Step-by-Step Tutorial: Quaye Works HAWK-H7 Flight Controller Programming Guide

This comprehensive tutorial will guide you through **setting up the QUAYEWORKS HAWK-H7 flight controller** based on the **STM32H743IIT6** microcontroller using **STM32CubeIDE and CubeMX**. We will cover configuration of all peripherals from the provided schematic/pinout, including UARTs, I2C, SPI, SDMMC, Timers/PWM for motors and servos, ultrasonic sensor interfaces, external interrupts, DMA, clock configuration, and USB power management. The tutorial uses **bare-metal C (HAL library, no RTOS)** and emphasizes a modular code structure. Follow the steps below to configure peripherals in CubeMX, generate initialization code, and organize your application logic.

0. STLINK-V2 Clone Setup and Programming in CubeProgrammer

This section details how to set up and program your flight controller using an STLink-V2 clone. Follow these clear steps to extract, install, and flash the proper bootloaders and firmware so your board can be programmed using the ST-Link utility and STM32CubeProgrammer.

0.1 Extract the STLinkV2_Clone Files:

1. **Download:**
   Download the **STLinkV2_Clone.zip** folder from the QuayeWorks GitHub repository.
2. **Extract:**
   Once the download is complete, extract the zip file into your preferred Downloads folder.
   You should see two files:
   - `Unprotected-2-1Bootloader.bin`
   - `en.stsw.link004.zip`

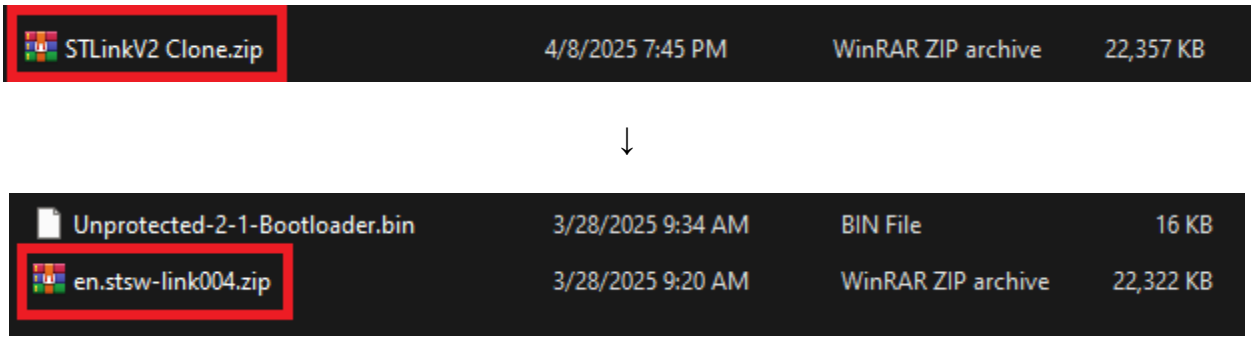


Figure 0.1"Screenshot showing the extracted files from STLinkV2_Clone.zip")

0.2 Install the ST-Link Utility:

1. **Extract Utility:**
   Extract the file named **STM32 ST-LINK Utility v4.1.0 setup.exe** from the downloaded package.
2. **Install Drivers:**
   Launch the setup program and follow the on-screen instructions to install the ST-Link Utility and the necessary drivers on your PC.

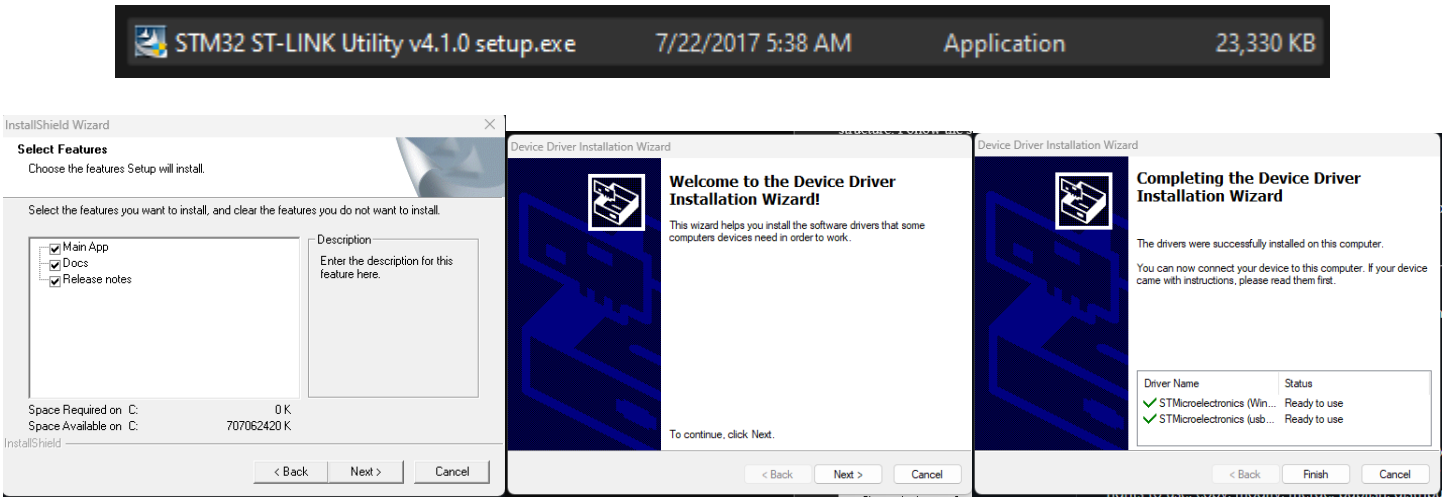


Figure 0.2: "Screenshot of the ST-Link Utility installation wizard")

0.3 Prepare STLink-V2 Clone Debug Ports:

Before flashing the firmware, you must connect the STLink-V2 clone to your Hawk-H7's debug port. Follow these wiring instructions carefully:

- **5V:** Connect the 5.0V output from the STLink clone to the 5.0V port on the Hawk-H7 board.
- **SWDIO:** Connect the STLink's SWDIO pin to the TMS (Test Mode Select) pin on the board.
- **SWDCK:** Connect the STLink's SWDCK (clock) pin to the TCK (Test Clock) pin.
- **GND:** Connect the ground (GND) wires together.

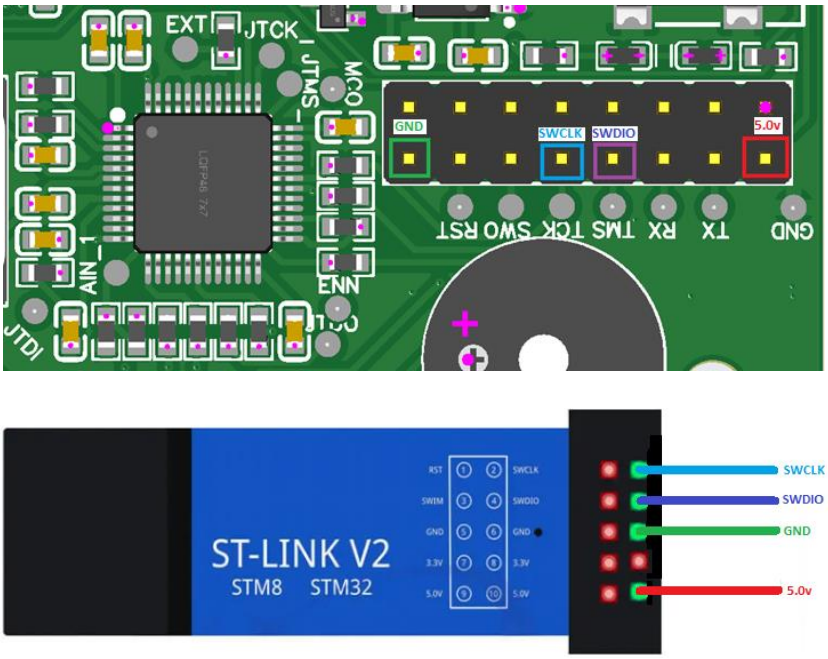


Figure 0.3: "Diagram showing the proper wiring connections between the STLink-V2 clone and the debug port, highlighting the 5V, SWDIO, SWDCK, and GND connections."

0.4 Flash the Unprotected Bootloader:

1. **Connect STLink-V2 Clone:**
   Attach your STLink-V2 clone to your computer via USB.
2. **Connect to the Target:**
   Using the ST-Link Utility, connect to your target board.
3. **Erase the Chip:**
   In the utility, perform a chip erase to clear the previous firmware.
4. **Re-Connect the STLink-V2 Clone:**
   Disconnect the STLink-V2 clone, then unplug and replug it to refresh its connection.
5. **Flash Bootloader:**
   Open the `Unprotected-2-1Bootloader.bin` file in the ST-Link Utility and flash it to the board.
6. **Disconnect:**
   Once flashing is complete, remove the STLink-V2 clone's connections from the debug port.

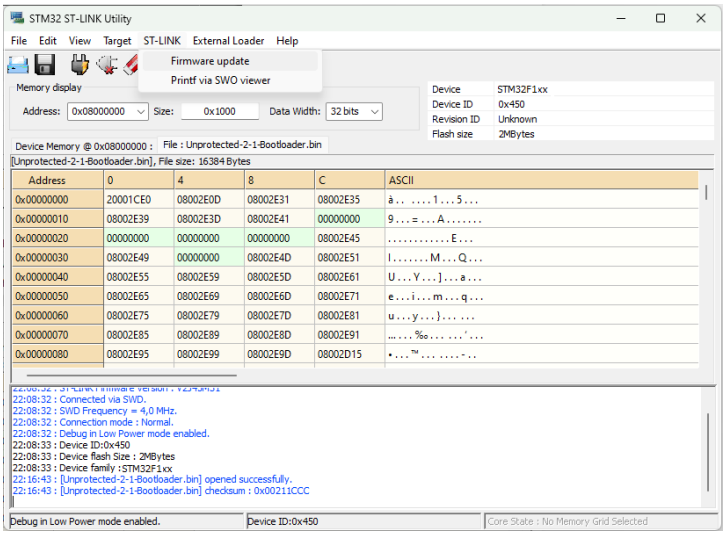
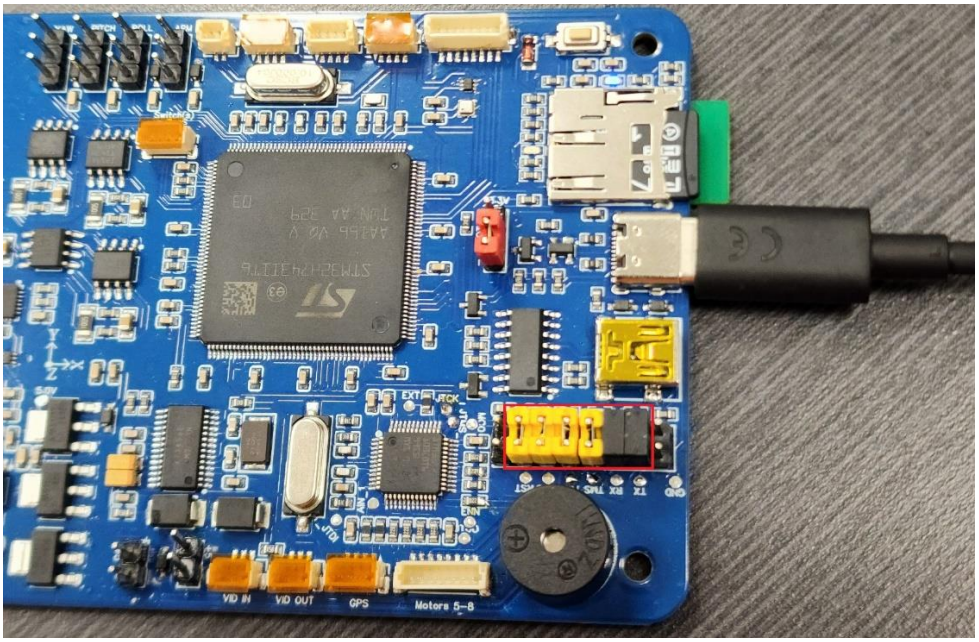


Figure 0.4: "Screenshot of the ST-Link Utility showing the process of flashing the Unprotected-2-1Bootloader.bin file.")

0.5 Upgrade the F103CBT6 Firmware:

1. **Set Up Jumper Connections:**
   Using six jumpers, connect all the necessary pins (excluding ground and power pins) on the board to prepare the debug port for firmware upgrade.
2. **Connect to PC:**
   Attach the Hawk-H7 board to your PC via the USB-C cable. Ensure the debug connection is stable.
3. **Launch ST-Link Utility:**
   Open the utility to update the firmware. Follow these steps:
   - Open the **Firmware Update** page within the ST-Link Utility.
   - Click **Device Connect**.
   - Select the **STM32+MSD+VCP** product option.
   - Confirm by clicking **Yes** to update the firmware.

Your board should now register as an ST-Link-V2.



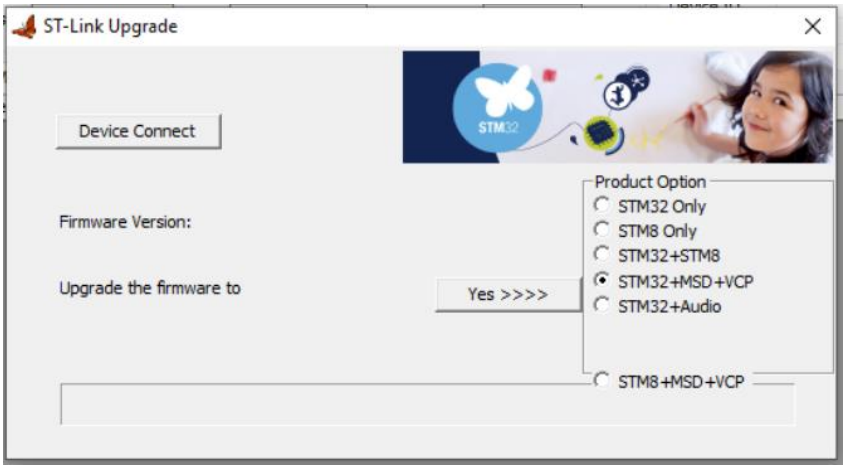


Figure 0.5: "Image showing the ST-Link Utility Firmware Update page with the STM32+MSD+VCP option selected."

0.6 Upgrade to the Latest Firmware:f

1. **Launch Cube Programmer:**
   Install the latest **STM32CubeProgrammer** if you haven't already. Open its "firmware upgrade" page.
2. **Enter Update Mode:**
   Click **Open in Update Mode** to reload the device for update. The device should show an ST-Link ID with an unknown firmware version (this is normal).
3. **Perform Upgrade:**
   Click the **Upgrade** button. Maintain the **STM32+MSD+VCP** option throughout the process.
   Note: This board is equipped with the STM32F103CBT6 for its ST-Link-V2 functionality. It is more capable than the older C8T6 variant, so selecting another option might downgrade its performance.
4. **Reconnect:**
   After upgrading, disconnect the flight controller's USB cable from the PC and then reconnect it.

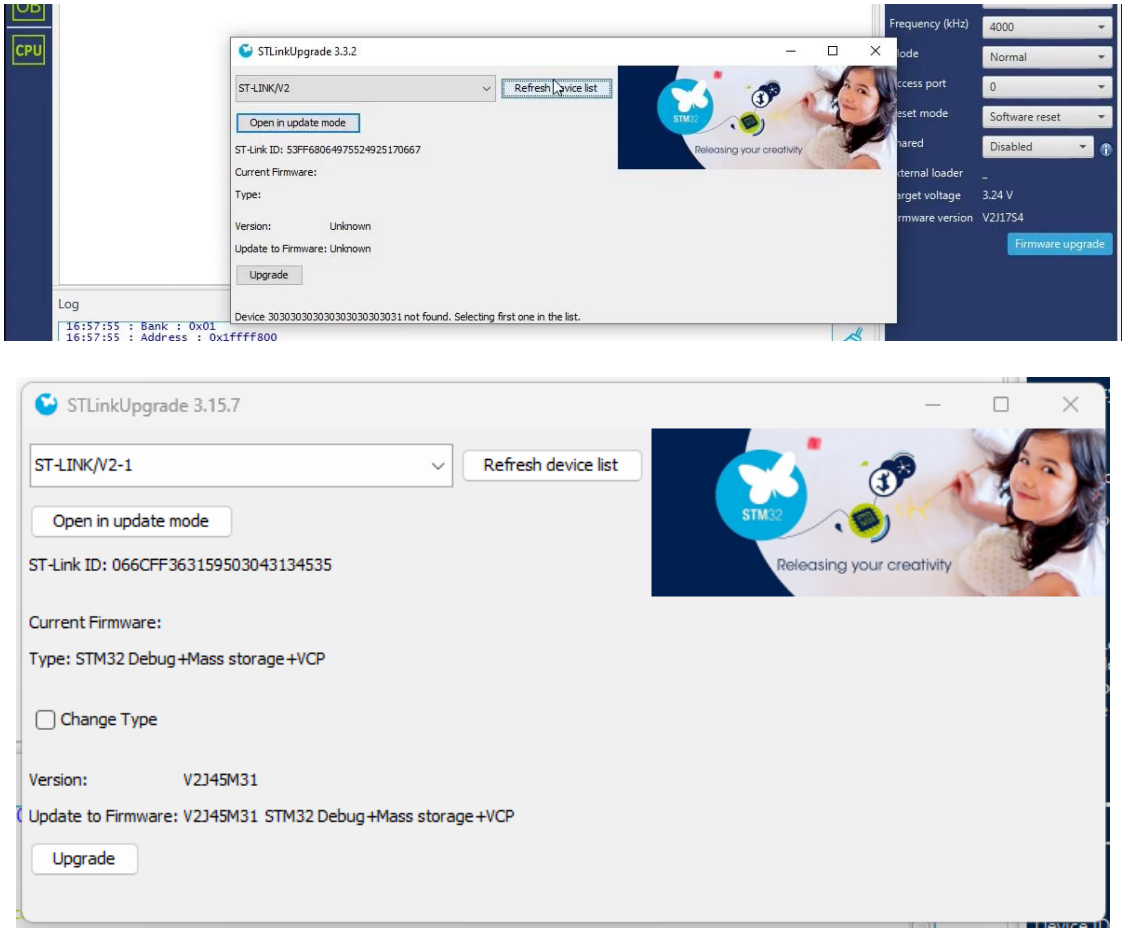


Figure 0.6: "Screenshot of STM32CubeProgrammer's firmware upgrade page showing the device in update mode, ready for upgrade, with the STM32+MSD+VCP option selected."

0.7 Connect to the Hawk-H7 Board:

1. **Verify ST-Link-V2 Connection:**
   After the firmware upgrade, the ST-Link-V2 clone should display a new ST-Link ID and updated firmware version.
2. **Re-Connect Debug Ports:**
   Ensure that all jumpers remain connected as previously described and reattach the ST-Link-V2 to the board.
3. **Final Verification:**
   Connect to the Hawk-H7 board through the ST-Link-V2 interface. The utility should now show that you are connected to the Hawk-H7 board for debugging and UART programming.

Your ST-Link connection is now configured for debug serial and programming of the Hawk-H7 board.

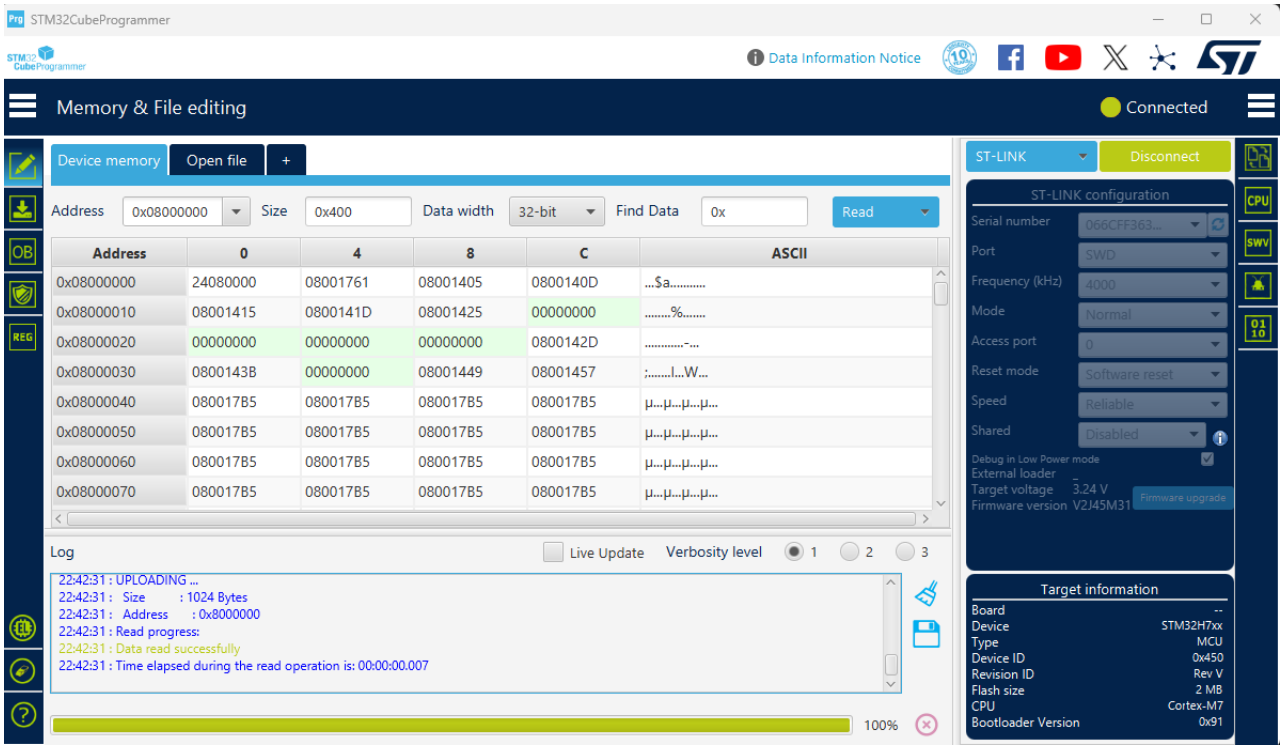


Figure 0.7: "Screenshot showing the ST-Link-V2 connection established with the H7 board, indicating debug and UART programming connections."

1. Project Setup in STM32CubeIDE and CubeMX

1.1 Create a New Project: Launch **STM32CubeIDE**, go to **File → New STM32 Project**. In the Target Selector, search for **STM32H743IIT6** (our MCU) and select it. CubeMX will open with the MCU's pinout.

1.2 Configure the Clock Source: Open the **Clock Configuration** tab. According to the schematic, the board has an external 10 MHz crystal (pins **PH0/PH1**) and a 32.768 kHz LSE (pins **PC14/PC15**)

Do the following:

- Set **HSE (High Speed External)** as the clock source. Input frequency = 10 MHz (if that's the crystal on your board).
- Configure the PLL to achieve the desired system core clock (e.g. 400 MHz for STM32H7). For example, set PLL1 with appropriate M, N, P, Q dividers to get 400 MHz for SYSCLK and 200 MHz for APB timers, etc.
- Enable the **LSE** if you need the RTC or precise low-power timers (optional).
- Ensure **PLL3** is set to provide a 48 MHz clock if you plan to use SDMMC or USB (SDMMC often requires 48 MHz for the SD clock base).

1.3 Power and Voltage Settings: In CubeMX's **System Core → RCC**, make sure the voltage scale is correct (Voltage Scale 1 for high frequency). Also ensure **PB14/PB15** are not used for OSC (they are used as UART1 TX/RX in our design). CubeMX usually handles this based on chosen peripherals.

1.4 Enable CPU Caches: The STM32H7 has I/D caches. By default, `HAL_Init()` enables them. Leave this as default but remember when using DMA you may need to manage cache coherency (we will mention this later). For now, proceed with default settings.

2. Pinout Configuration Overview

Using the provided pinout, we will assign each peripheral to the correct pins in CubeMX. The table below summarizes the peripheral-to-pin mapping from the schematic:

- **UART Interfaces:**
  - **UART1:** `PB14` (TX) and `PB15` (RX) – connected to ST-Link Virtual COM port (for debugging/logging).
  - **UART3:** `PB10` (TX) and `PB11` (RX) – connected to Bluetooth **HC-05** module.
  - **UART6:** `PC6` (TX) and `PC7` (RX) – connected to **GPS (Ublox)**.
- **I2C1 Bus:** `PB8` (SCL) and `PB9` (SDA) – connected to **MPU6050**, **BMP388**, **QMC5883L**, **INA219** (current sensor), and four **Buck converters** (if they have I2C control)
- **SPI4 Interface:**
  - `PE12` (SPI4_SCK), `PE14` (SPI4_MOSI), `PE13` (SPI4_MISO), and `PE11` as a GPIO for chip select – connected to **AT7456E OSD** chip
- **SDMMC1 (SD Card):**
  - D0-D3: `PC8`, `PC9`, `PC10`, `PC11`
  - CLK: `PC12`
  - CMD: `PD2`
  - (These pins form the 4-bit wide SDIO interface for the SD card slot)
- **PWM Outputs for 8 Motors:**
  - Using **TIM1** for Motors 1-4 on channels 1-4: pins `PA8`, `PA9`, `PA10`, `PA11`
  - Using **TIM3** for Motors 5-8 on channels 1-4: pins `PB4`, `PB5`, `PB0`, `PB1`
- **PWM Outputs for 4 Servos:**
  - We will use **TIM5** for Servos (4 channels): pins `PA0`, `PA1`, `PA2`, `PA3` for Yaw, Pitch, Roll, Arm servos respectively

    (TIM5 channels 1-4 map to PA0-PA3 on STM32H7.)

- **Ultrasonic Sensors (HC-SR04) x3:**
  - Trigger pins (outputs): `PF6`, `PF7`, `PF8` – one for each sensor's trigger.
  - Echo pins (inputs): `PF10`, `PF11`, `PF12` – each receives the echo pulse from a sensor.
  - We will use timers in input capture mode or EXTI interrupts on these echo pins to measure pulse width (distance). The pinout suggests using timers **TIM16, TIM17, TIM13** for triggers

    (each of those timers has a channel 1 that we can use on PF6, PF7, PF8 respectively). We'll configure input capture on the echo pins (specific timer channels chosen in CubeMX based on pin availability, e.g., TIMx CHy that maps to PF10, PF11, PF12).

- **PPM Input (Radio Receiver):** Pin `PF9` is designated for PPM input

  We can use a timer input capture here as well (or external interrupt) to decode incoming radio commands if needed.

- **Other GPIOs:**
  - Enable pins for sensors: e.g., `PA15` for MPU6050 Enable, `PD1` for QMC5883L Enable, `PD0` for BMP388 Enable, `PA12` for HC-05 Enable. Set these pins as **GPIO_Output** in CubeMX so we can control power to those modules (as per schematic notes).
  - HC-05 State pin: `PA5` is an input from the BT module state indicator

    (can be used to detect Bluetooth connection status, set as GPIO_Input).

  - Buck converter ON/OFF pins: `PI0`, `PG13`, `PG14`, `PG15` for Yaw/Roll/Pitch/Arm LM2596 regulator enable control

    Configure as outputs (likely active high or low per design; the schematic indicates MOSFET control).

  - Buzzer pin: `PH6` (digital output for a buzzer)
  - Battery voltage analog pin: `PH2` (ADC input for battery voltage divider)

    We will set up an ADC channel on this pin with appropriate ADC (likely ADC3 on H7 for PH2).

Pin Assignment in CubeMX: Go to the **Pinout & Configuration** tab and assign each pin:

- Click pin by pin on the MCU diagram or use the categories on the left (e.g., go to **Connectivity → USART3** to assign PB10/PB11, etc.).
- For each peripheral (UART, I2C, SPI, etc.), enable it and assign the pins as per above list. CubeMX will automatically mark the pins in green once assigned.

3. UART Configuration (HC-05, GPS, PC Serial, Debug)

We have four UARTs to set up. In CubeMX, under **Connectivity**, enable the following:

3.1 USART1 (or UART1) for ST-Link Virtual COM – This will be used to print debug messages via the on-board ST-Link (the ST-Link MCU is connected to our STM32H7's PB14/PB15):

- Mode: Asynchronous.
- Pins: Set **PB14** as TX, **PB15** as RX
- Parameter Settings: Choose a Baud Rate (e.g., **115200** bps for debug). No hardware flow control.
- NVIC: Enable the global interrupt for USART1 if you plan to use interrupts or DMA for reception.

3.2 USART3 for Bluetooth HC-05 – Connects to HC-05 Bluetooth:

- Mode: Asynchronous.
- Pins: **PB10** (TX), **PB11** (RX)
- Baud Rate: HC-05 default is typically **9600** or **38400** (check your module, default is often 9600). Set accordingly.
- NVIC: Enable USART3 global interrupt (for receiving commands from Bluetooth via interrupt).
- Optionally, you can enable **DMA** for USART3 RX if expecting continuous data from BT (not usually heavy, but possible for higher baud).

3.3 USART6 for GPS (Ublox) – Connects to GPS module:

- Mode: Asynchronous.
- Pins: **PC6** (TX – goes to GPS RX), **PC7** (RX – from GPS TX)
- Baud Rate: Ublox GPS modules often default to **9600** baud (NMEA) or sometimes 38400. Check module settings; 9600 is safe to start.
- NVIC: Enable USART6 interrupt (GPS will send data continuously, so we often use either interrupts or DMA to handle incoming NMEA sentences).
- DMA: It's recommended to enable a DMA channel for USART6 RX in circular mode, to continuously capture GPS data with minimal CPU load. In CubeMX, under USART6 → Add DMA (choose RX Stream, e.g., DMA1 Stream0, peripheral-to-memory, circular).

After enabling these, CubeMX will list the configured UARTs in **Connectivity > USART/UART** section. Double-check no pin conflicts (CubeMX flags conflicts in red). The pinout text also indicated that **PB14/PB15** might be used by the ST-Link MCU, but if we configure them for our UART1, it's correct (the ST-Link is physically connected to those pins for VCP).

Interrupt vs DMA for UART: For relatively slow or infrequent data (like Bluetooth commands or debug prints), UART interrupt mode is fine. For continuous high-rate data (GPS at 10Hz NMEA or higher update rates), DMA is beneficial. We'll enable DMA for at least GPS (and maybe for others for demonstration).

Make sure to also configure the **UART pin output settings** if needed (CubeMX by default sets them as push-pull alt function, which is fine for UART TX). No pull-up needed on RX typically (the devices likely have internal pull-ups or are driven).

4. I2C1 Configuration (IMU, Barometer, Magnetometer, INA219, Bucks)

We will use **I2C1** on pins PB8/PB9 to interface with multiple sensors:

- In CubeMX, enable **I2C1** under Connectivity.
- Pins: **PB8** as I2C1_SCL, **PB9** as I2C1_SDA
- Mode: I2C (leave as I2C peripheral, no SMBus).
- Speed: Set Fast Mode (400 kHz) if all devices support it (MPU6050, BMP388, QMC5883L, INA219 usually can handle 400kHz; if not, Standard 100kHz is safe).
- Acknowledge: Enable.
- Do NOT enable analog filter if not needed (default on).
- DMA: You can add DMA for I2C1 TX and RX if you plan to read sensors in the background. However, sensor readings are typically short (few bytes) so DMA is optional. If you expect to stream large chunks (not common for these sensors), DMA might help. We'll proceed without DMA on I2C for simplicity.
- NVIC: Enable I2C1 event and error interrupts (CubeMX might enable by default). This allows using HAL I2C in interrupt mode if desired, or at least get error callbacks (like bus error, NACK).

Note: Since multiple devices share this bus, each has a unique address. Later in code, we'll initialize each sensor by addressing it (e.g., MPU6050 at 0x68, QMC5883L around 0x0D, etc.). Ensure proper pull-up resistors on SCL/SDA lines (the schematic likely has them, e.g., 4.7kΩ to 3.3V, often included in sensor modules). We also have four **Buck converters** listed on this I2C bus. If these are I2C-controlled (some regulators allow control or telemetry via I2C), ensure you have their addresses and any libraries or command sets. If they're simply on/off via GPIO (which we have as PI0, PG13, PG14, PG15), then I2C might not be heavily used for them except possibly monitoring. The **INA219** is an I2C current sensor (address typically 0x40-0x45 range). We will later write a driver to read battery current via INA219 over I2C1.

5. SPI4 Configuration (AT7456E On-Screen Display)

The AT7456E (OSD chip) communicates via SPI. We will use **SPI4**:

- In CubeMX, enable **SPI4** under Connectivity.
- Mode: Full-Duplex Master.
- Pins assignment:
  - **PE12** as SPI4_SCK,
  - **PE13** as SPI4_MISO,
  - **PE14** as SPI4_MOSI,
  - **PE11** will be used as the Chip Select (NSS) but we can manage it manually as a GPIO.
- In CubeMX, you have two options for CS:
  - **Option A:** Designate PE11 as SPI4_NSS (Slave Select) in CubeMX. If you do this, set SPI mode to "Hardware NSS Signal" and CubeMX will configure that pin. However, hardware NSS in master mode on STM32 is not always convenient (it toggles NSS automatically only under certain conditions).
  - **Option B (Recommended):** Leave SPI4_NSS unassigned (disable it in CubeMX) and configure **PE11** as a regular GPIO output. We will manually control PE11 in software (set it low before a transaction to select the OSD, and high after).
- SPI Parameters:
  - Set the Baud rate prescaler such that SPI clock is within AT7456E limits (if the OSD chip can handle up to e.g. 8 MHz, choose a prescaler accordingly based on H7's APB clock).
  - Data Size: 8 bits.
  - Clock Polarity and Phase: According to AT7456E datasheet (if it's like MAX7456, likely CPOL=0, CPHA=0 or 1; we should confirm. Let's assume Mode 0 (CPOL=0, CPHA=0) for now).
  - First bit: MSB first.
- NVIC: Enable SPI4 global interrupt if you plan to use interrupt-driven SPI. If using DMA, also add DMA channels:
  - You could add a DMA for SPI4 TX (memory-to-peripheral) especially if sending large OSD data (like updating many characters on screen).
  - SPI4 RX DMA not usually needed unless reading back a lot of data from OSD (the OSD might send some status bytes).

Later in code, we will create an `OSD_driver` module that uses HAL SPI functions to send commands to AT7456E (like writing characters to the video overlay). The OSD chip typically needs initialization (setting up video mode, etc.), and then you write to its registers to place text or graphics on the screen.

6. SDMMC1 Configuration (SD Card Interface)

For logging flight data or configurations, an SD card is often used. We use **SDMMC1** peripheral with 4-bit bus:

- In CubeMX, enable **SDMMC1** (sometimes under Connectivity as well).
- Pins to assign:
  - **PC8** → D0,
  - **PC9** → D1,
  - **PC10** → D2,
  - **PC11** → D3,
  - **PC12** → CLK,
  - **PD2** → CMD
- In the SDMMC settings, choose **Wide Bus (4 bits)**.
- Enable **DMA** for SDMMC1 (CubeMX might automatically assign DMA for SDMMC, as SDMMC heavily relies on it for data transfer).
- NVIC: Enable the SDMMC global interrupt (for handling transfer complete or errors).
- **FATFS:** In CubeMX, go to **Middleware → FATFS** and add a FATFS with SD Card interface. This will pull in the FatFS library and link it to SDMMC1. It makes it easier to write files to the SD in our application code using `<fatfs.h>` APIs (like `f_open`, `f_write`, etc.). CubeMX will generate the disk I/O template functions that call the HAL SD driver.

Clock: Ensure the SDMMC kernel clock is 48MHz (in Clock Configuration, set PLL3 if needed and assign to SDMMC). The HAL driver will likely use that to derive the initial clock (usually starts at <400 kHz for card init, then up to 24MHz or more for data). The STM32H7 SDMMC can potentially go higher speed (with HS mode), but start with default.

Later in code, we will mount the SD card (using `f_mount`), and test writing/reading a file to verify the interface. Keep in mind SD card access should be done when the system is idle or logging at intervals, as it can be slow (use DMA and large block writes to optimize).

7. Timer Configuration for PWM Outputs (Motors & Servos)

Now we configure timers for generating PWM signals for 8 BLDC motors (for ESCs) and 4 servo outputs.

7.1 TIM1 for Motor1-4 PWM:

- Enable **TIM1** under **Timers → Configuration** in CubeMX.
- In **TIM1** settings: choose **PWM Generation (CH)** mode for channels 1, 2, 3, 4.
- Assign channels to pins:
    - TIM1_CH1 → **PA8** (Motor 4 in our list, but it's okay – we can remap logically later. For consistency, maybe assign CH1=Motor1 (PA8), CH2=Motor2 (PA9), etc. or the reverse; just note which is which.)
    - TIM1_CH2 → **PA9**,
    - TIM1_CH3 → **PA10**,
    - TIM1_CH4 → **PA11**

- Parameter settings:
    - Set **Prescaler** and **Counter Period** to get the desired PWM frequency. For drone ESCs (if using standard servo PWM signals), a frequency of **50 Hz** is typical (20 ms period). However, many ESCs can handle faster rates (e.g. 400 Hz for faster response). Let's assume 50 Hz for now to be safe for both ESCs and servos.
        - Compute timer values: TIM1 is a 16-bit advanced timer on APB2 (which might be 200 MHz if core is 400 MHz with APB2 prescaler 2). For 50 Hz: period = 20,000 µs. One approach: use a prescaler to scale down clock, then period for 50Hz. For example, with 200 MHz timer clock:
            - Prescaler = 1999 (to get 100 kHz tick), Counter period = 20000 (ticks) for 0.2s? Actually, 100kHz tick means 10 µs resolution. 20ms period = 2000 ticks. Let's do simpler: Prescaler = 1999, Counter Period = 5000 gives 0.5s... hmm, let's do precisely:
            - For 50Hz: Period = (TimerClock / 50) – 1. If TimerClock = 200 MHz, /50 = 4e6. 4e6 is above 16-bit range (65535), so use prescaler:
            - e.g. Prescaler = 199 (divide by 200), TimerClock becomes 1 MHz, then period = 1MHz/50 = 20000 counts for 20ms. 20000 fits in 16-bit. So Prescaler=199, Period=20000 gives 50 Hz with 1 µs resolution.
        - Alternatively, if we want higher frequency like 400 Hz: 1/400=2.5ms. Using 1 MHz clock ticks, 2500 counts period yields 400Hz.
        - For now, set Prescaler = 199, Counter Period = 20000 in TIM1 config for 50Hz. (We can adjust in code if needed.)
    - **PWM Mode:** ensure each channel is in PWM Generation CHx (CubeMX might label it as PWM1 mode).
    - Polarity: Active High (default).
    - Enable **ARR Preload** so that changes to period apply on update (Cube should do by default for PWM).
- Output Compare: CubeMX might allow setting initial pulse value for each channel; set 0 for now (motors off initially).
- Break/Dead-time: Not needed for driving ESC signal (we are not driving a half-bridge directly, so no complementary outputs or dead-time needed).

- NVIC: If you want, enable TIM1 update interrupt (not strictly needed just for PWM output). We might use an update interrupt later if we want to do something every PWM period (like a control loop tick at 50Hz could sync with this).

7.2 TIM3 for Motor5-8 PWM:

- Enable **TIM3**.
- Mode: PWM Generation on CH1-CH4.
- Pins assignment:
  - TIM3_CH1 → **PB4**,
  - TIM3_CH2 → **PB5**,
  - TIM3_CH3 → **PB0**,
  - TIM3_CH4 → **PB1**
  - (CubeMX will list available alt functions; choose the mapping that matches these pins, likely AF2 for TIM3 on those PB pins).
- Prescaler & Period: Use the same values as TIM1 if you want the same frequency (e.g., prescaler 199, period 20000 for 50Hz) so all motors update together frequency-wise. TIM3 is on APB1 (which might be 100 MHz if APB1 prescaler is 4? Actually H7 might have APB1 at 100MHz if 400/4, or 200MHz if 400/2 depending on config). Ensure to account for actual timer clock; however, CubeMX computes it if clock config is done.
- No need for dead-time or complementary outputs either.
- NVIC: Similarly, not required to enable interrupt unless using it for timing events.

7.3 TIM5 for Servo1-4 PWM:

- Enable **TIM5** (a 32-bit general-purpose timer).
- Mode: PWM Generation CH1-CH4.
- Pins:
  - TIM5_CH1 → **PA0** (Arm-Servo),
  - TIM5_CH2 → **PA1** (Roll-Servo),
  - TIM5_CH3 → **PA2** (Pitch-Servo),
  - TIM5_CH4 → **PA3** (Yaw-Servo)
  - (*Note:* The pinout text labels Yaw-Servo at PA3, Pitch at PA2, Roll at PA1, Arm at PA0. Depending on how you want to map channels, you could also do CH1=Yaw, etc. The exact mapping of servo function to timer channel can be decided in code later; for now just ensure those pins are configured as PWM outputs.)
- Prescaler & Period: Servos typically expect 50Hz signal (standard RC servos). We can use the same 50Hz settings as motors. If using the same frequency, it's okay. If motors were set to higher like 400Hz, you might want servos still at 50Hz (some servos can't handle high update rates). In that case, we deliberately use a different timer for servos so we can keep it at 50Hz while motors could go higher. For simplicity, keep TIM5 at 50Hz (Prescaler and Period similar as above, adjusting if TIM5 clock differs).
- NVIC: Not needed generally for just output.

After configuring these timers, CubeMX will mark the corresponding pins (PA0-PA3, PA8-PA11, PB0,1,4,5) as in use by timers. In the **Pinout view**, they'll show their function (TIMx_CHy).

Summary: We now have 12 PWM outputs configured. In code, we will use HAL functions to set the pulse widths (CCR values) to control motors and servos. The typical range for servo/ESC pulses is 1ms (1000µs) to 2ms (2000µs) high pulse out of 20ms period for 0-100% command. With our timer set to 1µs resolution (if 1MHz tick), that means CCR values of ~1000 to 2000 for min to max. We'll verify these in code.

8. Timer/Interrupt Configuration for Ultrasonic Sensors (HC-SR04)

Each HC-SR04 ultrasonic sensor uses a trigger pin (output) and an echo pin (input) to measure distance:

- We need to generate a 10µs pulse on the **trigger pins** (`PF6`, `PF7`, `PF8`) and then measure the width of the pulse on the **echo pins** (`PF10`, `PF11`, `PF12`), which goes high for a duration proportional to distance (approximately 58 µs per cm of distance).

There are two common approaches: **(A) Use Timer Input Capture** on the echo pins to directly measure the high time. **(B) Use external GPIO interrupts and read a free-running timer or micros counter.**

We will set up **Timer Input Capture** for precision:

- We can repurpose three timers (one for each sensor) or use one timer with multiple channels. The H7 has many timers, but since we already used TIM1,3,5, we still have timers like TIM2, TIM4, TIM8, TIM12, TIM13, TIM14, TIM15, TIM16, TIM17, etc. The pin mapping in the text suggests using **TIM16, TIM17, TIM13** (each has a channel 1) for triggers. However, we might use them for captures as well:
    - **TIM16**: a 16-bit timer with CH1 on PF6 possible (for trigger) and CH1 can capture echo on PF10? Unlikely, since one channel cannot be at two pins. Instead, we will separate trigger and echo handling.

Plan: Use one timer for generating periodic triggers (or trigger via software), and use input capture on a separate timer for the echoes:

- We can use a single timer (like TIM2 which is 32-bit) running at high frequency as a timebase for all ultrasonic measurements, and use external interrupts on PF10,11,12 for capture:
    - Or use three timer channels from a single timer that supports those pins as input channels. For example, **TIM8** might have channels on PF10, PF11, PF12 (we need to check alt functions).

Alternatively, simpler:

- Set up **GPIO external interrupts** on PF10, PF11, PF12 (rising and falling edges) and use a free-running microsecond timer (TIM2 32-bit).
- And use **basic timers or software** for triggers.

Given CubeMX:

- Configure **PF6, PF7, PF8** as **GPIO_Output** (no alternate function) and name them e.g. TRIG1, TRIG2, TRIG3.
- Configure **PF10, PF11, PF12** as **GPIO_EXTI** (external interrupt mode) or as timer input capture channels if available.

Let’s attempt Timer Input Capture via CubeMX:

- Enable **TIM2** (32-bit). We’ll use it as a general timebase.
- For simplicity, do not assign TIM2 pins; just set TIM2 in **Internal Clock mode** and enable one channel in input capture (even if not tied to a pin, we can trigger capture events in code by software).
- Set TIM2 prescaler to get 1 MHz (1 µs tick) like earlier (if APB1 is 100MHz, prescaler 99 gives 1MHz).

- Set TIM2 period to max (0xFFFFFFFF for 32-bit, or just leave as default max, so it free-runs and overflows rarely).
- Enable **Update Event interrupt** for TIM2 (so we can handle overflow if needed, though 32-bit at 1MHz overflows in ~4295 seconds, which is fine).
- Alternatively, we could capture directly via timer channels:
    - If TIM2 CHx can map to PF10 etc, we could route PF10 to TIM2_CH1 (for example) in CubeMX if that mapping exists. But PF10 often is TIM14_CH1 on many MCUs; not sure for H7. If available, we could do:
        - TIM2_CH1 <- PF10,
        - TIM2_CH2 <- PF11,
        - TIM2_CH3 <- PF12, But need to confirm if PF10/PF11/PF12 have TIM2 AF. Uncertain, so let's stick to EXTI for those to avoid confusion.
- **External Interrupts:** In CubeMX, for pins PF10, PF11, PF12, select GPIO Mode as "External Interrupt with Rising/Falling edge" (if available). Alternatively, set as input and later enable both edges. CubeMX typically allows selecting one edge (rising) by default, but we can handle the rest in code.
- Enable EXTI lines in NVIC: In the NVIC settings, enable interrupts for the EXTI lines corresponding to PF10, PF11, PF12. (These correspond to EXTI10, EXTI11, EXTI12, which on STM32 map by pin number. They may share interrupt vectors in groups: EXTI9_5 for 5-9, EXTI15_10 for 10-15, etc. On H7, PF10/11/12 would all fall under EXTI15_10 IRQ.)

Now, how to generate trigger pulses:

- We could manually toggle PF6/7/8 in code with `HAL_GPIO_WritePin` and `HAL_Delay` or a microsecond delay.
- Or use timers TIM13, TIM16, TIM17 as mentioned to generate a pulse:
    - For example, **TIM16** can be set up in PWM mode on PF6 (trigger1). If we configure TIM16 CH1 on PF6, we can program it to generate a pulse of 10 µs at a desired repetition rate (say 20 Hz reading per sensor to avoid interference).
    - This might be overkill for now, we can do it in software in the main loop or a periodic timer interrupt.

To keep it straightforward:

- Do **not** set trigger pins to a timer PWM in CubeMX (we'll toggle them in code).
- So PF6, PF7, PF8 remain general GPIO outputs.

Recap config for ultrasonic:

- PF6, PF7, PF8: GPIO_Output (Trigger pins, initially set Low).
- PF10, PF11, PF12: GPIO_EXTI (interrupt on both edges).
- TIM2: 32-bit timer as time base at 1MHz (for measuring echo pulse width).
- NVIC: Enable EXTI15_10 (covers PF10,11,12 interrupts) and TIM2 global interrupt if using it.

We will write code such that:

- To measure distance: set trigger high on one of PF6/7/8 for 10 µs, then low. When the echo pin goes high (EXTI rising), record TIM2 count. When it falls (EXTI falling), record TIM2 count again and compute difference = pulse duration in µs. Then distance (cm) = duration / 58 (approx, since sound travels ~0.0343 cm/µs round trip, divide by 2 gives 0.01715 cm/µs, invert ~58 µs/cm).

Code Example for EXTI Callback:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    static uint32_t echo_start[3];
    uint32_t now = __HAL_TIM_GET_COUNTER(&htim2);

    if (GPIO_Pin == GPIO_PIN_10) { // Sensor 1
        if (HAL_GPIO_ReadPin(GPIOF, GPIO_PIN_10) == GPIO_PIN_SET) {
            echo_start[0] = now; // Rising edge
        } else {
            uint32_t duration = (now >= echo_start[0]) ? (now - echo_start[0])
                                                       : (0xFFFFFFFF - echo_start[0] + now + 1);
            distance_cm_1 = duration / 58.0f;
        }
    }
    // Repeat for GPIO_PIN_11 and GPIO_PIN_12 for sensors 2 and 3.
}
```

Generate a 10 µs trigger pulse (can be done in code):

```
// Function to generate a trigger pulse for an ultrasonic sensor
void triggerUltrasonic(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin) {
    HAL_GPIO_WritePin(GPIOx, GPIO_Pin, GPIO_PIN_SET);
    Delay_us(10);  // Custom microsecond delay function
    HAL_GPIO_WritePin(GPIOx, GPIO_Pin, GPIO_PIN_RESET);
}
```

9. External Interrupts and DMA Configuration Summary

We have already enabled various interrupts and DMA in previous steps. Let's summarize and ensure they are properly set:

External Interrupts (EXTI):

- Ultrasonic Echo pins: PF10, PF11, PF12 on EXTI (both edges).
- (Optional) PPM input on PF9: If using a PPM receiver, you might set PF9 as EXTI as well. Alternatively, use a timer input capture for PPM decoding. CubeMX: PF9 -> GPIO_EXTI, NVIC enable EXTI9_5 (since PF9 is EXTI9).
- (Optional) Any user button or failsafe line if present (not in our list, aside from NRST which is hardware reset).

CubeMX will generate `HAL_GPIO_EXTI_Callback()` in stm32h7xx_it.c to handle these. We will add logic there to handle rising/falling edges for echo (we can differentiate by checking `GPIO_PIN_x` in the callback).

DMA:

- We configured DMA for:
  - USART6 RX (GPS) – e.g. DMA1 Stream something.
  - Possibly USART3 RX (if set).
  - UART8 or USART1 TX/RX if needed (not critical, up to your usage).
  - SPI4 TX (for OSD).
  - SDMMC1 (for SD card, handled by HAL SD driver).
  - (We skipped ADC DMA for now, but we might add for battery ADC).
- Check CubeMX **DMA Settings** tab: ensure each chosen DMA has a unique stream and priority. CubeMX usually auto-assigns appropriate streams. For example, if using DMA1 for some and DMA2 for others on H7 (since H7 has multiple DMA controllers).
- **Memory considerations:** On STM32H7, if DMA is used with data in D-cacheable memory (e.g., SRAM1), you must either disable caching for that region or use functions like `SCB_CleanDCache()`/`InvalidateDCache()` around DMA. A simpler way is to place DMA buffers in DTCM or SRAM2 (which are not cached). CubeIDE linker file often has a section like `.dma_buffer`. For our tutorial, we assume small data or that we manage by flushing cache when needed (for instance, after DMA fills a GPS buffer, invalidate the cache for that region before reading). Keep this in mind during coding.

NVIC Priorities: In CubeMX, under System Core → NVIC, you can set priority groupings and individual priorities. For a flight controller:

- Give high priority to time-critical interrupts like the ultrasonic echo capture or PPM input (since you want precise timing).
- Medium priority to sensor reading or DMA complete events.
- Lower priority to UART communications, etc.
- Since no RTOS, we'll mostly rely on interrupt handlers setting flags and then main loop processing.

For now, default priorities are fine, but it's good to ensure, for example, the EXTI for ultrasonic (EXTI15_10) is higher priority (lower numerical value) than, say, UART interrupts.

10. Generating Project and Code Structure Planning

At this point, all peripherals are configured in CubeMX. **Save** the .ioc file and click **GENERATE CODE**. STM32CubeIDE will generate a project with initialization code for all these peripherals in `Core/Src/main.c` and `Core/Src/stm32h7xx_hal_msp.c` and the respective peripheral `.c` files.

Now, we will organize the code in a **modular structure**. Instead of putting all logic in `main.c`, we will create separate files for different subsystems:

- **motor_control.c/h:** Functions to initialize motors/servos (if not already done in MX_Init) and to set PWM duty cycles for each motor/servo.
- **sensors.c/h:** Drivers for IMU (MPU6050), magnetometer (QMC5883L), barometer (BMP388), current sensor (INA219). Each can also be separate (mpu6050.c, etc.) if preferred.
- **osd.c/h:** Functions to communicate with the AT7456E OSD (e.g., init OSD, write characters).
- **gps.c/h:** Functions to parse GPS data (NMEA or UBX) coming on UART6.
- **bluetooth.c/h:** Handle commands from HC-05 (if any, e.g., to change settings or control the drone via a phone).
- **ppm.c/h or rc_input.c/h:** Functions to capture and interpret the PPM signal from PF9 (if used).
- **ultrasonic.c/h:** Functions for ultrasonic sensor trigger and reading distance.
- **power.c/h:** Handle the buck converter enables (PI0, PG13-15) and battery monitoring (ADC on PH2, and INA219 over I2C).
- **main.c:** Will contain `main()` which calls all init functions and then loops, calling update functions in a timely manner.

This structure helps keep each part of the code focused and manageable.

CubeMX by default puts all initialization (MX_UART_Init, MX_I2C_Init, etc.) in `main.c`. We can leave those but call them through our own init functions for clarity: For example, in `motor_control.c` we might have:

```
void MotorControl_Init(void) {
    MX_TIM1_Init();
    MX_TIM3_Init();
    MX_TIM5_Init();
    // Start PWM on all channels
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
    ...
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
    ...
    HAL_TIM_PWM_Start(&htim5, TIM_CHANNEL_1);
}
```

However, since `MX_TIMx_Init` is generated static in main.c by default, we have two options: (1) Change their scope to non-static and call externally, or (2) copy the initialization code into our module. A simpler way: Keep `MX_xxx_Init` calls in main.c but call `HAL_TIM_PWM_Start` for each channel in our `MotorControl_Init` after those calls.

Alternatively, disable "static" in CubeMX code generation settings for init funcs. For this tutorial, assume we can call them or move them.

11. Writing Initialization Code and Peripheral Drivers

Now we write the code to initialize and use each peripheral. We'll illustrate key parts:

11.1 Clock and HAL Init: CubeIDE generates `SystemClock_Config()` in main.c based on our settings. It also calls `HAL_Init()` which sets up SysTick, etc. Ensure `HAL_Init()` is at the beginning of main.

11.2 GPIO Initialization: CubeMX will generate `MX_GPIO_Init()` that sets all the GPIO modes we configured (including setting the output level for pins like sensor EN pins, trigger pins etc.). We can adjust default states:

- e.g., Set all motor PWM GPIOs to low (initial CCR=0 anyway).
- Set sensor EN pins high (to power sensors if those are active-high enables) – check schematic, likely those enable pins (PA15, PD0, PD1, PA12) might need to be set High to turn on sensors. We can do `HAL_GPIO_WritePin(GPIOx, PIN, GPIO_PIN_SET)` after initializing GPIO.
- Keep triggers low initially.

11.3 UART Drivers:

- After `MX_USART1_UART_Init()` etc., we have `huart1, huart3, huart6, huart8` handles.
- Start receiving data via interrupt or DMA:
  - For example, for GPS on UART6, we might do:

```
HAL_UART_Receive_DMA(&huart6, gpsRxBuffer, GPS_RX_BUF_LEN);
```

where `gpsRxBuffer` is a global array to hold incoming data (e.g., 256 bytes circular).

- For Bluetooth on UART3, if expecting command strings, we could use interrupt mode:

```
HAL_UART_Receive_IT(&huart3, &btRxByte, 1);
```

(receiving one byte at a time and processing in callback, or better, a DMA as well).

- For CH340 (UART8) and debug (UART1), perhaps we only transmit on those (for logging). We can still set up a receive in IT if needed for user input.

- Implement callback for UART:
  - HAL library uses `void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)` for when a UART RX is complete (either one byte in IT mode or full buffer in DMA mode).
  - For DMA circular (GPS), there is no "complete" unless buffer wraps; instead use UART Idle line detection: enable UART Idle interrupt (by enabling IRQ and in ISR check for `IDLE` flag), which triggers when no data arrives for a frame period. In `USART6_IRQHandler`, if `__HAL_UART_GET_FLAG(huart6, UART_FLAG_IDLE)`, then call `HAL_UART_IDLECallback()` or similar (HAL does not have it by default, you write it). This is advanced; as an alternative, we can process GPS data in main loop by periodically checking how much DMA has filled (using `__HAL_DMA_GET_COUNTER`).

- **GPS parsing:** Inside the `gps.c` module, parse NMEA sentences from the buffer. For instance, look for `$GNGGA` or `$GPRMC` lines in the received data, then extract latitude, longitude, etc. Ublox can also be configured to output binary UBX messages for more efficient parsing, but NMEA is easier to start with. This tutorial won't detail parsing logic, but one can use strtok or state machine to parse GPS data once a full sentence is received (they end with `\r\n`).

11.4 I2C/Sensor Drivers:

- After `MX_I2C1_Init()`, use `HAL_I2C_Master_Transmit/Receive` or implement higher-level functions:
  - Write functions to initialize each sensor:
    - **MPU6050:** Needs waking from sleep, setting sample rate, etc. For example, write 0x00 to PWR_MGMT_1 register.
    - **QMC5883L:** Set control register for continuous mode.
    - **BMP388:** Configure pressure/temperature sampling rates.
    - **INA219:** Configure calibration register for current sensing.
  - These initializations involve writing to I2C addresses. For example:

```
uint8_t buf[2];
// MPU6050: write PWR_MGMT_1 register (0x6B) = 0x00
buf[0] = 0x6B; buf[1] = 0x00;
HAL_I2C_Master_Transmit(&hi2c1, MPU6050_ADDR<<1, buf, 2, HAL_MAX_DELAY);
```

(Where `MPU6050_ADDR` is 0x68 by default. We shift left 1 for 7-bit to 8-bit address in HAL.)

- Similarly for other sensors. Modularize each sensor's init and read functions:

  - `MPU6050_ReadAccelGyro(ax, ay, az, gx, gy, gz)`
  - `BMP388_ReadPressureTemp(&pressure, &temperature)`
  - etc., using HAL_I2C to read the required registers.

- If sensors have data-ready interrupts (not in our pin list, but e.g., MPU6050 has an INT pin that could be wired to an MCU pin to signal data ready), in our schematic they did not mention it, so we'll use polling or timed reads.

Using DMA for I2C: If you plan to read sensors frequently (e.g., IMU at 1 kHz), using I2C DMA can offload the CPU. Setup I2C1 RX DMA in CubeMX if needed and then use `HAL_I2C_Master_Transmit_DMA` / `HAL_I2C_Master_Receive_DMA`. For simplicity, using blocking or interrupt mode for sensor reads is okay if done in a low frequency (like barometer at 50 Hz, magnetometer 100 Hz, etc.). The IMU might need higher rate; if so, consider reading it in an interrupt routine (e.g., TIM interrupt triggers the read).

11.5 PWM (Motor & Servo) Control Code:

- CubeMX sets up the timers, but we need to start PWM outputs:
  - After `MX_TIM1_Init`, etc., call `HAL_TIM_PWM_Start()` for each channel. For brevity, we can loop:

```
// Start TIM1 channels
for(int ch = TIM_CHANNEL_1; ch <= TIM_CHANNEL_4; ch += 0x4) {
    HAL_TIM_PWM_Start(&htim1, ch);
}
// Start TIM3 channels
for(int ch = TIM_CHANNEL_1; ch <= TIM_CHANNEL_4; ch += 0x4) {
    HAL_TIM_PWM_Start(&htim3, ch);
}
// Start TIM5 channels
for(int ch = TIM_CHANNEL_1; ch <= TIM_CHANNEL_4; ch += 0x4) {
    HAL_TIM_PWM_Start(&htim5, ch);
}
```

(Note: incrementing by 0x4 works because TIM_CHANNEL_2 is 0x4, CHANNEL_3 is 0x8, etc., but better use explicit calls.)

- To set a motor's speed (PWM duty), use `__HAL_TIM_SET_COMPARE(&htimX, channel, compareValue)`. For example, to set Motor1 (on TIM1 CH1) to a value:

```
// Motor 1 (PA8, TIM1_CH1) to 1500us pulse (assuming 1us ticks as configured)
__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, 1500);
```

This sets the CCR1 = 1500, which if period=20000, is a 7.5% duty (~1.5ms on 20ms period). Do this for each motor or servo as needed.

- We can create a function `MotorControl_SetMotorPWM(uint8_t motorIndex, uint16_t pulse_us)` that maps motor index 1-8 to the correct timer/channel and sets compare. Similarly `MotorControl_SetServoPWM(uint8_t servoIndex, uint16_t pulse_us)` for servos.

11.6 Ultrasonic Sensor Code:

- **Trigger pulse:** To generate a 10µs pulse on a trigger pin:

```
HAL_GPIO_WritePin(GPIOF, GPIO_PIN_6, GPIO_PIN_SET); // PF6 High
Delay_us(10);  // a microsecond delay function, see below
HAL_GPIO_WritePin(GPIOF, GPIO_PIN_6, GPIO_PIN_RESET); // PF6 Low
```

We need a `Delay_us` function since HAL only has ms delay. We can create one using the DWT cycle counter or using TIM2 that we set up:

- If TIM2 is running at 1 MHz, we can poll its CNT for microsecond delays (not super precise under an OS, but fine here as interrupts off).
- Or enable DWT (Data Watchpoint Timer) CYCCNT if not already enabled by HAL (some HAL init enable it), and use core clock cycles to delay.
- Simpler: use `for` loop with volatile if slight imprecision is acceptable. But given we have high speed MCU, a quick busy-wait is fine for 10µs.

- **Capturing echo:** Since we configured PF10, PF11, PF12 as EXTI:

  - In `stm32h7xx_it.c`, CubeMX will set up `EXTI15_10_IRQHandler` to call `HAL_GPIO_EXTI_IRQHandler` for the respective pins.
  - We implement `HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)` in our code (maybe in main.c or a common file):

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  static uint32_t echo_start[3]; // store start times for 3 sensors
  uint32_t now = __HAL_TIM_GET_COUNTER(&htim2);
  if(GPIO_Pin == GPIO_PIN_10) { // Echo1
    if(HAL_GPIO_ReadPin(GPIOF, GPIO_PIN_10) == GPIO_PIN_SET) {
      // Rising edge - echo pulse start
      echo_start[0] = now;
    } else {
      // Falling edge - echo pulse end
      uint32_t duration = (now >= echo_start[0]) ? (now - echo_start[0])
                                                 : ((0xFFFFFFFFu - echo_start[0]) + now + 1);
      distance_cm_1 = duration * 0.01715f; // or (duration / 58.0f);
    }
  }
  // Similarly for GPIO_PIN_11 and 12...
}
```

This captures the timer count at rising edge and computes difference at falling edge to get pulse length. We handle overflow wrap by logic above (though with 32-bit timer and short intervals, overflow not an issue unless > ~4295 sec between pulses).

- We need to start TIM2 before using (so in init, do `HAL_TIM_Base_Start(&htim2);` to run the free-running counter).
- `distance_cm_1` would be a global or static variable that holds the measured distance for sensor1. We'd have distance_cm_2 and _3 similarly.
- *Important:* The sound wave from one sensor can bounce to another if triggers are simultaneous. It's best to trigger them sequentially, not all at once. We can trigger sensor 1, measure, then trigger 2, etc., in round-robin in the main loop with some delay in between to avoid cross-talk.

• Alternatively, if using timer input capture channels, HAL would do similar internally, but our EXTI method is clear and works.

11.7 PPM Input (if used):

- If PF9 is PPM input (a combined signal of multiple RC channels):
  - Configure an input capture on a timer (like TIM14 CH1 on PF9 if available, or use EXTI like ultrasonic but we need to measure multiple pulse widths in sequence).
  - Simpler: use EXTI on PF9, measure successive high pulses via the same TIM2 counter or a separate high-res timer.
  - PPM consists of a series of pulses (0.5ms to 2ms) for each channel and a longer gap between frames. Decoding PPM can be done by measuring time between edges.
  - Implementation (briefly): On every rising edge of PPM line, record time; compute difference from last rising. If difference is large (> e.g. 5ms), that indicates start of a new frame. Otherwise, that difference is a channel value. Store it in an array of channels sequentially.
  - This can be done in HAL_GPIO_EXTI_Callback for PF9 similarly.
  - Ensure PF9's EXTI is enabled (EXTI9_5 IRQ).

11.8 OSD (AT7456E) Driver:

- Initialize the SPI interface: `HAL_SPI_Init(&hspi4);` (CubeMX already did in MX_SPI4_Init).
- Ensure chip select pin PE11 is configured as GPIO and set **high** (inactive) initially.
- Basic OSD communication example:
  - To write to a register on the AT7456E, typically you lower CS, send an address byte (with write flag), then send data, then raise CS.

```
HAL_GPIO_WritePin(GPIOE, GPIO_PIN_11, GPIO_PIN_RESET); // CS low
uint8_t cmd[2] = { regAddress | 0x80, value }; // assuming 0x80 bit for write
HAL_SPI_Transmit(&hspi4, cmd, 2, HAL_MAX_DELAY);
HAL_GPIO_WritePin(GPIOE, GPIO_PIN_11, GPIO_PIN_SET); // CS high)
```

For reading, you'd send address (without write flag), then receive data.

- The specifics depend on the AT7456E datasheet (register map is similar to MAX7456). For example, to clear screen, write 0x04 to the Clear Display register.
- In a function `OSD_WriteChar(x, y, char)`, you would set cursor position registers and then write the char code to video memory via SPI.

We won't detail OSD further, as it's a specialized chip, but ensure timing (some writes may need delays). Use DMA or interrupts if updating a lot of characters to avoid slowing the loop.

11.9 USB Power Management:

- The board has **USB power control circuitry** as per schematic (likely to switch 5V to the board and control back-powering). From the pinout:
  - It shows a section "USB Power DONE" and possibly a MOSFET arrangement. Also "MOSFET Power Control" and those buck ON/OFF pins.
  - Likely, if the board is powered via USB, we might not want to enable all loads (motors, etc.) to avoid drawing too much current from USB (500mA limit).
  - How to handle in code:
    - If there is a way to detect USB vs battery power. Possibly a voltage sensing or a pin from the power chip. Some boards have a GPIO indicating if VBUS (USB 5V) is present. If our board does, CubeMX would configure a pin for VBUS sensing (not explicitly given, but maybe the F103 ST-Link part handles it).
    - Alternatively, measure battery voltage via ADC. If battery voltage is below some threshold (or zero), we assume we're on USB only.
    - In such case, we might keep the buck converters (PI0, PG13-15) off to cut power to motors/servos while on USB, to protect the PC's USB port.
    - Implement logic in `power.c`:

```
float batt = ReadBatteryVoltage();
if(batt < 6.0f) { // assume USB (5V) and no battery (if a LiPo, would be >6V for 2S, etc.)
    // Turn off heavy loads
    HAL_GPIO_WritePin(GPIOI, GPIO_PIN_0, GPIO_PIN_RESET);  // Yaw buck off
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_13, GPIO_PIN_RESET); // Roll buck off
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_RESET); // Pitch buck off
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_15, GPIO_PIN_RESET); // Arm buck off
} else {
    // Battery present
    HAL_GPIO_WritePin(GPIOI, GPIO_PIN_0, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_13, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_15, GPIO_PIN_SET);
}
```

- Also, when on USB only, you might not want to arm the motors in software as there's no proper power. So enforce a check in arming logic (don't spin motors if battery not detected).

- If the microcontroller itself is USB capable (STM32H7 has USB OTG FS/HS), one would manage USB power in USB peripheral config. But here we rely on external CH340C, so our code doesn't interface with USB directly, only through UART.

11.10 DMA and Interrupt Enablement in Code:

- After initialization, ensure to start any DMA-based peripherals:
  - We already mentioned UART6 DMA start.
  - For SD card, the FATFS library call `f_read`/`f_write` will internally use the DMA via HAL, so no need to manually start it aside from ensuring the SD card is initialized (through `BSP_SD_Init()` or `HAL_SD_Init` which Cube does).
  - For I2C, if using DMA, call HAL_I2C...DMA as needed.
  - Check that in `stm32h7xx_it.c`, the interrupt handlers for DMA (e.g., DMAx_Streamx_IRQHandler) are calling `HAL_DMA_IRQHandler` for the correct handle (CubeMX sets this up).
  - Likewise, UART IRQ handlers call `HAL_UART_IRQHandler`, etc. These call into the callbacks we implement (RxCplt, TxCplt, etc.)

12. Modulated Code Areas

12.1 Motor Control Module Header + Src file:

The motor control module handles all operations related to driving the motors (via Electronic Speed Controllers) and servos. This module initializes the PWM outputs by starting the appropriate timers and channels (TIM1 and TIM3 for motors, TIM5 for servos). It provides higher-level functions to set the PWM duty cycle in microseconds for each motor and servo, abstracting away the low-level hardware details. This allows the flight control logic (e.g., PID controllers) to simply update motor or servo values without dealing with direct timer register manipulation.

Key Functions:

- `MotorControl_Init()`: Starts the PWM outputs on TIM1, TIM3, and TIM5 as configured in CubeMX.
- `MotorControl_SetMotorPWM(motorIndex, pulse_us)`**:** Maps a motor index (1–8) to a specific timer channel and sets its compare value, which determines the PWM pulse width.
- `MotorControl_SetServoPWM(servoIndex, pulse_us)`: Similar function for setting servo outputs.

motor_control.h

```
#ifndef MOTOR_CONTROL_H
#define MOTOR_CONTROL_H

#include "stm32h7xx_hal.h"

// Initializes PWM for motors and servos
void MotorControl_Init(void);

// Set motor PWM pulse in microseconds for motorIndex (1-8)
void MotorControl_SetMotorPWM(uint8_t motorIndex, uint16_t pulse_us);

// Set servo PWM pulse in microseconds for servoIndex (1-4)
void MotorControl_SetServoPWM(uint8_t servoIndex, uint16_t pulse_us);

#endif
```

```c
#include "motor_control.h"

// External timer handles (must be declared in your main project)
extern TIM_HandleTypeDef htim1;
extern TIM_HandleTypeDef htim3;
extern TIM_HandleTypeDef htim5;

void MotorControl_Init(void) {
    // Start PWM outputs for motors (TIM1 for motors 1-4)
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_4);

    // Start PWM outputs for motors (TIM3 for motors 5-8)
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_4);

    // Start PWM outputs for servos (TIM5 for servos 1-4)
    HAL_TIM_PWM_Start(&htim5, TIM_CHANNEL_1);
    HAL_TIM_PWM_Start(&htim5, TIM_CHANNEL_2);
    HAL_TIM_PWM_Start(&htim5, TIM_CHANNEL_3);
    HAL_TIM_PWM_Start(&htim5, TIM_CHANNEL_4);
}

void MotorControl_SetMotorPWM(uint8_t motorIndex, uint16_t pulse_us) {
    // Map motors 1-4 to TIM1 and motors 5-8 to TIM3
    if (motorIndex >= 1 && motorIndex <= 4) {
        uint32_t channel;
        switch(motorIndex) {
            case 1: channel = TIM_CHANNEL_1; break;
            case 2: channel = TIM_CHANNEL_2; break;
            case 3: channel = TIM_CHANNEL_3; break;
            case 4: channel = TIM_CHANNEL_4; break;
            default: channel = TIM_CHANNEL_1; break;
        }
        __HAL_TIM_SET_COMPARE(&htim1, channel, pulse_us);
    }
    else if (motorIndex >= 5 && motorIndex <= 8) {
        uint32_t channel;
        switch(motorIndex) {
            case 5: channel = TIM_CHANNEL_1; break;
            case 6: channel = TIM_CHANNEL_2; break;
            case 7: channel = TIM_CHANNEL_3; break;
            case 8: channel = TIM_CHANNEL_4; break;
            default: channel = TIM_CHANNEL_1; break;
        }
        __HAL_TIM_SET_COMPARE(&htim3, channel, pulse_us);
    }
}
```

12.2 Sensor Module Header + Src file:

This module is dedicated to sensor initialization and data acquisition. It covers various onboard sensors including the MPU6050 IMU, QMC5883L magnetometer, BMP388 barometer, and INA219 current sensor. Each sensor is initialized to prepare for data readout, and dedicated functions fetch sensor readings. The modular structure allows for independent calibration and future upgrades (e.g., sensor fusion algorithms) without mixing sensor code with other system logic.

Key Functions:

- `MPU6050_Init()` **and** `MPU6050_ReadAccelGyro()`: Initialize the MPU6050, wake it up and read accelerometer and gyroscopic data.
- `QMC5883L_Init()` **and** `QMC5883L_ReadMag()`: Set up and retrieve magnetometer readings.
- `BMP388_Init()` **and** `BMP388_ReadPressureTemp()`: Configure the barometric sensor and read temperature and pressure values.
- `INA219_Init()` **and** `INA219_ReadCurrent()`: Initialize and retrieve current measurements from the INA219 sensor.

sensors.h

```
#ifndef SENSORS_H
#define SENSORS_H

#include "stm32h7xx_hal.h"

// Structure to hold accelerometer and gyroscope data from MPU6050
typedef struct {
    int16_t ax;
    int16_t ay;
    int16_t az;
    int16_t gx;
    int16_t gy;
    int16_t gz;
} MPU6050_Data;

// Function prototypes for sensor initialization and data reading
void MPU6050_Init(void);
void MPU6050_ReadAccelGyro(MPU6050_Data* data);

void QMC5883L_Init(void);
void QMC5883L_ReadMag(int16_t* mx, int16_t* my, int16_t* mz);

void BMP388_Init(void);
void BMP388_ReadPressureTemp(float* pressure, float* temperature);

void INA219_Init(void);
float INA219_ReadCurrent(void);

#endif
```

sensors.c

```c
#include "sensors.h"

// External I2C handle (assumed to be hi2c1)
extern I2C_HandleTypeDef hi2c1;

// Define sensor I2C addresses (shifted left by 1 for HAL)
#define MPU6050_ADDR         (0x68 << 1)
#define MPU6050_PWR_MGMT_1   0x6B

void MPU6050_Init(void) {
    uint8_t data[2];
    data[0] = MPU6050_PWR_MGMT_1;
    data[1] = 0x00;  // Clear sleep mode to wake up the sensor
    HAL_I2C_Master_Transmit(&hi2c1, MPU6050_ADDR, data, 2, HAL_MAX_DELAY);
}

void MPU6050_ReadAccelGyro(MPU6050_Data* data) {
    uint8_t reg = 0x3B;  // Starting register for accelerometer data
    uint8_t buf[14];
    HAL_I2C_Master_Transmit(&hi2c1, MPU6050_ADDR, &reg, 1, HAL_MAX_DELAY);
    HAL_I2C_Master_Receive(&hi2c1, MPU6050_ADDR, buf, 14, HAL_MAX_DELAY);

    data->ax = (buf[0] << 8) | buf[1];
    data->ay = (buf[2] << 8) | buf[3];
    data->az = (buf[4] << 8) | buf[5];
    data->gx = (buf[8] << 8) | buf[9];
    data->gy = (buf[10] << 8) | buf[11];
    data->gz = (buf[12] << 8) | buf[13];
}

void QMC5883L_Init(void) {
    // Placeholder: Write necessary configuration for continuous measurement mode
}

void QMC5883L_ReadMag(int16_t* mx, int16_t* my, int16_t* mz) {
    // Placeholder: Read magnetometer registers using I2C and store values
    *mx = 0; *my = 0; *mz = 0;
}

void BMP388_Init(void) {
    // Placeholder: Configure BMP388 registers for pressure and temperature measurement
}

void BMP388_ReadPressureTemp(float* pressure, float* temperature) {
    // Placeholder: Read registers and compute pressure (in hPa) and temperature (°C)
    *pressure = 1013.25f;
    *temperature = 25.0f;
}

void INA219_Init(void) {
    // Placeholder: Configure INA219 for current measurement
}

float INA219_ReadCurrent(void) {
    // Placeholder: Read current data from INA219 and convert to amperes
    return 0.0f;
}
```

12.3 OSD Module Header + Src file:

The OSD module provides functions to control the AT7456E on-screen display chip. Its purpose is to overlay flight data (such as altitude, speed, or battery voltage) onto an FPV video feed. The module initializes the OSD, handles communication over SPI, and offers functions to write characters or clear the display. It abstracts the specific communication protocol so that flight data can be presented visually for the pilot's convenience.

Key Functions:

- `OSD_Init()`: Sends the initialization commands to the OSD chip, including clearing the screen and setting video overlay parameters.
- `OSD_WriteChar(x, y, ch)`: Writes a character to a specified position on the OSD.
- `OSD_ClearScreen()`: Clears the entire display.

osd.h

```h
#ifndef OSD_H
#define OSD_H

#include "stm32h7xx_hal.h"

// Initialize the OSD chip
void OSD_Init(void);

// Write a character at position (x, y) on the OSD
void OSD_WriteChar(uint8_t x, uint8_t y, char ch);

// Clear the OSD screen
void OSD_ClearScreen(void);

#endif
```

osd.c

```c
#include "osd.h"

// External SPI handle and CS (chip select) definitions—ensure these are set per your CubeMX configuration.
extern SPI_HandleTypeDef hspi4;
extern GPIO_TypeDef* OSD_CS_GPIO_Port;
extern uint16_t OSD_CS_Pin;

void OSD_Init(void) {
    // Placeholder: Initialization commands for the AT7456E OSD chip
    OSD_ClearScreen();
}

void OSD_WriteChar(uint8_t x, uint8_t y, char ch) {
    uint8_t data[2];
    // Placeholder: Prepare command data based on desired OSD protocol.
    // (For example, setting cursor position then writing character.)
    data[0] = (x & 0xFF);  // Example: x-coordinate
    data[1] = ch;          // The character to display

    // Manually control chip select: set low, transmit, then high
    HAL_GPIO_WritePin(OSD_CS_GPIO_Port, OSD_CS_Pin, GPIO_PIN_RESET);
    HAL_SPI_Transmit(&hspi4, data, 2, HAL_MAX_DELAY);
    HAL_GPIO_WritePin(OSD_CS_GPIO_Port, OSD_CS_Pin, GPIO_PIN_SET);
}

void OSD_ClearScreen(void) {
    // Placeholder: Issue command to clear the display.
    uint8_t cmd = 0x04;  // Example command code for clearing OSD
    HAL_GPIO_WritePin(OSD_CS_GPIO_Port, OSD_CS_Pin, GPIO_PIN_RESET);
    HAL_SPI_Transmit(&hspi4, &cmd, 1, HAL_MAX_DELAY);
    HAL_GPIO_WritePin(OSD_CS_GPIO_Port, OSD_CS_Pin, GPIO_PIN_SET);
}
```

12.4 GPS Module Header + Src file:

The GPS module handles data reception from a connected GPS device (typically a Ublox module) via UART (using DMA for efficiency). This module initializes GPS communication, captures incoming NMEA (or UBX) sentences, and processes them to extract key positioning data such as latitude, longitude, altitude, and time. This enables functions like return-to-home and autonomous flight modes.

Key Functions:

- `GPS_Init()`: Starts UART reception in DMA mode to continuously capture GPS data.
- `GPS_ProcessData()`: Processes the circular DMA buffer to parse complete NMEA sentences and extract relevant information.

gps.h

```h
#ifndef GPS_H
#define GPS_H

#include "stm32h7xx_hal.h"

// Initialize GPS module reception via DMA
void GPS_Init(void);

// Process incoming GPS data (parse NMEA or UBX messages)
void GPS_ProcessData(void);

#endif
```

gps.c

```c
#include "gps.h"
#include <string.h>
#include <stdlib.h>

#define GPS_BUFFER_SIZE 256
char gpsBuffer[GPS_BUFFER_SIZE];
// Note: In a DMA circular buffer setup, you may use indices to keep track
volatile uint16_t gpsWriteIndex = 0;

extern UART_HandleTypeDef huart6;

void GPS_Init(void) {
    // Start DMA reception in circular mode for continuous GPS data.
    HAL_UART_Receive_DMA(&huart6, (uint8_t*)gpsBuffer, GPS_BUFFER_SIZE);
}

void GPS_ProcessData(void) {
    // Simple example: search for a $GNGGA sentence in the buffer.
    char* pStart = strstr(gpsBuffer, "$GNGGA");
    if (pStart != NULL) {
        char* pEnd = strstr(pStart, "\r\n");
        if (pEnd != NULL) {
            size_t sentenceLength = pEnd - pStart;
            char sentence[100];
            if(sentenceLength < sizeof(sentence)) {
                strncpy(sentence, pStart, sentenceLength);
                sentence[sentenceLength] = '\0';
                // Process the sentence: extract latitude, longitude, etc.
                // For demonstration, you might print it via UART1 for debugging.
                // printf("GPS: %s\n", sentence);
            }
        }
    }
}
```

12.5 Bluetooth Module Header + Src file:

The Bluetooth module manages communication with the HC-05 Bluetooth unit, which is used for wireless telemetry or control commands from a mobile device or PC. It initializes UART reception for the Bluetooth module and contains routines for parsing incoming commands. This module allows for non-intrusive adjustments and monitoring of flight data.

Key Functions:

- `Bluetooth_Init()`: Sets up UART (or DMA, if preferred) for Bluetooth communication using interrupt mode.
- `Bluetooth_ProcessCommand()`: Processes any received Bluetooth data to interpret control or configuration commands.

bluetooth.h

```
#ifndef BLUETOOTH_H
#define BLUETOOTH_H

#include "stm32h7xx_hal.h"

// Initialize Bluetooth (HC-05) on its UART
void Bluetooth_Init(void);

// Process any received Bluetooth commands
void Bluetooth_ProcessCommand(void);

#endif
```

bluetooth.c

```
#include "bluetooth.h"

extern UART_HandleTypeDef huart3;
volatile uint8_t btRxByte;

void Bluetooth_Init(void) {
    // Start UART reception for Bluetooth commands (using interrupt mode)
    HAL_UART_Receive_IT(&huart3, &btRxByte, 1);
}

void Bluetooth_ProcessCommand(void) {
    // Placeholder: If a full command has been received, parse and act on it.
    // For example, you might toggle a flight mode or adjust PID parameters.
}
```

12.6 rc_input(ppm) Module Header + Src file:

Designed to capture radio control (RC) signals, this module reads in either a combined PPM signal or serial RC data (e.g., SBUS) from the receiver. It uses either external interrupts or timer input capture to decode pulse width values for individual channels. This module maps the RC inputs to usable channel data that the flight controller uses for manual control and stabilization.

Key Functions:

- `RC_Input_Init()`: Configures the necessary EXTI lines or timer channels for receiving RC signals.
- `Global Channel Array (rcChannels[])`: Stores the pulse-width values for each RC channel.
- `HAL_GPIO_EXTI_Callback()`: Captures pulse edges from the RC input and decodes channel values.

rc_input.h (or ppm.h)

```
#ifndef RC_INPUT_H
#define RC_INPUT_H

#include "stm32h7xx_hal.h"

#define RC_CHANNELS 8

// Array to hold RC channel values (in microseconds or as appropriate)
extern volatile uint16_t rcChannels[RC_CHANNELS];

// Initialize RC input reception (e.g., PPM)
void RC_Input_Init(void);

#endif
```

rc_input.c

```
#include "rc_input.h"

volatile uint16_t rcChannels[RC_CHANNELS] = {0};

// Example using an external interrupt on PF9 for a PPM signal.
volatile uint32_t ppm_last_tick = 0;

void RC_Input_Init(void) {
    // Initialization of EXTI for PF9 is assumed to be done in CubeMX.
}

// Callback for EXTI interrupt (to be called from HAL_GPIO_EXTI_Callback)
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == GPIO_PIN_9) {  // assuming PF9 is the PPM input
        uint32_t now = HAL_GetTick(); // use a microsecond counter for more precision in real code
        uint32_t pulseWidth = now - ppm_last_tick;
        ppm_last_tick = now;

        // For a real PPM decoder, check pulse width to differentiate between channel pulse and frame
delimiter
        static uint8_t channel = 0;
        if (pulseWidth > 5) { // threshold value (ms placeholder)
            rcChannels[channel] = pulseWidth;
            channel = (channel + 1) % RC_CHANNELS;
        }
    }
}
```

12.7 Ultrasonic Module Header + Src file:

The ultrasonic module provides functions to trigger and capture distance measurements from ultrasonic sensors (e.g., HC-SR04). It uses general-purpose GPIO to output a trigger pulse and external interrupts (or timer input capture) to record the echo pulse width via a free-running microsecond timer. The duration of the echo pulse is then converted into distance (typically using a conversion factor of roughly 58 µs per centimeter).

Key Functions:

- `Ultrasonic_Init()`: Starts the necessary timer used for measuring echo pulse duration.
- `Ultrasonic_Trigger(sensorIndex)`: Generates a 10 µs pulse on the designated trigger pin.
- `Ultrasonic_GetDistance(sensorIndex)`: Reads the echo pulse duration captured by the interrupt callback and converts it to a distance in centimeters.
- `HAL_GPIO_EXTI_Callback()`**:** Processes rising and falling edge interrupts to capture start and end times for each sensor's echo signal.

ultrasonic.h

```
#ifndef ULTRASONIC_H
#define ULTRASONIC_H

#include "stm32h7xx_hal.h"

// Initialize ultrasonic sensor functions
void Ultrasonic_Init(void);

// Trigger an ultrasonic sensor (sensorIndex: 0-2)
void Ultrasonic_Trigger(uint8_t sensorIndex);

// Retrieve the measured distance in centimeters for the specified sensor (if ready)
float Ultrasonic_GetDistance(uint8_t sensorIndex);

#endif
```

```c
#include "ultrasonic.h"
#include "stm32h7xx_hal.h"

extern TIM_HandleTypeDef htim2;  // Free-running timer at 1 MHz

volatile uint32_t ultrasonic_start[3] = {0};
volatile uint32_t ultrasonic_duration[3] = {0};
volatile uint8_t ultrasonic_ready[3] = {0};

void Ultrasonic_Init(void) {
    // Start timer TIM2 as a free-running counter (configured for 1 µs resolution)
    HAL_TIM_Base_Start(&htim2);
}

void Ultrasonic_Trigger(uint8_t sensorIndex) {
    GPIO_TypeDef* port = GPIOF;
    uint16_t pin;
    if(sensorIndex == 0)
        pin = GPIO_PIN_6;
    else if(sensorIndex == 1)
        pin = GPIO_PIN_7;
    else if(sensorIndex == 2)
        pin = GPIO_PIN_8;
    else
        return;

    // Generate a 10 µs pulse on the trigger pin
    HAL_GPIO_WritePin(port, pin, GPIO_PIN_SET);
    // Simple delay loop (for demonstration; replace with a proper microsecond delay if available)
    for(volatile int i = 0; i < 50; i++);
    HAL_GPIO_WritePin(port, pin, GPIO_PIN_RESET);
}

float Ultrasonic_GetDistance(uint8_t sensorIndex) {
    // Compute distance in cm: duration (in microseconds) divided by 58 approx.
    if(ultrasonic_ready[sensorIndex]) {
        ultrasonic_ready[sensorIndex] = 0;
        return ((float)ultrasonic_duration[sensorIndex]) / 58.0f;
    }
    return 0.0f;
}

// EXTI callback for handling echo signal edges.
// This function should be invoked in the HAL_GPIO_EXTI_Callback in your project.
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    uint32_t now = __HAL_TIM_GET_COUNTER(&htim2);
    if(GPIO_Pin == GPIO_PIN_10) {  // Echo for sensor 0
        if(HAL_GPIO_ReadPin(GPIOF, GPIO_PIN_10) == GPIO_PIN_SET) {
            ultrasonic_start[0] = now;
        } else {
            ultrasonic_duration[0] = (now >= ultrasonic_start[0]) ?
                                     (now - ultrasonic_start[0]) :
                                     ((0xFFFFFFFF - ultrasonic_start[0]) + now + 1);
            ultrasonic_ready[0] = 1;
        }
    }
    else if(GPIO_Pin == GPIO_PIN_11) {  // Echo for sensor 1
        if(HAL_GPIO_ReadPin(GPIOF, GPIO_PIN_11) == GPIO_PIN_SET) {
            ultrasonic_start[1] = now;
        } else {
            ultrasonic_duration[1] = (now >= ultrasonic_start[1]) ?
                                     (now - ultrasonic_start[1]) :
                                     ((0xFFFFFFFF - ultrasonic_start[1]) + now + 1);
            ultrasonic_ready[1] = 1;
        }
    }
    else if(GPIO_Pin == GPIO_PIN_12) {  // Echo for sensor 2
        if(HAL_GPIO_ReadPin(GPIOF, GPIO_PIN_12) == GPIO_PIN_SET) {
            ultrasonic_start[2] = now;
        } else {
            ultrasonic_duration[2] = (now >= ultrasonic_start[2]) ?
                                     (now - ultrasonic_start[2]) :
                                     ((0xFFFFFFFF - ultrasonic_start[2]) + now + 1);
            ultrasonic_ready[2] = 1;
        }
    }
}
```

12.8 Power Module Header + Src file:

This module oversees battery monitoring and controlling buck converter outputs. It reads the battery voltage using the ADC (after an appropriate voltage divider) and provides a function to control the enable pins of the buck converters. This ensures that the flight controller and peripherals receive stable and regulated voltage, and it also helps prevent the motors from being armed when under USB-only power conditions.

Key Functions:

- `Power_Init()`: Carries out any additional power management initialization.
- `Power_ReadBatteryVoltage()`: Uses ADC measurements to calculate and return the battery voltage.
- `Power_SetBuckEnable(buckID, state)`: Controls the output enable for buck converters based on your hardware mapping

power.h

```
#ifndef POWER_H
#define POWER_H

#include "stm32h7xx_hal.h"

// Initialize power management functions
void Power_Init(void);

// Read the battery voltage (returns voltage in Volts)
float Power_ReadBatteryVoltage(void);

// Control buck converter enables; buckID from 0 to 3 and state 0 (off) or 1 (on)
void Power_SetBuckEnable(uint8_t buckID, uint8_t state);

#endif
```

power.c

```
#include "power.h"

// External ADC handle (assumed to be configured for battery voltage)
extern ADC_HandleTypeDef hadc1;

// Example conversion factors: adjust according to your ADC resolution, Vref, and voltage divider ratio.
float Power_ReadBatteryVoltage(void) {
    uint32_t adcValue = 0;
    HAL_ADC_Start(&hadc1);
    if (HAL_ADC_PollForConversion(&hadc1, 10) == HAL_OK) {
        adcValue = HAL_ADC_GetValue(&hadc1);
    }
    HAL_ADC_Stop(&hadc1);
    // For a 12-bit ADC (0-4095), assuming 3.3V reference and a divider of 2:
    float voltage = ((float)adcValue) * (3.3f / 4095.0f) * 2.0f;
    return voltage;
}

void Power_SetBuckEnable(uint8_t buckID, uint8_t state) {
    // Example mapping:
    // buckID 0 => PI0, 1 => PG13, 2 => PG14, 3 => PG15
    GPIO_TypeDef* port;
    uint16_t pin;
    switch(buckID) {
        case 0: port = GPIOI; pin = GPIO_PIN_0; break;
        case 1: port = GPIOG; pin = GPIO_PIN_13; break;
        case 2: port = GPIOG; pin = GPIO_PIN_14; break;
        case 3: port = GPIOG; pin = GPIO_PIN_15; break;
        default: return;
    }
    HAL_GPIO_WritePin(port, pin, (state ? GPIO_PIN_SET : GPIO_PIN_RESET));
}
```

12.9 Main Application (main.c) – Overall System Integration:

With all subsystems initialized, we design the main loop to continuously handle sensor updates, control, and communication. A possible structure:

```c
#include "main.h"
#include "motor_control.h"
#include "sensors.h"
#include "osd.h"
#include "gps.h"
#include "bluetooth.h"
#include "rc_input.h"    // also known as ppm.c/h if you prefer that naming
#include "ultrasonic.h"
#include "power.h"
#include "fatfs.h"       // FatFS for SD card file system

// Global variables for sensor data
MPU6050_Data imuData;
int16_t magX, magY, magZ;
float pressure, temperature;
float batteryVoltage;

// External handles (generated by CubeMX)
extern TIM_HandleTypeDef htim1;
extern TIM_HandleTypeDef htim2;
extern TIM_HandleTypeDef htim3;
extern TIM_HandleTypeDef htim5;
extern UART_HandleTypeDef huart1;
extern UART_HandleTypeDef huart3;
extern UART_HandleTypeDef huart6;
extern UART_HandleTypeDef huart8;
extern ADC_HandleTypeDef hadc1;
extern I2C_HandleTypeDef hi2c1;
extern SPI_HandleTypeDef hspi4;

// File system objects
FATFS SDFatFS;
char SDPath[4];  // Path to SD card (e.g., "0:")

int main(void)
{
    /* Reset of all peripherals, initialize the Flash interface and the Systick. */
    HAL_Init();
    SystemClock_Config();  // Generated clock configuration from CubeMX

    /* Initialize all configured peripherals (these functions are generated by CubeMX) */
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_I2C1_Init();
    MX_SPI4_Init();
    MX_USART1_UART_Init();  // Debug/UART1
    MX_USART3_UART_Init();  // Bluetooth HC-05
    MX_USART6_UART_Init();  // GPS module
    MX_UART8_Init();        // CH340C USB-to-UART
    MX_SDMMC1_SD_Init();
    MX_FATFS_Init();
    MX_TIM1_Init();         // PWM for motors 1-4
    MX_TIM3_Init();         // PWM for motors 5-8
    MX_TIM5_Init();         // PWM for servos
    MX_TIM2_Init();         // Free-running timer (e.g., for ultrasonic timing)
    MX_ADC1_Init();         // ADC for battery voltage
```

↓
Continued Below

↓

Continued Here

```
/* Start free-running timer for ultrasonic measurement at 1 µs resolution */
    HAL_TIM_Base_Start(&htim2);

    /* Initialize individual modules */
    MotorControl_Init();     // Starts PWM outputs on TIM1, TIM3 and TIM5 channels.
    GPS_Init();              // Starts UART6 reception in DMA mode.
    Bluetooth_Init();        // Begins UART3 interrupt mode reception.
    RC_Input_Init();         // Sets up EXTI on the designated RC/PPM input (e.g., PF9).
    Ultrasonic_Init();       // Starts the free-running timer used for measuring echo timing.
    Power_Init();            // Additional power monitoring setup if required.

    /* Enable sensors using dedicated GPIO pins (ensure these match your schematic) */
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_SET); // Enable MPU6050
    HAL_GPIO_WritePin(GPIOD, GPIO_PIN_0, GPIO_PIN_SET);  // Enable BMP388
    HAL_GPIO_WritePin(GPIOD, GPIO_PIN_1, GPIO_PIN_SET);  // Enable QMC5883L
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12, GPIO_PIN_SET); // Enable HC-05

    /* Initialize sensor drivers */
    MPU6050_Init();
    QMC5883L_Init();
    BMP388_Init();
    INA219_Init();

    /* Initialize OSD for FPV overlay */
    OSD_Init();

    /* Mount the SD card for logging purposes */
    if (f_mount(&SDFatFS, SDPath, 1) != FR_OK) {
        // Handle error (e.g., print a message via UART1)
    } else {
        // SD Card mounted successfully
    }

    /* Optional delay to allow sensors and peripherals to stabilize */
    HAL_Delay(500);

    /* Main loop variables for scheduling tasks */
    uint32_t lastSensorTime = HAL_GetTick();
    uint32_t lastGPSProcessTime = HAL_GetTick();
    uint32_t lastUltraTime = HAL_GetTick();
```

↓

Continued Below

↓
Continued Here

```
while (1)
    {
        uint32_t now = HAL_GetTick();

        /* Sensor reading (e.g., IMU) at ~500 Hz (every 2 ms) */
        if (now - lastSensorTime >= 2) {
            lastSensorTime = now;
            MPU6050_ReadAccelGyro(&imuData);
            // Note: Add sensor fusion algorithms (Kalman, complementary) as needed.
        }

        /* GPS data processing every 100 ms */
        if (now - lastGPSProcessTime >= 100) {
            lastGPSProcessTime = now;
            GPS_ProcessData();
        }

        /* Update battery voltage via ADC */
        batteryVoltage = Power_ReadBatteryVoltage();

        /* Trigger ultrasonic sensor (example: sensor 0) every 60 ms */
        if (now - lastUltraTime >= 60) {
            lastUltraTime = now;
            Ultrasonic_Trigger(0);
        }
        float distance0 = Ultrasonic_GetDistance(0);
        // Use 'distance0' as needed in your obstacle avoidance or landing algorithms.

        /* Motor control update:
         * In a full flight controller, here you'd perform PID calculations using sensor data,
         * compute the desired motor outputs, and update PWM values accordingly.
         * For now, setting all motor outputs to a neutral 1500 µs value.
         */
        for (uint8_t i = 1; i <= 8; i++) {
            MotorControl_SetMotorPWM(i, 1500);
        }

        /* Servo control update: set servos to neutral */
        for (uint8_t i = 1; i <= 4; i++) {
            MotorControl_SetServoPWM(i, 1500);
        }

        /* Process any incoming Bluetooth commands, if available */
        Bluetooth_ProcessCommand();

        /* Additional tasks can be inserted here such as:
         * - Logging telemetry data to the SD card
         * - Updating OSD overlays with real-time flight data
         * - Handling RC/PPM inputs and switching flight modes.
         */

    } // End while loop
}
```

The above pseudocode illustrates how the main loop might manage periodic tasks. We use `HAL_GetTick()` (ms ticks via SysTick) for scheduling slower events (since we didn't set up a dedicated scheduler). High-rate tasks like IMU read use small delays (2ms). For even higher rates, one could use a timer interrupt to get consistent timing (e.g., use TIM7 as a 1kHz interrupt to trigger IMU read).

Modularity: In practice, each section (IMU read, baro read, etc.) can be refactored to functions in their modules, and the main loop just calls those if ready. For example, have `Sensors_Update()` that handles calling each sensor's read at appropriate rate (perhaps using internal static counters or an RTOS-like ticker). Without RTOS, careful use of timers or tick counts can achieve multi-rate looping.

No RTOS considerations: We rely on the main loop and interrupts. Make sure none of the HAL functions block for too long. I2C and SPI HAL calls by default are blocking with timeouts; they will poll until done. If using them in the main loop as above, it's okay if the timeouts are not hit. Alternatively, use the _IT or _DMA versions to do non-blocking.

Safety and Fail-safes: This is beyond scope of tutorial, but always consider:

- **Watchdog Timer:** Enable an Independent Watchdog to reset the MCU if the code hangs (CubeMX can set IWDG).
- **Calibration:** Ensure to calibrate sensors (e.g., gyros) on startup (collect bias for a few seconds).
- **Sensor Fusion:** If building a real drone controller, you'd implement a filter for attitude (like using accel + gyro in complementary or Kalman filter), and a PID control for stability. Those would be separate modules as well, feeding into motor outputs.
- **Power checks:** As mentioned, do not run motors if on USB power only.

12.10 FatFS (SD Card Logging) optional:

Although integrated primarily through CubeMX middleware, the FatFS module enables file system operations on a microSD card. It allows logging of telemetry data, sensor data, or other flight-related logs. This module works alongside the SDMMC peripheral and uses DMA transfers to optimize data throughput. The functions provided by FatFS (such as f_mount, f_open, f_write, and f_close) are standard and can be incorporated into your logging routines within the main loop or a dedicated logging task.

Key Considerations:

- **Initialization in CubeMX:** FatFS is configured to mount the SD card automatically.
- **Non-Blocking Operation:** Logging operations are scheduled periodically or on-demand to avoid interfering with real-time control.
- **Data Logging:** Useful for post-flight analysis and debugging of flight behavior.

13. Conclusion and Next Steps

By following this tutorial, you have configured all major peripherals of the STM32H743-based flight controller:

- UARTs for Bluetooth, GPS, PC communication, and debugging.
- I2C for IMU, magnetometer, barometer, current sensor, and possibly smart buck converters.
- SPI for the on-screen display module.
- SDMMC for logging to an SD card.
- Timers for PWM outputs to motors (8 ESCs) and servos (4 channels).
- GPIO and timers for ultrasonic distance sensors.
- External interrupts and DMA to efficiently handle sensor inputs and communication.
- A structured approach to initialization and main loop design without an RTOS.

From here, you can proceed to implement the flight control algorithms:

- Use sensor data (orientation from IMU, heading from magnetometer, altitude from barometer, position from GPS, distance from ultrasonic) to make control decisions.
- Use RC input (PPM or via Bluetooth commands) to control the drone.
- Adjust motor speeds via PWM to stabilize and move the drone as commanded.
- Display important info on the OSD for FPV (like horizon line, battery, etc.).
- Log data to SD for analysis of flight performance.