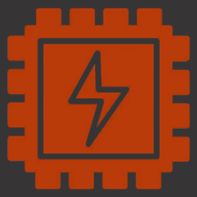

Open Energy Profiler Toolset

Drive innovations in the field of low-powered technologies

Documentation

Energy Profiler Probe

- Firmware -

Developer Guide

30. June 2025

Revision History

Version	Date	Description
1.0.0	30.06.2025.	Initial draft

Content

1. Introduction

This document serves as the official developer guide for the firmware running on the **E**nergy **P**rofiler **P**robe (**EPP**), a critical embedded component within the broader **Open E**nergy **P**rofiler **T**oolset (**OpenEPT**) ecosystem. It is specifically designed for embedded systems developers, system integrators, researchers, and contributors who aim to understand, maintain, or extend the capabilities of this firmware.

The Energy Profiler Probe (EPP) is an essential tool for the development and optimization of modern energy-aware systems, providing both high-speed voltage and current measurement capabilities, up to 1 MSPS sampling rate, and the ability to generate arbitrary current profiles. This combination of features enables developers to perform precise, high-resolution monitoring of power consumption while also actively stimulating and analyzing the behavior of battery-powered devices. Such capabilities are particularly valuable for battery profiling, power efficiency tuning, and real-time energy diagnostics. By unifying data acquisition and signal generation within a single, cohesive hardware platform, the EPP significantly streamlines the process of designing, validating, and optimizing low-power embedded applications, making it an indispensable asset in energy-focused development environments.

The firmware described in this guide is architected for reliability, modularity, and performance, and is optimized to run on STM32 dual-core processors (Cortex-M7 and Cortex-M4). It utilizes FreeRTOS for real-time scheduling and LWIP for lightweight IP stack support, in addition to the official STM32 HAL libraries. This software stack ensures that the EPP meets the demands of both low-latency streaming and high-throughput data processing.

A core advantage of the firmware lies in its layered and extensible architecture. It is logically divided into functional layers that include device and platform drivers, middleware services, application logic, and a central system management layer. This structure not only facilitates clear separation of concerns but also makes the codebase easy to scale and adapt, whether for new hardware peripherals, emerging protocols, or application-specific extensions.

Each software block, such as AnalogIN, AnalogOUT, Energy Debugging, Control Service, EEZ DIB Interface, and Sample Stream, is implemented as a self-contained, thread-safe module with clearly defined interfaces. These services operate independently yet cooperatively under FreeRTOS supervision, ensuring robust performance in real-time scenarios. Whether the goal is to integrate a new external ADC, introduce an advanced streaming format, or enable system-wide energy event tracking, this architecture provides a clean starting point.

This guide offers in-depth coverage of the firmware system, including conceptual overviews, architectural diagrams, peripheral configuration details, and real-world usage scenarios. It walks the reader through essential topics such as stream synchronization, multi-buffered DMA management, sample packet construction, network task coordination, and service initialization sequences. For each functional component, associated source code locations and configuration parameters are clearly identified.

While this document presents a complete top-down explanation of the firmware's operational principles, it is recommended to complement your reading with the auto-generated Doxygen documentation [xxx]. The Doxygen reference provides detailed insights into API declarations, internal data structures, configuration macros, and callback mechanisms, all of which are essential for confidently navigating and extending the codebase. It also includes function-level documentation for low-level drivers and middleware services that may not be fully elaborated in this guide.

2. Architecture

The architecture of the firmware for EPP is presented in the Figure 2.1:

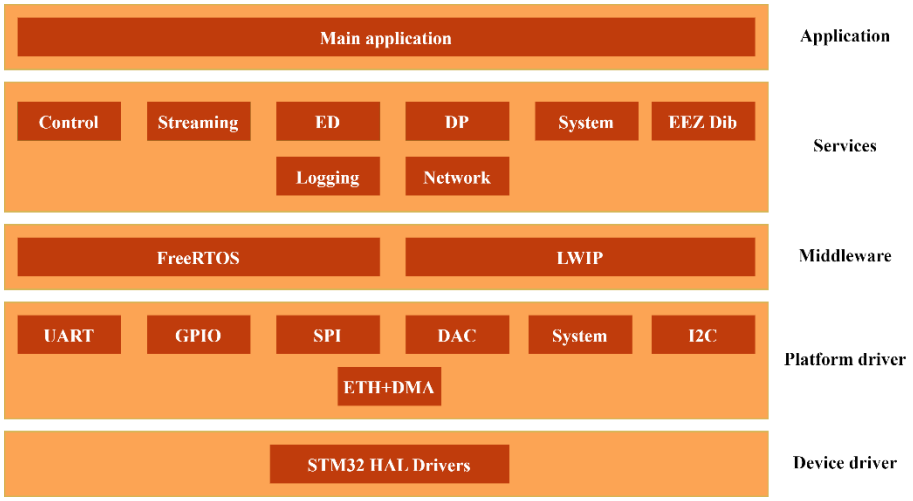

Figure 2.1 – EPP's firmware architecture

The presented architecture includes multiple layers each composed of various functional software blocks. The primary architecture layers include:

- Device drivers' layer
- Platform drivers' layer
- Middleware layer
- Services layer
- Application layer

The standard STM32H7 device HAL drivers are wrapped within the device driver layer.

The introduction of the platform drivers layer ensures that the STM32H7 device driver is thread safe. From a higher perspective, the drivers within this layer act as wrappers around standard device drivers, enhancing the portability of our solution across various platforms.

Platform driver layer and higher software layers utilize different mechanisms from third parties' libraries such as FreeRTOS and LWIP library. FreeRTOS library

Within the Middleware layer, we have implemented a collection of unique functionalities, carefully divided into individual RTOS tasks. These tasks are designed to harness the capabilities of specific lower software layers, creating a well-organized structure that enhances the modularity and efficiency of our system architecture. This approach not only promotes clarity and maintainability but also allows for seamless integration and scaling of functionalities within the overall system framework.

The Application layer serves as the embodiment of the main firmware logic, taking on the responsibility of initializing all lower layers and initiating the RTOS scheduler. This layer acts as the orchestrator, setting the stage for seamless interaction and collaboration of the various components within the system. By encapsulating the core functionality, it establishes a cohesive framework that ensures the proper execution and synchronization of tasks throughout the entire system.

3. Source Code Organization

Complete project source code is available under *Firmware* repository on the official OpenEPT organization on the GitHub.

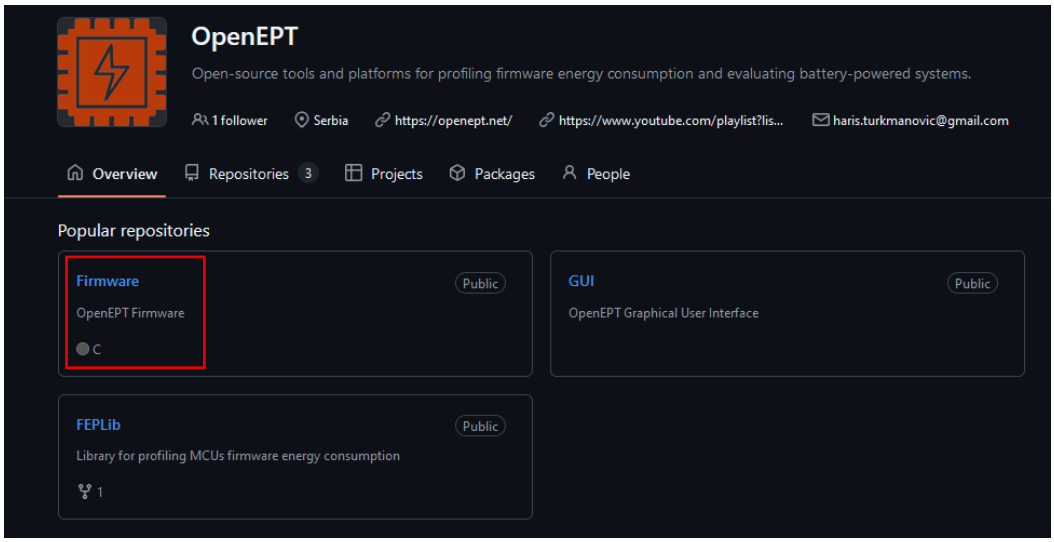

Figure 3.1 - OpenEPT Organization on GitHub and Firmware repository

The Firmware repository is organized as it is presented on Figure 3.2:

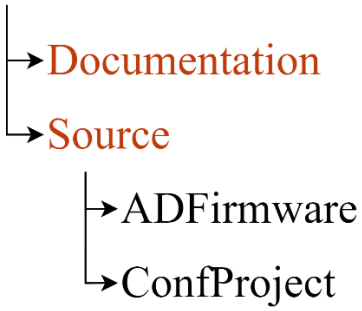

Figure 3.2 - Firmware repository top level

Top level repository hierarchy contains two subdirectories:

- **Documentation**
  Here are located scripts to generate documentation based on code comments (Doxygen)
- **Source**
  Here are located two subdirectories:
  - ADFirmware – STM32 Cube IDE project that contains all source code for Acquisition Device.
  -ConfProject - STM32 Cube IDE project that contains *.ioc* file that is useful for fast device configuration (this is used during development phase, and it is left for testing purpose only).

Therefore, all source code of Acquisition device is located on path *Source/ADFirmware* which top directory structure is illustrated on

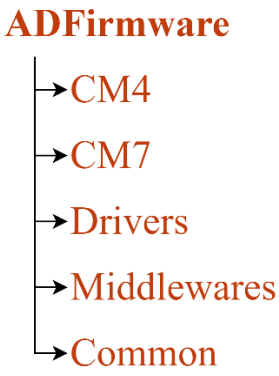

Figure 3.3 - ADFirmware directory structure

- **CM4**
  Complete source code that will be run on Cortex M4 core.
- **CM7**
  Complete source code that will be run on Cortex M7 core.
- **Common**
  Startup file directory to booth all cores successfully.
- **Drivers**
  STM32 driver library from the official STM32H7 Cube git repository
- **Middleware**
  Third parties' libraries are common for all cores (for example FreeRTOS, LwIP, etc)

Within each core directory (CM4 and CM7) code directory organization is implemented to correspond to the overall firmware architecture presented on Figure 2.1. This organization is illustrated on Figure 3.4.

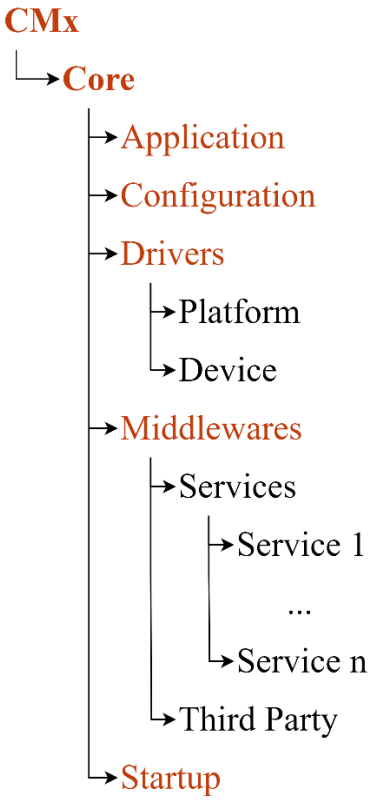

Figure 3.4 - Firmware code directory structure

- **Application**
  Contains firmware top-level source code.

- **Configuration**
  Contains all configuration files such as global firmware configuration, LwIP configuration, FreeRTOS configuration
- **Drivers**
  This directory contains two sub-directories: Platform and Device. The platform directory contains thread-safe platform-independent drivers for different peripherals (UART, SPI, I2C) while the Device directory is linked to the Driver's directory.
- **Middleware**
  This directory contains two subdirectories: Services and Third party. Each service functionality, that corresponds to the Acquisition device architecture described here, will be implemented within a separate subdirectory.

4. Functional Software Blocks

4.1. Drivers

4.1.1. Analog IN

Block Summary			
Name	Analog IN	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/AnalogIN/drv_ain.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/AnalogIN/drv_ain.h`			
ADS9224R ADC Source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/AnalogIN/ADS9224R/ads9224r.c`			
ADS9224R ADC Header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/AnalogIN/ADS9224R/ads9224r.h`			

Continuous voltage and current sample acquisition is implemented within the Analog IN driver's software block whose main components are illustrated in Figure 4.1.

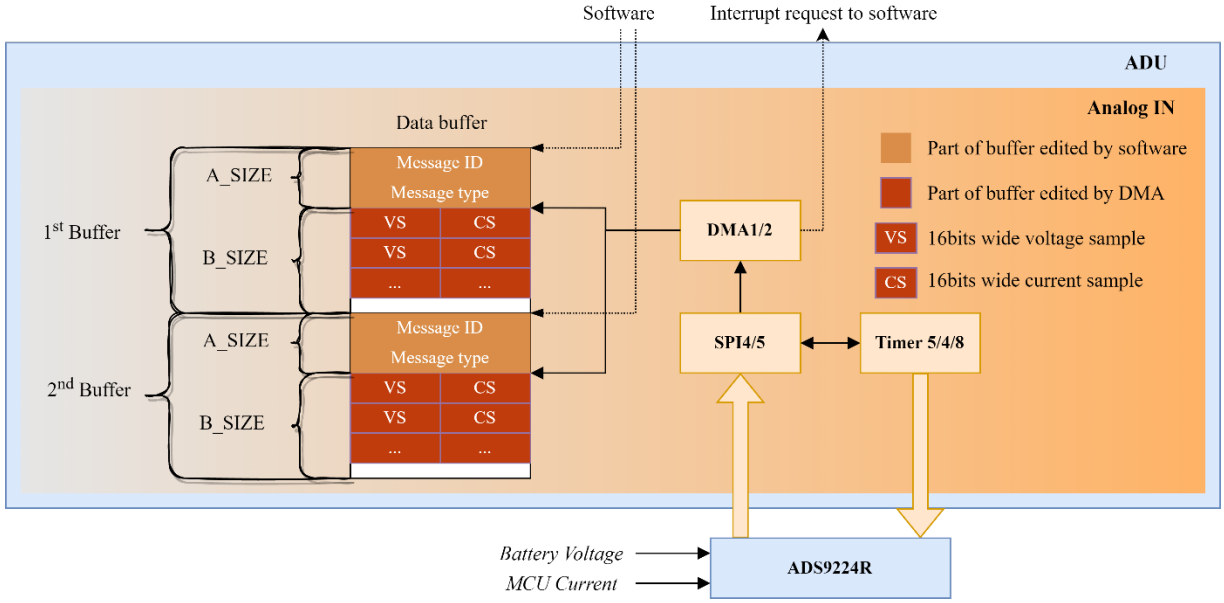

Figure 4.1 - Functional elements of AnalogIN software block

This software block is in charge to communicate with external ADC and to continuously, with minimal software assistance, acquire, read and store multiple voltage and current samples into a single packet of data. For such actions, the Timer and DMA peripherals, both connected with SPI, are configured to periodically trigger sample conversion on external ADC. Voltage and Current sampling are performed sequentially after the request to start conversion. At each end of the sampling process, start Timers that generate clock which initiate samples transfer from external ADC to ADU. Sample transfer triggers DMA to store voltage current samples to the specific part inside MCU memory. An interrupt is generated when the buffer is filled with several samples defined with the *DRV_AIN_ADC_BUFFER_MAX_SIZE* macro. Inside ISR is called the previously registered callback function from the Samples Stream service which has forwarded buffer address value.

Analog IN driver is designed to support two operational modes: configuration and acquisition. These two modes are related to ADS9224R operating modes:

- **configuration** – used to configure ADS9224R ADC and to prepare it for acquisition.
- **acquisition** – mode where it is expected to receive sampling request and to generate CLK to read samples from ADS9224R.

After ADS9224R is powered up, and firmware is run on the STM32 MCU platform, it is important to check that ADS9224R is ready to be configured. The function inside the ADS9224R driver, in charge of properly configuring ADS9224R, is named *ADS9224R_Init* and its definition is presented on Figure 4.2.

```
ads9224r_status_t   ADS9224R_Init(ads9224r_config_t *ads9224r_config_t, uint32_t timeout)
{
    uint8_t registersContent[8] = {0};

    memset(&prvADS9224R_DATA, 0, sizeof(ads9224r_handle_t));

    /* Ping ADC Start */
    if(prvADS9224R_PowerDown(1000) != ADS9224R_STATUS_OK) return ADS9224R_STATUS_ERROR;

    if(prvADS9224R_PowerUp(timeout) != ADS9224R_STATUS_OK) return ADS9224R_STATUS_ERROR;

    /* Set configuration state */
    if(prvADS9224R_CONF_SetState() != ADS9224R_STATUS_OK) return ADS9224R_STATUS_ERROR;

    prvADS9224R_DATA.init = ADS9224R_INIT_STATE_INIT;

    /*default sampling period is max*/
    ADS9224R_SetSamplingRate(200-1, 0);

    /*Do initial read */
    for(uint8_t i =0; i < 8; i++)
    {
        if(prvADS9224R_CONF_SPI_Master_ReadReg(i, &registersContent[i], timeout) != ADS9224R_STATUS_OK)
            return ADS9224R_STATUS_ERROR;
    }

    /*Do false write */
    if(ADS9224R_SetPatternState(ADS9224R_FPATTERN_STATE_ENABLED, timeout) != ADS9224R_STATUS_OK)
            return ADS9224R_STATUS_ERROR;

    /*Do false write */
    if(ADS9224R_SetPatternState(ADS9224R_FPATTERN_STATE_DISABLED, timeout) != ADS9224R_STATUS_OK)
            return ADS9224R_STATUS_ERROR;

    return ADS9224R_STATUS_OK;

}
```

Figure 4.2 - ADS9224R_Init function definition

One of the first steps performed within this function is to “ping” the device to check if ADS9224R is present and if it is ready to be configured. This “ping” operation considers powering down and then powering up the device to force ADS9224R to generate a ready signal pulse. The device is power-down by pulling the #PD/RST pin and holding it minimum t_{WL_PD} period defined within a datasheet, which is in our case about 1 ms. After this, the device is powered up by pulling the #PD/RST pin high. When ADS9224R detects a rising edge, if it is ready and present within a system, consequently it will generate a READY signal too high, and the duration of the high level will be T_{PU} = 0.9ms. This is illustrated on Figure 4.3.

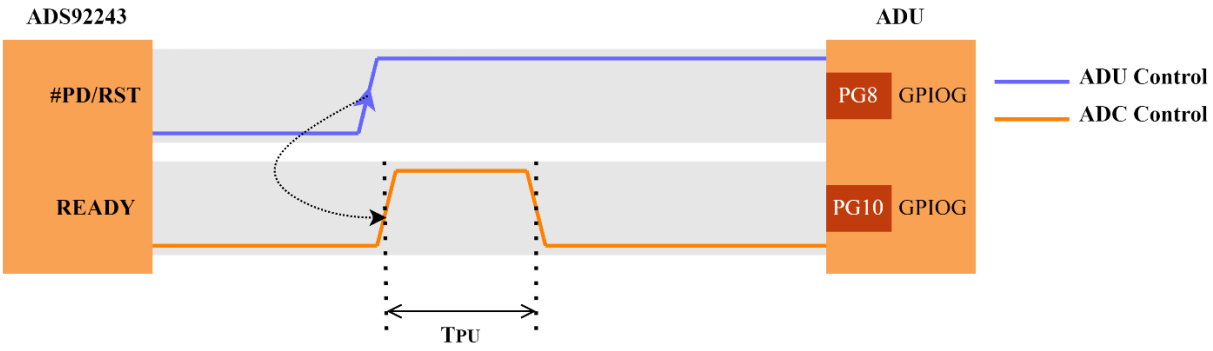

Figure 4.3 - ADS9224R Powerup cycle

After the device is responded to by a rising Ready signal, it is ready to be configured and the ADS9224R driver is moved to Configuration mode by calling private function *prvADS9224R_CONF_SetState*. In Configuration mode is mainly performed writing and reading device registers over standard SPI protocol. On the ADU side is used SPI3 instance of SPI that is configured to operate in Master mode and few GPIO pins. Lines between ADS9224R and ADU used within Configuration mode are illustrated in Figure 4.4.

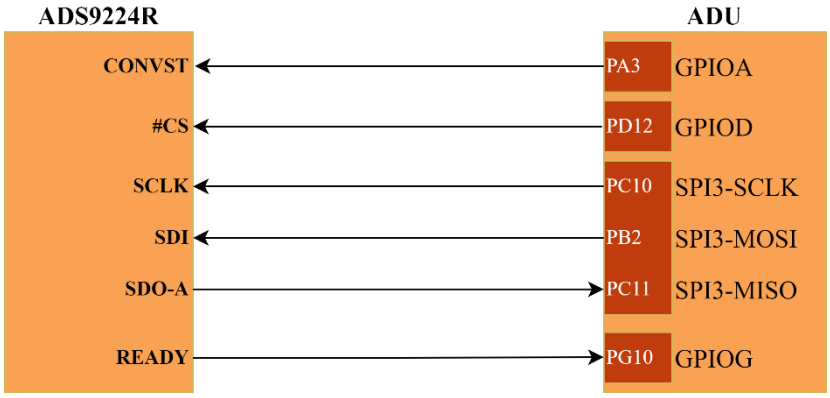

Figure 4.4 - Lines in configuration mode

SPI3 instance on the STM32 side is configured to operate in Master mode. Pin 12 of GPIOD is configured as output and it is used as Chip Select signal for SPI while Pin 10 of GPIOG is configured as input and is used for reading operations. Pin 3 of GPIOA is used to keep the CONVST input of ADS9224R high during the configuration stage.

Different types of operations over ADS9224R registers are supported within ADS9224R. Operations are performed by sending the corresponding commands to the ADS9224R device over SPI. Even if there are multiple supported operations, within a current firmware version, two operations are extensively used: Register write, and Register read.

To perform register, and write operations, the standard procedure of sending two bytes over SPI is performed. This procedure is described in official documentation [1]. However, there are a few important notices related to the reading operations. When the register content wants to be read, two bytes are sent from MCU where the first 4 bits indicate the read command. After these two bytes are sent to the device, the device raises a READY pin which indicates that the requested register content is ready to be sent to MCU. After the Ready pin is set high, STM32 can perform an SPI read of one byte. This sequence is illustrated in Figure 4.5.

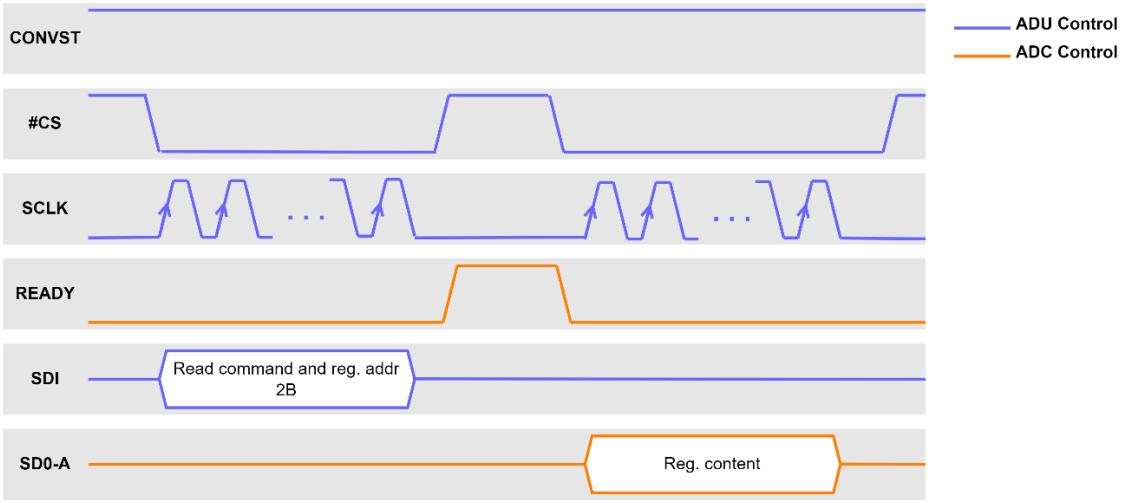

Figure 4.5 - Read the sequence timing diagram

There are two possible conversion control and data transfer frame modes supported by ADS9224R: Zone 1 and Zone 2. These two modes are detailed and described in official ADC documentation [1] and their described protocol between ADC and MCU. For our project, Zone 1 is more suitable even if it requires a few more hardware resources.

To enable streaming from ADC to MCU, the following peripherals are used:

- Timer instance No 5 (TIM5) – Configured in master mode and it used to Periodically trigger conversion
- Timer instance No 4 (TIM4) – Configured in slave mode and it is used to control Chip select (CS) signal
- Timer instance No 8 (TIM8) – Configured in slave mode and it is used to generate a Clock for data transfer
- SPI instance No 5 (SPI5) – Configured in slave mode and it is used to receive data from channel A
- SPI instance No 4 (SPI4) – Configured in slave mode and it is used to receive data from channel B

Schematic which illustrates details related to the connections between ADU and ADS9224R is presented in Figure 4.6.

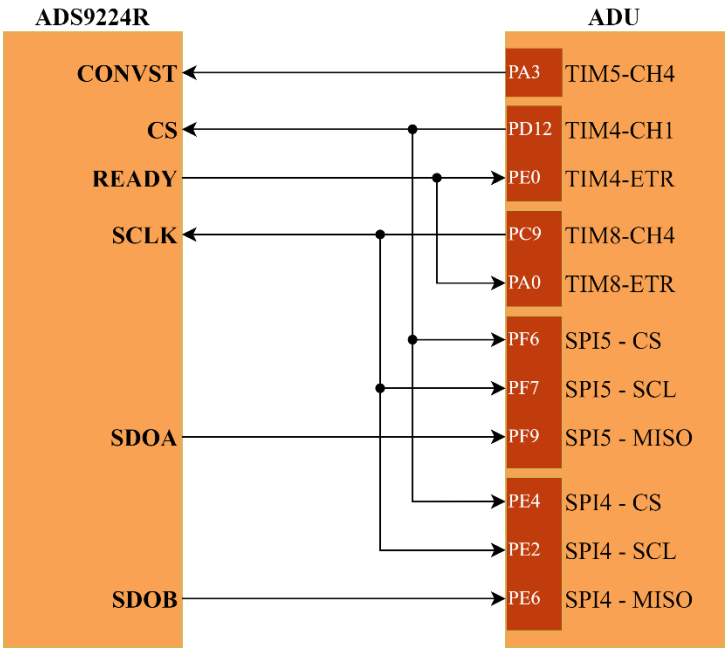

Figure 4.6 - Lines used in Acquisition mode

TIM5 is utilized to periodically trigger the conversion process by generating short high-level pulses on the CONVST input of the ADS9224R ADC. The timer is configured in master mode, with channel 4 operating in PWM mode. TIM5 is connected to STM32's internal APB bus, with its input clock frequency set to the maximum of 200 MHz. To achieve the desired sampling rate, the timer's prescaler and PWM duty cycle are adjusted based on the sampling rate defined by the OpenEPT GUI. All timer configurations related to this process are encapsulated within the *prvADS9224R_TIMER_CONVST_Init* function, while the sampling rate can be configured using the *ADS9224R_SetSamplingRate* function.

TIM4 is responsible for controlling the SPI Chip Select (CS) input of the ADS9224R ADC. It is set to operate in slave mode as a One Pulse timer, triggered by the rising edge of the READY signal generated by the ADS9224R. Channel 1 of the timer is configured in PWM mode, with its active level set to zero. Like TIM5, TIM4 is also connected to the STM32's internal APB bus, with its input clock frequency set to 200 MHz. The total high- and low-level duration of the signal remains constant across all sampling periods. All configurations related to this timer are handled within the *prvADS9224R_TIMER_CS_Init* function.

TIM8 is used to generate clock signal for SPI communication. Like TIM4, TIM8 is configured to operate in slave mode, triggered by the falling edge of the READY signal. Channel 4 is set to PWM mode, with a 50% duty cycle, and the repetition counter is set to 16 (corresponding to one clock period per data bit). The clock frequency remains constant regardless of the sampling period. As with TIM5 and TIM4, TIM8 is also connected to the STM32's internal APB bus, with its input clock frequency set to the maximum of 200 MHz. All timer configurations for this process are encapsulated within the *prvADS9224R_TIMER_SCLK_Init* function.

An overview of Timers' configurations is presented in Table 2

Table 2 – Timer's configuration overview

Timer Instance	Mode	Function	External Trigger	Base CLK [MHZ]	Channel/Mode
TIM4	Slave	Chip Selext	Rising Edge of Ready Signal	200	CH1/PWM
TIM5	Master	Conversion start	-	200	CH4/PWM
TIM8	Slave	SCLK	Falling edge of Ready signal	200	CH1/PWM

After we are presented with the timer's configuration it is important to explain the way they are synchronized to achieve timing for Zone 1. This synchronization is presented in Figure 4.7.

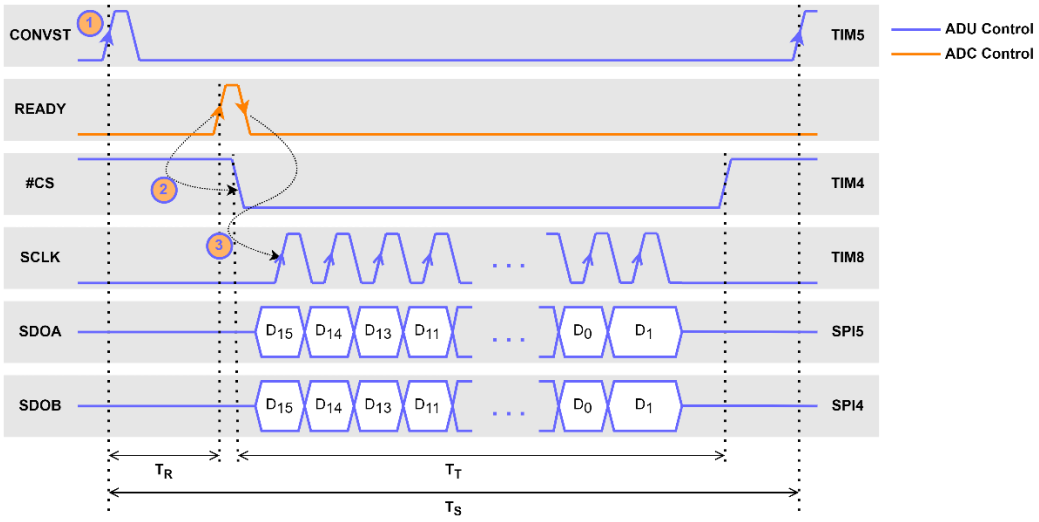

Figure 4.7 - Zone 1 timing diagram

Once the ADS9224R configuration is complete, the device is ready to transmit the acquired samples from channels A and B. The conversion process is initiated by the rising edge of the CONVST signal (1). The minimum high-level duration of this signal is defined by the switching characteristics outlined in [1]. As previously mentioned, this signal is controlled by TIM5, which is configured in master mode to generate a PWM signal with the pre-configured period T_S. When the rising edge of the CONVST signal is detected, after a delay of T_R (which, according to [1], does not exceed 315ns), the READY signal is generated. This signal plays a key role as it triggers TIM4 and TIM8, which are responsible for the CS and SCLK signals, respectively. The rising edge of the READY signal (2) starts TIM4, which pulls the CS signal low to activate all SPI slaves. The low-level duration of the CS signal, denoted as T_T, corresponds to the time needed to transfer all 16 bits of data. The falling edge of the READY (3) signal then triggers TIM8, which generates 16 clock cycles to facilitate data transfer.

It is important to highlight three critical timing parameters:

- T_R: The time interval between the rising edge of the CONVST signal and the rising edge of the READY signal, controlled by the ADS9224R, which is specified in [1] and does not exceed 315ns.
- T_S: The sampling period is determined by the value set by the user through the GUI application.
- T_T: The time interval during which data transfer is active, which must be less than T_S - T_R.

In the current software version, 16 clock cycles are generated during each T_T interval. To ensure high-speed data transfer, the T_T interval remains constant, regardless of the sampling period. This setup allows the transmission of the 16-bit sampled value from the corresponding ADC channel, even when the sampling rate is 1MSps.

To manage data acquisition, three timers generate the CS, SCLK, and CONVST signals for Zone 1, while SPI instances 4 and 5 are used to receive data and store it in local memory. These SPI peripherals are connected to DMA controllers, which offload the CPU by handling data transfer. The DMA retrieves one byte at a time from the SPI and continues until N samples have been received. Unlike the internal ADC setup in the Analog IN software block, where data from both channels is stored sequentially in a single buffer, in case the software utilizes the external ADC, each buffer is divided into two equal parts, with each part storing N samples from one channel. Since the DMA operates in double-buffer mode, when one buffer fills with N samples, the DMA automatically switches the data stream to the second buffer. These buffers, declared as non-cacheable within the AIN software block, store the data. Once both buffer sections are full, an interrupted service routine (ISR) notifies the software to read and transmit the data over Ethernet, while the DMA continues to fill the alternate buffer. This process is illustrated in Figure 4.8.

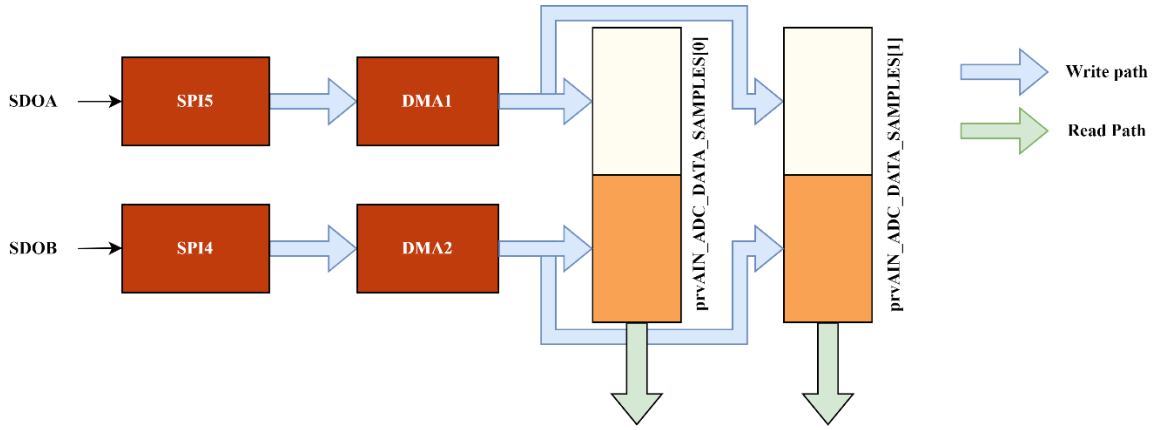

Figure 4.8 - Samples transfer logic

When the "START ACQUISITION" command is received by the MCU from the GUI, the acquisition process using the external ADC begins with a call to the *ADS9224R_StartAcquisition* function. The function first checks whether the ADS9224R driver is already in acquisition mode; if not, it configures

the driver for acquisition by calling *prvADS9224R_ACQ_SetState*. An important modification made to the STM32 HAL library ensures that the DMA operates in double-buffer mode. This modification involves replacing the *HAL_DMA_Start_IT* function with *HAL_DMAEx_MultiBufferStart_IT* inside the *HAL_SPI_Receive_DMA* function. This change is illustrated in Figure 4.9.

```
2218   /* Enable the Rx DMA Stream/Channel  */
2219 //  if (HAL_OK != HAL_DMA_Start_IT(hspi->hdmarx, (uint32_t)&hspi->Instance->RXDR, (uint32_t)hspi->pRxBuffPtr,
2220 //                                 hspi->RxXferCount))
2221 //  {
2222   if(HAL_OK != HAL_DMAEx_MultiBufferStart_IT(hspi->hdmarx, (uint32_t)&hspi->Instance->RXDR, (uint32_t)hspi->pRxBuffPtr, (uint32_t)pData1, Size))
2223   {
2224     /* Update SPI error code */
2225     SET_BIT(hspi->ErrorCode, HAL_SPI_ERROR_DMA);
2226
2227     /* Unlock the process */
2228     __HAL_UNLOCK(hspi);
2229
2230     hspi->State = HAL_SPI_STATE_READY;
2231     errorcode = HAL_ERROR;
2232     return errorcode;
```

Figure 4.9 - Part of the STM32 HAL library that is modified to support DMA double buffer mode

Once it is confirmed that the driver is in acquisition mode, the PWM channels are enabled in the following order: TIM4 (CS), TIM8 (SCLK), and TIM5 (CONVST). When the TIM5 timer is activated, it generates pulses that initiate the conversion process, starting the data transfer cycles on the SPI bus.

4.1.2. Analog OUT

Block Summary			
Name	Analog OUT	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/AnalogOUT/drv_aout.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/AnalogOUT/drv_aout.h`			

This driver oversees configuring DAC value used for current sense part of the circuit. Within the current version, this driver block defines three public functions: *DRV_AOUT_Init*, *DRV_AOUT_SetEnable* and *DRV_AOUT_SetValue*. *DRV_AOUT_Init* is called during system initialization, and this function is in charge to properly initializing all functionalities related to STM32 DAC. *DRV_AOUT_SetEnable* enable or disable STM32 DAC channel while *DRV_AOUT_SetValue* set DAC value.

4.1.3. Network

Block Summary			
Name	Network	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/LwIP/ethernetif.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/LwIP/ethernetif.h`			

The Ethernet Interface (ETHIF) software block provides RTOS compatible network interface driver for the LwIP TCP/IP stack on STM32H7 microcontrollers. This block implements all the required low-level hardware abstraction and DMA-based packet handling mechanisms

The ethernet interface driver is composed of three primary layers:

- Low-level hardware control (HAL_ETH)
- LwIP driver glue logic (*ethernetif_init*, *low_level_input/output*)
- Dedicated RTOS tasks and semaphores for asynchronous event handling

The initialization sequence begins with *ethernetif_init*, which is typically passed to *netif_add* during network stack setup. Internally, this function invokes *low_level_init* to configure the STM32H7 Ethernet peripheral, initialize the PHY driver (LAN8742), allocate Rx/Tx descriptors, and create memory pools for zero-copy RX buffers (*RX_POOL*). The Ethernet MAC is started in interrupt mode, and a dedicated FreeRTOS thread *ethernetif_input* is spawned to process incoming packets signaled by a semaphore *RxPktSemaphore*. Link status and PHY negotiation (duplex and speed) are managed via the LAN8742 driver, which is abstracted using a CMSIS-compliant I/O layer.

Transmission is handled through *low_level_output*, where the *pbuf* chain is traversed and packaged into *ETH_BufferTypeDef* buffers. The data is sent using the STM32 HAL function *HAL_ETH_Transmit_IT*, with blocking behavior implemented via *TxPktSemaphore* in case of DMA descriptor unavailability. Upon completion of transmission, *HAL_ETH_TxCpltCallback* releases the semaphore to unblock pending operations. Similarly, reception flow begins with the ISR-driven *HAL_ETH_RxCpltCallback*, which signals the input thread to retrieve packets via *low_level_input*. Definition of *low_level_input* function is presented on Figure 4.10.

```
void ethernetif_input( void* argument )
{
    struct pbuf *p = NULL;
    struct netif *netif = (struct netif *) argument;

    for( ;; )
    {
        if (osSemaphoreAcquire( RxPktSemaphore, TIME_WAITING_FOR_INPUT)==osOK)
        {
            do
            {
                p = low_level_input( netif );
                if (p != NULL)
                {
                    if (netif->input( p, netif) != ERR_OK )
                    {
                        pbuf_free(p);
                    }
                }

            }while(p != NULL);
        }
    }
}
```

Figure 4.10 - *low_leve_input* function definition

The driver fully supports zero-copy reception. RX pbufs are allocated using the *HAL_ETH_RxAllocateCallback* hook (Figure 4.11), which provides aligned memory from the *RX_POOL*. Received packets are linked using *HAL_ETH_RxLinkCallback*, and coherency is ensured via *SCB_InvalidateDCache_by_Addr*. Freed buffers are reclaimed through *pbuf_free_custom*, ensuring efficient memory reuse without fragmentation. On the transmission side, LwIP-managed pbufs are released using *HAL_ETH_TxFreeCallback*.

```
void HAL_ETH_RxLinkCallback(void **pStart, void **pEnd, uint8_t *buff, uint16_t Length)
{
    struct pbuf **ppStart = (struct pbuf **)pStart;
    struct pbuf **ppEnd = (struct pbuf **)pEnd;
    struct pbuf *p = NULL;

    /* Get the struct pbuf from the buff address. */
    p = (struct pbuf *)(buff - offsetof(RxBuff_t, buff));
    p->next = NULL;
    p->tot_len = 0;
    p->len = Length;

    /* Chain the buffer. */
    if (!*ppStart)
    {
        /* The first buffer of the packet. */
        *ppStart = p;
    }
    else
    {
        /* Chain the buffer to the end of the packet. */
        (*ppEnd)->next = p;
    }

    *ppEnd  = p;

    /* Update the total length of all the buffers of the chain. Each pbuf in the chain should have its tot_len
    * set to its own length, plus the length of all the following pbufs in the chain. */
    for (p = *ppStart; p != NULL; p = p->next)
    {
        p->tot_len += Length;
    }

    /* Invalidate data cache because Rx DMA's writing to physical memory makes it stale. */
    SCB_InvalidateDCache_by_Addr((uint32_t *)buff, Length);
}
```

Figure 4.11 – HAL_ETH_RxLinkCallback function definition

4.1.4. GPIO

Block Summary			
Name	GPIO	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/GPIO/drv_gpio.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/GPIO/drv_gpio.h`			

The General-Purpose Input/Output (GPIO) software block provides a thread-safe abstraction layer for configuring and controlling digital input and output pins across the STM32H7 platform. This software block manages initialization, runtime state manipulation, and interrupt servicing of GPIO pins. The block is built with FreeRTOS compatibility in mind and leverages synchronization mechanisms to ensure safe concurrent access in real-time systems.

All GPIO port-level operations are safeguarded using dedicated mutex semaphores. During the initialization of a GPIO port (invoked via *DRV_GPIO_Port_Init*), a mutex is created and associated with the corresponding port handle. This lock is used to ensure atomic access when configuring or manipulating pins associated with that port. Public APIs such as *DRV_GPIO_Pin_Init* and *DRV_GPIO_Pin_SetState*, which definition is presented on Figure 4.12, acquire this lock before performing HAL-level operations on the port registers. Similarly, pin deinitialization and state reading are protected to prevent data races and ensure consistent behavior in multi-threaded execution environments. For ISR contexts, specialized functions *DRV_GPIO_Pin_SetStateFromISR*, *DRV_GPIO_Pin_ToogleFromISR* are provided to avoid context-switch violations while maintaining thread safety.

```
drv_gpio_status_t DRV_GPIO_Pin_SetState(drv_gpio_port_t port, drv_gpio_pin pin, drv_gpio_pin_state_t state)
{
    if(prvDRV_GPIO_PORTS[port].initState != DRV_GPIO_PORT_INIT_STATUS_INITIALIZED || prvDRV_GPIO_PORTS[port].lock == NULL) return DRV_GPIO_STATUS_ERROR;
    if(pin > DRV_GPIO_PIN_MAX_NUMBER) return DRV_GPIO_STATUS_ERROR;

    if(xSemaphoreTake(prvDRV_GPIO_PORTS[port].lock, portMAX_DELAY) == pdFALSE ) return DRV_GPIO_STATUS_ERROR;

    HAL_GPIO_WritePin((GPIO_TypeDef*)prvDRV_GPIO_PORTS[port].portInstance, 1 << pin, state);

    if(xSemaphoreGive(prvDRV_GPIO_PORTS[port].lock) == pdFALSE ) return DRV_GPIO_STATUS_ERROR;

    return DRV_GPIO_STATUS_OK;
}
```

Figure 4.12 – *DRV_GPIO_Pin_SetState* function definition

Interrupt support is implemented through a shared registry of callback functions, where each pin with interrupting capability is associated with a user-defined handler function. The *DRV_GPIO_RegisterCallback*, which definition is presented on **Figure 4.13**, and *DRV_GPIO_Pin_EnableInt* functions configure the pin in interrupt mode and register the corresponding ISR callback. At runtime, the global EXTI interrupt handlers delegate control to the user-defined callback based on the pin number, enabling fast and modular response to GPIO events. All pin interrupt callbacks are stored in a statically allocated array *prvDRV_GPIO_PINS_INTERRUPTS*, while the corresponding port state is tracked in *prvDRV_GPIO_PORTS*.

```c
drv_gpio_status_t DRV_GPIO_RegisterCallback(drv_gpio_port_t port, drv_gpio_pin pin, drv_gpio_pin_isr_callback callback, uint32_t priority)
{
    // Ensure the port is initialized
    if(prvDRV_GPIO_PORTS[port].initState != DRV_GPIO_PORT_INIT_STATUS_INITIALIZED || prvDRV_GPIO_PORTS[port].lock == NULL)
        return DRV_GPIO_STATUS_ERROR;

    // Ensure the pin number is valid
    if(pin > DRV_GPIO_PIN_MAX_NUMBER || pin > DRV_GPIO_INTERRUPTS_MAX_NUMBER)
        return DRV_GPIO_STATUS_ERROR;

    // Store the callback
    prvDRV_GPIO_PINS_INTERRUPTS[pin] = callback;

    // Enable the interrupt for the specified pin
    drv_gpio_status_t status = DRV_GPIO_Pin_EnableInt(port, pin, priority, callback);

    return status;
}
```

Figure 4.13 - *DRV_GPIO_RegisterCallback* function's definition

4.1.5. Interrupts

BLOCK SUMMARY			
Name	Interrupts	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/Interrupts/stm32h7xx_it.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/Interrupts/stm32h7xx_it.h`			

The Interrupt Service Routine (ISR) software block is responsible for handling all processor exceptions and peripheral interruptions generated on the STM32H7 platform. This module acts as the centralized dispatch layer for routing hardware interrupts to their respective handlers, ensuring deterministic and responsive behavior of time-critical software components.

A significant portion of the ISR logic is dedicated to external interrupt (EXTI) line handling, especially for GPIO events. These handlers, such as *EXTI15_10_IRQHandler* and *EXTI9_5_IRQHandler*, check which EXTI line has triggered the interrupt using *__HAL_GPIO_EXTI_GET_IT* and clear the interrupt flag after servicing via *__HAL_GPIO_EXTI_CLEAR_IT*. Each triggered line then invokes the corresponding HAL-level callback via *HAL_GPIO_EXTI_IRQHandler*, which is further routed to user-defined callback functions registered in the GPIO driver layer. This layered approach ensures clear separation of responsibilities, where the ISR module provides the hardware-level dispatch mechanism while higher-level services (e.g., drv_gpio) handle the application-specific logic.

```
/**
  * @brief This function handles EXTI line[15:10] interrupts.
  */
void EXTI15_10_IRQHandler(void)
{
  if (__HAL_GPIO_EXTI_GET_IT(0x1 << EXTI_LINE10))
  {
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_10);
    __HAL_GPIO_EXTI_CLEAR_IT(0x1 << EXTI_LINE10);
  }
  if (__HAL_GPIO_EXTI_GET_IT(0x1 << EXTI_LINE11))
  {
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_11);
    __HAL_GPIO_EXTI_CLEAR_IT(0x1 << EXTI_LINE11);
  }
  if (__HAL_GPIO_EXTI_GET_IT(0x1 << EXTI_LINE12))
  {
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
    __HAL_GPIO_EXTI_CLEAR_IT(0x1 << EXTI_LINE12);
  }
  if (__HAL_GPIO_EXTI_GET_IT(0x1 << EXTI_LINE13))
  {
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
    __HAL_GPIO_EXTI_CLEAR_IT(0x1 << EXTI_LINE13);
  }
  if (__HAL_GPIO_EXTI_GET_IT(0x1 << EXTI_LINE14))
  {
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_14);
    __HAL_GPIO_EXTI_CLEAR_IT(0x1 << EXTI_LINE14);
  }
  if (__HAL_GPIO_EXTI_GET_IT(0x1 << EXTI_LINE15))
  {
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_15);
    __HAL_GPIO_EXTI_CLEAR_IT(0x1 << EXTI_LINE15);
  }
}
```

Figure 4.14 - EXTI Service Routine

In addition to GPIO interrupts, this ISR block manages interrupts from Ethernet (ETH), DAC and Timer peripherals (TIM6_DAC), and others. For instance, *TIM6_DAC_IRQHandler* simultaneously checks

the state of the DAC and triggers both DAC and Timer-related handlers. This combined handling is essential in cases where peripherals share interruption lines or when coordinated processing is required (e.g., DAC output under Timer control). The implementation is modular and allows HAL drivers to manage their internal state, while application logic is typically offloaded to registered callbacks or deferred via task notifications if operating within a FreeRTOS-based system.

For fault-related exceptions (e.g., *HardFault_Handler*, *BusFault_Handler*), the current implementation enters infinite loops, serving as breakpoints for debugging unrecoverable errors. These can later be extended to include logging or safe system reset procedures depending on system-level fault tolerance strategies. Overall, the ISR block provides a robust and scalable foundation for real-time event processing and is tightly integrated with the platform's peripheral driver and HAL layers.

4.1.6. SPI

Block Summary			
Name	SPI	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/SPI/drv_spi.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/SPI/drv_spi.h`			

The SPI Interface (DRV_SPI) software block provides a modular, FreeRTOS-compatible abstraction layer for configuring and operating SPI peripherals. The SPI software block is structured around a centralized handle management system (*drv_spi_handle_t*) that stores initialization state, user configuration, FreeRTOS lock handle, and an embedded HAL SPI handler *SPI_HandleTypeDef*. A fixed-size array, *prvDRV_SPI_INSTANCES*, holds these handles for all SPI instances, enabling the driver to manage multiple peripherals concurrently in a thread-safe manner. Each SPI instance is guarded by a binary mutex, ensuring thread-safe operation for concurrent access to transmission and reception routines in a multitasking environment.

```
typedef struct drv_spi_handle_t
{
    drv_spi_instance_t                  instance;
    drv_spi_initialization_status_t     initState;
    drv_spi_config_t                    config;
    SemaphoreHandle_t                   lock;
    SPI_HandleTypeDef                   deviceHandler;
}drv_spi_handle_t;
```

Figure 4.15 - *drv_spi_handle_t* structure definition

Initialization is performed using *DRV_SPI_Instance_Init*, where users configure mode (master/slave), clock polarity, and clock phase. Default values are applied for advanced parameters such as CRC, NSS behavior, and FIFO settings, but these can be extended for future configurability. Deinitialization is handled via *DRV_SPI_Instance_DeInit* to allow dynamic reconfiguration or power management.

Data communication is supported through both blocking *DRV_SPI_TransmitData*, *DRV_SPI_ReceiveData* and non-blocking interrupt/DMA-based APIs. The *DRV_SPI_EnableITData* function triggers full-duplex DMA transfers, while interrupt-based completion is serviced by *HAL_SPI_TxRxCpltCallback* and routed to user-registered callbacks via *DRV_SPI_Instance_RegisterRxCallback*. These mechanisms are especially useful in latency-sensitive scenarios where SPI peripherals require low-overhead, high-throughput communication.

The SPI block also includes full integration with the HAL MSP layer. GPIO and DMA configuration for each SPI peripheral (SPI2, SPI3, SPI4, SPI5) is handled in *HAL_SPI_MspInit* and *HAL_SPI_MspDeInit* functions. Each peripheral is conditionally configured with specific GPIO ports, DMA streams, and clock sources. For instance, SPI2 uses *DMA1_Stream4/5* and operates in circular DMA mode, suitable for continuous streaming scenarios.

4.1.7. System

Block Summary			
Name	System	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/System/drv_system.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/System/drv_system.h`			

The System Initialization (DRV_SYSTEM) software block is responsible for performing all essential platform-level initialization tasks prior to enabling specific services. This block ensures a well-defined and safe startup sequence for STM32H7-based dual-core (Cortex-M7 + Cortex-M4) embedded systems. It provides two key entry points: *DRV_SYSTEM_InitCoreFunc* for early core/system configuration and *DRV_SYSTEM_InitDrivers* for initializing platform-wide peripheral drivers. Definition of these two functions is presented on Figure 4.16.

```
drv_system_status_t DRV_SYSTEM_InitCoreFunc()
{
    prvDRV_SYSTEM_CACHE_Enable();
    if(prvDRV_SYSTEM_CPU2_Wait(0xFFFF) != DRV_SYSTEM_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;
    if(HAL_Init() != HAL_OK) return DRV_SYSTEM_STATUS_ERROR;
    if(prvDRV_SYSTEM_MPU_Init() != DRV_SYSTEM_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;
    if(prvDRV_SYSTEM_CLOCK_Init() != DRV_SYSTEM_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;

    __HAL_RCC_HSEM_CLK_ENABLE();
    /*Take HSEM */
    HAL_HSEM_FastTake(HSEM_ID_0);
    /*Release HSEM in order to notify the CPU2(CM4)*/
    HAL_HSEM_Release(HSEM_ID_0,0);


    if(prvDRV_SYSTEM_CPU2_Wait(0xFFFF) != DRV_SYSTEM_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;


    return DRV_SYSTEM_STATUS_OK;
}
drv_system_status_t DRV_SYSTEM_InitDrivers()
{
    if(DRV_GPIO_Init() != DRV_GPIO_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;
    if(DRV_UART_Init() != DRV_UART_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;
    if(DRV_SPI_Init() != DRV_SPI_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;
    if(DRV_AIN_Init(DRV_AIN_ADC_3, NULL) != DRV_AIN_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;

    if(DRV_Timer_Init() != DRV_TIMER_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;
    if(DRV_AOUT_Init() != DRV_AOUT_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;
    if(DRV_I2C_Init() != DRV_AOUT_STATUS_OK) return DRV_SYSTEM_STATUS_ERROR;


    return DRV_SYSTEM_STATUS_OK;
}
```

Figure 4.16 - System initialization functions

The core initialization path *DRV_SYSTEM_InitCoreFunc* begins by enabling instruction and data caches *SCB_EnableICache* and *SCB_EnableDCache* to improve execution performance. In dual-core configurations, synchronization with CPU2 (typically the Cortex-M4 core) is handled through hardware semaphores and RCC flags. Specifically, *prvDRV_SYSTEM_CPU2_Wait* waits for the M4 core to reach stop mode before proceeding, ensuring coordination between both processing units. A shared semaphore (HSEM_ID_0) is taken and released to signal startup readiness between the cores.

Next, the Memory Protection Unit (MPU) is configured using *prvDRV_SYSTEM_MPU_Init* to define memory attributes for key regions such as LwIP buffers, DMA descriptors, and reserved regions. Each MPU region is carefully assigned access permissions, caching behavior, and execution rights,

enforcing memory safety and optimizing performance for DMA operations (e.g., ETH RX descriptors are marked as non-cacheable, bufferable, and shareable). This MPU configuration is crucial for maintaining data coherency in systems using cache and peripheral DMA access.

```
/** Initializes Memory Region that belongs to AnalogIN buffers
*/
MPU_InitStruct.Enable           = MPU_REGION_ENABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER3;
MPU_InitStruct.BaseAddress      = 0x38000000;
MPU_InitStruct.Size             = MPU_REGION_SIZE_8KB;
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

Figure 4.17 - Initialization of Memory region that belongs to AnalogIN buffer.

The system clock configuration is then performed via *prvDRV_SYSTEM_CLOCK_Init*, which uses the HSI oscillator as the PLL source to derive all main system clocks (SYSCLK, HCLK, APBx) with appropriate dividers. This ensures that peripherals operate at well-defined and validated frequencies. The configuration targets high performance with voltage scaling set to *PWR_REGULATOR_VOLTAGE_SCALE1* and PLL settings customized for the specific STM32H7 system design.

Once the core system is fully initialized, *DRV_SYSTEM_InitDrivers* is called to bring up all platform-level hardware abstraction layers. This includes GPIO, UART, SPI, analog input, analog output (DAC), timers, and I2C. Each driver is initialized in sequence, and any failure in peripheral initialization leads to a controlled error response. This centralization of driver initialization ensures deterministic system behavior and simplifies diagnostics during bring-up and deployment.

4.1.8. Timer

Block Summary			
Name	Timer	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/Timer/drv_timer.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/Timer/drv_timer.h`			

The DRV_TIMER software block serves as a modular, and thread-safe abstraction layer for managing the STM32H7's timers. It is designed to support a wide range of use cases, including periodic task scheduling, PWM signal generation, and precise time-based control, all within the context of FreeRTOS-based embedded systems.

This driver supports multiple hardware timer instances (e.g., TIM1–TIM4), each with up to four independent channels. The configuration for each timer is defined using the *drv_timer_config_t* structure, which includes the prescaler value, counter mode (up/down), clock divider, auto-reload behavior, and other key parameters. Upon initialization, the selected timer is configured in base and PWM modes using STM32 HAL functions, and a FreeRTOS mutex is created to guard all interactions with that specific timer instance. This ensures thread-safe operations even when multiple tasks attempt concurrent access.

The initialization sequence starts with *DRV_Timer_Init*, which resets all driver-internal state and marks the timer module as ready. Each timer instance is then individually configured using *DRV_Timer_Init_Instance*. This function handles the allocation of hardware resources, such as enabling peripheral clocks and configuring GPIOs via *HAL_TIM_Base_MspInit*. The timer is then brought up in PWM-ready mode using *HAL_TIM_Base_Init*, *HAL_TIM_PWM_Init*, and clock configuration routines.

Each channel within a timer can be independently configured using *DRV_Timer_Channel_Init*. As of the current implementation, only PWM1 mode is supported, which allows for duty-cycle-based waveform generation. Configuration is performed through *HAL_TIM_PWM_ConfigChannel*, and upon successful setup, the channel's metadata is stored in the driver's internal handle structure.

To start signal generation, the application calls *DRV_Timer_Channel_PWM_Start*, specifying the timer, channel, target pulse width (period), and a timeout value. Internally, the driver acquires the mutex associated with the timer instance, validates that both the timer and channel are initialized, and then updates the corresponding CCR register to set the PWM duty cycle. The channel then started using *HAL_TIM_PWM_Start*, and the mutex is released.

All critical operations in this driver are protected using FreeRTOS binary semaphores, making this module inherently safe for use in preemptive multitasking environments. Mutexes ensure that no race conditions occur when starting or modifying timer output, even if multiple tasks are involved. The per-instance locking model also ensures that unrelated timers do not interfere with one another.

Internally, the driver maintains two key static arrays:

- *prvDRV_TIMER_PLATFORM_HANDLER* contains HAL-level *TIM_HandleTypeDef* structures.
- *prvDRV_TIMER_HANDLER* maintains driver-specific metadata, including channel configuration and state flags.

This static allocation model ensures deterministic memory usage, a critical requirement in real-time embedded systems where heap allocation is often restricted or discouraged.

4.1.9. UART

Block Summary			
Name	UART	**Layer**	Driver
Version	1.03		
Related files			
Driver top source file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/UART/drv_uart.c`			
Driver top header file			
`Source/ADFirmware/CM7/Core/Drivers/Platform/UART/drv_uart.h`			

The DRV_UART software block is a thread-safe abstraction layer designed to manage UART peripheral instances on STM32H7-based platform. It provides configuration, data transmission, and receive-callback registration services over multiple UART ports. The implementation is tailored for use in FreeRTOS environments and supports concurrent access using mutexes for each UART instance.

The driver supports up to *CONF_UART_INSTANCES_MAX_NUMBER* UART interfaces, which are statically allocated in the internal *prvDRV_UART_INSTANCES* array. Each UART instance is associated with a dedicated *UART_HandleTypeDef* structure for HAL interactions, as well as a FreeRTOS semaphore to protect shared access to the interface. The *DRV_UART_Init* function initializes internal structures, clearing any previous state, and marks the driver ready for instance-level configuration.

Individual UARTs are initialized using *DRV_UART_Instance_Init*. This function sets up the selected UART peripheral based on a user-supplied configuration structure *drv_uart_config_t*. Although currently only the baud rate is configurable via this structure, the initialization routine sets the word length, parity, stop bits, hardware flow control, FIFO thresholds, and disables FIFO mode using standard STM32 HAL functions. Once initialized, a dedicated mutex (lock) is created per instance, enabling thread-safe transmission.

Data transmission is performed through the *DRV_UART_TransferData* function. This function first acquires the instance's mutex before calling *HAL_UART_Transmit* to send the data buffer. If the mutex acquisition or HAL transmission fails, the function returns an error. The use of per-instance semaphores ensures that only one task at a time may access a UART resource, thus avoiding race conditions in concurrent environments.

Reception is supported via interrupt mode and a lightweight callback mechanism. The *DRV_UART_Instance_RegisterRxCallback* function allows the user to register a type *drv_uart_rx_isr_callback*, which is invoked each time a new byte is received. Upon registering, the function also starts with the reception interrupt using *HAL_UART_Receive_IT*. When data arrives, the *HAL_UART_RxCpltCallback* ISR invokes the appropriate callback and immediately restarts the reception for the next byte. This design supports byte-wise interrupt-driven reception and is well-suited for protocols that require immediate handling of control or framing bytes.

The driver also implements peripheral clock configuration and GPIO alternate function mapping in the *HAL_UART_MspInit* routine. This low-level function ensures that each UART instance is properly clocked and that TX/RX pins are initialized with the correct GPIO settings, using STM32Cube HAL APIs. Interrupts are configured and prioritized to support asynchronous receive operation.

The DRV_UART module is designed with modularity and safety in mind. By encapsulating all hardware and synchronization resources per instance, the driver supports multiple concurrent UART sessions with deterministic behavior and minimal resource contention. This makes it ideal for embedded applications requiring robust serial communication, such as CLI interfaces, sensor networks, debug logging,

and inter-MCU data exchange. Future enhancements may include DMA support, more advanced parity/stop bit configuration, and framing-layer abstraction.

4.2. Services

4.2.1. Control

Block Summary			
Name	Control	**Layer**	Middleware
Version	1.03		
Related files			
Top source file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/Control/control.c`			
Top header file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/Control/control.h`			
Command parser source file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/Control/CMParse/cmparse.c`			
Command parser header file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/Control/CMParse/cmparse.h`			

Within the device firmware, the logic for control and status messages is encapsulated within a distinct service. This service is responsible for receiving control messages, parsing their content, executing the appropriate actions based on the message content, and generating responses accordingly. The core logic and operational mechanisms of this service are depicted in the Figure 4.18.

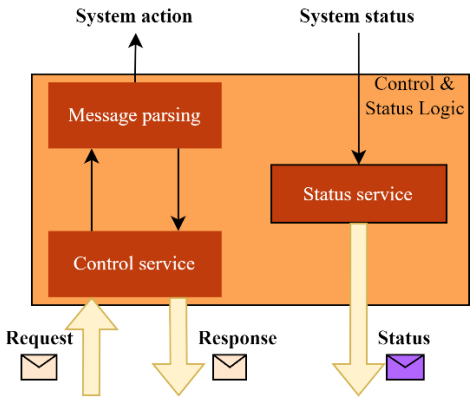

Figure 4.18 - Control service working principle

Control messages are sent to the Acquisition device in the form of a control message request. These requests are sent from the HOST side on the previously opened TCP port on the acquisition device. *A control message request* is received inside the ***Control service*** server task, and it is forwarded to ***Message parsing*** logic to analyse control message content. Message parsing logic is part of Control service logic in charge of analyzing the content of received control messages and calling corresponding callback functions previously assigned to specific control messages. Part of the Control service code that creates a mapping between callback functions and control messages is presented in Figure 4.19. When callback functions that correspond to request control messages are executed, corresponding control message responses are generated based on callback function execution results. This response is sent back to the HOST machine.

```
CMPARSE_AddCommand("",                              prvCONTROL_UndefinedCommand);
CMPARSE_AddCommand("device hello",                  prvCONTROL_GetDeviceName);
CMPARSE_AddCommand("device setname",                prvCONTROL_SetDeviceName);
CMPARSE_AddCommand("device slink create",           prvCONTROL_CreateStatusLink);
CMPARSE_AddCommand("device slink send",             prvCONTROL_StatusLinkSendMessage);
CMPARSE_AddCommand("device eplink create",          prvCONTROL_EPLinkCreate);
CMPARSE_AddCommand("device stream create",          prvCONTROL_StreamCreate);
CMPARSE_AddCommand("device stream start",           prvCONTROL_StreamStart);
CMPARSE_AddCommand("device stream stop",            prvCONTROL_StreamStop);

CMPARSE_AddCommand("device adc chresolution set",   prvCONTROL_SetResolution);
CMPARSE_AddCommand("device adc chresolution get",   prvCONTROL_GetResolution);
CMPARSE_AddCommand("device adc chclkdiv set",       prvCONTROL_SetClkdiv);
CMPARSE_AddCommand("device adc chclkdiv get",       prvCONTROL_GetClkdiv);
CMPARSE_AddCommand("device adc chstime set",        prvCONTROL_SetChSamplingtime);
CMPARSE_AddCommand("device adc chstime get",        prvCONTROL_GetChSamplingtime);
CMPARSE_AddCommand("device adc chavrratio set",     prvCONTROL_SetAveragingratio);
CMPARSE_AddCommand("device adc chavrratio get",     prvCONTROL_GetAveragingratio);
CMPARSE_AddCommand("device adc speriod set",        prvCONTROL_SetSamplingtime);
CMPARSE_AddCommand("device adc speriod get",        prvCONTROL_GetSamplingtime);
CMPARSE_AddCommand("device adc voffset set",        prvCONTROL_SetVoltageoffset);
CMPARSE_AddCommand("device adc voffset get",        prvCONTROL_GetVoltageoffset);
CMPARSE_AddCommand("device adc coffset set",        prvCONTROL_SetCurrentoffset);
CMPARSE_AddCommand("device adc coffset get",        prvCONTROL_GetCurrentoffset);
CMPARSE_AddCommand("device adc clk get",            prvCONTROL_GetADCInputClk);
CMPARSE_AddCommand("device adc value get",          prvCONTROL_GetADCValue);
CMPARSE_AddCommand("device adc samplesno set",      prvCONTROL_SetSamplesNo);
```

Figure 4.19 - Part of the supported commands list with corresponding callback functions

All control messages are in ASCII format with predefined structures. Control message structures, as well as commands implemented within the current software version, are presented in [2]

Besides control messages, there are also status messages that are used to inform the host machine about the system execution status. To enable this service, the device-side control message for configuring the status messages server should be utilized.

4.2.2. Stream

Block Summary			
Name	Stream	**Layer**	Middleware
Version	1.03		
Related files			
Top source file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/SamplesStream/sstream.c`			
Top header file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/SamplesStream/sstream.h`			

Figure 4.20 illustrates all firmware software blocks that are parts of streaming functionalities.

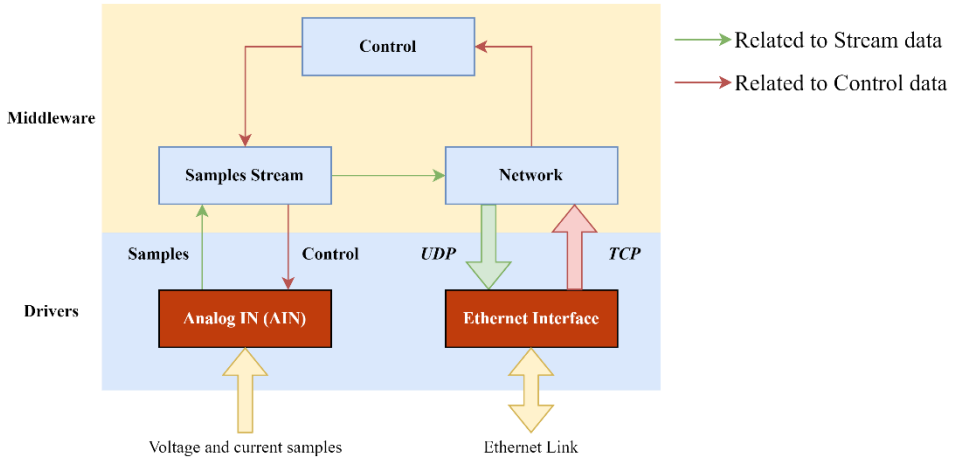

Figure 4.20 – Streaming service's functional software blocks

Presented blocks are:

- **Analog IN**
  In charge of acquiring voltage and current samples and storing them in corresponding buffers
- **Ethernet Interface and Network service**
  Software IP stack
- Samples Stream Service
  In charge of receiving sample data packets from lower software layers, processing them, and forwarding them to the host side
- **Control Service**
  In charge of parsing control commands.

After MCU firmware is successfully run and the ethernet link is established, the host initiates the data streaming process by transmitting a control message. This message, as defined within the [2] file, contains the server's IP address where obtained samples should be sent. This established streaming channel facilitates the continuous transmission of data from the MCU to the server for real-time processing or further analysis. For clarity, the "create stream" command format with some IP address information is provided below:

```
device stream create -ip=192.168.1.200 -port=5000
```

Upon receiving the control message, the Control Service parses it and forwards the "Create Stream" action to the **Samples Stream Service**. This service, based on the message content, creates a stream for sending data to the server at 192.168.1.200, port 5000.

Stream is created by calling *SSTREAM_CreateChannel* which is key part of the Sample Stream (SSTREAM) service. New streaming channel is established based on the provided connection information.

It ensures that each new channel is properly initialized, both in terms of synchronization primitives and task management, before the system begins using it for data streaming. The function is designed to be thread-safe and supports multiple concurrent stream connections, with a maximum number defined in the system configuration.

The process begins by checking if the number of currently active connections has reached the configured upper limit (*SSTREAM_CONNECTIONS_MAX_NO*). If so, it returns an error to avoid resource over-allocation. Assuming there is room for another channel, the function assigns a new connection ID based on the current active count. It then updates internal bookkeeping structures for control and stream handling with this ID, and copies over the connection parameters (such as IP address and port) from the provided *connectionHandler*.

To support synchronization between the main system and the newly created tasks, the function initializes binary semaphores (initSig) and mutexes (guard) for both the control and stream components of the channel. These semaphores serve as signals indicating successful task startup, while the mutexes ensure safe access to shared resources in a multithreaded environment.

Once synchronization objects are successfully created, the function proceeds to launch two FreeRTOS tasks: the control task and the stream task. These tasks are responsible for handling the control interface and the actual data streaming respectively. After each task is created, the function waits on its corresponding semaphore with a timeout to confirm that the task has initialized correctly. If any of the task creation or synchronization steps fail, the function exits with an error status.

If all steps complete successfully, the new channel's connection ID is finalized, and the system's active connection counter is incremented. The function then returns SSTREAM_STATUS_OK, signalling that the new streaming channel is fully initialized and ready for use.

Following stream creation, the host can send commands for configuring parameters like sampling rate and resolution. The current software version supports the configuration of the Sampling period and number of samples within single stream packet. The [2] provides a comprehensive list of commands used to configure specific parameter values.

After the data stream is established and configured, the host may trigger data transmission by sending a "Start Stream" command. Upon receiving this command, the **Stream Service** leverages the Stream ID (SID) value to identify the relevant stream. The designated stream then begins continuous sample acquisition and transmits the data to the pre-configured server. This ongoing data transmission process can be dynamically controlled by the host through specific pause or stop commands.

When streaming is active, the corresponding stream task is triggered, by Analog IN driver, for further processing when current and voltage samples are stored inside a corresponding buffer within Analog IN driver. Processing logic implemented inside this task includes two steps:

1. Incrementing the Message ID (**MID**) sequentially
2. Appending a Message Type (**MTI**)

The sequential MID allows the host system to detect potential packet loss during transmission because unreliable, very fast, UDP protocol is used for packet transmission. The MTI serves to identify the specific data type contained within the packet. While the current version handles only one data type, this design offers flexibility for future implementations that may include different data streaming types. Data samples, together with MID and MTI, create a Stream message packet which is forwarded to the network service where it is encapsulated inside the corresponding ethernet packet and sent to the host side. Stream message packet creation over different software services is illustrated in **Figure 4.21**.

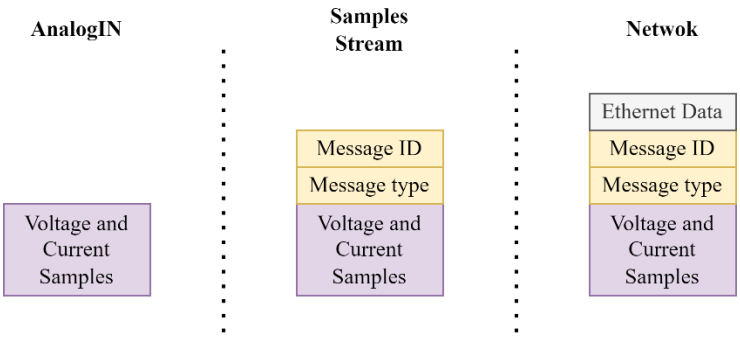

Figure 4.21 - Stream message content among firmware's services

4.2.3. Energy Debugging

Block Summary			
Name	Energy Debugging	**Layer**	Middleware
Version	1.03		
Related files			
Top source file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/ED/energy_debugger.c`			
Top header file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/ED/energy_debugger.h`			

Inside OpenEPT Device's firmware is implemented energy debugging service which primary functionality is to receive and process energy sampling request by recording *PacketID* and *SampleID* when energy request is issued. Figure 4.22 illustrates software architecture of this service. To fully understand algorithms implemented inside this service, it is important to understand samples streaming and network functionalities first.

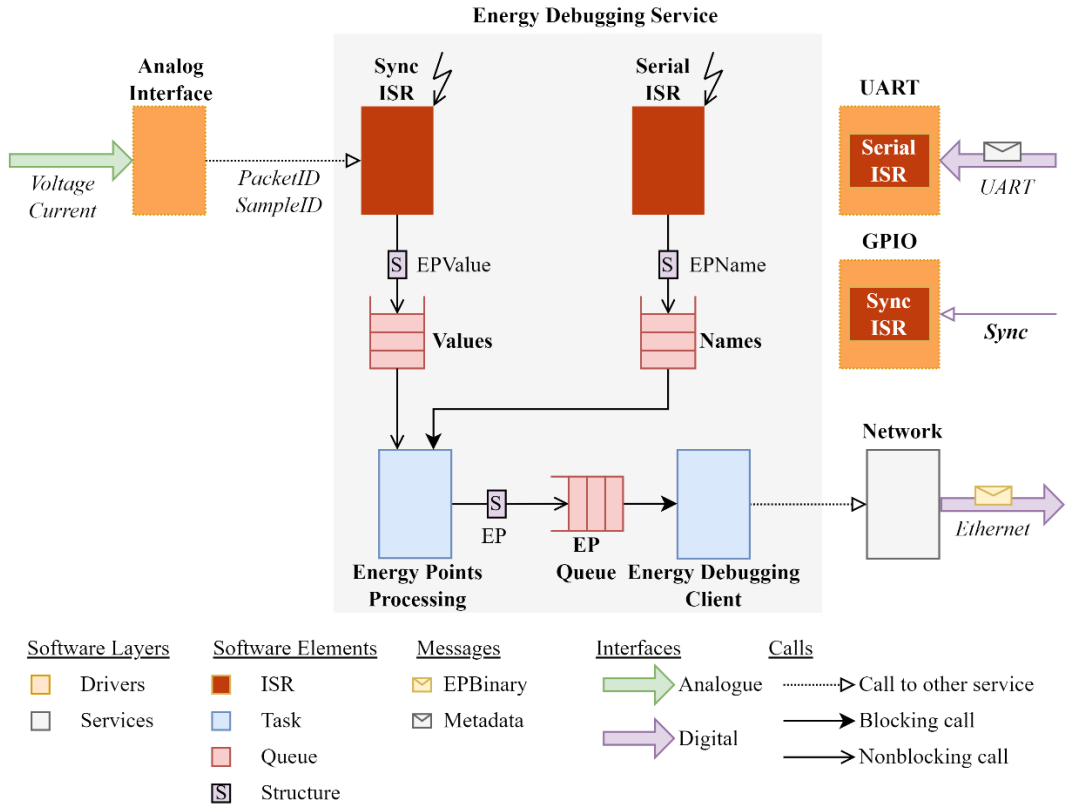

Figure 4.22 - Software Architecture Overview of Energy Debugging Service within OpenEPT Device's firmware

When a rising or falling edge of the Sync signal is detected, the corresponding Sync Interrupt Service Routine (ISR) is triggered. As the first step within this ISR, a request is made to the Analog Interface (AIN) software block to retrieve two values: *PacketID* and *SampleID*. *PacketID* uniquely identifies a streaming packet, as ADC samples are transmitted in chunks of N samples. *SampleID* represents the specific sample within that chunk. The *PacketID* matches the ID of the packet that will be sent over the UDP interface, while the *SampleID* is obtained from the DMA counter. Once these values are retrieved, they are stored in the corresponding Value structure, which is then added to the Values queue.

The Serial ISR is responsible for receiving metadata messages. It gathers characters received over the serial interface, configured for use within the Energy Debugging Interface (currently implemented as UART in the OpenEPT firmware), and stores them in a corresponding buffer. When a \r (carriage return)

character is detected, the buffer contents are transferred to a corresponding Name structure and then added to the Names queue.

The Energy Points Processing (EPP) task integrates the core functionalities of the Energy Debugging mechanism. This task reads from two queues, with only the Names queue being a blocking read. When the Serial ISR writes a Name structure instance to the Names queue, it unblocks the EPP task, which then analyses the received metadata messages to determine their type. Part of this task content, responsible for processing these queues, is illustrated on Figure 4.23.

```
if(xQueueReceive(prvENERGY_DEBUGGER_QUEUE_EBP_NAME, &ebpName, portMAX_DELAY) != pdTRUE)
{
    LOGGING_Write("Energy point service",LOGGING_MSG_TYPE_ERROR,  "Unable to get EP info from callback\r\n");
    prvENERGY_DEBUGGER_DATA.mainTaskState = ENERGY_DEBUGGER_STATE_ERROR;
    break;
}

//Get id
if(xQueueReceive(prvENERGY_DEBUGGER_QUEUE_ID, &id, 0) != pdTRUE)
{
    LOGGING_Write("Energy point service",LOGGING_MSG_TYPE_WARNNING,  "EP ID mismatch\r\n");
    continue;
}


LOGGING_Write("Energy point service",LOGGING_MSG_TYPE_INFO,  "Energy point successfully received\r\n");
ebp.id = id;
memcpy(&ebp.name, &ebpName, sizeof(energy_debugger_ebp_name_t));
LOGGING_Write("Energy point service",LOGGING_MSG_TYPE_INFO,  "Energy point name: %s\r\n", ebpName.name);
LOGGING_Write("Energy point service",LOGGING_MSG_TYPE_INFO,  "Energy point ID: %d\r\n", id);

//send info
for(int i = 0; i < prvENERGY_DEBUGGER_DATA.activeConnectionsNo; i++)
{
    if(xQueueSendToBack(prvENERGY_DEBUGGER_DATA.activeLink[i].ebp, &ebp, 0) != pdTRUE)
    {
        LOGGING_Write("Energy point service",LOGGING_MSG_TYPE_ERROR,  "Unable to send EP info\r\n");
        prvENERGY_DEBUGGER_DATA.mainTaskState = ENERGY_DEBUGGER_STATE_ERROR;
        break;
    }
}

//clear buffer
memset(&ebp, 0, sizeof(energy_debugger_breakpoint_info_t));
memset(&ebpName, 0, sizeof(energy_debugger_ebp_name_t));

break;
```

Figure 4.23 - Part of EPP task responsible for processing Value structure

Three processing scenarios are possible:

- If the message is a START or STOP command, the system initializes communication by setting up all relevant buffers and generate response that will be sent back to the DUT
- If the message is of type Energy Point Name, the task performs a non-blocking read from the Values queue. If the Values queue is empty, an error is generated; otherwise, the retrieved PacketID, SampleID, and Sample Name are combined to create a corresponding Energy Point. The generated Energy Point message is then written to the Energy Point (EP) queue.
- If the message is logging message, received message content is forwarded to the OpenEPT logging interface

Energy Debugging Client (EDC) Task perform blocking read from EP queue to obtain EP messages after which it extracts content from EP messages and transform it to binary format message named EPBinary. This message will be sent over TCP to OpenEPT GUI application, and its format is illustrated on Figure 4.24.

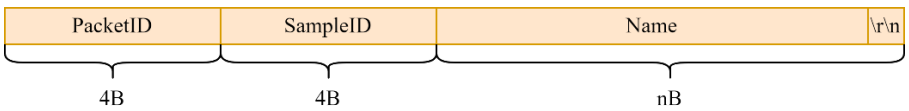

Figure 4.24 - EP binary message structure

This EPBinary message format consists of following fields:

- **PacketID** (4B wide) – represents packetID to which EP message is related to
- **SampleID** (4B wide) – id of sample inside the packet to which EP message is related to
- **Name** (n Bytes wide end with \r\n characters) – name of the EP

4.2.4. EEZ DIB

Block Summary			
Name	EEZ DIB	**Layer**	Middleware
Version	1.03		
Related files			
Top source file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/EezDib/eez_dib.c`			
Top header file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/EezDib/eez_dib.h`			

Main EEZ DIB Service task, with relevant synchronization elements with other system services, is illustrated on Figure 4.25.

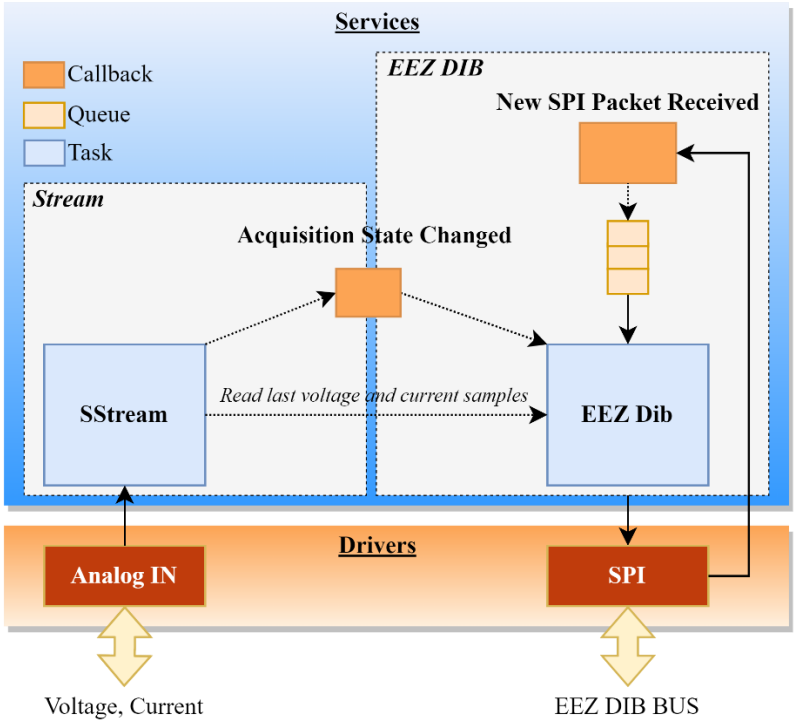

Figure 4.25 - EEZ DIB Service's task and relevant synchronization elements

The SStream service is responsible for acquiring voltage and current samples from the AnalogIN driver, processing them, and transmitting the data to the OpenEPT GUI application running on the host side. This service supports two acquisition modes: Active and Inactive. The current acquisition state of the SStream service is critical for the operation of the EEZ DIB service. To facilitate this interaction, the SStream service has been extended to support the registration of a callback function that is invoked whenever the acquisition state changes. During service initialization, performed within the System Task, the *prvSYSTEM_AcquisitionStateChanged* function is registered as the callback handler. This function, implemented in *system.c* notifies the EEZ DIB service whenever a change in the acquisition state occurs. Within the EEZ DIB service, this state recorded within the first bit of a service-specific variable named STATUS. This acquisition state information is crucial for the EEZ DIB service to determine the appropriate timing for initiating sample collection from the SSample service.

The EEZ DIB Service Task operates in three distinct states: Initialization, Service, and Error. During the Initialization state, all relevant peripherals (such as SPI Instance 2) and internal variables are set up. The Error state handles fault recovery mechanisms if an error occurs in any of the other states. The core functionality resides in the Service state, where the main task logic is executed. In this state, the task begins by reading the queue that stores incoming request messages from EEZ DIB MCU. Once a message is

retrieved, it is processed, and the appropriate action is executed, and response is sent back to the EEZ DIB MCU.

Communication between the EEZ DIB Main MCU and the OpenEPT MCU is established over a full-duplex SPI interface. In this setup, the EEZ DIB side operates as the SPI master, while the OpenEPT side functions as the SPI slave. Messages exchanged between the two MCUs are limited to a maximum of 10 bytes and can be either in ASCII or binary format. There are two message types:

- **Request** - Sent from the EEZ DIB MCU to the OpenEPT MCU, in ASCII format, over the MOSI line.
- **Response** – Sent from the OpenEPT MCU back to the EEZ DIB MCU, in binary format, over the MISO line.

When a Request message is transmitted during an SPI transaction, it is received by the OpenEPT MCU using a dedicated SPI DMA channel. Upon reception, a corresponding callback is triggered. The message is then encapsulated into a request message structure and send to the queue on which EEZ DIB Service task is blocked. Writing to this queue will unblock task, and message processing will start. As processing results, binary response is prepared it will be transmitted during the next SPI frame.

As previously mentioned, request messages are sent from the EEZ DIB MCU to the OpenEPT MCU over the MOSI line in ASCII format. Each request consists of a command string followed by the \r character, which signifies the end of the message. The current version of the OpenEPT firmware supports a single request command: "GetVC\r". Upon receiving this command, OpenEPT responds by sending a binary-formatted message within the SPI frame. This response includes the following fields:

- Voltage (2 bytes)
- Current (2 bytes)
- STATUS variable (1 byte)
- End-of-Message (EOM) marker (2 bytes)

This binary response format is illustrated in the Figure 4.26.

Voltage *LSB*	**Voltage** *MSB*	**Current** *LSB*	**Current** *MSB*	**STATUS**	**EOM** *LSB*	**EOM** *MSB*

Figure 4.26 - EEZ DIB Response message format

4.2.5. Logging

Block Summary			
Name	Logging	**Layer**	Middleware
Version	1.03		
Related files			
Top source file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/Logging/logging.c`			
Top header file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/Logging/logging.h`			

The LOGGING service is built on FreeRTOS. It provides an abstraction for formatted log generation, queuing, and asynchronous transmission through configurable output channels. This service is designed to support modular, multitasking applications by offloading the message formatting and transmission logic into a dedicated logging task.

The service uses a FreeRTOS queue to buffer logging messages and a background task to process and transmit these logs. Messages are formed using standard printf-style formatting with variable arguments, allowing to embed context-rich diagnostic information. Log messages are categorized by severity using the *logging_msg_type_t* enum, which includes INFO, WARNING, and ERROR types. Each message is prepended with a service identifier string, and a consistent message format is applied for clarity.

The initialization process begins with *LOGGING_Init*, which creates the message queue and the binary semaphore used to signal readiness. A dedicated task (*prvLOGGING_TaskFunc*) is created and enters a finite-state machine with three states: INIT, SERVICE, and ERROR. In the INIT state, the task initializes its output channels (currently UART3) by configuring and activating the corresponding UART peripheral using the DRV_UART driver. Once the UART is ready, the service transitions to the SERVICE state and signals that it is initialized. Should any failure occur during initialization or runtime message processing, the service enters the ERROR state and signals a low-level error through the *SYSTEM_ReportError* interface.

```
static void prvLOGGING_TaskFunc(void* pvParameters){
    logging_message_t tmpBuffer;
    for(;;){
        switch(prvLOGGING_DATA.state)
        {
        case LOGGING_STATE_INIT:
            if(prvLOGGING_InitChannels() != LOGGING_STATUS_OK)
            {
                prvLOGGING_DATA.state   = LOGGING_STATE_ERROR;
                break;
            }
            prvLOGGING_DATA.state   = LOGGING_STATE_SERVICE;
            xSemaphoreGive(prvLOGGING_DATA.initSig);
            break;
        case LOGGING_STATE_SERVICE:
            if(xQueueReceive(prvLOGGING_DATA.txMsgQueue, &tmpBuffer, portMAX_DELAY) != pdPASS)
            {
                prvLOGGING_DATA.state   = LOGGING_STATE_ERROR;
                break;
            }
            if(prvLOGGING_SendLogMessage(tmpBuffer.buffer, tmpBuffer.size) != LOGGING_STATUS_OK)
            {
                prvLOGGING_DATA.state   = LOGGING_STATE_ERROR;
                break;
            }
            break;
        case LOGGING_STATE_ERROR:
            SYSTEM_ReportError(SYSTEM_ERROR_LEVEL_LOW);
            vTaskDelay(portMAX_DELAY);
            break;
        }
    }
}
```

Figure 4.27 - Logging Service's main task

Log messages are constructed and dispatched using the *LOGGING_Write* function. If the FreeRTOS scheduler has not yet started, the message is sent synchronously over the UART using a blocking transmit

function. Otherwise, the message is queued into the logging queue to be handled asynchronously by the background task. This dual-mode behavior ensures the system can produce logs even during early startup before multitasking begins.

Internally, each log message is encapsulated in a *logging_message_t* structure that holds a statically sized buffer and its length. The *prvLOGGING_SendLogMessage* function ensures all data is transmitted using the currently initialized output channel. In this implementation, UART3 is used exclusively, and its configuration is hardcoded within *prvLOGGING_InitChannels*.

4.2.6. Network

Block Summary			
Name	Network	**Layer**	Middleware
Version	1.03		
Related files			
Top source file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/Network/network.c`			
Top header file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/Network/network.h`			

The Network service handles Ethernet-based communication and integrates with the LwIP stack to provide IP connectivity for the system. This service operates as a dedicated FreeRTOS task and manages link detection, interface configuration, and runtime monitoring of the physical and MAC layers. It is built to support a static IP configuration and does not rely on DHCP, simplifying integration in systems with fixed addressing schemes.

During initialization, the service sets up the LwIP TCP/IP stack, configures the network interface with predefined IP, subnet mask, and gateway values, and adds it to the system. It then registers a callback to handle link state changes, which allows the service to update the internal state and inform the rest of the system when the connection is established or lost. Status information, such as connection speed and duplex mode, is reported through the centralized logging service, aiding in system diagnostics.

The runtime task periodically checks the state of the Ethernet PHY using the LAN8742 driver. If a link-up or link-down condition is detected, it reconfigures the MAC layer accordingly and brings the interface up or down as needed. This mechanism ensures that the system maintains a consistent view of network availability and reacts appropriately to changes in link conditions. The task uses LOCK_TCPIP_CORE and UNLOCK_TCPIP_CORE to protect LwIP operations, ensuring thread safety during status transitions.

In case of initialization or runtime failures, the service enters a safe error state and reports the problem via the system error handler. The *NETWORK_Init* function is the entry point for external modules and is responsible for creating the task, setting up synchronization, and waiting for the link configuration to complete within a given timeout. Overall, the service provides a clean and isolated way to manage Ethernet networking without exposing internal logic or requiring manual setup from application code.

4.2.7. System

Block Summary			
Name	System	**Layer**	Middleware
Version	1.03		
Related files			
Top source file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/System/system.c`			
Top header file			
`Source/ADFirmware/CM7/Core/Middlewares/Services/System/system.h`			

The System service is the central service responsible for initializing, supervising, and coordinating key firmware modules. It is implemented as a dedicated FreeRTOS task that operates across three well-defined states: INIT, SERVICE, and ERROR. Its main purpose is to establish a stable runtime environment by initializing dependent services, managing global system parameters, and providing synchronized access to shared state variables. This service ensures consistent behavior during system startup and runtime, with robust fault handling and status indication mechanisms.

During the INIT phase, the service sequentially initializes critical hardware interfaces and middleware services. This includes GPIO configuration for system status indicators, PWM timer setup for RGB LED control, and external interrupt configuration for user input (e.g., button press). It then proceeds to initialize a few software components, namely the Logging, Network, Control, Sample Stream (SSTREAM), Discharge Profile Control (DPCONTROL), Energy Debugger, and EEZ DIB services. Each component is initialized with a timeout mechanism, and any failure during this phase results in a transition to the ERROR state, with corresponding logging output and LED indication.

The System service manages a global RGB LED whose colour can be dynamically updated to reflect various system statuses (e.g., idle, error, or active acquisition). The current colour values are stored in a shared data structure and updated using a thread-safe mechanism with FreeRTOS semaphores. Updates are propagated to the main system task via task notifications, ensuring that PWM adjustments are performed within the correct task context. This design prevents concurrency issues and eliminates the need for direct hardware access from external tasks.

Another critical responsibility of the System service is linking status monitoring. It provides a public API (*SYSTEM_SetLinkStatus*) that allows the network service (or other modules) to update the current network connection state. The link status is indicated using a dedicated GPIO-controlled diode, and internal state updates are guarded by a semaphore to ensure safe access from multiple contexts.

The service also supports dynamic assignment of a device name, which is stored internally and can be accessed or updated by external components via *SYSTEM_SetDeviceName* and *SYSTEM_GetDeviceName*. The data is protected by the same mutex used for other shared resources, maintaining consistency across the system.

If an error occurs at any stage, the System service provides centralized fault handling via the *SYSTEM_ReportError* API. Depending on the severity (low, medium, or high), the function configures the RGB LED to a corresponding red intensity and activates the error status diode. This visual feedback is useful for debugging and system monitoring in embedded environments where console access might be limited.

Once all components are initialized and no faults have been detected, the system transitions into the SERVICE state. In this state, the task waits for asynchronous events (e.g., RGB LED updates) and maintains the system's stable operation. The core logic of the application may also be placed here, although typically higher-level modules will handle runtime logic beyond the scope of basic system management.

Overall, the System service plays a pivotal role in the architecture by initializing and supervising all major services, providing reliable runtime control, and managing system-wide status indicators in a thread-

safe manner. This modular and guarded design ensures scalability, robustness, and ease of integration across a wide range of embedded firmware projects.

4.3. Configuration

The *globalConfig.h* file (*Source/ADFirmware/CM7/Core/Configuration/globalConfig.h*) serves as a centralized configuration header that defines compile-time constants used across the firmware architecture. It encapsulates the system-wide definitions required to configure various software services, hardware abstraction layers, driver interfaces, and task-related parameters. This file enables the firmware to be easily adjusted for different hardware configurations, operational modes, and performance tuning without requiring deep changes in implementation files.

5. Build and Run Instructions

To successfully build and run the DAU firmware, the STM32Cube IDE must be installed and configured with the appropriate dependencies. The following steps guide you through the full setup process.

Step 1: Download and install STM32CubeIDE

The latest version of STM32CubeIDE should be obtained from this link. After downloading, STM32CubeIDE should be installed on the machine by following standard installation instructions.

Step 2: Clone project from official GitHub account

There are two primary methods for downloading the project from the official GitHub repository. The first method is using a dedicated Git console, such as Git Bash Application, which allows cloning the repository directly via the command line. The second method involves navigating to the repository's GitHub page and clicking the green Code button, then selecting the Download ZIP option. This will download the project files as a compressed archive. The latter method is visually illustrated in the corresponding Figure 5.1.

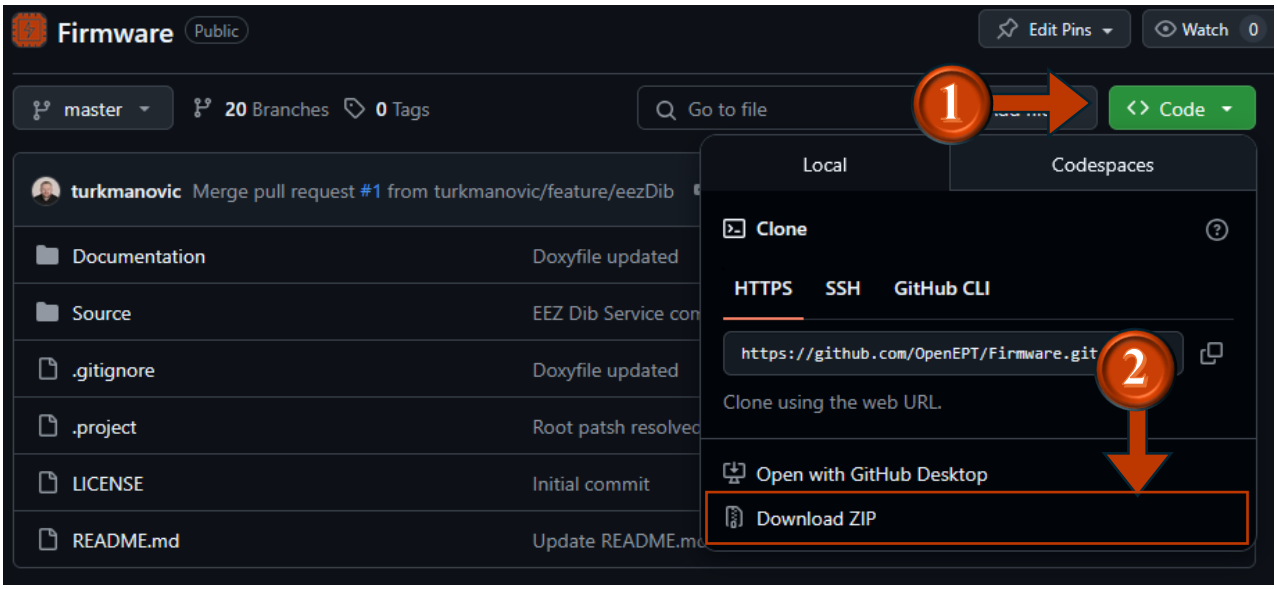

Figure 5.1 - Download project from the official GitHub repository

Step 3: Import project

Once STM32CubeIDE is launched and the workspace path is configured, navigate to *File → Open Projects from File System* (**1, 2** in Figure 5.2).

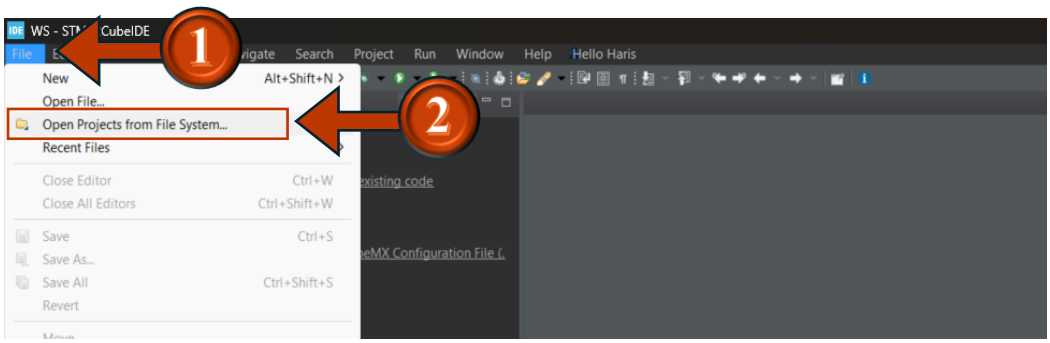

Figure 5.2 - Open project from file system option

Clicking *Open Projects from File System* opens the *Import Project from File System or Archive* window. In this dialog, you need to specify the directory where the project resides (in our case, *Source/ADFirmware*). To do so, click the *Directory* button (**1** in Figure 5.3).

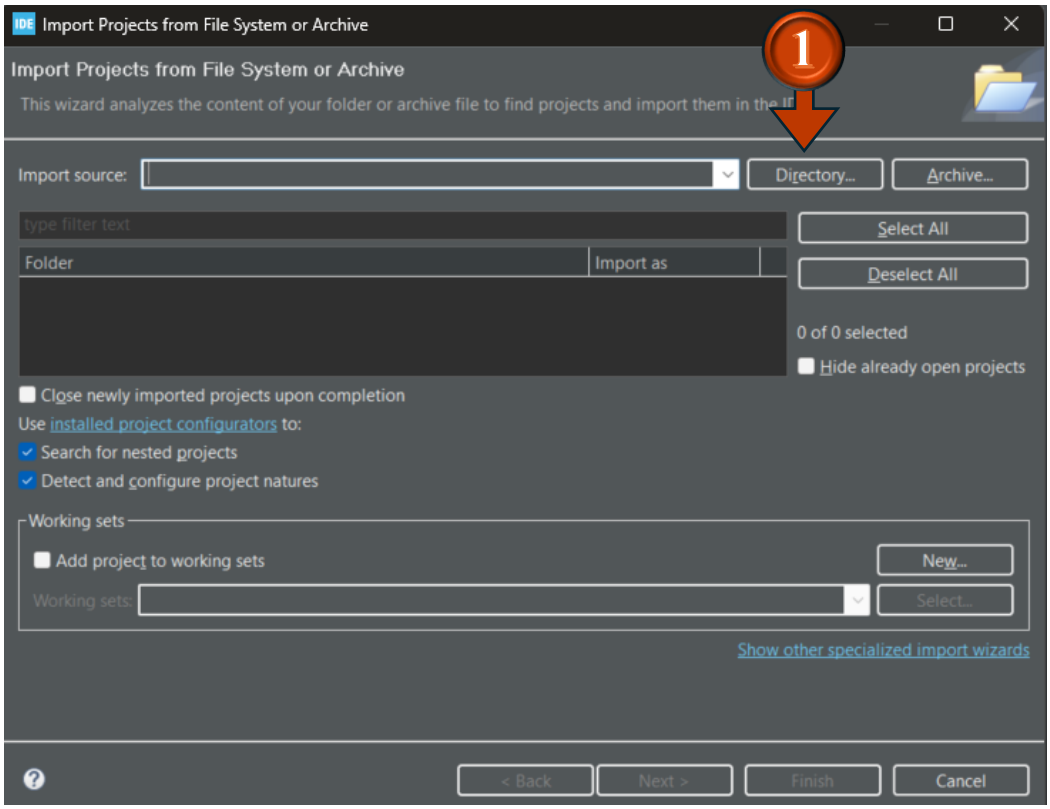

Figure 5.3 - Import project from file system or archive

After clicking the *Directory* button, the *Browse for Folder* window appears (Figure 5.4). Here, navigate to the *Source/ADFirmware* directory (step **1**), and click *Select Folder* (step **2**).

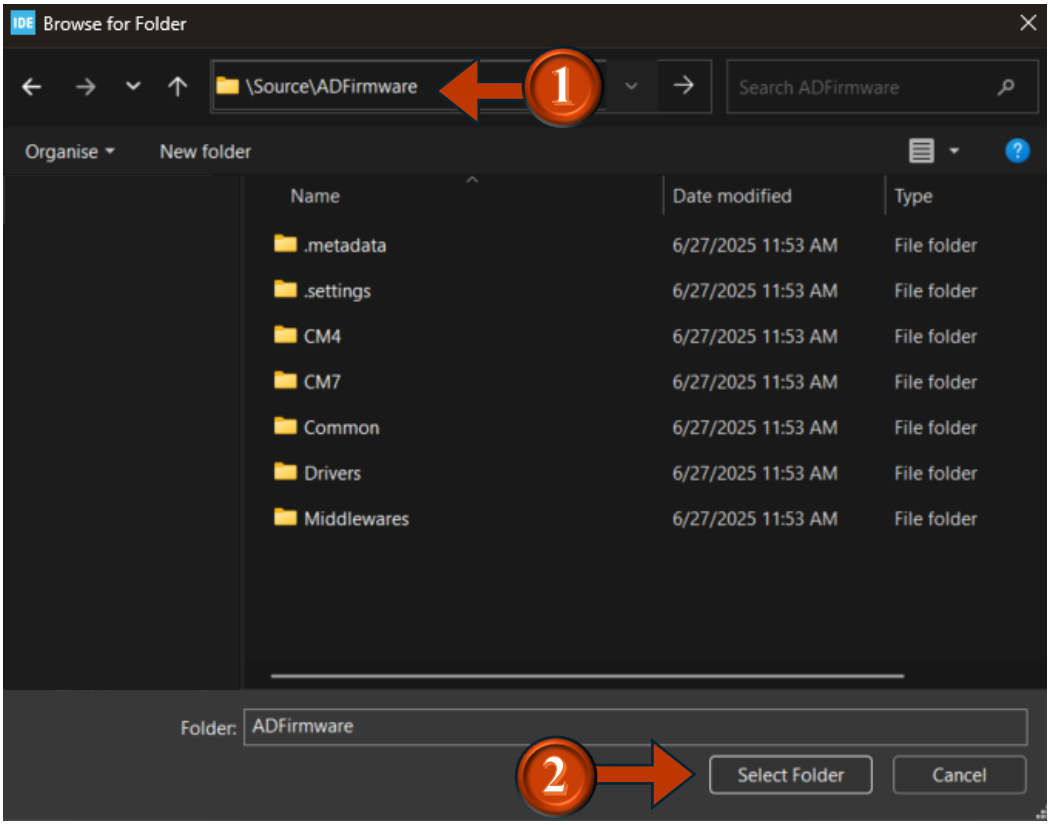

Figure 5.4 - Browse for Folder window

If the project is successfully detected in the selected directory, the window from Figure 5.3 will be updated to reflect the discovered project, as shown in Figure 5.5. To proceed with the import, click the *Finish* button (step 1).

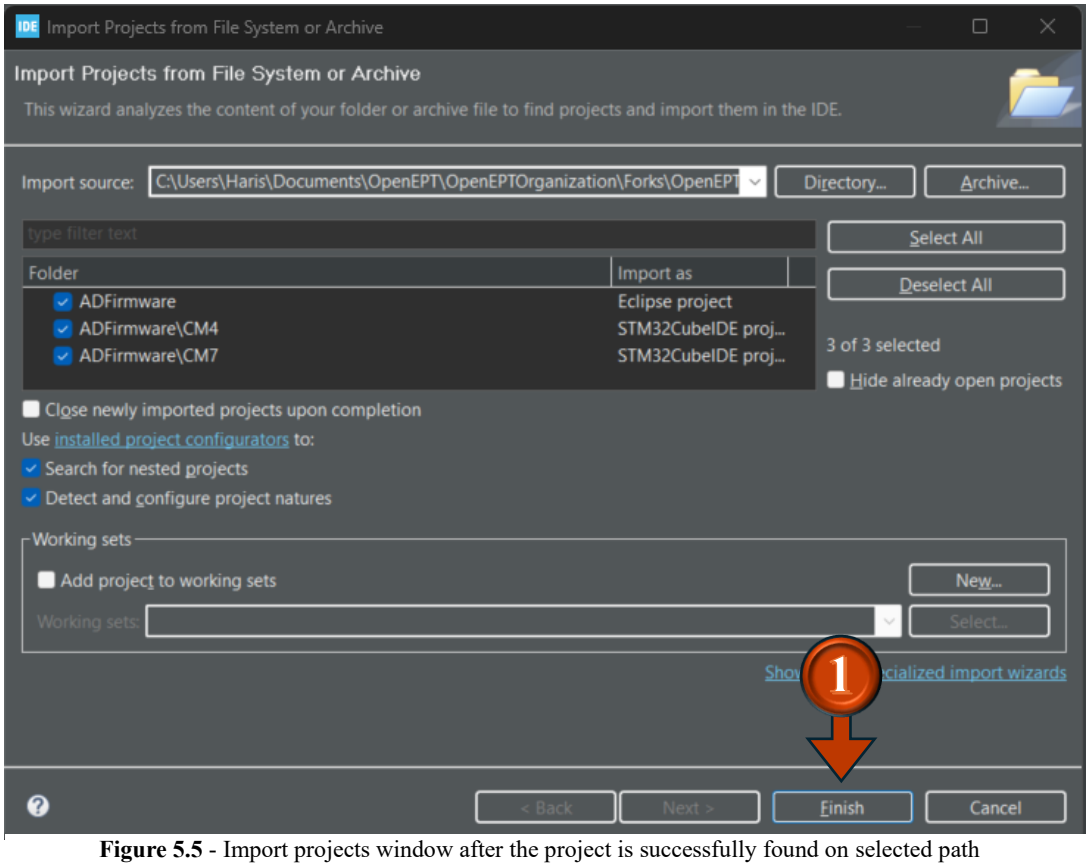

Figure 5.5 - Import projects window after the project is successfully found on selected path

Once the project is successfully imported, it will be displayed in the *Project Explorer* panel, as shown in Figure 5.6.

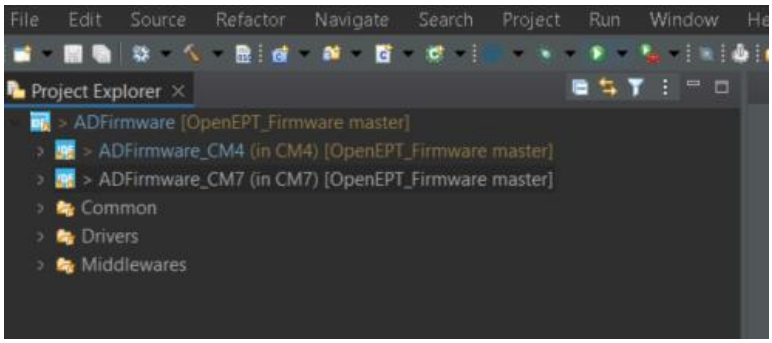

Figure 5.6 - Project Explorer window after the project is successfully imported

Step 4: Configure Global path

The PROJECT_PATH variable serves as a reference point for many relative include paths used throughout the project. Instead of hardcoding absolute paths, source files and build configurations use this variable to dynamically resolve locations of headers and source files within the project directory. This approach improves project portability and maintainability, especially when working across different systems or when the project is moved to a different directory structure.

To define the PROJECT_PATH build variable, right-click on the project named ADFirmware_CM7 in the *Project Explorer*, then select Properties. In the Properties window, navigate to *C/C++ Build → Build Variables* (Figure 5.7). Locate the variable named PROJECT_PATH, which is essential since many of the project's include paths are defined relative to it.

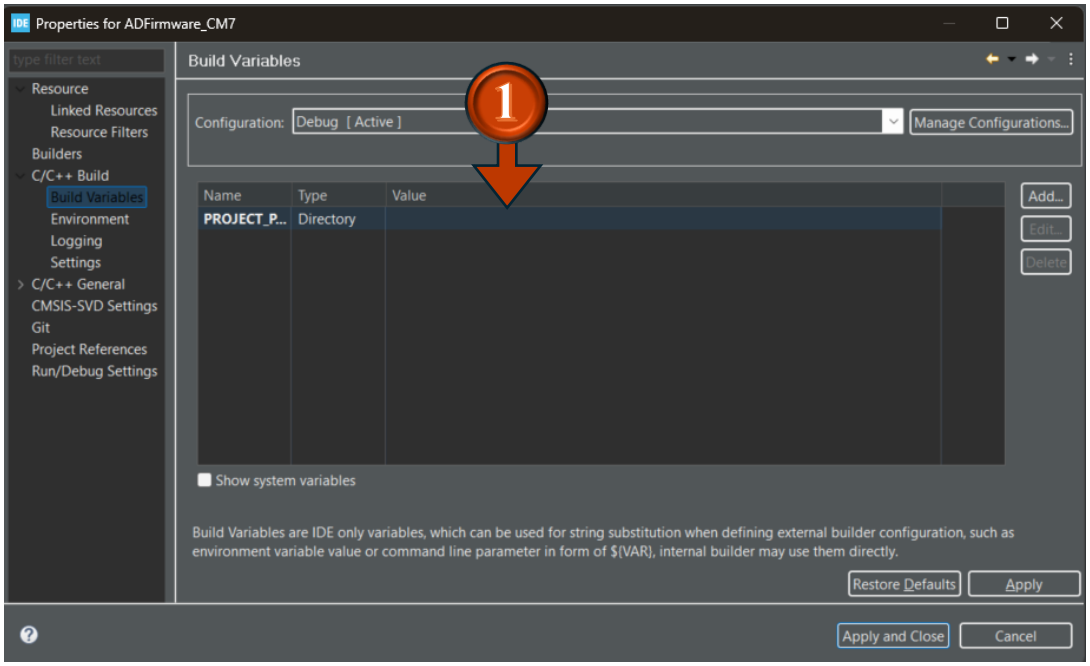

Figure 5.7 - Build variables part of Properties window

After locating the PROJECT_PATH variable, double-click on it to open the *Edit Existing Build Variable* dialog (1 in Figure 5.7). In this window, click *Browse* (1 in Figure 5.8). and navigate to the project directory, specifically *Source/ADFirmware*. Select this folder and confirm by clicking *OK*. This assigns the correct path to the PROJECT_PATH variable, ensuring that include files are properly resolved during the build process.

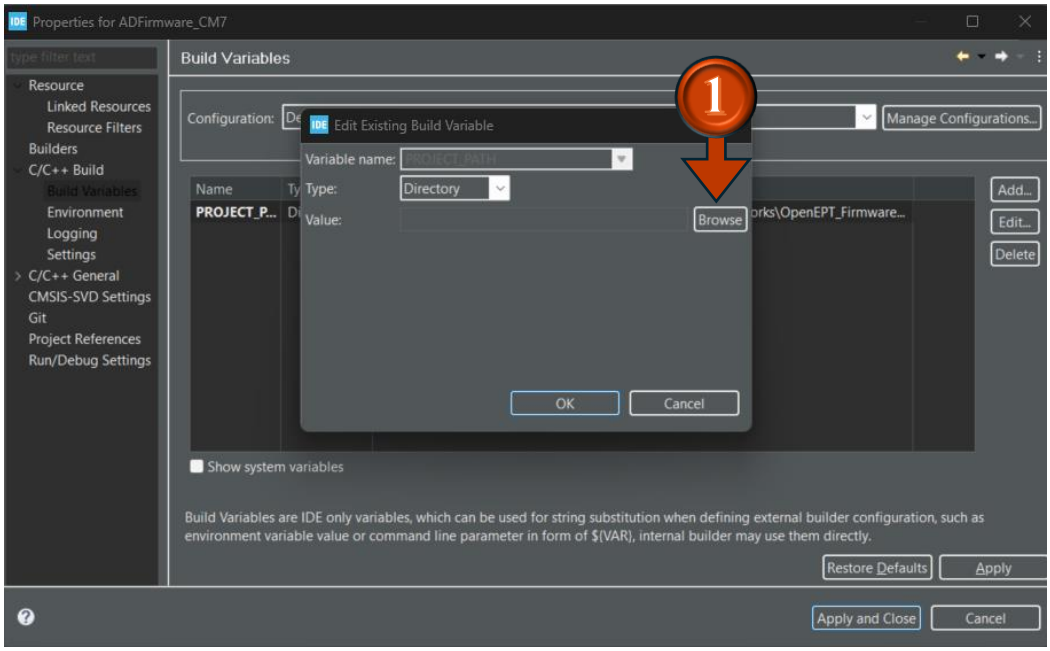

Figure 5.8 - Edit Existing Build Variable

After clicking *OK* in the *Edit Existing Build Variable* dialog, return to the *Properties* window shown in Figure 5.7. To finalize the change, click *Apply* and *Close*. If the specified path is valid and correctly set, the include paths listed under *ADFirmware_CM7* → *Includes* in the *Project Explorer* will update automatically. The folder icon should change from a transparent folder with a yellow warning triangle in the bottom-right corner (Figure 5.9) to a solid blue folder icon (Figure 5.10), indicating that the include path has been successfully resolved.

Figure 5.9 - Paths before Build variable path is resolved

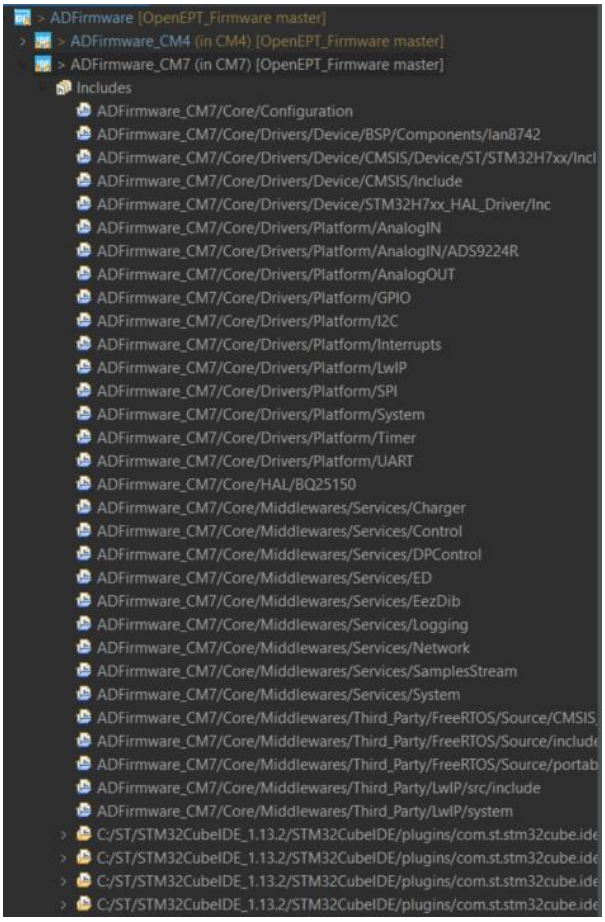

Figure 5.10 - Paths after Build variable path is resolved

Step 4: Build and Run the project and run first Debug session

Once the project is fully configured and successfully imported, the build process can be initiated. To begin, right-click on the *ADFirmware_CM7* project in the *Project Explorer*. It is recommended to first perform a clean build to ensure that all previously generated files are removed. To do this, select *Clean Project* from the context menu. After the cleaning process is complete, right-click on the project again and select *Build Project*. This will start the compilation process using the defined settings and paths. If building process is successfully done, *ADFirmware_CM7.elf* is generated under *ADFirmware_CM7* → *Binaries* (**1** in Figure 5.11). To run the project, click on bug symbol from tool bar (**2** in Figure 5.11).

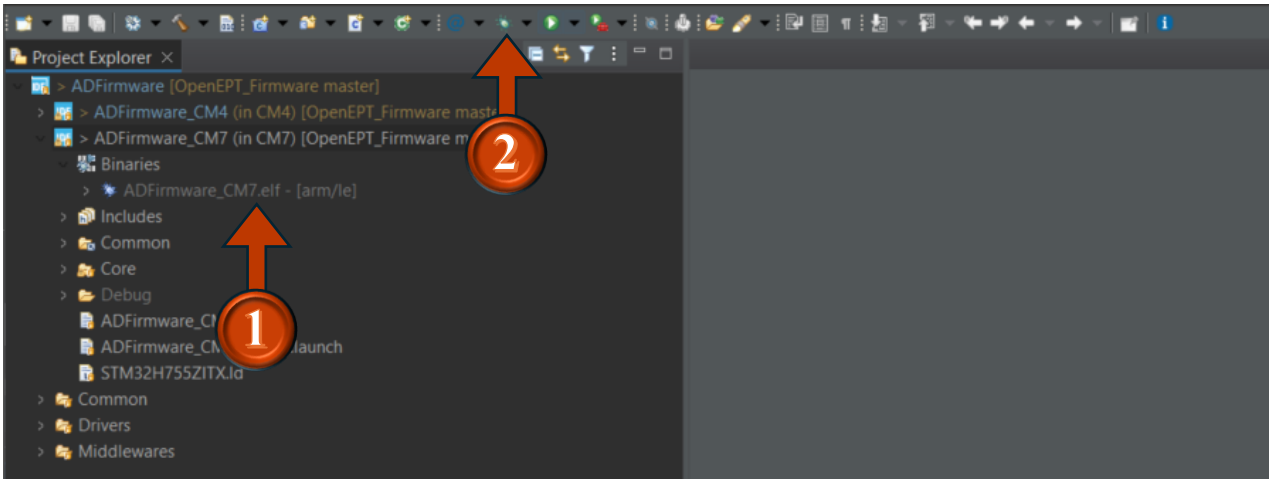

Figure 5.11 - Project binary and first debug run

List of Figures

References

[1] ADS92x4R Dual, Low-Latency, Simultaneous-Sampling SAR ADC Datasheet, June 2019

[2] Command list document