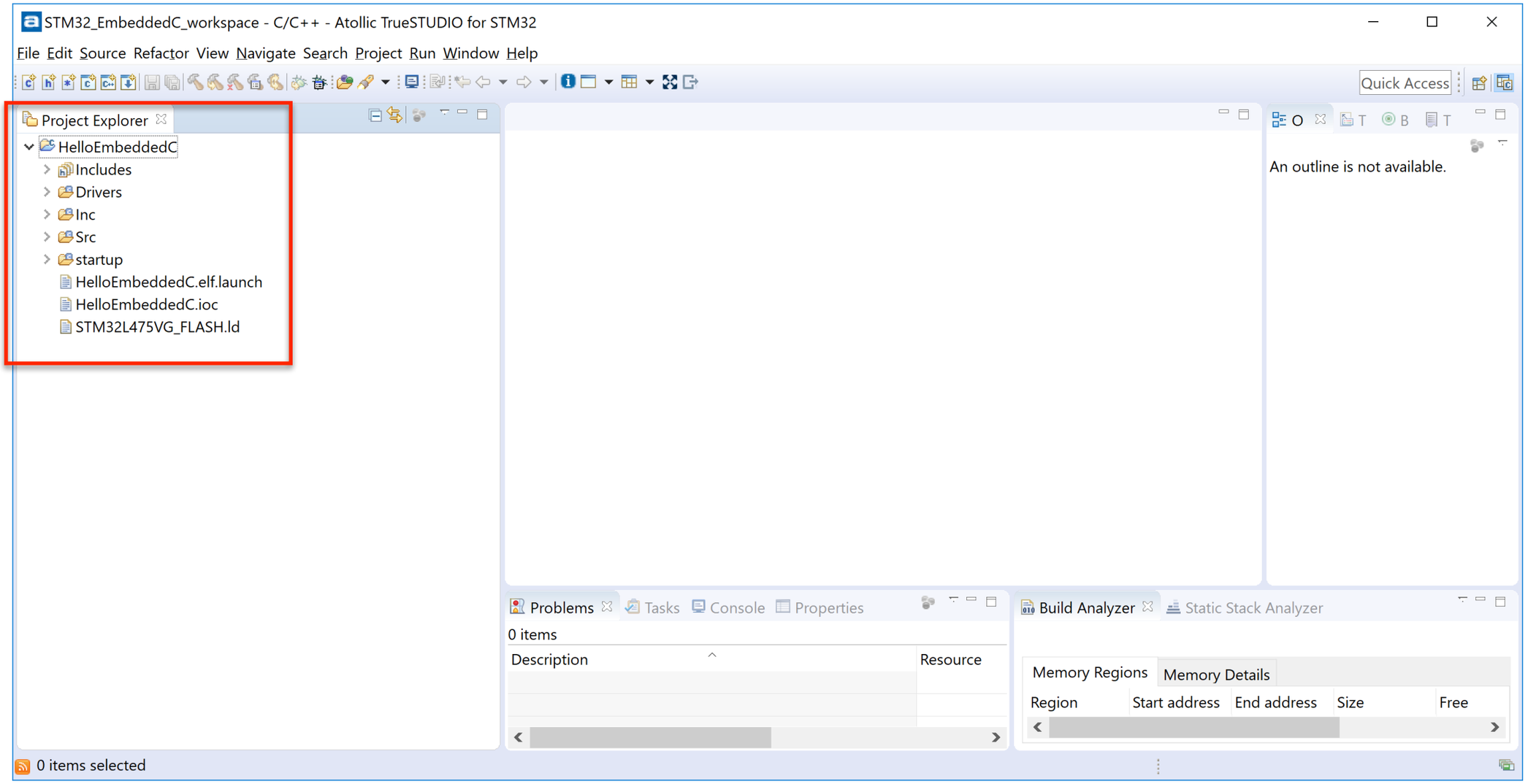
Step: File, Import, Projects, General, Existing Projects into Workspace



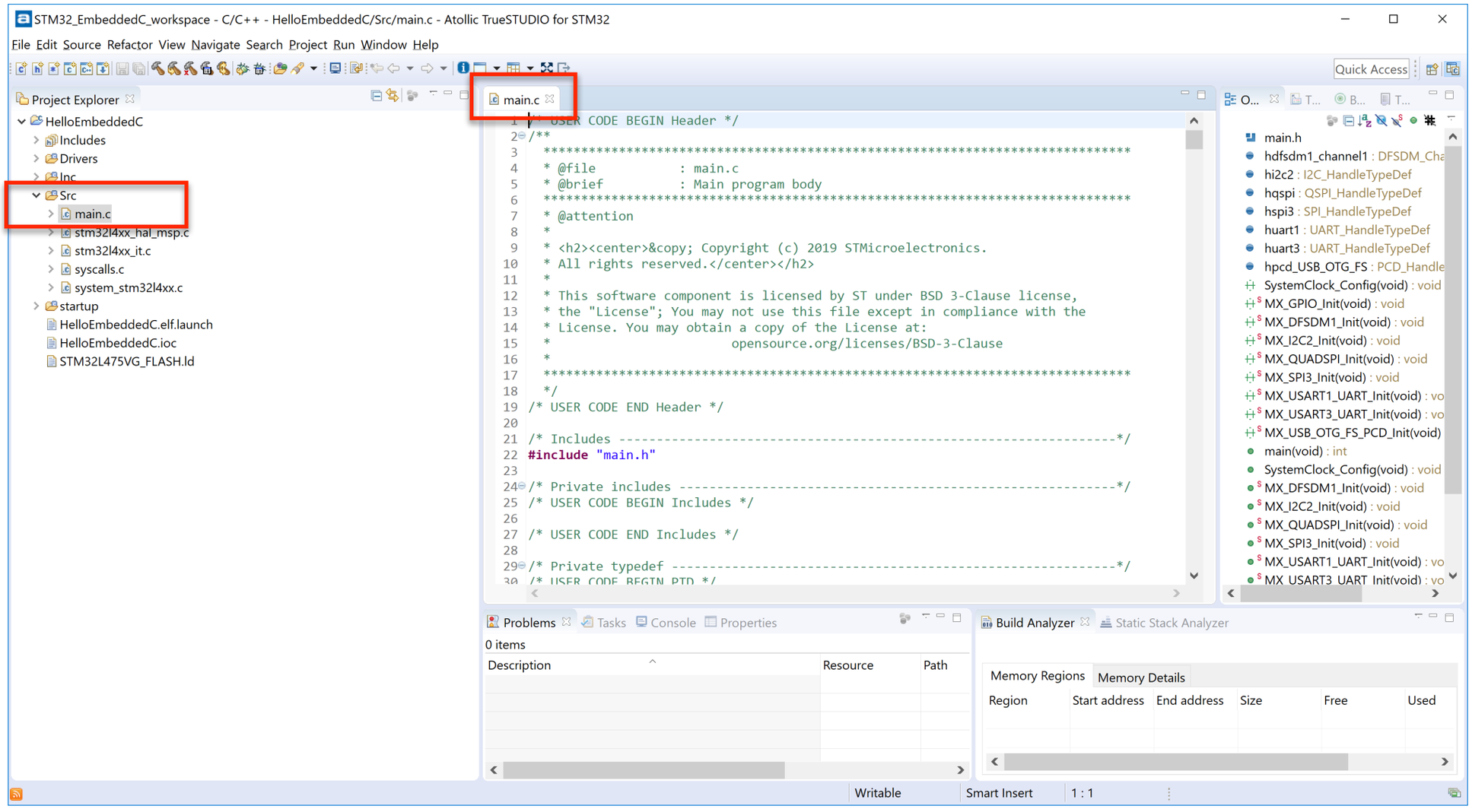
Step: Select Root Directory (Browse to directory as needed)



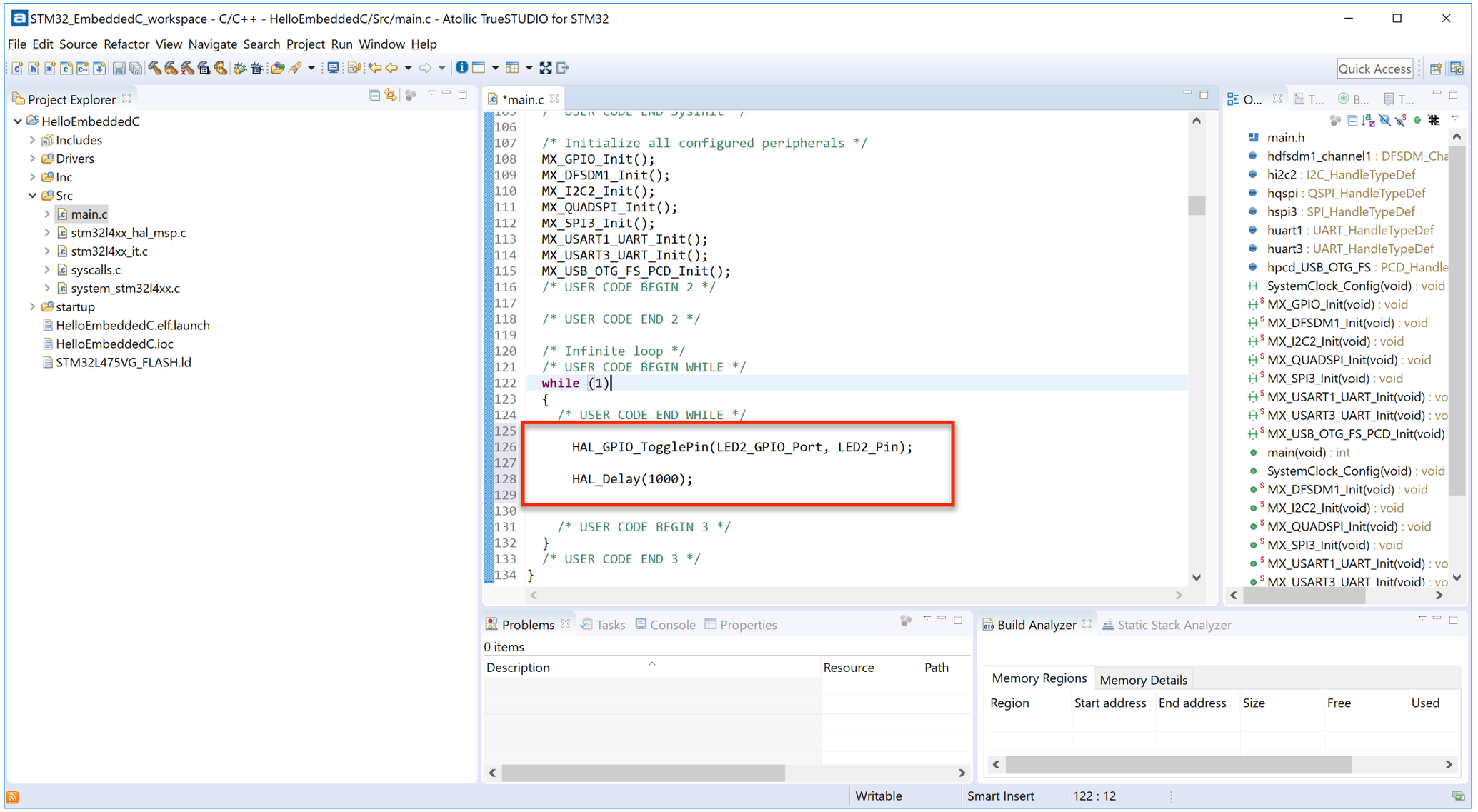
Step: Observe Results



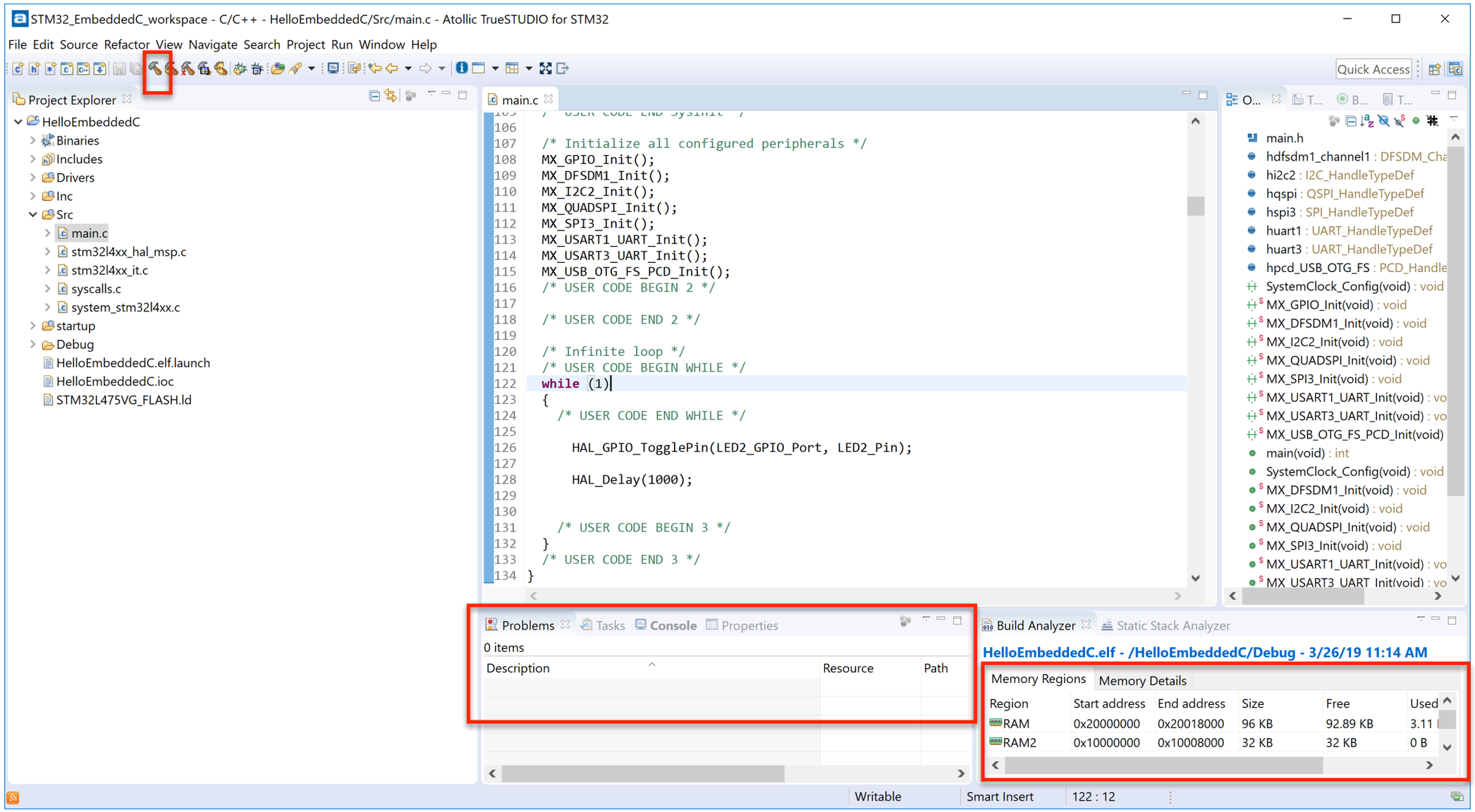
Step: Open main.c



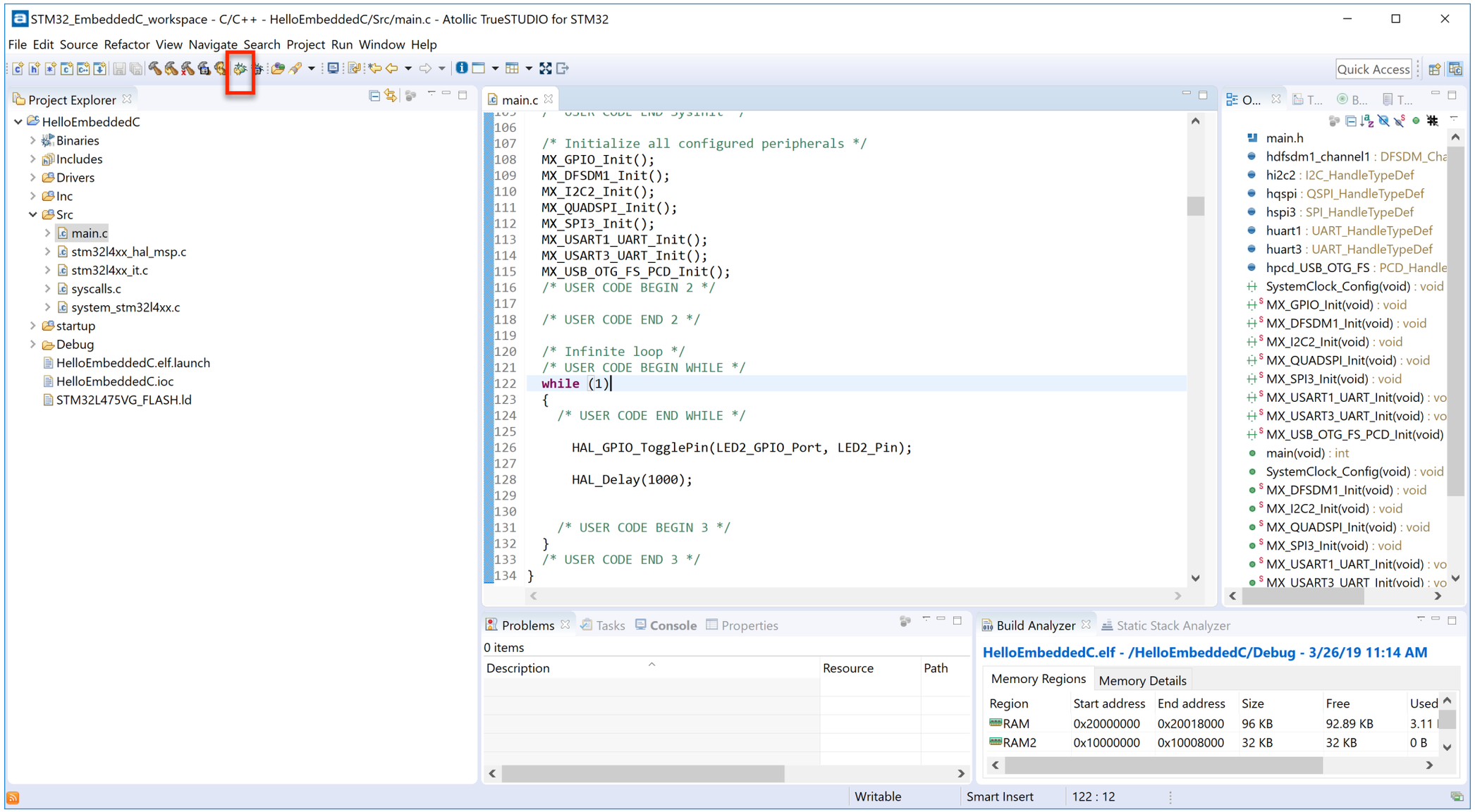
Step: Edit main.c - Add two functions: HAL_GPIO_TogglePin() and HAL_Delay()



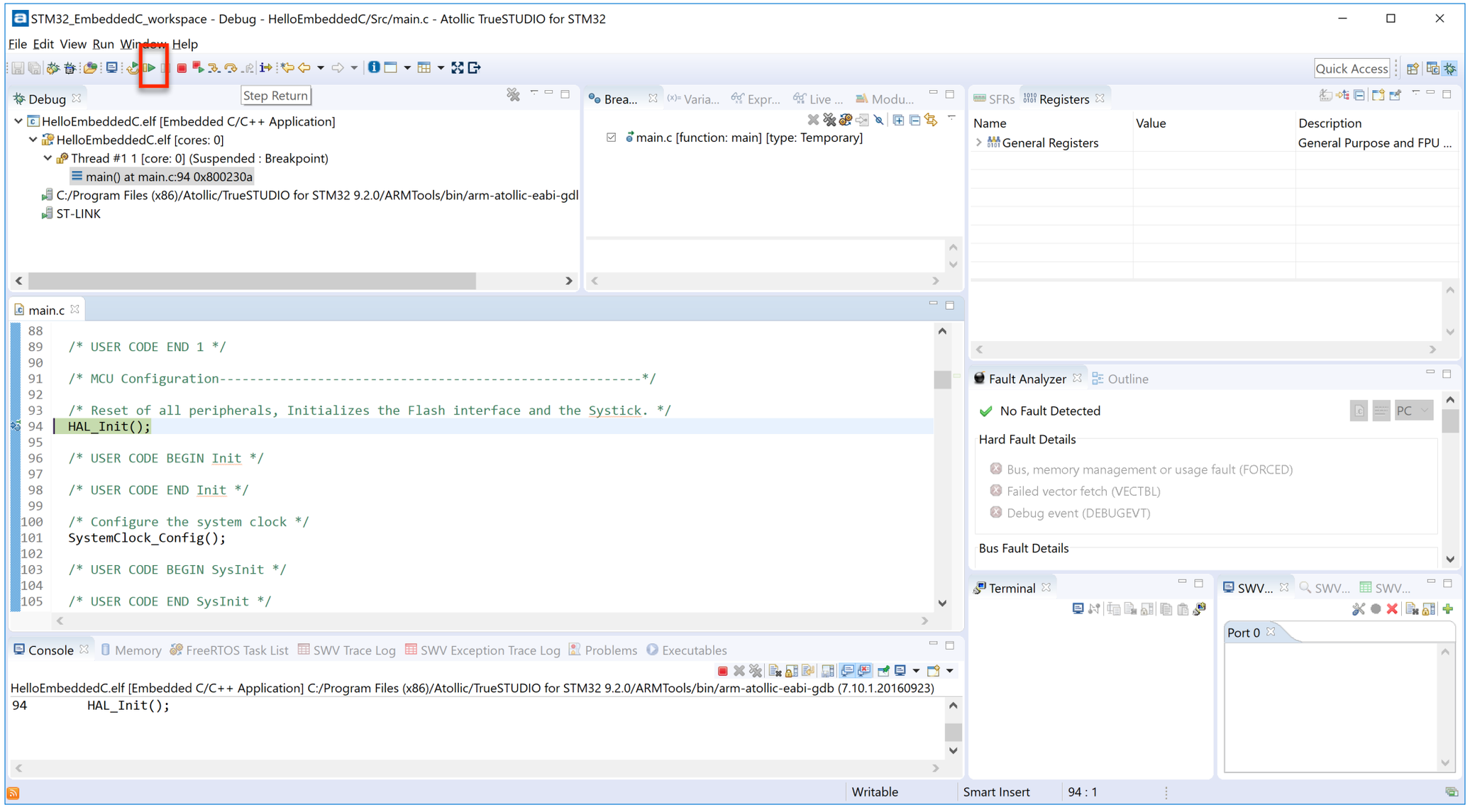
Step: Build Project



Step: Run Project in Debug Mode

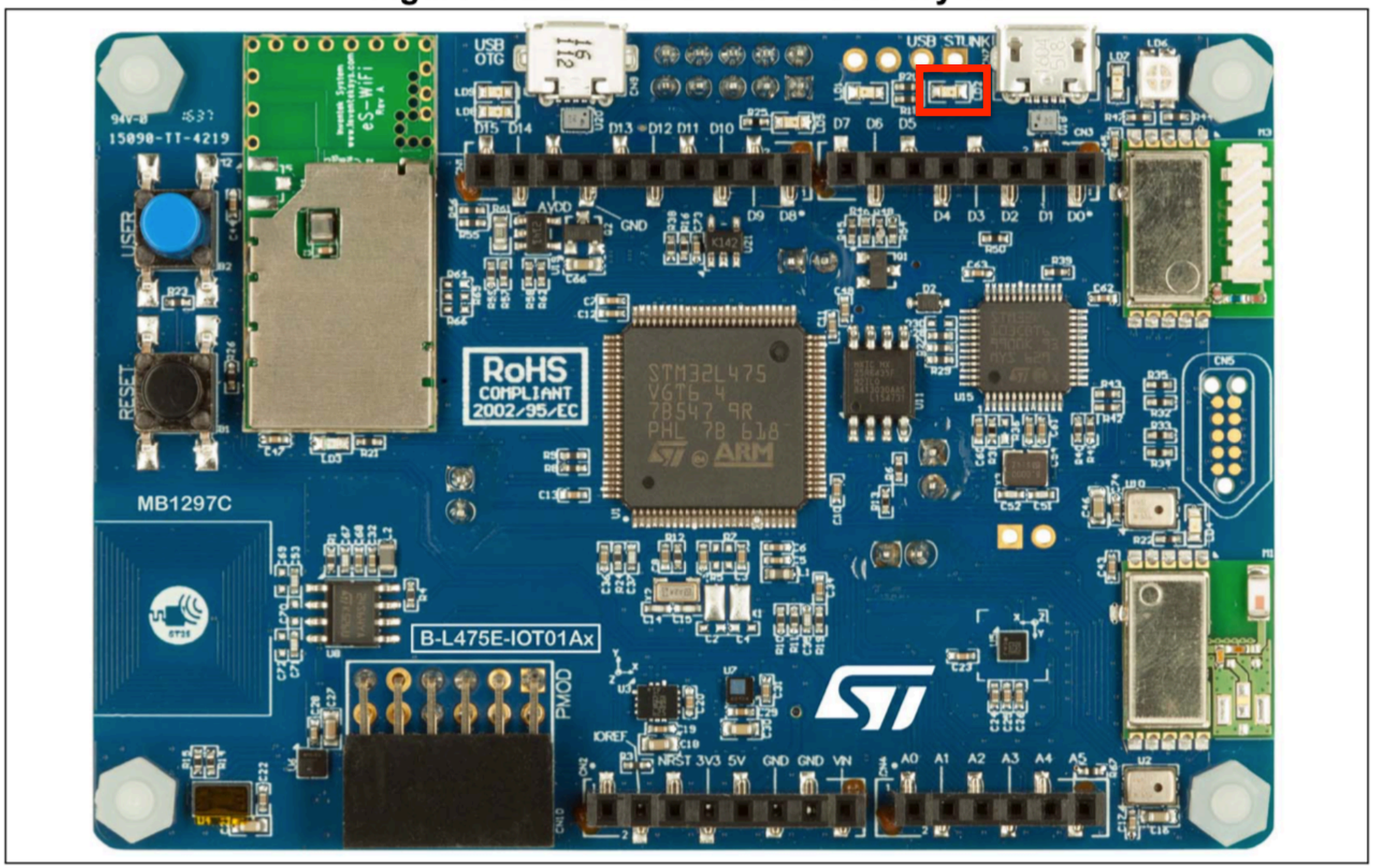


Step: Continue Running (after breakpoint hit)



Step: Look for LED blinking on STM Discovery Board at 1 second on/off rate

Figure 1. B-L475E-IOT01A Discovery kit



Summary

- Introduction to Embedded C
- STM32CubeMX
- Tour of Generated Project
- TrueStudio
- Hello Blinking LED