NUCLEO-G474RE

Power Electronics Project Series

Project Documentation

01

Basic Led Blink

Implementation Frameworks:

STM32CubeIDE | MATLAB Embedded Coder

Document Version: 1.0

Last Updated: February 9, 2026

GitHub: `https://github.com/Anmol-G-K/NUCLEO-G474RE-PowerElectronics-Guide`

Contents

1 Project Overview

1.1 Objective

The primary objective of this project is to familiarise yourself with the STM32CubeIDE and MATLAB Embedded Coder development environments for programming the STM32 NUCLEO-G474RE microcontroller. This is accomplished through a simple yet effective LED blinking application.

For the STM32CubeIDE implementation, the on-board green LED (connected to pin PA5) blinks with a fixed delay of **100 milliseconds** (on and off). For the MATLAB Embedded Coder implementation, the same LED blinks with a delay of **1 second**. These different timing values serve to illustrate the distinct approaches of each development environment.

1.2 Applications

Whilst this project itself is introductory in nature, it establishes the foundational knowledge required for developing more complex systems:

- GPIO configuration and control using HAL (Hardware Abstraction Layer)
- Understanding development workflow in STM32CubeIDE
- Code generation and deployment using MATLAB Embedded Coder
- Digital output control for relay activation based on commands or feedback (future expansion)

1.3 Key Concepts

The fundamental concepts covered in this project include:

- Configuring General Purpose Input/Output (GPIO) pins using HAL APIs
- GPIO configuration using MATLAB Embedded Coder's STM32 Support Package
- Understanding the differences between manual firmware development and model-based design

1.4 Learning Outcomes

By completing this project, you will understand:

- How to use GPIO pins for digital output control
- Basic familiarity with STM32CubeIDE and its project structure
- How to generate and configure embedded systems using STM32CubeMX
- The workflow and capabilities of MATLAB Embedded Coder for STM32 targets
- The practical differences between traditional embedded C programming and model-based design approaches

2 STM32CubeIDE Implementation

2.1 Prerequisites

Before starting this implementation, ensure you have:

- *STM32CubeIDE version 1.10 or later installed*
- *NUCLEO-G474RE development board*
- *USB cable for programming and debugging*
- *Basic familiarity with embedded C programming*
- *Knowledge of microcontroller peripherals*

2.2 Hardware Configuration

2.2.1 Pin Configuration

The NUCLEO-G474RE development board features an on-board green LED connected to pin PA5. This project utilises this LED for the blinking demonstration.

Table 1: Pin Configuration for LED Blink Project

Pin	Function	Description
PA5	GPIO Output	On-board green LED (LD2)

2.2.2 Peripheral Configuration

The GPIO peripheral configuration for this project is defined as follows:

- **GPIO Output Level:** Low (LED off by default)
- **Mode:** Output Push-Pull (provides both source and sink capability)
- **GPIO Pull-up/Pull-down:** No pull-up or pull-down required
- **Maximum Output Speed:** Low speed (sufficient for LED driving)
- **User Label:** LD2 (on-board green LED)

Note: These configurations are automatically applied when the STM32CubeMX project is created for the NUCLEO-G474RE board. The .ioc (I/O Configuration) file contains all the peripheral settings and can be modified through the graphical interface.

2.3 Software Architecture

2.3.1 Project Structure

The STM32CubeIDE project generated for the NUCLEO-G474RE board follows a standard structure that separates device drivers, hardware abstraction layers, and application code:

```
CUBE_IDE/
|-- Core/
|   |-- Inc/
|   |   |-- main.h
|   |   `-- stm32g4xx_it.h
|   `-- Src/
|       |-- main.c
|       `-- stm32g4xx_it.c
|-- Drivers/
|   |-- STM32G4xx_HAL_Driver/
|   `-- CMSIS/
|-- Startup/
|   `-- startup_stm32g474xx.s
`-- .mxproject
```

Listing 1: STM32CubeIDE Project Directory Structure

The **Core/Src/main.c** file is where the primary application logic resides. STM32CubeIDE provides protected code regions marked with special comments, allowing developers to add custom code without it being overwritten during subsequent code generation cycles.

2.3.2 Initialisation Flow

The initialisation sequence of an STM32 microcontroller follows a standard pattern:

1. **System Clock Configuration**: The CPU clock is configured to the desired frequency
2. **Peripheral Initialisation**: All required peripherals (GPIO, timers, ADC, etc.) are initialised
3. **HAL Initialisation**: The Hardware Abstraction Layer is set up
4. **Application Code Loop**: The infinite loop in `main()` executes the application logic

For this LED blink project, we utilise the built-in HAL functions to control the GPIO pin, setting it high and low at regular intervals using the `HAL_Delay()` function.

2.3.3 Project Creation Workflow

The following images illustrate the step-by-step process of creating a new STM32CubeIDE project for the NUCLEO-G474RE board.

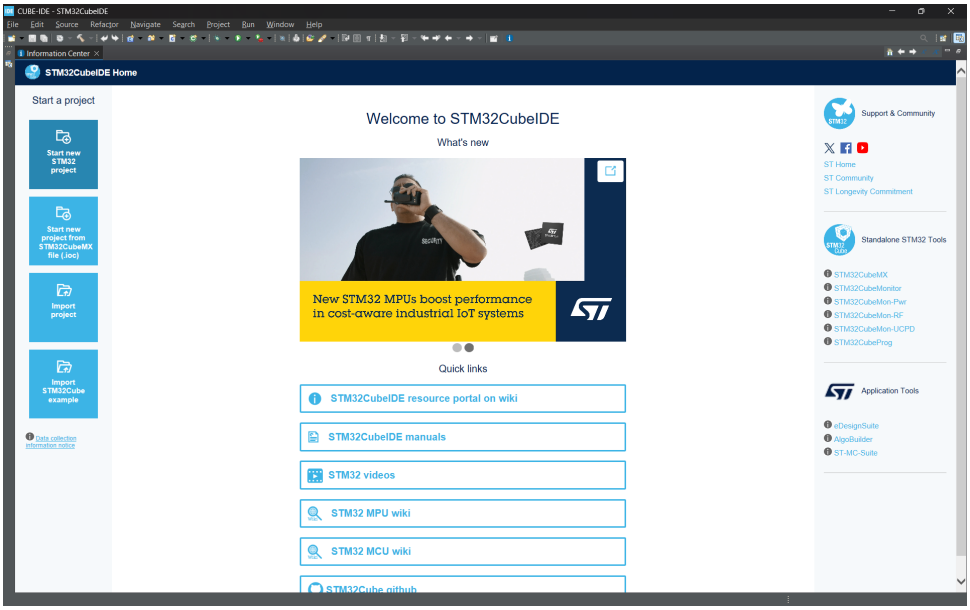

Figure 1: STM32CubeIDE default landing page. Click on *Start New STM32 Project* to begin.

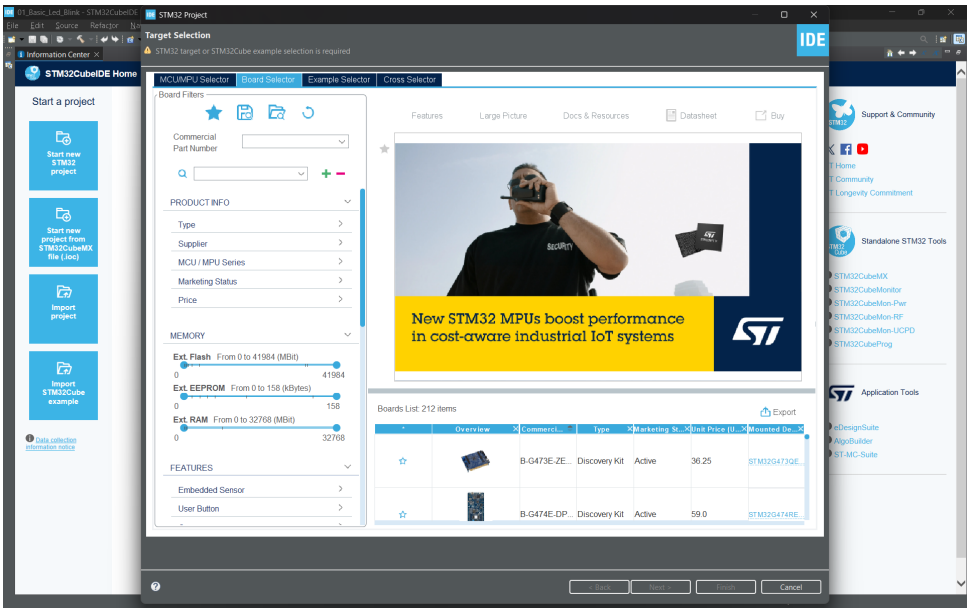

Figure 2: Board selector interface. Switch to *Board Selector* tab and search for G474RE.

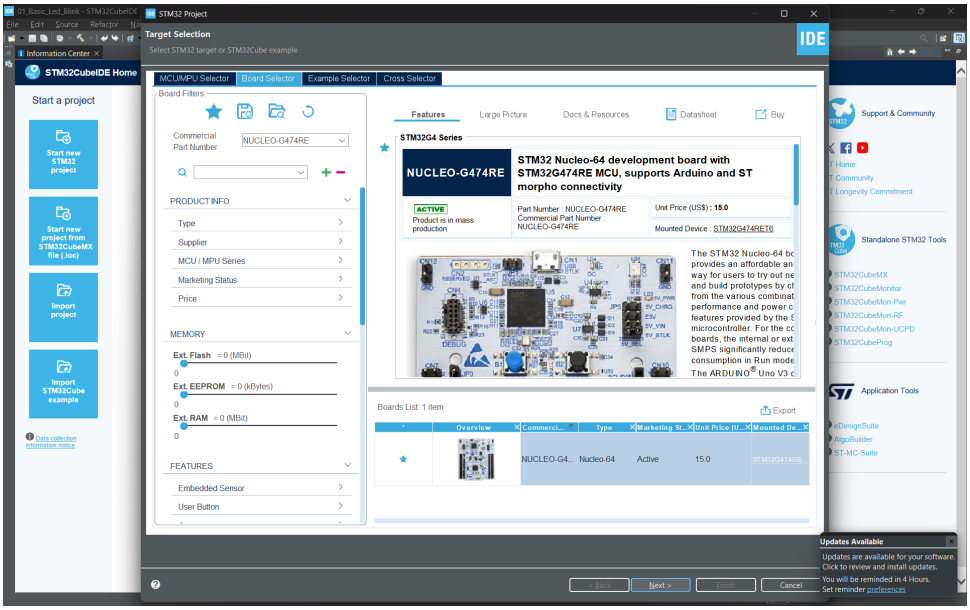

Figure 3: Product selection confirmation. Verify NUCLEO-G474RE is selected before proceeding.

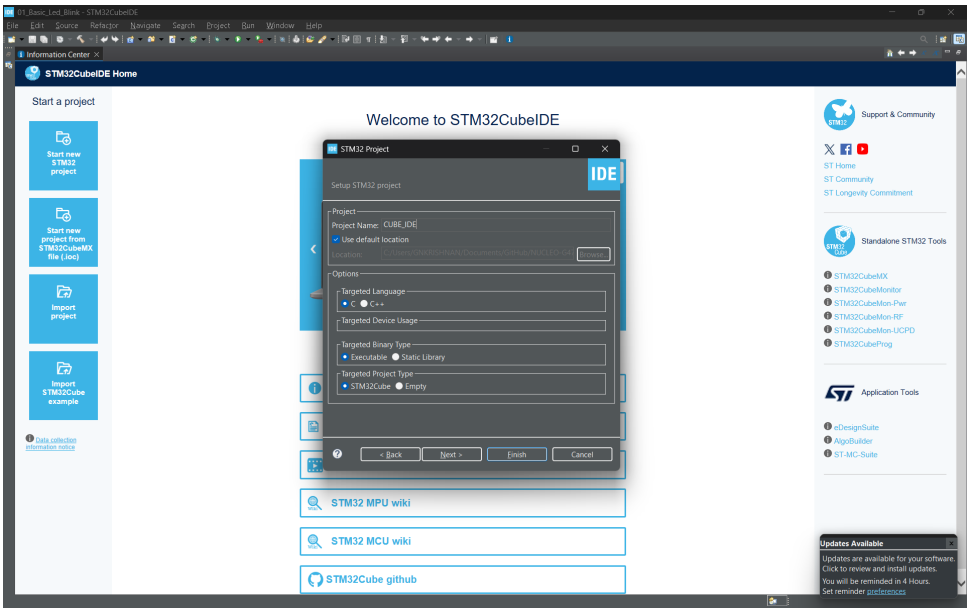

Figure 4: Project name and settings. Use snake_case naming convention. Accept default settings.

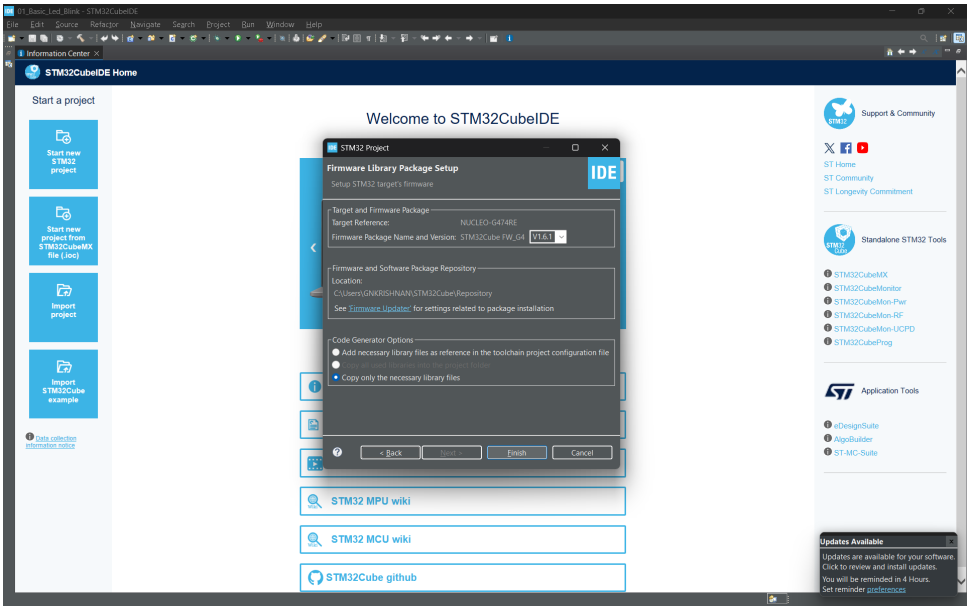

Figure 5: Library selection. Ensure *Copy Only Necessary Library* is checked.

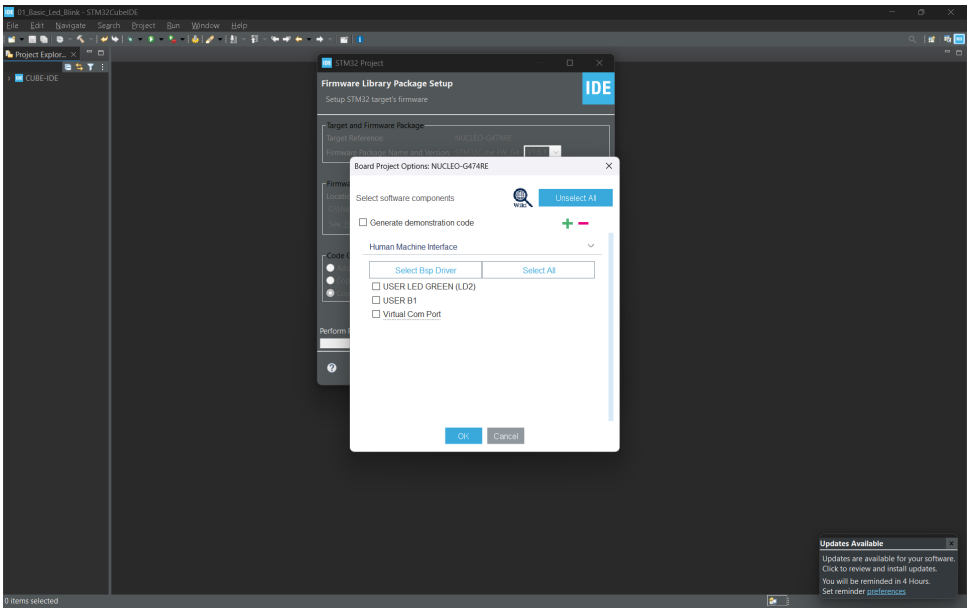

Figure 6: Board support package configuration. Uncheck unnecessary peripherals (USER LED GREEN, USER B1, Virtual COM Port).

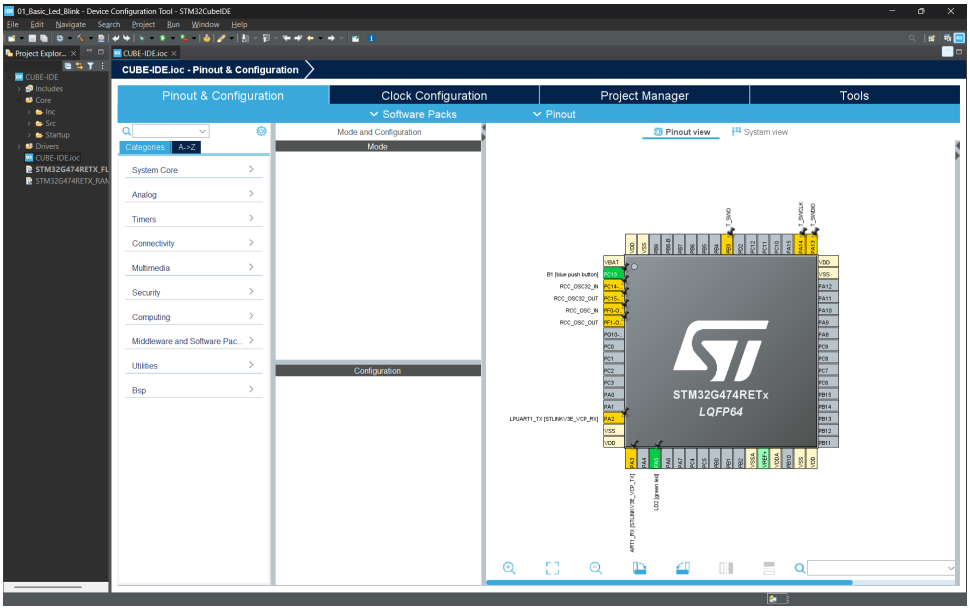

Figure 7: STM32CubeMX I/O configuration interface. The .ioc file is displayed with PA5 (LD2) pre-configured as GPIO output.

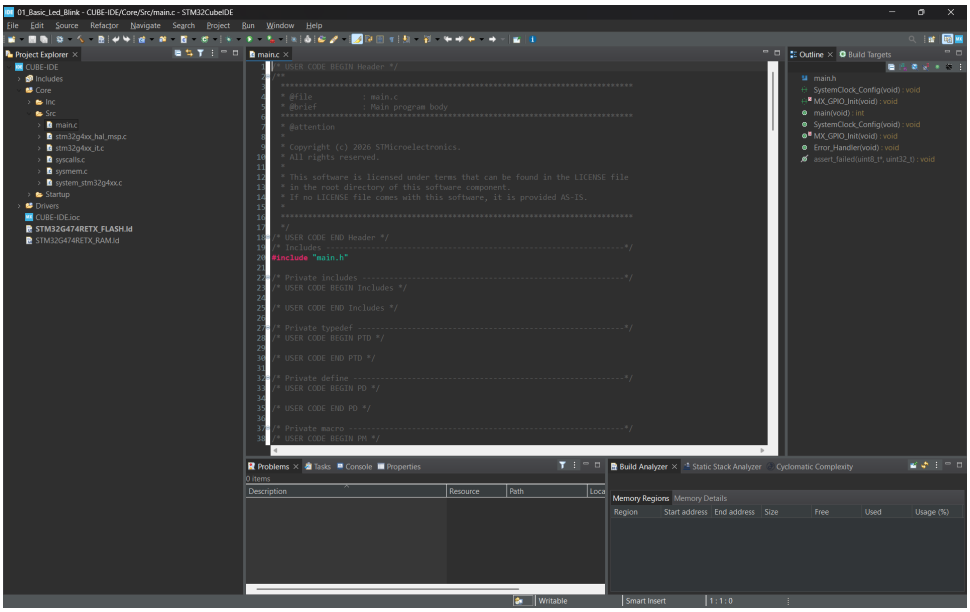

Figure 8: Project file structure. Navigate to Core/Src/main.c to implement the LED blink logic.

2.4 Code Implementation

2.4.1 Main Initialisation

The system initialisation code is automatically generated by STM32CubeMX and includes:

```
void SystemClock_Config(void)
{
  // System clock configuration
  // Auto-generated by STM32CubeMX
}
```

```
6
7  void MX_GPIO_Init(void)
8  {
9    // GPIO initialisation for PA5
10   // Auto-configured by STM32CubeMX
11 }
```

Listing 2: System Initialisation Function

2.4.2 Core Functionality

The main logic for blinking the LED is implemented in the infinite loop within `main.c`.

Insert the following code between the USER CODE markers in the main while loop:

```
1  /* USER CODE BEGIN 3 */
2  // Set PA5 (LED) to HIGH
3  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, SET);
4  // Wait 100ms
5  HAL_Delay(100);
6  // Set PA5 (LED) to LOW
7  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, RESET);
8  // Wait 100ms
9  HAL_Delay(100);
10 /* USER CODE END 3 */
```

Listing 3: LED Blink Implementation (100ms delay)

The `HAL_GPIO_WritePin()` function takes three parameters:

- **GPIOx**: The GPIO port (GPIOA in this case)
- **GPIO_Pin**: The pin number (GPIO_PIN_5 for PA5)
- **PinState**: The desired pin state (SET for high, RESET for low)

The `HAL_Delay()` function provides a blocking delay in milliseconds. For precise timing requirements in more complex applications, timer-based approaches would be preferred.

2.4.3 Interrupt Service Routines

This project does not utilise interrupt service routines, as the LED blinking is managed through simple polling in the main loop.

Note: For more detailed information on GPIO configuration and the HAL API, refer to Section 9 of the STM32G474RE Reference Manual [1].

2.5 Building and Flashing

The process of building and deploying the firmware to the NUCLEO-G474RE board is straightforward and entirely integrated within the STM32CubeIDE environment.

2.5.1 Build Process

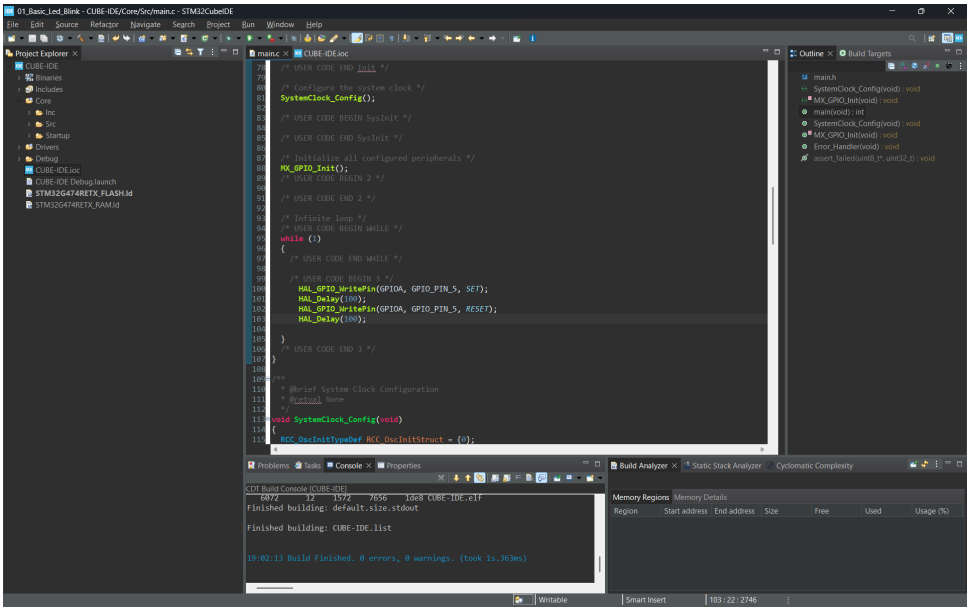

Figure 9: Project build. Click the hammer icon in the toolbar or right-click the project and select *Build Project*.

Upon successful compilation, the console displays *Build Finished 0 errors, 0 warnings*.

2.5.2 Execution Steps

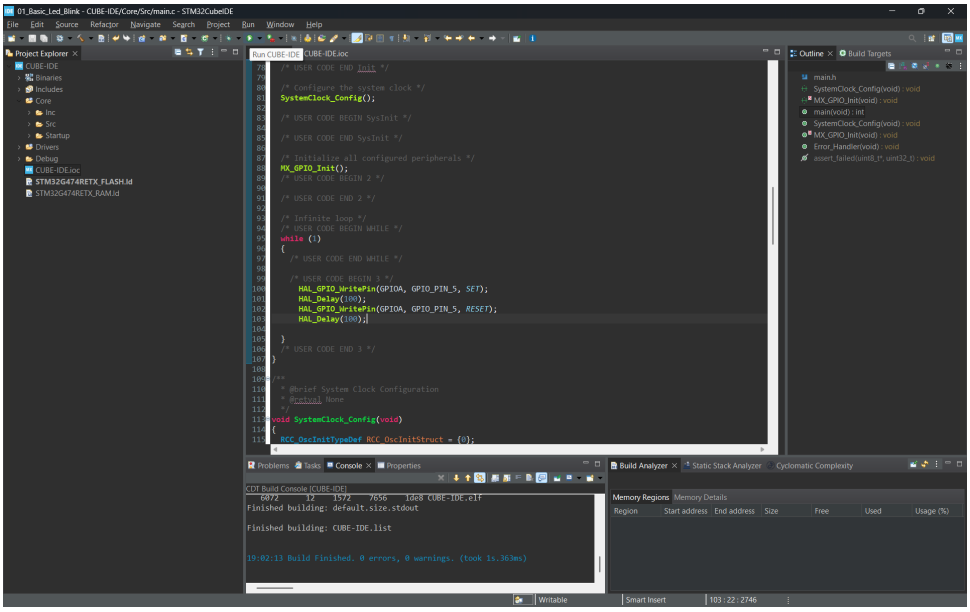

Figure 10: Running the application. Click the green play button in the toolbar to execute.

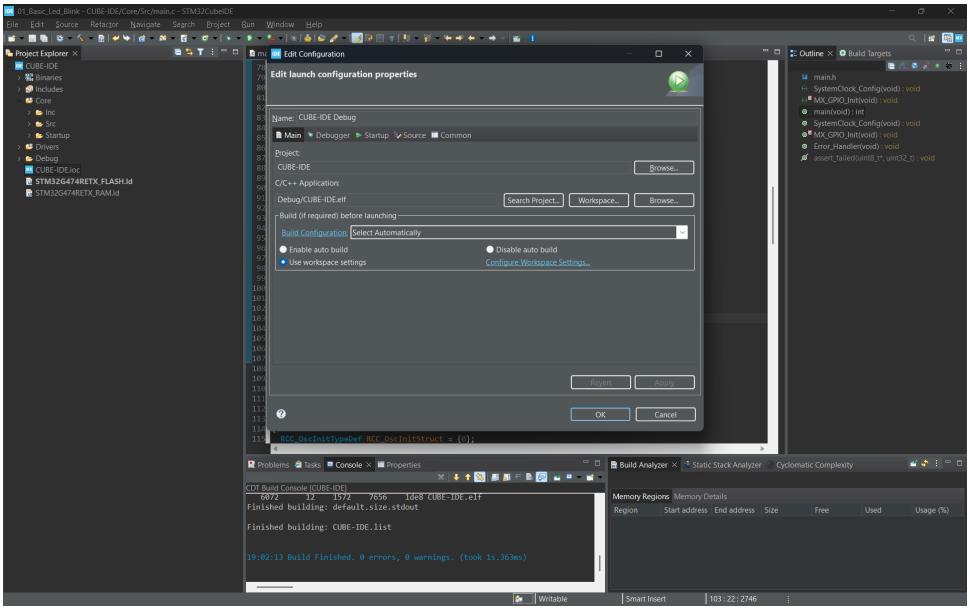

Figure 11: Run configuration dialog. Accept default settings and click *OK* to proceed with flashing.

After successful flashing, the on-board green LED (LD2) begins blinking at the configured 100 millisecond interval.

2.6 Debugging

For an introductory project like this, debugging is relatively straightforward. However, STM32CubeIDE provides powerful debugging capabilities for more complex applications.

Exploration and Learning:

To deepen your understanding of the system, it is recommended to:

- Explore the **.ioc file** using STM32CubeMX to understand how GPIO pins are configured. This file contains the complete peripheral configuration and can be graphically inspected.
- Review the **Reference Manual** (Section 9) to understand GPIO functionality at a deeper level, including port registers and advanced features.
- Examine the generated `stm32g4xx_hal_gpio.c` and `stm32g4xx_hal_gpio.h` files to see how the HAL abstracts hardware registers.

Using the Integrated Debugger:

If you wish to use the debugger:

- Set breakpoints by clicking in the left margin of the code editor
- Run the project in debug mode via **Debug As → STM32 C/C++ Application**
- Inspect variable values and processor state at breakpoints
- Use the *Variables* and *Registers* panels to monitor changes

Note: The STM32CubeIDE User Manual [4] provides comprehensive documentation on using the integrated debugger, including memory inspection, register viewing, and breakpoint management.

3 MATLAB Embedded Coder Implementation

3.1 Prerequisites

Before starting the MATLAB Embedded Coder implementation, ensure you have:

• *MATLAB R2021b or later with Embedded Coder toolbox installed* • *Embedded Coder Support Package for STMicroelectronics STM32 (install via Add-On Explorer)* • *ARM GCC Compiler configured in MATLAB (change from default to GNU Tools for ARM Embedded Processors)* • *NUCLEO-G474RE development board* • *Basic familiarity with Simulink and block diagrams*

Note: The ARM GCC compiler configuration is critical for successful code generation. By default, MATLAB may be configured with a different toolchain. This must be changed to **GNU Tools for ARM Embedded Processors** for STM32 compatibility.

3.2 MATLAB/Simulink Model Design

For this introductory LED blink project, the Simulink model is exceptionally simple, consisting of only two blocks: a discrete pulse generator and a digital port write block. This simplicity allows focus on the development workflow rather than complex control algorithms.

3.2.1 Creating a New Simulink Model

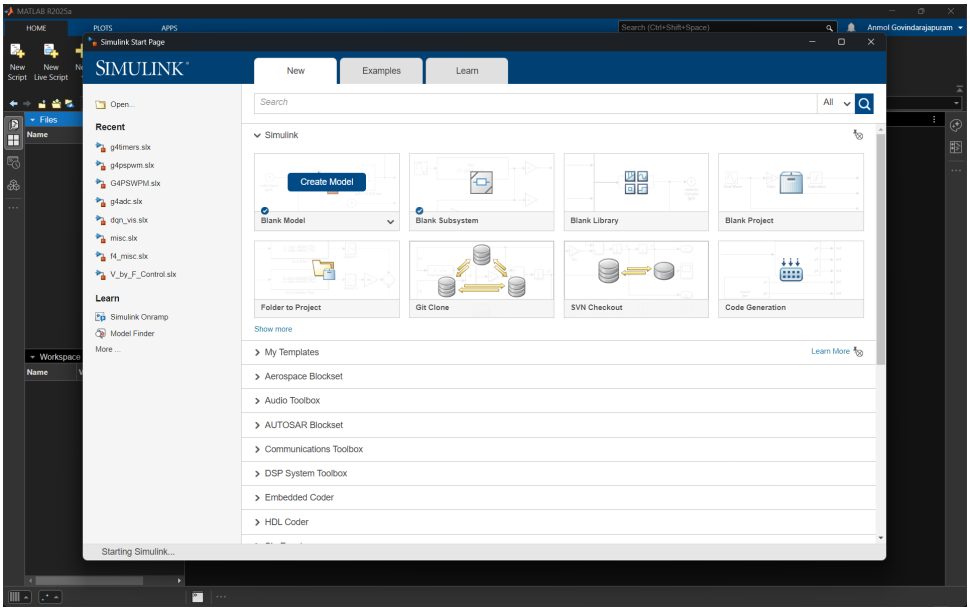

Figure 12: MATLAB startup. Launch MATLAB and create a new blank Simulink model.

3.2.2 Model Architecture

The model architecture for this project comprises:

1. **Discrete Pulse Block**: Generates a periodic pulse train with a 1-second period (on for 0.5 seconds, off for 0.5 seconds)
2. **Digital Port Write Block**: Writes the pulse signal to GPIO pin PA5 (the on-board LED)

This straightforward architecture demonstrates the fundamental workflow of model-based design whilst avoiding unnecessary complexity.

3.2.3 Block Functionality

For the LED blink project, the functional blocks serve distinct purposes:

Table 2: Simulink Block Functionality

Block	Purpose
Discrete Pulse	Generates timing signal and delay for LED blinking
Digital Port Write	Interfaces with GPIO pin PA5 to control the LED

3.3 Model Configuration

The Simulink model configuration for STM32 deployment requires specific settings to ensure proper code generation and real-time execution.

3.3.1 Solver Settings

The Simulink solver must be configured in discrete mode for embedded systems:

- **Solver Type**: Fixed-step (deterministic execution required for real-time systems)
- **Solver**: Discrete mode (no continuous-time states in this model)
- **Fixed Step Size**: 0.01 seconds (10 milliseconds, suitable for GPIO-based applications)

These settings are automatically configured by MATLAB when the STM32G4 hardware board is selected in the Hardware Implementation settings.

Note: For more complex projects with continuous-time dynamics (such as motor control with PID loops), the solver configuration may require adjustment. The fixed-step discrete solver ensures predictable execution timing on the microcontroller.

3.4 Building and Deploying the Simulink Model

The Embedded Coder Support Package for STM32 simplifies the build and deploy process by automating much of the code generation and compilation.

3.4.1 Hardware Board Selection and Configuration

Before building the model, the hardware target must be configured. This is accomplished through the Configuration Parameters dialog:

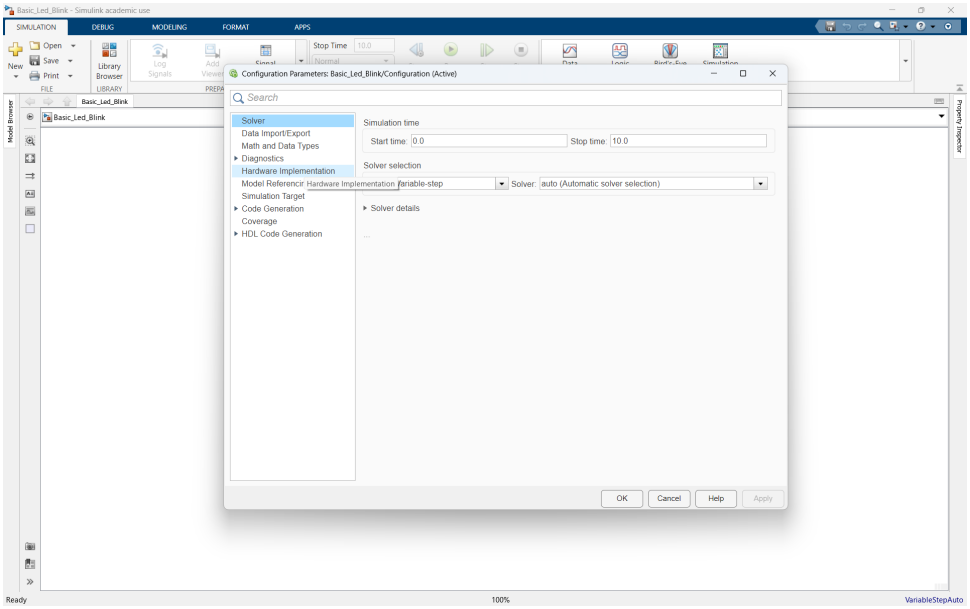

Figure 13: Configuration Parameters dialog. Press Ctrl+E to open. Navigate to *Hardware Implementation*.

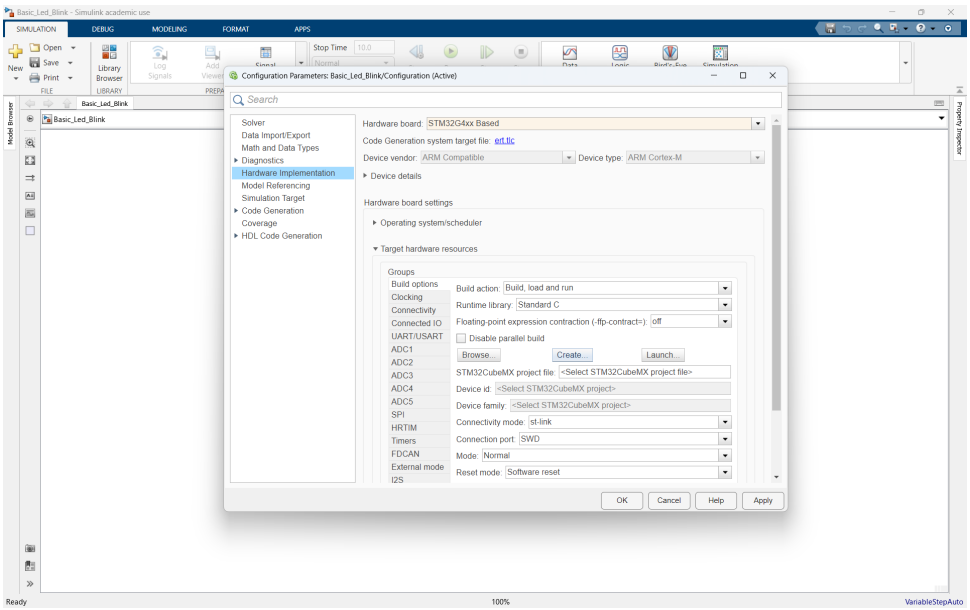

Figure 14: Hardware selection. Choose *STM32G4xx based* and allow MATLAB to apply configuration changes.

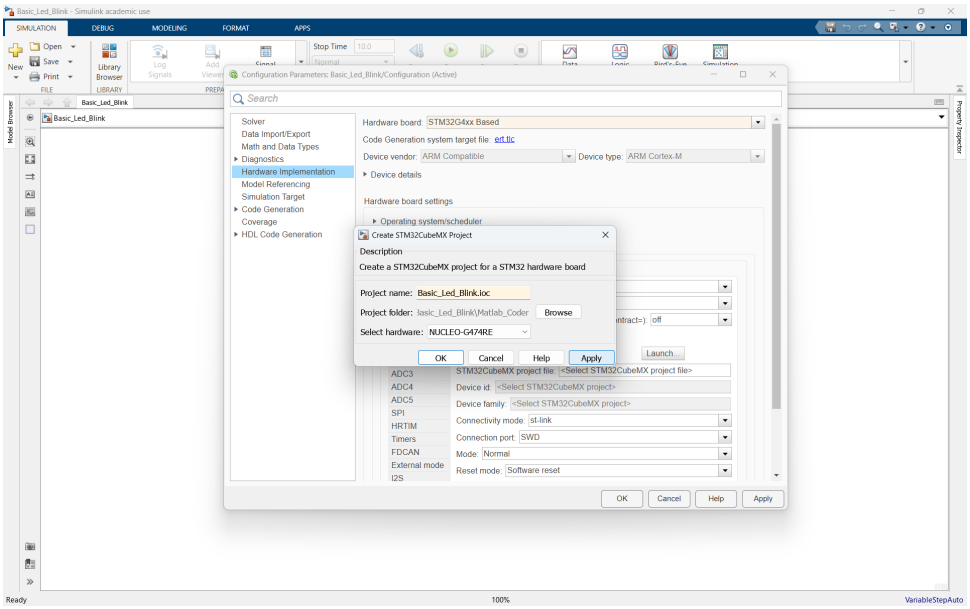

Figure 15: I/O configuration file creation. Select the correct board (NUCLEO-G474RE) and specify project location.

3.4.2 CubeMX Configuration Verification

After hardware selection, the .ioc file is created in STM32CubeMX. Verify the configuration settings:

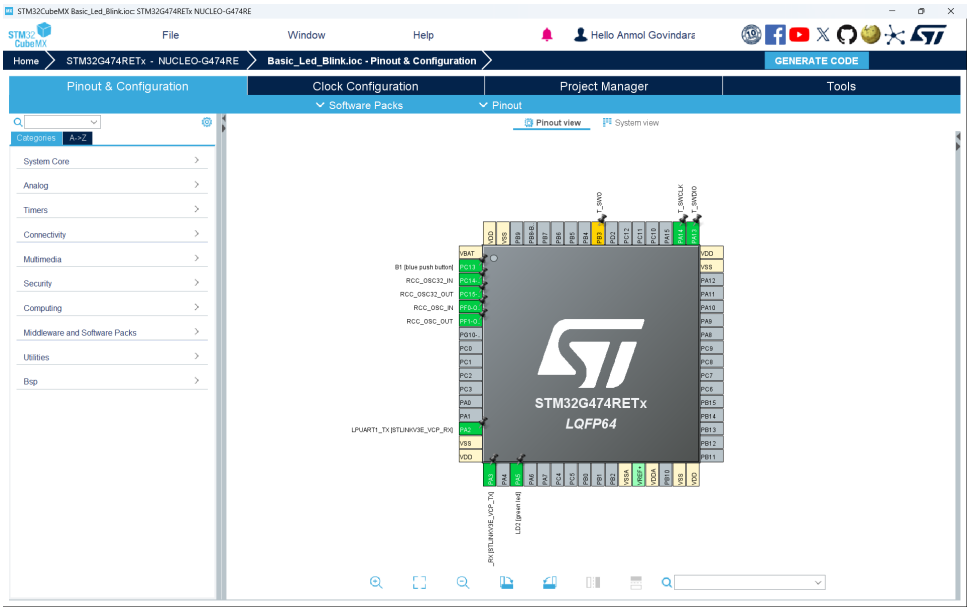

Figure 16: STM32CubeMX I/O configuration interface. This interface is identical to the standalone STM32CubeMX tool.

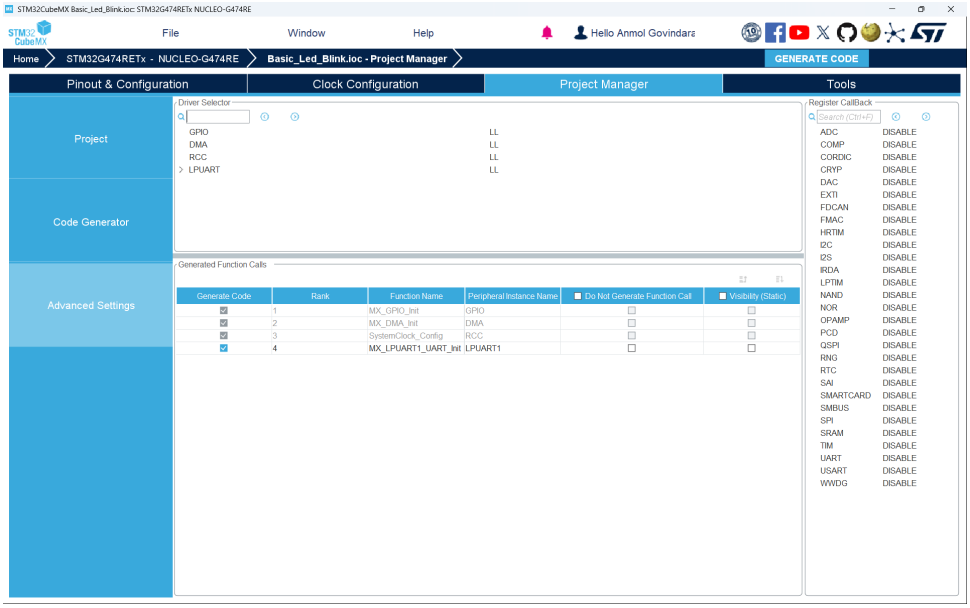

Figure 17: Advanced settings verification. Navigate to Project Manager → Advanced Settings. Verify all driver layers are LL (Low Level) and function call visibility is Static.

3.4.3 Compiler Configuration

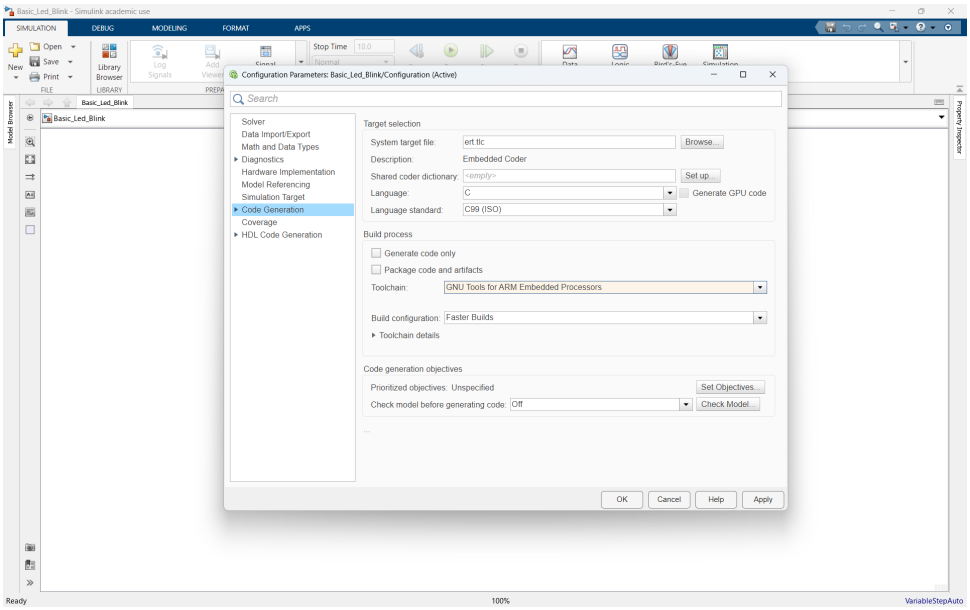

Figure 18: Compiler selection. In Configuration Parameters, under Code Generation, change the toolchain to *GNU Tools for ARM Embedded Processors*.

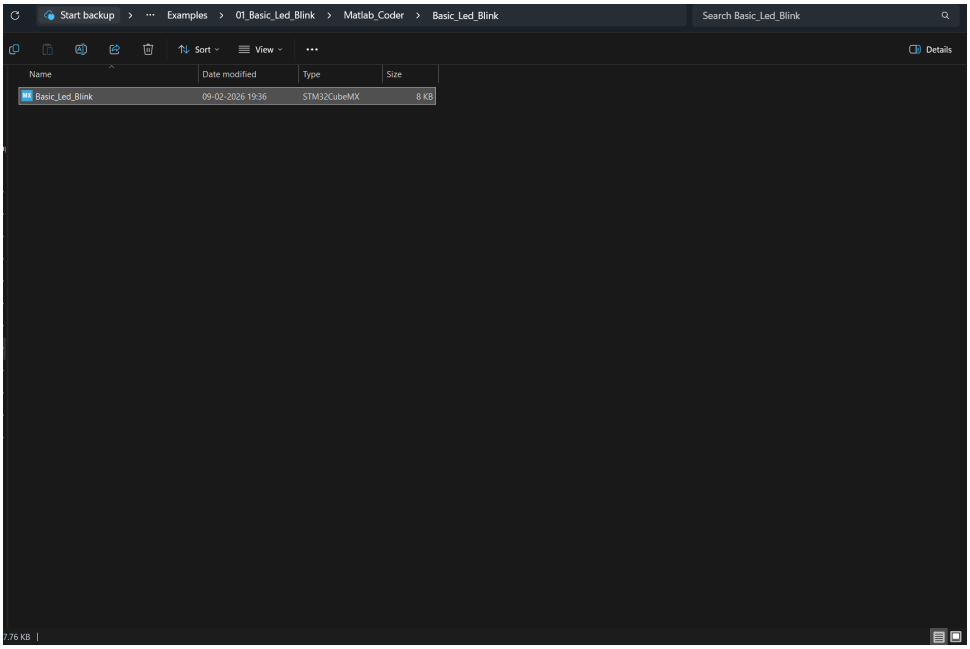

Figure 19: I/O configuration file confirmation. The .ioc file is now configured for MATLAB code generation.

3.4.4 GPIO Port Configuration

Configure the GPIO digital output block in Simulink for pin PA5:

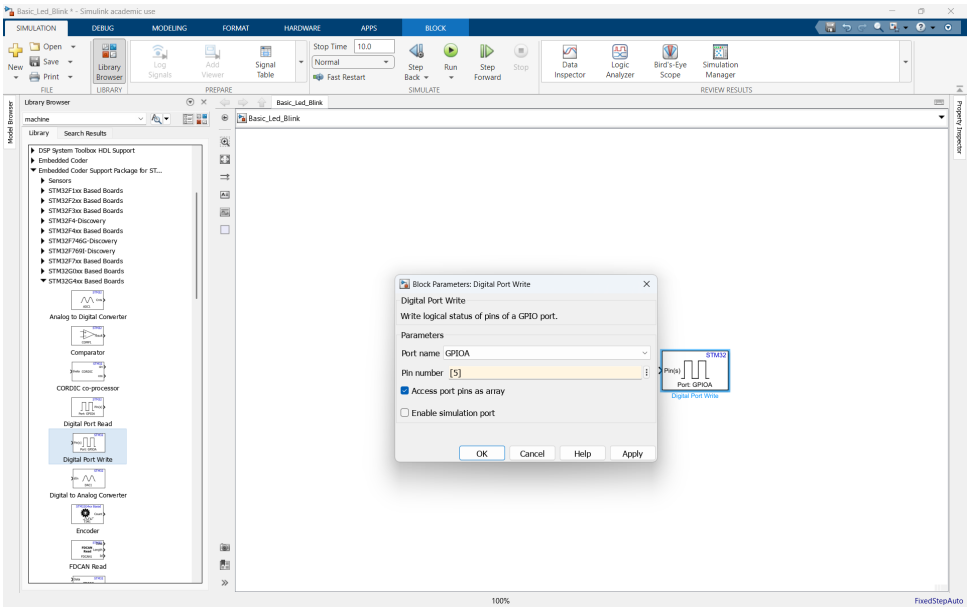

Figure 20: Digital Port Write block configuration. Drag and drop the block from the STM32G4xx library and configure it for GPIO Port A, Pin 5.

3.4.5 Discrete Pulse Configuration

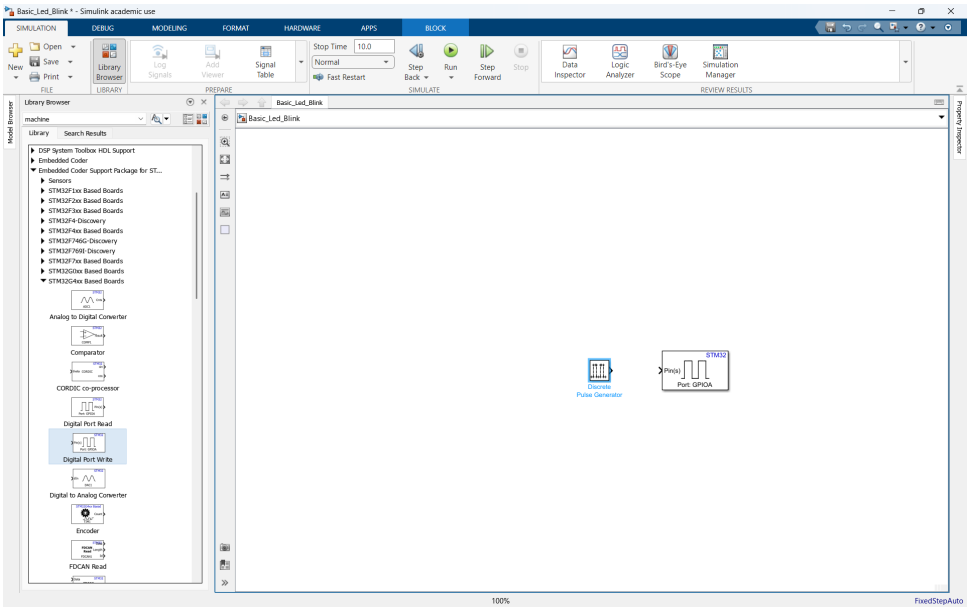

Figure 21: Discrete Pulse block selection. Select from the Simulink Sources library to create the timing signal.

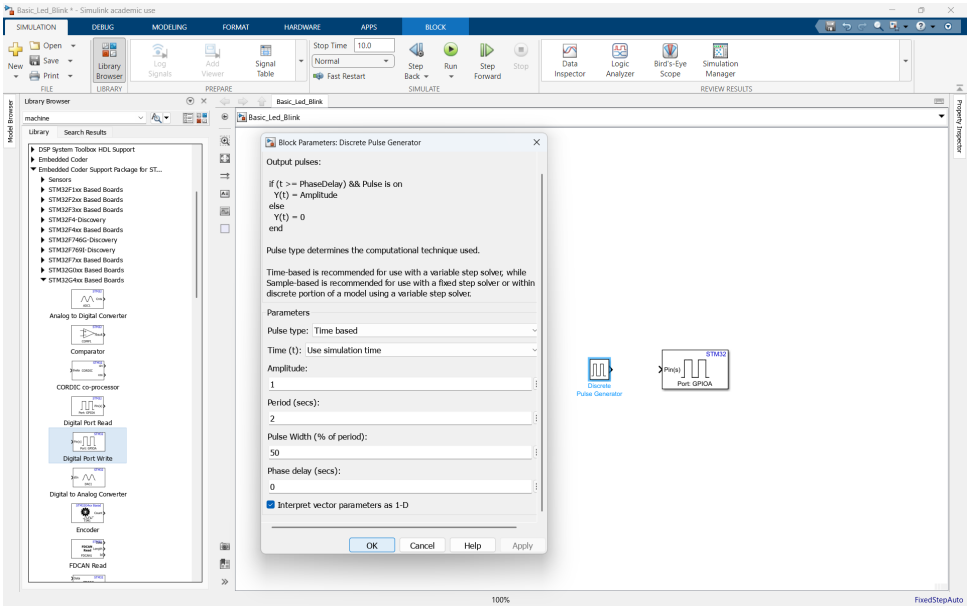

Figure 22: Discrete Pulse configuration. Set pulse period to 1 second (100 time steps at 0.01 s sampling time), pulse width to 0.5 seconds.

3.4.6 Complete Model Layout

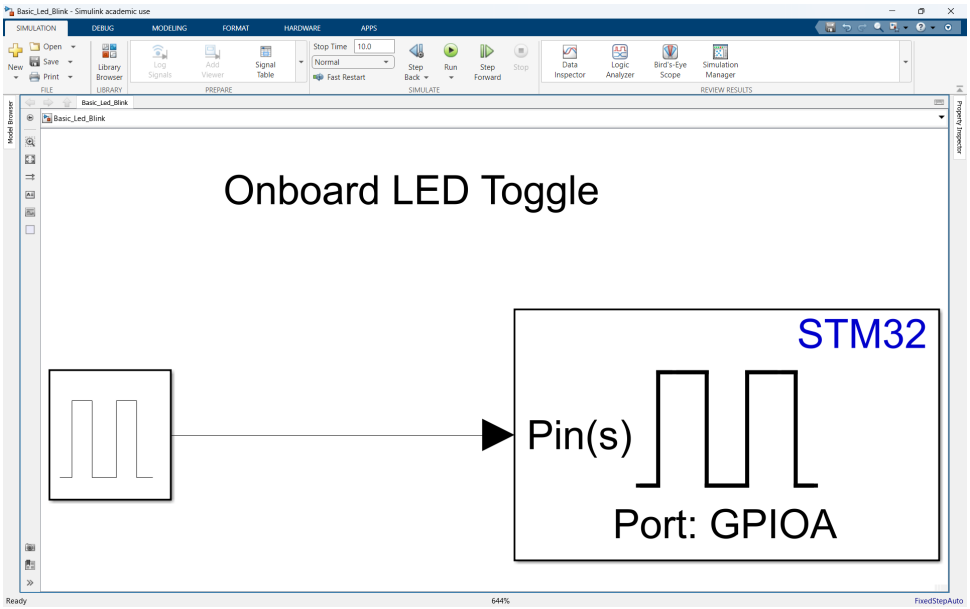

Figure 23: Final Simulink model. Connect the Discrete Pulse block output to the Digital Port Write block input. Save the model with Ctrl+S.

3.4.7 Building the Model

Once the Simulink model is complete and configured, the build process is initiated via the Hardware menu:

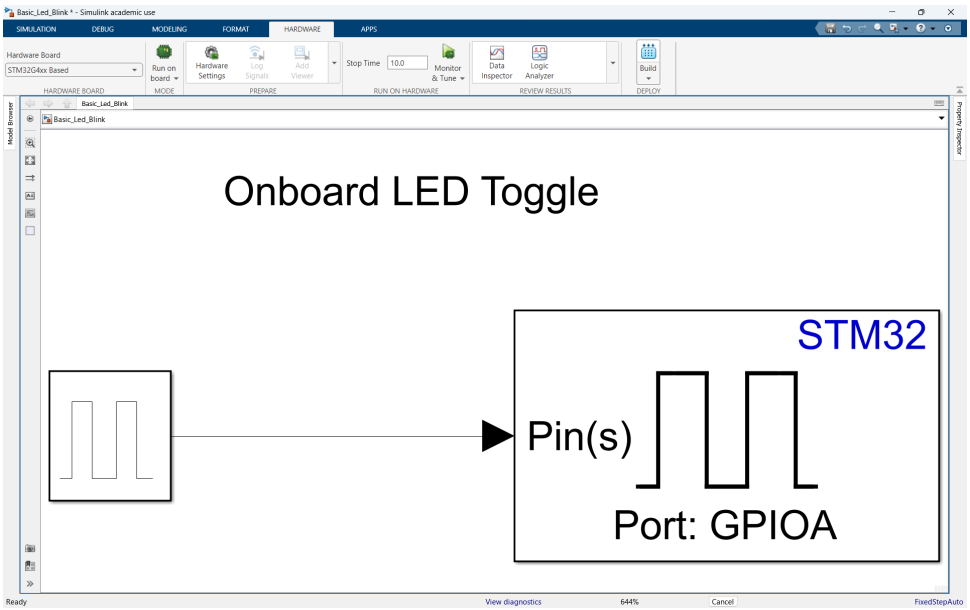

Figure 24: Hardware build button. Click the dropdown arrow in the *Hardware* section and select *Build* to generate code.

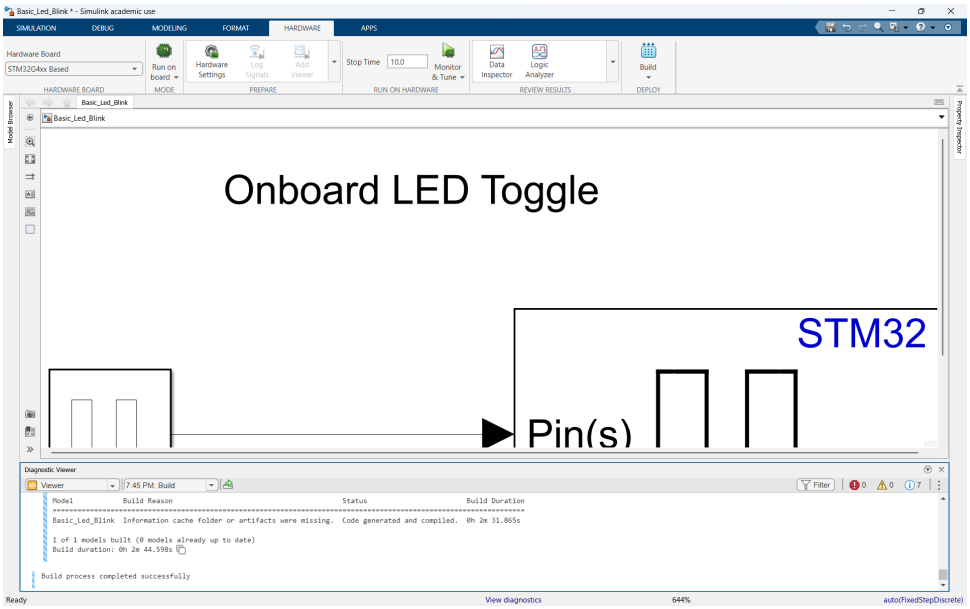

Figure 25: Successful build confirmation. Monitor the MATLAB Command Window for progress. Look for *Code generation successful* message.

The code generation process creates C code from the Simulink blocks, compiles it using the ARM GCC compiler, and generates the binary firmware file for deployment.

3.4.8 Building, Deploying, and Starting Execution

To simultaneously build, deploy to hardware, and start execution:

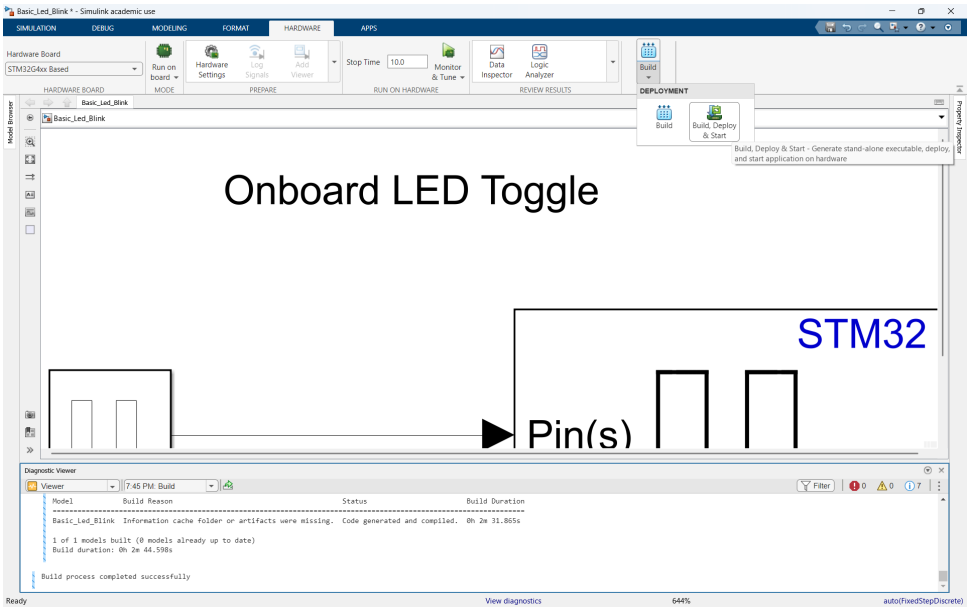

Figure 26: Build, Deploy and Start command. Click the dropdown arrow and select *Build Deploy and Start*. Connect the board via USB.

MATLAB will compile, generate code, and flash the firmware to the microcontroller. The on-board green LED (LD2) should begin blinking with a 1-second period (0.5 seconds on, 0.5 seconds off).

Note: The first build may take longer as MATLAB generates the project structure and all required files. Subsequent builds are faster.

3.5 Model-to-Code Mapping

Understanding how Simulink blocks map to generated C code is valuable for debugging and optimisation:

- **Discrete Pulse Block**: Generates C code implementing a counter that increments each time step, toggling an output when the counter reaches the specified period
- **Digital Port Write Block**: Generates HAL function calls equivalent to `HAL_GPIO_WritePin()` for controlling the GPIO pin

The generated code structure closely mirrors the Simulink block hierarchy, making it straightforward to understand the relationship between the model and the implementation.

4 STM32CubeIDE vs. MATLAB Embedded Coder

4.1 Implementation Comparison

For this introductory LED blink project, both approaches demonstrate fundamentally different philosophies for embedded systems development:

Table 3: Comparison of STM32CubeIDE and MATLAB Embedded Coder

Aspect	**STM32CubeIDE (C)**	**MATLAB Embedded Coder**
Development Approach	Direct C programming with HAL APIs	Visual model-based block diagram design
Code Generation	Manual code writing	Automatic C code generation
Blinking Delay	100 milliseconds	1 second
Learning Path	Understand GPIO registers and HAL functions	Understand Simulink blocks and model semantics
Development Speed	Moderate for simple projects	Fast for complex algorithm design
Code Inspection	Directly readable C code	Auto-generated code (less intuitive)
Debugging	Step through C code with debugger	Validate model before code generation

4.2 Key Differences in this Project

STM32CubeIDE Implementation:

The C implementation uses the Hardware Abstraction Layer to directly manipulate GPIO registers. The developer writes explicit function calls (`HAL_GPIO_WritePin`, `HAL_Delay`) that are mapped to hardware operations. The timing (100 ms) is hardcoded in the C source.

MATLAB Embedded Coder Implementation:

The model-based approach uses visual blocks representing functional operations. The timing (1 second) is configured through block parameters. The generated C code is automatically created from the block diagram, abstracting away the low-level register manipulation.

4.3 Recommended Approach for Different Scenarios

Use STM32CubeIDE when:

- You need precise control over hardware registers
- Developing low-level drivers or bootloaders
- Memory optimisation is critical (embedded systems with severe constraints)
- You prefer directly reading and modifying C code
- You are familiar with traditional embedded C programming

Use MATLAB Embedded Coder when:

- Designing complex control algorithms that benefit from simulation
- Rapid prototyping and iteration are priorities
- You have a background in control systems and MATLAB
- You want automatic code generation with integrated verification
- The project involves significant mathematical modelling

4.4 Hybrid Approach Recommendation

For more complex power electronics projects, a hybrid approach is ideal:

1. Develop and simulate the control algorithm in MATLAB/Simulink
2. Validate performance through model simulation
3. Generate core algorithm code using Embedded Coder
4. Use STM32CubeIDE for hardware initialisation, I/O handling, and optimisation
5. Integrate both code components for final deployment

This project introduces both approaches independently, providing a foundation for selecting the appropriate tool for future projects.

5 Testing and Validation

For this introductory project, testing and validation are straightforward. The functionality is immediately observable: the on-board LED either blinks or it does not.

5.1 Visual Verification

The primary verification method is visual observation:

- **STM32CubeIDE Implementation**: Observe the green LED (LD2) blinking with a 100 millisecond period (quick, noticeable flicker)
- **MATLAB Embedded Coder Implementation**: Observe the same LED blinking with a 1-second period (slow, easy to count: on for 0.5 seconds, off for 0.5 seconds)

5.2 Functional Testing

The project is considered successful if:

- The LED blinks continuously without crashing
- The timing is consistent and predictable
- No error messages appear in the IDE console
- The blinking pattern matches the expected behaviour (100 ms vs. 1 second timing)

5.3 Performance Characteristics

The LED blink project has minimal performance requirements:

- **Execution Time**: Negligible (GPIO operations take microseconds)
- **CPU Utilisation**: Less than 1 % (the processor spends most time in `HAL_Delay()` blocking calls)
- **Memory Usage**: Minimal (simple GPIO operations require only a few hundred bytes of code)

6 Results and Analysis

6.1 Expected Outcomes

STM32CubeIDE Implementation Result:

Upon successful flashing and execution, the on-board green LED (LD2) connected to pin PA5 blinks continuously with a period of 200 milliseconds (100 ms on, 100 ms off). This rapid blinking is easily perceived as a flickering light.

MATLAB Embedded Coder Implementation Result:

Upon successful build, deployment, and execution, the same on-board green LED blinks with a period of 2 seconds (1 second on, 1 second off). This slower blink rate allows manual timing verification and is more suitable for visual observation.

6.2 Key Observations

1. **Timing Difference**: The 100 ms vs. 1 second timing difference demonstrates the flexibility of both development approaches. Different applications may require different timing, easily configured in either method.

2. **Code Simplicity**: Both implementations are exceptionally simple, consisting of only a few lines of functional code. This simplicity allows focus on the development workflow rather than algorithmic complexity.

3. **Development Workflow**: The STM32CubeIDE approach emphasises direct C programming and hardware control, whilst MATLAB Embedded Coder emphasises model-based design and automatic code generation.

4. **Ease of Modification**: In STM32CubeIDE, changing the delay requires modifying the numeric constant in the C code. In MATLAB, changing the timing requires modifying the Discrete Pulse block parameters—arguably more intuitive for non-programmers.

6.3 Project Validation

This project successfully demonstrates:

- GPIO configuration and control on the STM32G474RE
- Successful use of both STM32CubeIDE and MATLAB Embedded Coder
- The practical workflow for embedded system development using two distinct tools
- Understanding of basic microcontroller timing and digital output control

7 Troubleshooting

7.1 Common Issues and Solutions

Table 4: Common Issues and Solutions

Issue	**Solution**
LED does not blink after flashing	Verify PA5 pin configuration in STM32CubeMX; check USB power supply
STM32CubeIDE compilation error	Ensure project is created correctly with NUCLEO-G474RE board selected
MATLAB code generation fails	Verify ARM GCC compiler is selected (not default compiler); check hardware configuration
Board not detected by IDE	Install ST-Link drivers; try different USB port; restart IDE
LED blinking very slowly/quickly	For CubeIDE: verify `HAL_Delay()` timing; for MATLAB: check Discrete Pulse block configuration

7.2 Debugging Recommendations

For this introductory project, debugging is primarily based on observation:

- **Visual Inspection**: The most direct verification method. If the LED blinks as expected, the implementation is correct.
- **Timing Verification**: Compare the observed blink rate with expected timing. Use a stopwatch or smartphone timer for the 1-second MATLAB version.
- **IDE Console**: Check the build console output for any errors or warnings.
- **Hardware Inspection**: Verify the USB cable is properly connected and the board is receiving power (look for the red power LED on the board).

8 Extensions and Enhancements

Whilst the basic LED blink project is intentionally simple, it provides a foundation for more advanced implementations. The following enhancements demonstrate how this basic project can be extended to address real-world applications.

8.1 Enhancement 1: Button-Controlled LED Toggle

Extend the project to include a push-button (on-board user button, pin PC13) that toggles the LED on and off. This introduces:

- GPIO input configuration and reading
- Debouncing techniques
- State machines for managing on/off states

Implementation: Read the push-button state using `HAL_GPIO_ReadPin()` and toggle a boolean variable that controls whether the LED blinks.

8.2 Enhancement 2: Complementary LED Control

Configure a second GPIO pin to control a complementary output: when PA5 is HIGH, the second pin is LOW, and vice versa. This demonstrates:

- Multiple GPIO pin control
- Synchronisation of outputs
- Potential application in push-pull driver configurations

Implementation: Use two GPIO pins configured for output push-pull. In the main loop, simultaneously set one HIGH while setting the other LOW.

8.3 Enhancement 3: Relay Control

Connect a 3.3V-compatible relay to a GPIO pin to control higher-voltage or higher-current loads. This extends the project to:

- Driving external devices from GPIO outputs
- Understanding electrical isolation through relays
- Practical applications in industrial automation

Caution: Relays require back-EMF protection (flywheel diodes). Do not connect a relay directly to a GPIO pin without proper driver circuitry (a transistor and flywheel diode). Refer to the Reference Manual for GPIO electrical characteristics and maximum current ratings [1].

8.4 Further Learning

Building on this project, subsequent projects in the NUCLEO-G474RE Power Electronics Guide introduce:

- ADC-based sensor reading and analogue-to-digital conversion
- PWM generation for controlling power electronics
- Timer-based interrupts for precise timing
- Advanced motor control and power conversion techniques

9 Conclusion

This project has successfully introduced both STM32CubeIDE and MATLAB Embedded Coder as development environments for the STM32 NUCLEO-G474RE microcontroller. Through a simple yet functional LED blinking application, we have demonstrated the fundamental workflows of both tools.

9.1 Key Takeaways

- **GPIO Fundamentals**: Understanding how to configure and control GPIO pins is foundational to all embedded systems. The HAL abstraction layer simplifies register-level operations into intuitive function calls.
- **STM32CubeIDE Workflow**: STM32CubeIDE provides an integrated development environment that combines peripheral configuration (via STM32CubeMX), code generation, compilation, and debugging. Direct C programming offers maximum control and transparency.
- **MATLAB Embedded Coder Workflow**: MATLAB Embedded Coder offers a model-based design approach, allowing developers to design algorithms visually using Simulink blocks. Automatic code generation from models can significantly accelerate development for complex algorithms.

- **Tool Selection**: The choice between tools depends on project requirements, developer expertise, and application complexity. For simple GPIO-based applications, STM32CubeIDE is straightforward. For complex control algorithms with mathematical modelling, MATLAB Embedded Coder excels.
- **Hardware-Software Integration**: Modern embedded systems development requires understanding both hardware capabilities (GPIO, timers, ADC) and software abstractions (HAL, code generation, real-time scheduling).

9.2 Next Steps

Having completed this introductory project, you are prepared to advance to more complex projects in the NUCLEO-G474RE Power Electronics Guide, including:

- ADC-based sensor acquisition (Project 02)
- PWM generation and motor control (Project 03-05)
- Multi-channel feedback control systems
- Advanced power electronics applications

Both development approaches—STM32CubeIDE and MATLAB Embedded Coder—will prove valuable in your embedded systems journey. This project establishes the foundation for proficiency with both tools.

References

References

[1] STMicroelectronics, *STM32G4 Series Advanced ARM-based 32-bit MCUs Reference Manual*, RM0440.

[2] STMicroelectronics, *STM32G474RE Datasheet*.

[3] STMicroelectronics, *STM32G4 NUCLEO-64 Boards User Manual*, UM2505.

[4] STMicroelectronics, *STM32CubeIDE Integrated Development Environment User Manual*, DM00629855.

[5] STMicroelectronics, *Description of STM32G4 HAL and Low-Layer Drivers*, UM2570.

[6] STMicroelectronics, *HRTIM Cookbook*, AN4539.

[7] MathWorks, *Embedded Coder Documentation*.

[8] MathWorks, *Embedded Coder Support Package for STMicroelectronics STM32*.

A Code Listings

Additional code examples can be placed here.

```
// Add extra code listings as needed
```

Listing 4: Additional Code Example

B Hardware Documentation

Refer to the official NUCLEO-G474RE schematic for complete hardware specifications.

C Configuration Files

Document important configuration files such as linker scripts and startup code.

Author

Anmol Govindarajapuram Krishnan

GitHub: `https://github.com/Anmol-G-K`

Email: cb.en.u4eee23103@cb.students.amrita.edu

Acknowledgments

We acknowledge the following for their contributions and support:

- **STMicroelectronics** for providing comprehensive microcontroller documentation and development tools
- **MathWorks** for excellent MATLAB and Embedded Coder resources
- The embedded systems and power electronics community for innovation and knowledge sharing
- Contributors and reviewers providing feedback

For updates and additional resources:
`https://github.com/Anmol-G-K/NUCLEO-G474RE-PowerElectronics-Guide`