NUCLEO-G474RE

Power Electronics Project Series

Basic ADC Sampling

Project #02

Interfacing with ADC in Polling Mode

Implementation Frameworks:

STM32CubeIDE | MATLAB Embedded Coder

Document Version: 1.0

Last Updated: February 14, 2026

Contents

1 Project Overview

1.1 Objective

Basic interfacing with ADC in polling mode, one of the primary requirements of embedded devices for sensing. ADC is a peripheral that achieves this by converting analogue signals into digital values for processing.

1.2 Applications

The ADC is fundamental to numerous embedded systems applications:

- **Voltage and Current Sensing**: Direct measurement of electrical parameters for feedback and monitoring
- **Sensor Readings**: Temperature, humidity, pressure, and other environmental sensor interfaces
- **Power Electronics Monitoring**: Voltage and current feedback for control algorithms

1.3 Key Concepts

The fundamental concepts covered in this project include:

- **Analog-to-Digital Conversion (ADC)**: Understanding the fundamental principles of converting continuous analogue signals into discrete digital values
- **Sampling Theory**: The principles governing how often we must measure a signal to accurately represent it
- **ADC Resolution**: Understanding how bit resolution determines the precision of measurements

1.4 Learning Outcomes

By completing this project, you will understand:

- How to interface an ADC peripheral with practical use cases
- Monitoring voltage using ADC with a simple potentiometer-based measurement system
- Configuring ADC sampling rates and resolution for embedded applications

Note: Go through Project 01 to understand the workflow.
GitHub Project Link – 01_Basic_Led_Blink

1.5 Fundamentals of Analog-to-Digital Conversion

1.5.1 What is an ADC?

In electronics, an analog-to-digital converter (ADC) is a system that converts an analogue signal which represents any real-world phenomenon into a digital signal that can be processed by digital computation circuits. This conversion is essential for embedded systems to interact with the physical world.

1.5.2 ADC Resolution

An ADC is characterised by its **resolution**, which determines the number of discrete analogue levels it can measure. Resolution is expressed in bits. The NUCLEO-G474RE features a **12-bit ADC**, meaning it can distinguish between $2^{12} = 4096$ discrete levels.

The measurable range is defined by the reference voltages:

- $V_{\textbf{ref}+}$: Reference positive voltage (typically 3.3V on our board)
- $V_{\textbf{ref}-}$: Reference negative voltage (typically 0V on our board)

Therefore, the measurable range is 3.3V divided into 4096 levels:

$$\text{LSB (Least Significant Bit)} = \frac{V_{\text{ref}+} - V_{\text{ref}-}}{2^n - 1} = \frac{3.3\text{ V}}{4095} = 805.664\mu\text{V} \tag{1}$$

Each step in the digital output represents approximately 805.664 microvolts of analogue change.

1.5.3 ADC Sampling Frequency

The second critical characteristic of an ADC is its **sampling frequency**, which determines how rapidly the ADC can acquire measurements. According to the **Nyquist Theorem**, the sampling frequency must be at least twice the maximum frequency of the signal being measured to avoid aliasing.

For the STM32 microcontroller, the sampling frequency is determined by:

$$T_{\text{conv}} = T_{\text{sampling}} + T_{\text{ADC\_CLK}} \tag{2}$$

Where:

- $T_{\textbf{sampling}}$: Sampling time (configurable, typically 12.5 clock cycles)
- $T_{\textbf{ADC\_CLK}}$: Fixed at 12.5 clock cycles for STM32G4 series

For our configuration:

- Total conversion time: $12.5 + 12.5 = 25$ clock cycles
- ADC prescaler: 4 (divides the system clock)
- ADC clock frequency: $\frac{170\text{ MHz}}{4} = 42.5$ MHz

- Conversion time: $\frac{25}{42.5\times10^6} = 588.235$ ns

This results in a maximum sampling rate of approximately 1.7 MHz, far exceeding typical requirements for voltage sensing applications.

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* • *A potentiometer (10 k recommended) for testing* • *Knowledge of GPIO configuration from Project 01*

2.2 Hardware Configuration

2.2.1 Pin Configuration

The ADC sampling project uses the on-board ADC connected to a potentiometer for variable voltage input:

Table 1: Pin Configuration for ADC Project

Pin	Function	Description
PA0	ADC1 Channel 1	Analogue input for voltage measurement

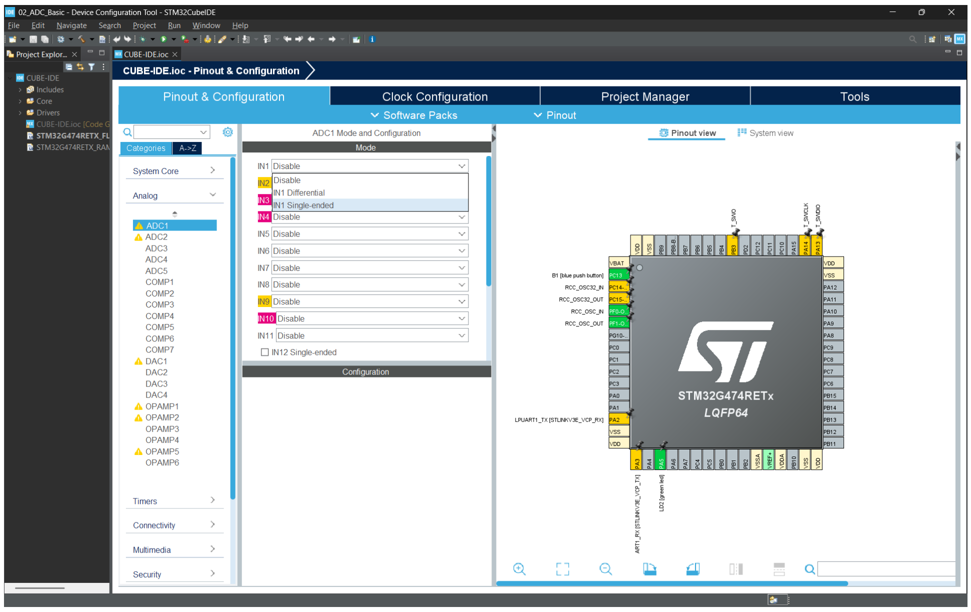
2.2.2 External Circuit

A potentiometer provides variable voltage input to the ADC:

- **Potentiometer connection**:
  - One end: Connect to +3.3V (from NUCLEO board)
  - Centre (wiper): Connect to PA0
  - Other end: Connect to GND (Ground)

By rotating the potentiometer, the voltage at PA0 varies between 0V and 3.3V, allowing observation of ADC conversions across the full measurement range.

2.2.3 Peripheral Configuration

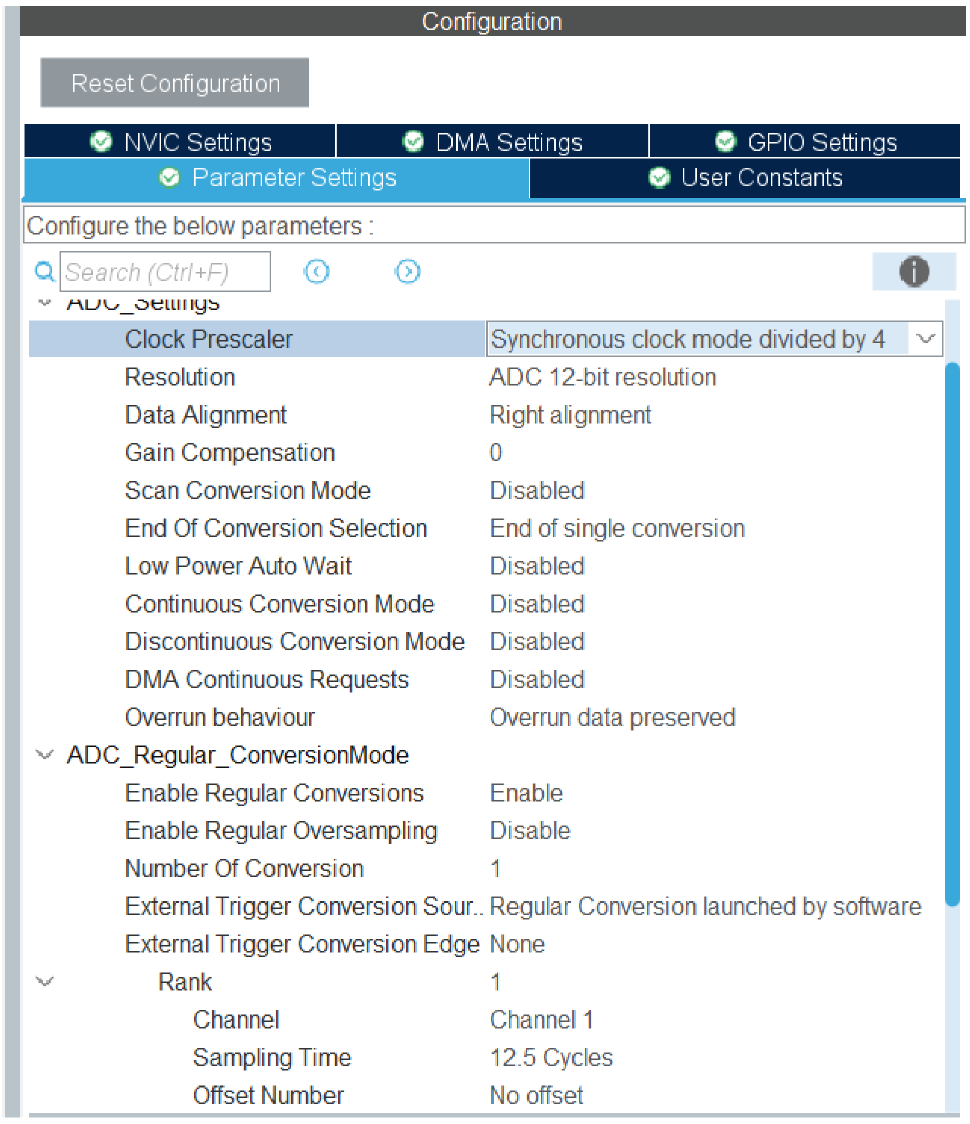


ADC1 Mode and Configuration

Mode

IN1 Disable
Disable
IN1 Differential
IN1 Single-ended
IN2
IN3
IN4 Disable
IN5 Disable
IN6 Disable
IN7 Disable
IN8 Disable
IN9 Disable
IN10 Disable
IN11 Disable
IN12 Single-ended

Figure 1: ADC1 channel 1 configuration in single-ended mode. Configure under the Analog section in STM32CubeMX.



ADC_Settings	
Clock Prescaler	Synchronous clock mode divided by 4
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Gain Compensation	0
Scan Conversion Mode	Disabled
End Of Conversion Selection	End of single conversion
Low Power Auto Wait	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
Overrun behaviour	Overrun data preserved
ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Enable Regular Oversampling	Disable
Number Of Conversion	1
External Trigger Conversion Sour..	Regular Conversion launched by software
External Trigger Conversion Edge	None
Rank	1
Channel	Channel 1
Sampling Time	12.5 Cycles
Offset Number	No offset

Figure 2: Detailed ADC configuration showing ADC prescaler set to 4 and sampling time set to 12.5 clock cycles.

2.3 Software Architecture

2.3.1 Initialisation Sequence

The ADC initialisation and operation sequence follows these steps:

1. **Start ADC**: Enable the ADC peripheral using `HAL_ADC_Start()`
2. **Wait for Sampling**: The ADC samples the analogue input. The End of Conversion (EOC) flag is set when sampling completes
3. **Poll for Conversion**: Use `HAL_ADC_PollForConversion()` to wait until conversion is complete
4. **Read Value**: Retrieve the digital value using `HAL_ADC_GetValue()`
5. **Repeat**: Add a small delay and repeat the cycle for continuous monitoring

2.4 Code Implementation

2.4.1 Variable Declaration

Insert the following code in the USER CODE BEGIN PV section to declare a variable for storing the raw ADC value:

```
/* USER CODE BEGIN PV */
uint16_t raw;
/* USER CODE END PV */
```

Listing 1: ADC Value Storage Variable

This 16-bit unsigned integer stores the digital value from the ADC (0 to 4095 for a 12-bit ADC).

2.4.2 Core ADC Sampling Function

Insert the following code in the main while loop (USER CODE BEGIN 3):

```
/* USER CODE BEGIN 3 */
// Start the ADC conversion
HAL_ADC_Start(&hadc1);
// Poll for conversion complete (timeout: 300ms)
HAL_ADC_PollForConversion(&hadc1, 300);
// Read the converted value
raw = HAL_ADC_GetValue(&hadc1);
// Small delay for next measurement
HAL_Delay(10);
/* USER CODE END 3 */
```

Listing 2: ADC Polling Implementation

Code Explanation:

- `HAL_ADC_Start(&hadc1)`: Initiates an ADC conversion on the configured channel
- `HAL_ADC_PollForConversion(&hadc1, 300)`: Blocks until conversion completes or 300ms elapses
- `HAL_ADC_GetValue(&hadc1)`: Returns the 12-bit digital result (0–4095)
- `HAL_Delay(10)`: Provides a 10ms delay before the next measurement

2.4.3 Interrupt Service Routines

This project does not utilise interrupt service routines. The polling approach is sufficient for demonstration purposes, though interrupt-driven ADC sampling would be more efficient for production code.

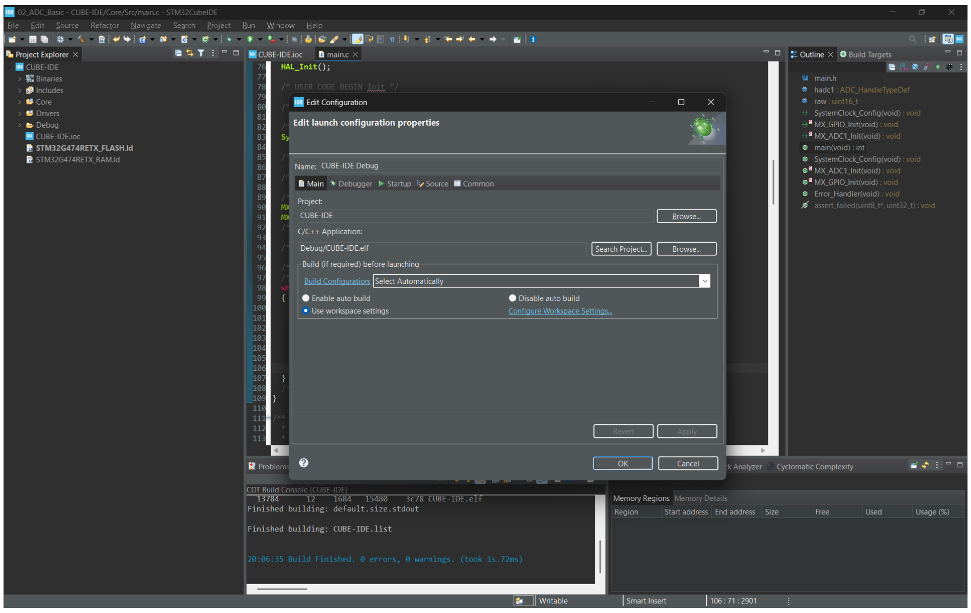
2.5 Building and Flashing

The build and flash procedures are identical to Project 01:

1. Right-click the project and select **Build Project**
2. Verify successful compilation with *Build Finished 0 errors, 0 warnings*
3. Connect the NUCLEO board via USB
4. Click the green play button or select **Run As** → **STM32 C/C++ Application**
5. The board is now running the ADC sampling firmware

2.6 Debugging and Observation

2.6.1 Setting Up the Debugger



Edit Configuration

Edit launch configuration properties

Name: CUBE-IDE Debug

Main | Debugger | Startup | Source | Common

Project: CUBE-IDE

C/C++ Application: Debug/CUBE-IDE.elf

Build (if required) before launching

Build Configuration: Select Automatically

Enable auto build | Disable auto build

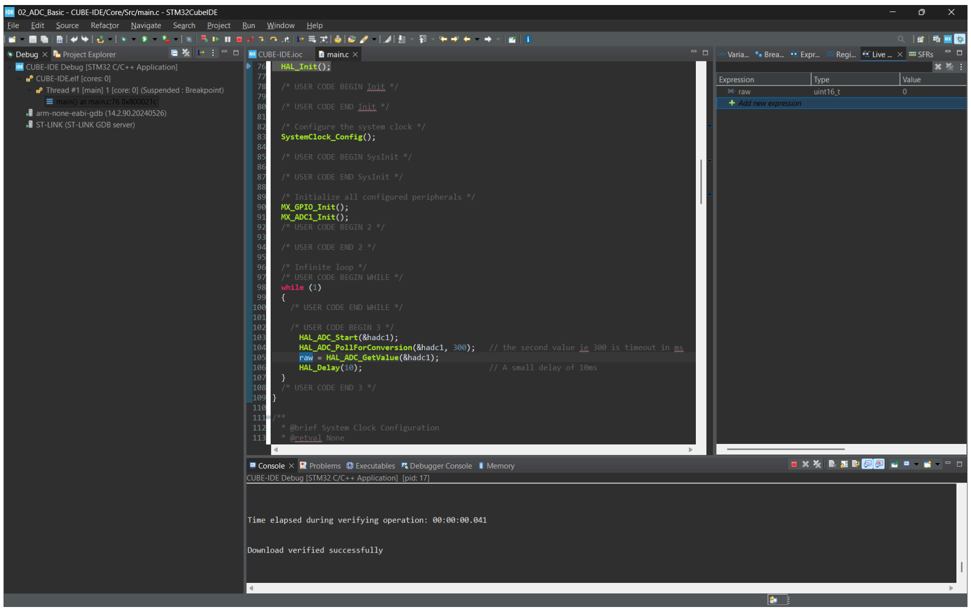
[✓ Use workspace settings] | Configure Workspace Settings...

CDT Build Console [CUBE-IDE]

```
  13784      12    1684   15480    3c78 CUBE-IDE.elf
Finished building: default.size.stdout

Finished building: CUBE-IDE.list


20:06:35 Build Finished. 0 errors, 0 warnings. (took 1s.72ms)
```

Figure 3: Debugger configuration and connection. Accept default settings and activate the debugger by clicking the debug button.

2.6.2 Observing Variable Changes



```
CUBE-IDE Debug [STM32 C/C++ Application]
  CUBE-IDE.elf [cores: 0]
    Thread #1 [main] 1 [core: 0] (Suspended : Breakpoint)
      main() at main.c:76 0x800021c
  arm-none-eabi-gdb (14.2.90.20240526)
  ST-LINK (ST-LINK GDB server)
```

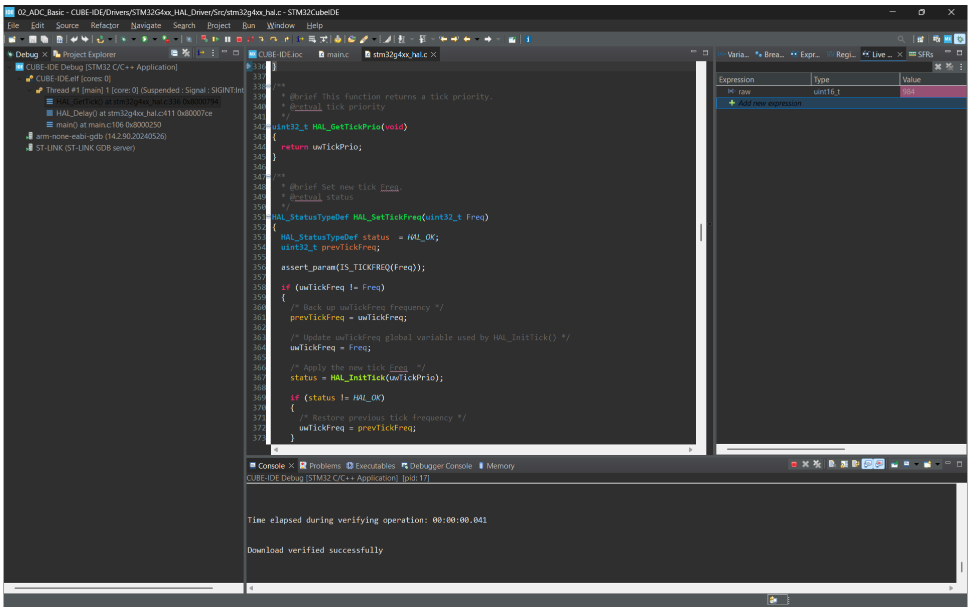
```c
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC1_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
      HAL_ADC_Start(&hadc1);
      HAL_ADC_PollForConversion(&hadc1, 300);   // the second value ie 300 is timeout in ms
      raw = HAL_ADC_GetValue(&hadc1);
      HAL_Delay(10);                            // A small delay of 10ms
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
```

Expression	Type	Value
raw	uint16_t	0
Add new expression		

```
CUBE-IDE Debug [STM32 C/C++ Application]  [pid: 17]

Time elapsed during verifying operation: 00:00:00.041

Download verified successfully
```

Figure 4: Live expression viewer. In the Variables pane (top-right), add the `raw` variable to watch window and observe its value in real-time as you adjust the potentiometer.

2.6.3 Real-Time Monitoring



```
CUBE-IDE Debug [STM32 C/C++ Application]
  CUBE-IDE.elf [cores: 0]
    Thread #1 [main] 1 [core: 0] (Suspended : Signal : SIGINT:Int
      HAL_GetTick() at stm32g4xx_hal.c:336 0x8000794
      HAL_Delay() at stm32g4xx_hal.c:411 0x80007ce
      main() at main.c:106 0x8000250
  arm-none-eabi-gdb (14.2.90.20240526)
  ST-LINK (ST-LINK GDB server)
```

```c
}

/**
  * @brief This function returns a tick priority.
  * @retval tick priority
  */
uint32_t HAL_GetTickPrio(void)
{
  return uwTickPrio;
}

/**
  * @brief Set new tick Freq.
  * @retval status
  */
HAL_StatusTypeDef HAL_SetTickFreq(uint32_t Freq)
{
  HAL_StatusTypeDef status  = HAL_OK;
  uint32_t prevTickFreq;

  assert_param(IS_TICKFREQ(Freq));

  if (uwTickFreq != Freq)
  {
    /* Back up uwTickFreq frequency */
    prevTickFreq = uwTickFreq;

    /* Update uwTickFreq global variable used by HAL_InitTick() */
    uwTickFreq = Freq;

    /* Apply the new tick Freq  */
    status = HAL_InitTick(uwTickPrio);

    if (status != HAL_OK)
    {
      /* Restore previous tick frequency */
      uwTickFreq = prevTickFreq;
    }
```

Expression	Type	Value
raw	uint16_t	984
Add new expression		

```
CUBE-IDE Debug [STM32 C/C++ Application]  [pid: 17]

Time elapsed during verifying operation: 00:00:00.041

Download verified successfully
```

Figure 5: Live variable output. The variable watch window displays the `raw` ADC value changing in real-time as the potentiometer is adjusted, confirming successful ADC operation.

Play (resume) the execution using the play button. Adjust the potentiometer and observe the `raw` variable value change in real-time, confirming that the ADC is correctly sampling the analogue input.

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* • *ARM GCC Compiler configured as **GNU Tools for ARM Embedded Processors*** • *NUCLEO-G474RE development board with potentiometer connected to PA0* • *Completion of Project 01 to understand the workflow* GitHub Project Link

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

3.2.1 Model Architecture

The Simulink model for ADC sampling consists of three functional blocks:

1. **ADC Block**: Reads the analogue value from pin PA0
2. **Gain Block**: Converts the raw 12-bit digital value (0–4095) to voltage (0–3.3V)
3. **Scope Block**: Logs and visualises the sampled voltage over time

This simple architecture demonstrates data acquisition and signal processing in a model-based design approach.

3.2.2 Block Functionality

Table 2: Simulink Block Descriptions for ADC Project

Block	**Purpose**
ADC Read	Acquires raw 12-bit digital samples from ADC channel 1
Gain	Converts digital values to voltage using Voltage = Raw $\times \frac{3.3}{4095}$
Scope	Displays the converted voltage values for real-time observation

3.3 Model Configuration

3.3.1 ADC Block Configuration

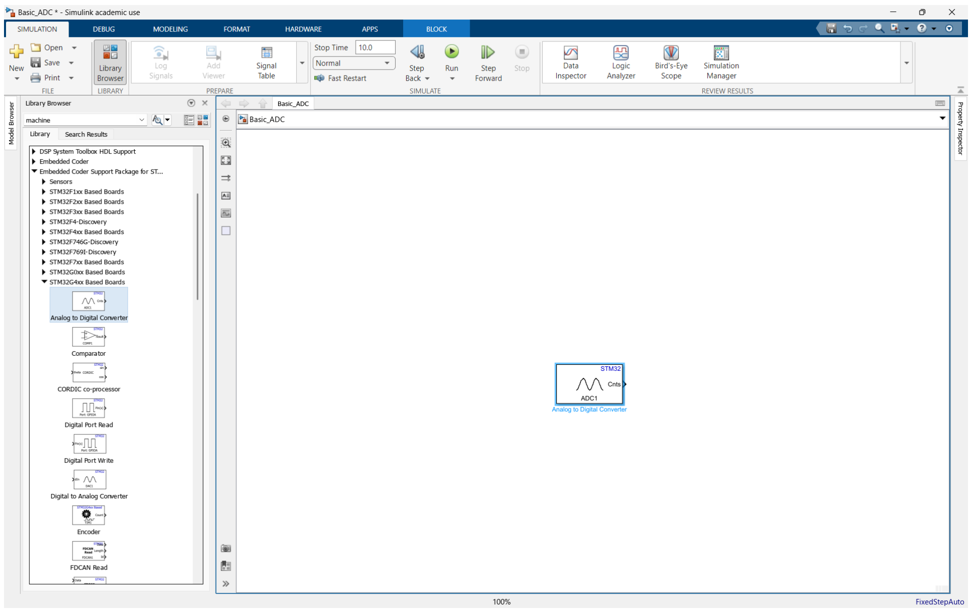
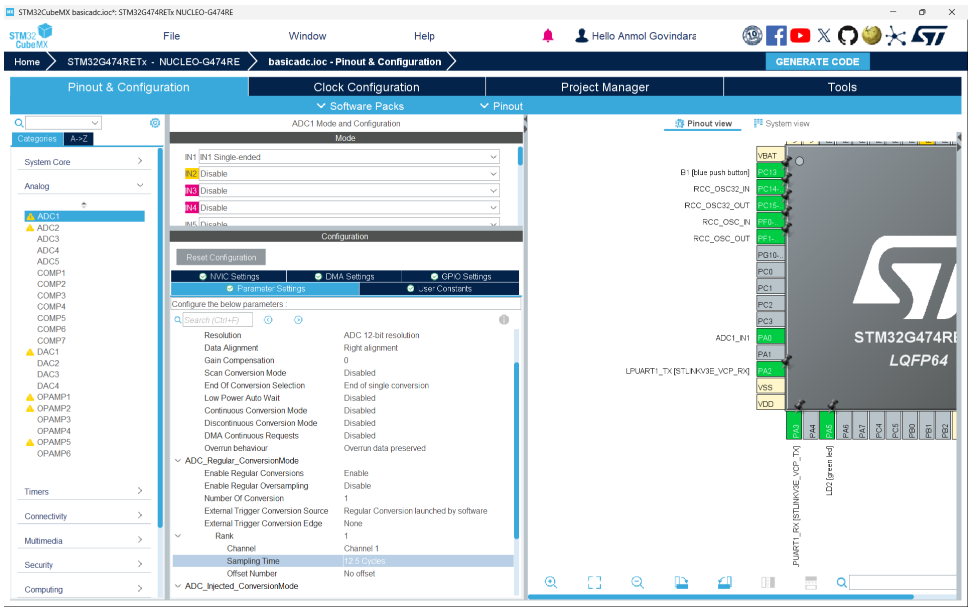


Figure 6: ADC block selection and insertion. Drag and drop the ADC block from the STM32G4xx library into the Simulink model.

3.3.2 STM32CubeMX Configuration



ADC1 Mode and Configuration

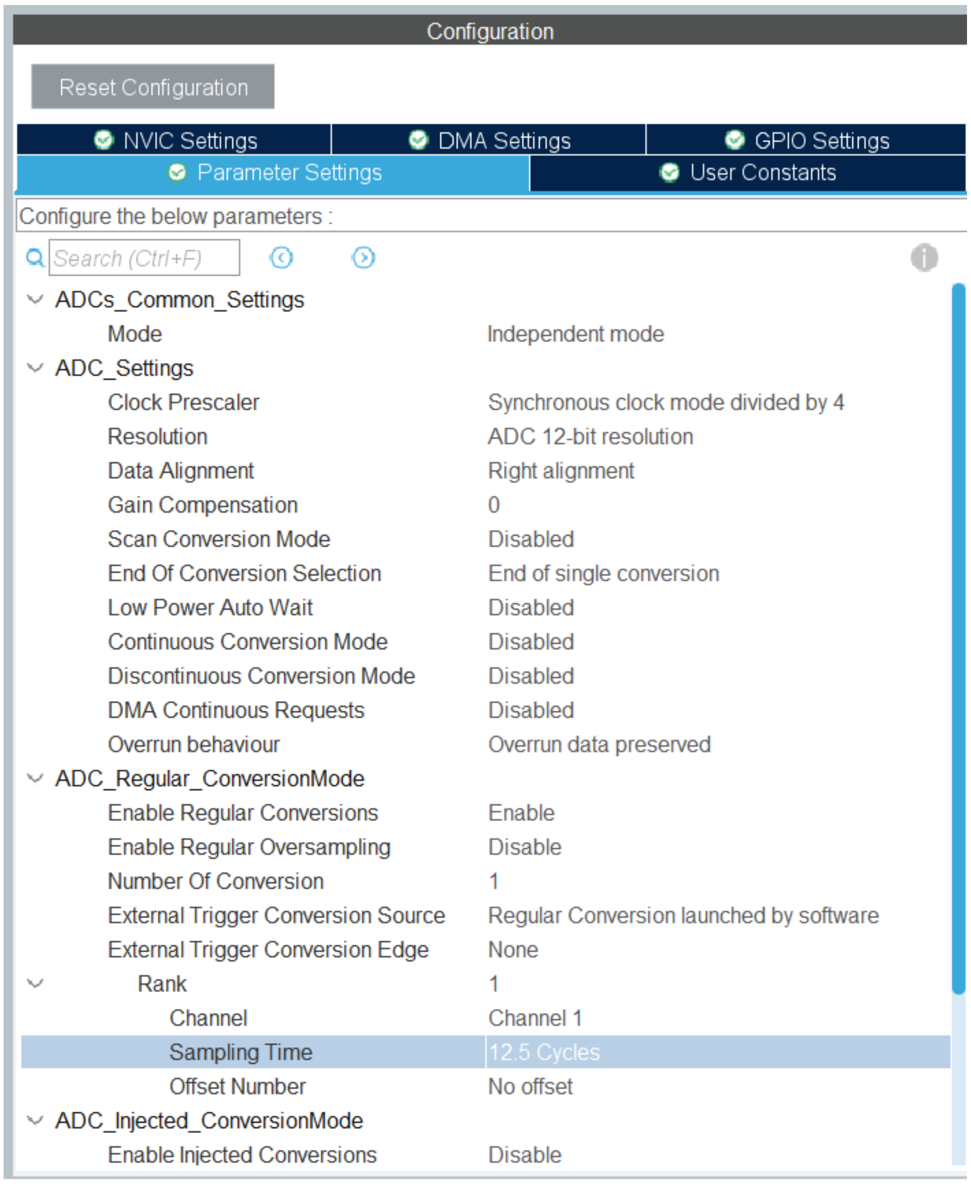
Mode

IN1 IN1 Single-ended
IN2 Disable
IN3 Disable
IN4 Disable

Configuration

Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Gain Compensation	0
Scan Conversion Mode	Disabled
End Of Conversion Selection	End of single conversion
Low Power Auto Wait	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
Overrun behaviour	Overrun data preserved
ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Enable Regular Oversampling	Disable
Number Of Conversion	1
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None
Rank	1
Channel	Channel 1
Sampling Time	12.5 Cycles
Offset Number	No offset
ADC_Injected_ConversionMode	

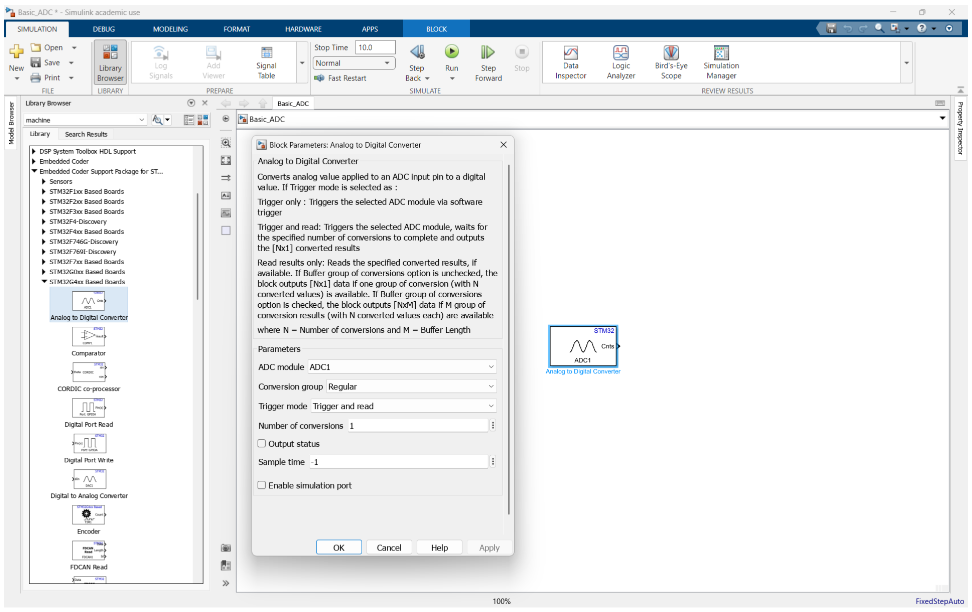
Figure 7: ADC channel configuration in STM32CubeMX. Configure ADC1 channel 1 in single-ended mode.



Configure the below parameters :

ADCs_Common_Settings	
Mode	Independent mode
ADC_Settings	
Clock Prescaler	Synchronous clock mode divided by 4
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Gain Compensation	0
Scan Conversion Mode	Disabled
End Of Conversion Selection	End of single conversion
Low Power Auto Wait	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
Overrun behaviour	Overrun data preserved
ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Enable Regular Oversampling	Disable
Number Of Conversion	1
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None
Rank	1
Channel	Channel 1
Sampling Time	12.5 Cycles
Offset Number	No offset
ADC_Injected_ConversionMode	
Enable Injected Conversions	Disable

Figure 8: ADC sampling parameters. Set sampling time to 12.5 clock cycles and ADC prescaler to 4.


Block Parameters: Analog to Digital Converter

Analog to Digital Converter

Converts analog value applied to an ADC input pin to a digital value. If Trigger mode is selected as :

Trigger only : Triggers the selected ADC module via software trigger

Trigger and read: Triggers the selected ADC module, waits for the specified number of conversions to complete and outputs the [Nx1] converted results

Read results only: Reads the specified converted results, if available. If Buffer group of conversions option is unchecked, the block outputs [Nx1] data if one group of conversion (with N converted values) is available. If Buffer group of conversions option is checked, the block outputs [NxM] data if M group of conversion results (with N converted values each) are available

where N = Number of conversions and M = Buffer Length

Parameters

ADC module: ADC1

Conversion group: Regular

Trigger mode: Trigger and read

Number of conversions: 1

☐ Output status

Sample time: -1

☐ Enable simulation port

Figure 9: ADC block trigger and read mode configuration. Ensure Trigger and Read Mode are properly selected.

3.3.3 Gain Block Configuration

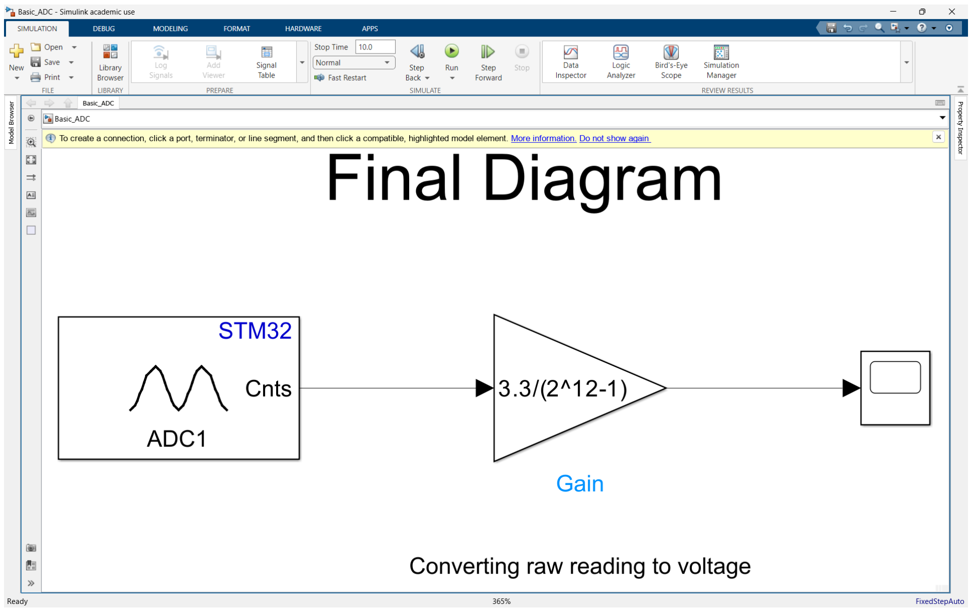
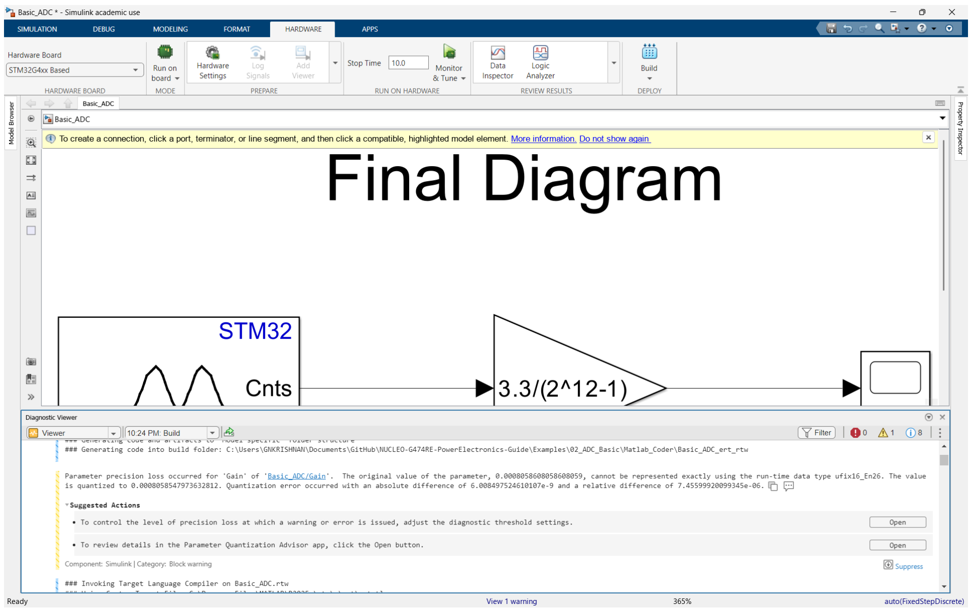


Figure 10: Complete model architecture with gain block. Add a gain block with value $\frac{3.3}{2^{12}-1} = \frac{3.3}{4095} = 0.000805664$ to convert raw ADC values to voltage.

3.3.4 Model Compilation

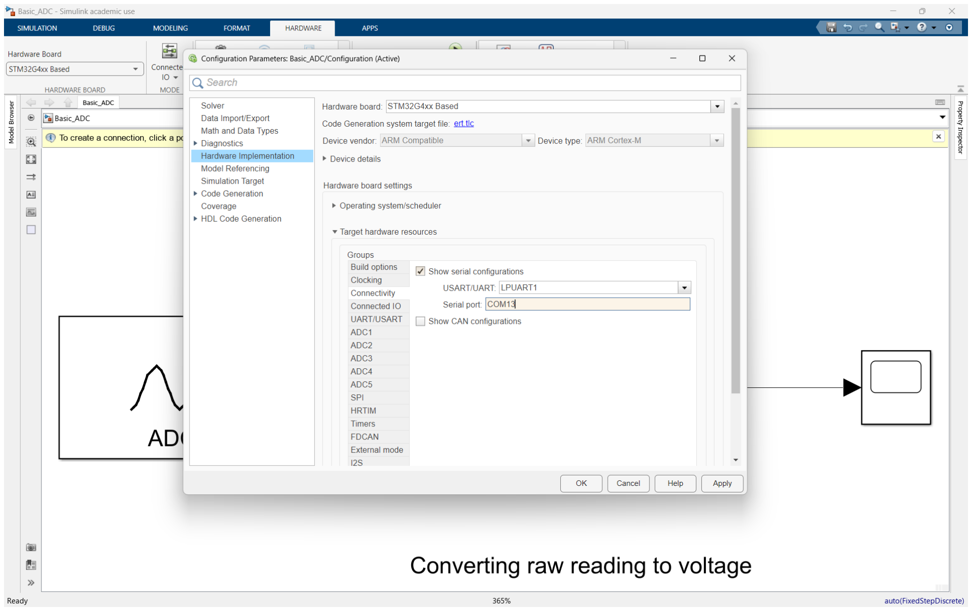


```
### Generating code into build folder: C:\Users\GNKRISHNAN\Documents\GitHub\NUCLEO-G474RE-PowerElectronics-Guide\Examples\02_ADC_Basic\Matlab_Coder\Basic_ADC_ert_rtw

Parameter precision loss occurred for 'Gain' of 'Basic_ADC/Gain'.  The original value of the parameter, 0.0008058608058608059, cannot be represented exactly using the run-time data type ufix16_En26. The value
is quantized to 0.0008058547973632812. Quantization error occurred with an absolute difference of 6.008497524610107e-9 and a relative difference of 7.45599920099345e-06.

Suggested Actions
  • To control the level of precision loss at which a warning or error is issued, adjust the diagnostic threshold settings.
  • To review details in the Parameter Quantization Advisor app, click the Open button.
Component: Simulink | Category: Block warning

### Invoking Target Language Compiler on Basic_ADC.rtw
```

Figure 11: Building the Simulink model. A warning regarding gain block accuracy may appear, but tolerance is well within acceptable limits. Click Build to compile.

3.4 Hardware Connectivity Configuration

3.4.1 Serial Port Configuration



Configuration Parameters: Basic_ADC/Configuration (Active)

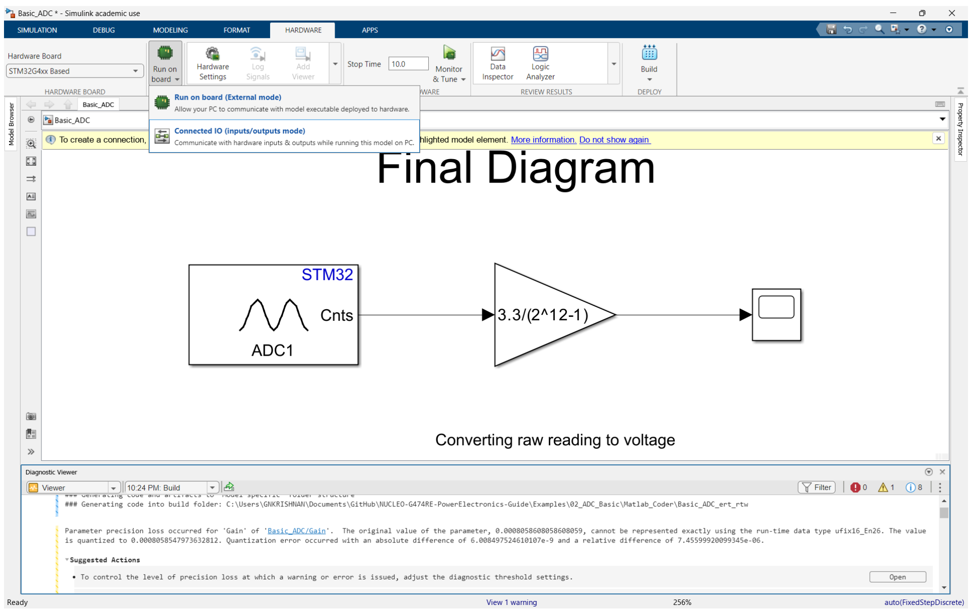
- Solver
- Data Import/Export
- Math and Data Types
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation
- Coverage
- HDL Code Generation

Hardware board: STM32G4xx Based
Code Generation system target file: ert.tlc
Device vendor: ARM Compatible Device type: ARM Cortex-M
Device details

Hardware board settings
Operating system/scheduler
Target hardware resources
Groups: Build options, Clocking, Connectivity, Connected IO, UART/USART, ADC1, ADC2, ADC3, ADC4, ADC5, SPI, HRTIM, Timers, FDCAN, External mode, I2S

[✓] Show serial configurations
USART/UART: LPUART1
Serial port: COM13
[ ] Show CAN configurations

Figure 12: Hardware connectivity setup. Navigate to Hardware Configuration and select the COM port where the STM32 is connected (visible in Device Manager or System Information).


Basic_ADC * - Simulink academic use
SIMULATION
DEBUG
MODELING
FORMAT
HARDWARE
APPS
Hardware Board
STM32G4xx Based
HARDWARE BOARD
Run on board
Hardware Settings
Log Signals
Add Viewer
Stop Time 10.0
Monitor & Tune
Data Inspector
Logic Analyzer
REVIEW RESULTS
Build
DEPLOY
Run on board (External mode)
Allow your PC to communicate with model executable deployed to hardware.
Connected IO (inputs/outputs mode)
Communicate with hardware inputs & outputs while running this model on PC.
To create a connection, ... hlighted model element. More information. Do not show again
Final Diagram
STM32
Cnts
ADC1
3.3/(2^12-1)
Converting raw reading to voltage
Diagnostic Viewer
Viewer
10:24 PM: Build
Filter
Ready
View 1 warning
256%
auto(FixedStepDiscrete)

```
### Generating code into build folder: C:\Users\GNKRISHNAN\Documents\GitHub\NUCLEO-G474RE-PowerElectronics-Guide\Examples\02_ADC_Basic\Matlab_Coder\Basic_ADC_ert_rtw

Parameter precision loss occurred for 'Gain' of 'Basic_ADC/Gain'.  The original value of the parameter, 0.0008058608058608059, cannot be represented exactly using the run-time data type ufix16_En26. The value
is quantized to 0.0008058547973632812. Quantization error occurred with an absolute difference of 6.008497524610107e-9 and a relative difference of 7.45599920099345e-06.

Suggested Actions
 • To control the level of precision loss at which a warning or error is issued, adjust the diagnostic threshold settings.
```

Figure 13: Connected I/O mode activation. Change the communication mode to Connected I/O to enable real-time data exchange between the computer and the board.

3.4.2 Simulation Settings

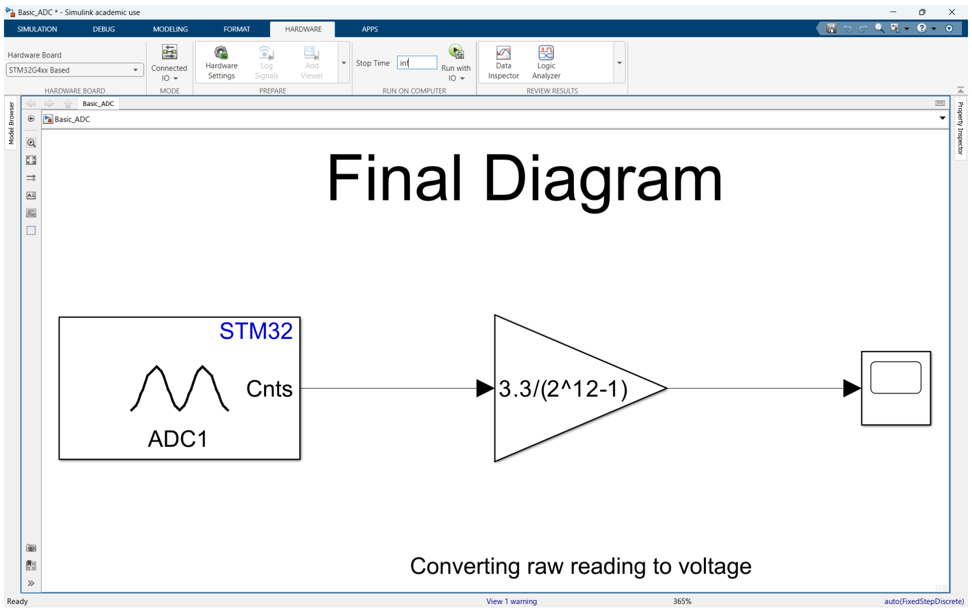
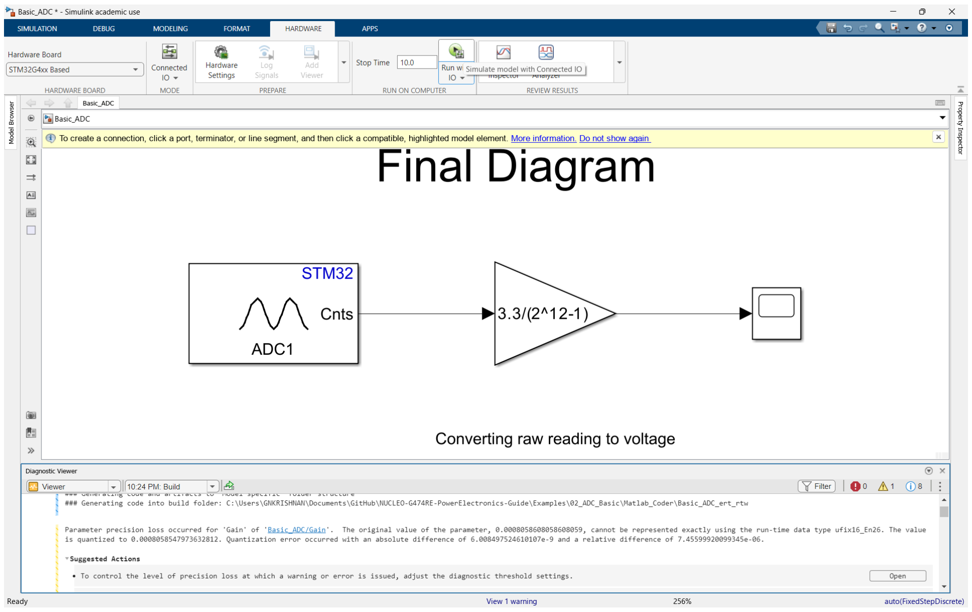


Figure 14: Simulation time configuration. Set the simulation time to *Infinity* (inf) to allow continuous monitoring of ADC values.

3.5 Running and Observing Results

3.5.1 Executing the Model



```
### Generating code into build folder: C:\Users\GNKRISHNAN\Documents\GitHub\NUCLEO-G474RE-PowerElectronics-Guide\Examples\02_ADC_Basic\Matlab_Coder\Basic_ADC_ert_rtw

Parameter precision loss occurred for 'Gain' of 'Basic_ADC/Gain'.  The original value of the parameter, 0.0008058608058608059, cannot be represented exactly using the run-time data type ufix16_En26. The value is quantized to 0.0008058547973632812. Quantization error occurred with an absolute difference of 6.008497524610107e-9 and a relative difference of 7.45599920099345e-06.

Suggested Actions
• To control the level of precision loss at which a warning or error is issued, adjust the diagnostic threshold settings.
```

Figure 15: Running the model in Connected I/O mode. Click the play button to start real-time data acquisition and processing.

3.5.2 Data Visualisation

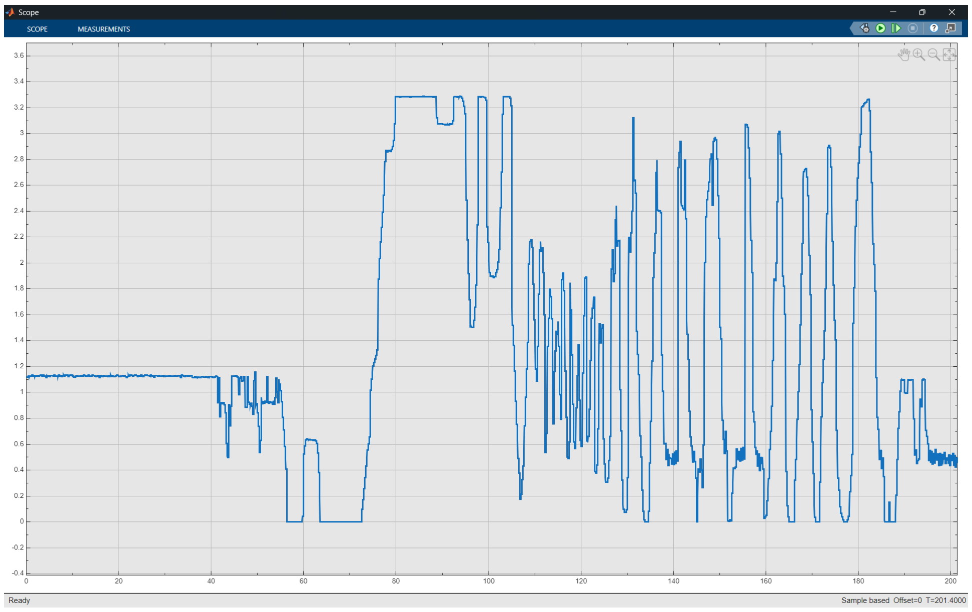


Figure 16: Real-time scope display. Observe the sampled voltage values in real-time as you adjust the potentiometer. The scope shows the actual voltage (0–3.3V) after gain conversion.

The scope display shows the converted voltage values in real time. Adjust the potentiometer and observe the voltage trace change smoothly from 0V to 3.3V, confirming successful ADC operation and model execution on the hardware.

4 STM32CubeIDE vs. MATLAB Embedded Coder

4.1 Implementation Comparison

Table 3: Comparison of ADC Implementation Approaches

Aspect	**STM32CubeIDE**	**MATLAB Embedded Coder**
Development	Direct HAL function calls for ADC control	Visual block diagram with automatic code generation
Code Control	Complete control over ADC polling sequence	Abstracted ADC implementation
Real-Time Monitoring	Debugger with variable watch window	Live scope and Connected I/O communication
Signal Processing	Manual gain calculation and application	Model-based gain blocks with automatic code generation
Verification	Step-by-step debugging and variable inspection	Real-time simulation with scope visualisation

4.2 Approach Selection

Use STM32CubeIDE for:

- Direct hardware control and optimisation
- Debugging at the C code level
- Simple ADC applications with minimal signal processing

Use MATLAB Embedded Coder for:

- Complex signal processing algorithms
- Rapid prototyping with visual modelling
- Real-time data acquisition and analysis
- Applications requiring multiple signal conversions

5 Results and Analysis

5.1 Expected Outcomes

STM32CubeIDE Implementation:

The `raw` variable in the debugger displays values from 0 to 4095 as the potentiometer is adjusted from 0V to 3.3V. Linear variation across the entire range confirms proper ADC operation.

MATLAB Embedded Coder Implementation:

The scope displays the converted voltage (0–3.3V) in real-time. Smooth, continuous traces indicate successful ADC sampling and gain conversion, with minimal quantisation noise.

5.2 Performance Observations

1. **Sampling Rate**: Both implementations sample at approximately 1.7 MHz, more than sufficient for voltage sensing
2. **Resolution**: 12-bit resolution provides ±1.6 mV measurement uncertainty
3. **Responsiveness**: Changes in potentiometer position are reflected in ADC values within microseconds
4. **Stability**: Steady-state readings show minimal jitter, indicating stable operation

5.3 Key Observations

- ADC sampling is deterministic and repeatable
- The 12-bit resolution is adequate for voltage monitoring applications
- Polling-based approach is simple but blocks execution during conversion
- Signal noise is minimal, suggesting adequate filtering and stable reference voltages

6 Troubleshooting

6.1 Common Issues and Solutions

Table 4: Common ADC Issues and Solutions

Issue	**Solution**
ADC reads constant zero or maximum	Verify potentiometer is correctly connected; check PA0 configuration in STM32CubeMX
ADC values erratic or noisy	Add capacitor to ADC input; check for EMI sources; verify power supply stability
MATLAB build fails	Confirm ARM GCC compiler is selected; verify hardware board is STM32G474RE
MATLAB scope shows no data	Check serial port configuration; verify board is connected; ensure model is running
Timeout in HAL_ADC_PollForConversion	Increase timeout value (second parameter); verify ADC clock is enabled

7 Extensions and Enhancements

7.1 Enhancement 1: Multiple Channel ADC

Expand the project to read from multiple ADC channels simultaneously (PA0, PA1, PA2) to monitor multiple analogue inputs such as voltage, current, and temperature.

7.2 Enhancement 2: Interrupt-Driven ADC

Replace the polling approach with interrupt-driven ADC sampling using `HAL_ADC_Start_IT()` and the ADC interrupt handler. This frees the CPU for other tasks whilst ADC conversion occurs in the background.

7.3 Enhancement 3: Digital Filtering

Implement a simple moving average or low-pass filter in the ADC value stream to reduce quantisation noise and improve stability of readings.

8 Conclusion

This project has successfully demonstrated ADC interfacing using both STM32CubeIDE and MATLAB Embedded Coder. The ADC is a fundamental peripheral for embedded systems, enabling measurement of real-world analogue signals.

8.1 Key Takeaways

- The ADC converts continuous analogue signals into discrete digital values
- Resolution and sampling frequency are critical design parameters
- Polling-based ADC sampling is simple but blocking
- STM32CubeIDE provides direct HAL control; MATLAB Embedded Coder offers model-based abstraction
- Real-time monitoring and debugging tools aid in verification and optimisation

8.2 Next Steps

Having mastered basic ADC sampling, the next projects introduce:

- PWM generation (Project 03)
- Interrupt-driven ADC for non-blocking operation
- Advanced signal processing and filtering techniques
- Closed-loop feedback control using ADC measurements
- Usage of differential mode ADC.

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] MathWorks, *Embedded Coder Documentation.*

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

A ADC Mathematical Reference

A.1 Digital to Voltage Conversion

To convert the raw 12-bit ADC value to voltage:

$$V_{\text{measured}} = \text{raw} \times \frac{V_{\text{ref}+} - V_{\text{ref}-}}{2^n - 1} \tag{3}$$

For our configuration:

$$V_{\text{measured}} = \text{raw} \times \frac{3.3}{4095} \tag{4}$$

A.2 Sampling Rate Calculation

$$f_{\text{sample}} = \frac{f_{\text{ADC clock}}}{T_{\text{conv}}} = \frac{42.5\ \text{MHz}}{25\ \text{cycles}} \approx 1.7\ \text{MHz} \tag{5}$$

B Potentiometer Circuit Diagram

The potentiometer is connected as a voltage divider:

- Pin 1: +3.3V (from NUCLEO board)
- Pin 2 (Wiper): PA0 (ADC input)
- Pin 3: GND (Ground)

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

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