以下範例為將前面所有內容整合後的 **Markdown** 文件示例，可作為專案文件的基礎結構。你可依需求增添或修改內容，讓整體更貼合實際專案需求。

專案結構與檔案功能說明

資料夾 / 檔案一覽

1. startup/ （例：main.c）

主要功能：

- 系統初始化與主程式入口：呼叫 ChibiOS RTOS 初始化（`halInit()`、`chSysInit()`）
- 建立 / 啟動應用層執行緒（Threads）

互動關係：

- 透過 `boards/` 進行硬體與時脈設定
- 呼叫 `library/ChibiOS_17.6.2/` 中的 HAL API 完成初始化
- 啟動 `application/`（感測器、控制、通訊）模組

---

2. application/

主要功能：

- VCU 應用層邏輯：包含感測器讀取、車輛控制、通訊等主要功能

互動關係：

- 與 `library/ChibiOS_17.6.2/` 互動：透過 HAL 函式（UART、CAN…）進行硬體通訊
- 與 `sensors/`、`control/`、`communication/` 子模組彼此協同

2.1 application/sensors/

- **主要功能**：感測器（如 GPS、速度、溫度等）讀取邏輯
- **互動關係**：
    - 使用 `hal_serial.c`（UART）或 `hal_can.c`（CAN）等驅動收集感測資料
    - 向 `control/` 模組提供感測器資料

2.2 application/control/

- **主要功能**：車輛控制邏輯與策略（如油門 / 扭力控制、煞車策略、圈數計算等）
- **互動關係**：
    - 從 `sensors/` 取得感測資訊（速度、位置等）
    - 演算法後透過 `communication/` 或直接呼叫 CAN / PWM 等 HAL 函式輸出控制指令

2.3 application/communication/

- **主要功能**：通訊模組：負責與其他 ECU (BMS、MCU)、顯示器或外部工具的資料交換
- **互動關係**：
    - 依靠 `hal_can.c`（CAN Bus）或 `hal_serial.c`（UART）進行收 / 發
    - 與 `control/`、`sensors/` 分享 / 交換通訊資料

---

3. library/ChibiOS_17.6.2/

主要功能：

- ChibiOS RTOS 核心：多工排程、Thread 管理、同步機制等
- 硬體抽象層 (HAL)：UART、CAN、PWM、I2C、SPI…驅動

互動關係：

- `startup/` 呼叫 `halInit()`、`chSysInit()` 等啟動 RTOS
- `application/` 各模組呼叫 HAL API 與硬體交互

3.1 library/ChibiOS_17.6.2/hal_serial.c

- **主要功能**：串列埠 (UART) 驅動：負責初始化 UART 及收 / 發字元函式
- **互動關係**：
    - 被 `sensors/`（GPS 讀取）或 `communication/`（UART 通訊）等模組使用
    - 與 `board.c` 中的 UART PIN 設定相互搭配

3.2 library/ChibiOS_17.6.2/hal_can.c

- **主要功能**：CAN Bus 驅動：初始化、設定濾波器 / ID、資料收發緩衝處理
- **互動關係**：
    - 被 `application/communication/`（車用 CAN）、`control/`（發送控制指令）呼叫
    - 與 `boards/` 的 CAN PIN 設定相互搭配

---

4. boards/ （例：board.c, board.h）

主要功能：

- 板級支援 (BSP)：定義 MCU 腳位、時脈、周邊初始化等

互動關係：

- ChibiOS RTOS 啟動時會載入此處設定
- 提供 GPIO、時脈頻率 等基礎參數，供 `library/ChibiOS_17.6.2/` HAL 驅動、`startup/` 使用

---

5. archieve/ （例：digital_io.c, speed_io.c）

主要功能：

- 早期專案或範例檔：可包含數位 IO、速度偵測等範例程式

互動關係：

- 可能已被更新版本取代
- 若需參考舊函式或功能，可從此匯入至新系統結構中

硬體與架構相關

1. SoC 整合

- 整合 CPU、GPU、NPU、週邊、記憶體控制器

2. MCU (Microcontroller)

- MCU = CPU + Flash + RAM + I/O
- MCU 內含 CPU 核心 (ARM Cortex-M、8051、AVR、PIC、RISC-V)

3. 電路與物理層

3.1 PCB 結構

- 多層板設計：確保高密度元件安裝與關鍵訊號的走線長度控制
- 接地平面 (Ground Plane) 與電源分層：降低雜訊耦合與阻抗不匹配
- 差分對 (Differential Pair) 與等長匹配 (Length Matching) 等高速訊號設計
- 降低過孔 (Via) 或適度安排過孔位置，以減少訊號折返與阻抗變化

3.2 EMC / EMI

- 針對高速訊號與電源回路的排版與過孔設計，避免產生大面積輻射
- 加入必要的濾波與屏蔽：如在通訊線上加 RC 濾波、在金屬外殼或關鍵區域做屏蔽
- 測試與認證：進行輻射 / 傳導干擾量測，確保符合國際或地區標準 (如 CE、FCC)
- 去耦電容（Decoupling Capacitors）放置在關鍵 IC 的電源腳旁，降低電壓漣波

---

4. 處理器架構

- 五大硬體組成 (ALU、CU、Memory、Input、Output)
- **存儲程式概念 (Stored-Program Concept)**
- **哈佛架構**
    - 指令與數據分離不同匯流排 (Instruction Bus、Data Bus)
- **馮紐曼架構**
    - 共用同一組匯流排，但一般會搭配快取分離 (I-Cache / D-Cache)

感測器與定位技術

- **OpenRTK330**（高精度 GNSS/INS 定位模組）
- **Ublox ZED-F9P**（GNSS 接收器晶片）
- 感測器和 MCU 比較（量測 vs. 控制 / 運算）
- **RTK**（Real-Time Kinematic，高精度衛星定位技術）
- **IMU / MEMS**（慣性感測器，用於姿態與動態量測）
- 以 GPS 訊號計算圈數的演算法概念（Line-Crossing / Geofencing）

圈數判斷演算法示例（C 語言）

以下提供兩個使用 C 語言撰寫的圈數判斷演算法範例，並在程式中附上詳細註解。主要差異在於：「偵測跨越一條虛擬線」或「進出一個設定好的區域」。

---

1. 線段交叉法 (Line-Crossing)

原理與步驟：

1. 在空間中定義一條「虛擬線」作為檢測邊界 (例如：賽道起終點)。

2. 連續讀取載具位置，若「舊位置」和「新位置」分別位於線的兩側，且符合條件 (如方向、速度)，則判定為已跨越線段。
3. 可用一個門檻距離 (threshold) 來避免在邊界附近小幅晃動時重複觸發。

```c
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

/*
 * 函式: lineCrossed
 * 目的: 判斷從上一個位置 (x_prev, y_prev) 到當前位置 (x_curr, y_curr)
 *       是否跨越了由 (x1, y1) ~ (x2, y2) 定義的虛擬線。
 * 原理: 利用向量叉積 (cross product) 來判斷兩點是否落在虛擬線的不同側。
 */
bool lineCrossed(double x_prev, double y_prev,
                 double x_curr, double y_curr,
                 double x1, double y1,
                 double x2, double y2)
{
    // side() 計算位置點與虛擬線的「側性」
    // 若兩個位置的側性符號不同 (相乘 < 0)，代表一個點在線的一側，另一個在相反側
    double side1 = (x_prev - x1) * (y2 - y1) - (y_prev - y1) * (x2 - x1);
    double side2 = (x_curr - x1) * (y2 - y1) - (y_curr - y1) * (x2 - x1);

    return (side1 * side2 < 0);
}

/*
 * 函式: distanceToLine
 * 目的: 計算點 (x, y) 到虛擬線 (x1, y1) ~ (x2, y2) 的最短距離 (垂直距離)。
 * 原理: 二維空間中，點到直線距離可利用向量投影與絕對值運算求得。
 */
double distanceToLine(double x, double y,
                      double x1, double y1,
                      double x2, double y2)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    double norm = sqrt(dx * dx + dy * dy);

    // 若線段定義實際上是同一個點，則直接回傳點距離
    if (norm == 0.0) {
        return sqrt((x - x1) * (x - x1) + (y - y1) * (y - y1));
```

```
    }
    // 計算點到無限長直線的垂直距離
    // 通過向量方程: 距離 = |dy*x - dx*y + x2*y1 - y2*x1| / sqrt(dx^2 + dy^2)
    return fabs(dy * x - dx * y + x2 * y1 - y2 * x1) / norm;
}

int main(void)
{
    // 模擬一系列 GPS 位置 (x, y), 假設載具向右上方移動
    double positions[][2] = {
        {0.0, 0.0},
        {0.5, 0.1},
        {1.0, 0.2},
        {1.5, 0.3},
        {2.0, 0.4},
        {2.5, 0.5},
        {3.0, 1.0},
        {3.5, 1.5},
        {4.0, 2.0},
        {4.5, 2.5},
        {5.0, 3.0},
        {5.5, 3.5}
    };
    int numPositions = sizeof(positions) / sizeof(positions[0]);

    // 定義虛擬線: 例如 (2, 0) ~ (2, 4), 一條垂直向上的線
    double x1 = 2.0, y1 = 0.0;
    double x2 = 2.0, y2 = 4.0;

    int lapCount = 0;           // 計數器, 累計跨越次數
    bool crossingFlag = false; // 避免在邊界附近重複觸發

    // 記錄前一時刻位置 (初始化為陣列第一個點)
    double x_prev = positions[0][0];
    double y_prev = positions[0][1];

    // 設定離線段多遠就代表「已經離開了起點線區域」
    double threshold = 0.5;
```

```
    // 從第 2 筆資料開始讀取，與前一筆比較
    for (int i = 1; i < numPositions; i++) {
        double x_curr = positions[i][0];
        double y_curr = positions[i][1];

        // 若跨越虛擬線且尚未進入「已跨越」狀態
        if (lineCrossed(x_prev, y_prev, x_curr, y_curr, x1, y1, x2, y2)
            && !crossingFlag)
        {
            lapCount++;
            crossingFlag = true;
            printf("Lap detected at position (%.2f, %.2f), lapCount = %d\n",
                   x_curr, y_curr, lapCount);
        }
        else {
            // 若當前位置與線段的距離大於某門檻值，表示已離開線段區域
            if (distanceToLine(x_curr, y_curr, x1, y1, x2, y2) > threshold) {
                crossingFlag = false;
            }
        }

        x_prev = x_curr;
        y_prev = y_curr;
    }

    printf("Final lap count = %d\n", lapCount);

    return 0;
}
```

方法特點

- **適用場景：** 有明確的「起終點線」或必須精準跨越某一條路徑的應用 (如賽道計時)。
- **需要注意：** 載具若在邊界處長時間停留或震動，需要加入距離閾值、方向判斷等機制。

---

2. 區域判定法 (Geofencing)

原理與步驟：

1. 在空間中定義一個「小範圍區域」，例如一個圓形作為起終點或檢測區。
2. 每次讀取定位座標，若從區外進入該區域，就代表成功通過並計算一次。
3. 從該區離開後才允許下一次計算。

```c
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

/*
 * 函式: distanceBetween
 * 目的: 計算 (x, y) 與 (xc, yc) 之間的歐幾里得距離。
 * 用途: 判斷是否進入或離開定義好的圓形區域。
 */
double distanceBetween(double x, double y, double xc, double yc)
{
    double dx = x - xc;
    double dy = y - yc;
    return sqrt(dx * dx + dy * dy);
}

int main(void)
{
    // 模擬一系列 GPS 位置 (x, y)
    double positions[][2] = {
        {0.0, 0.0},
        {0.5, 0.5},
        {1.0, 1.0},
        {1.5, 1.5},
        {2.0, 2.0},   // 進入區域
        {2.1, 2.1},
        {2.2, 2.2},
        {2.5, 2.5},   // 區域內
        {3.0, 3.0},   // 離開區域
        {2.0, 2.0}    // 再次進入
    };
    int numPositions = sizeof(positions) / sizeof(positions[0]);

    // 定義圓形區域: 中心 (2,2), 半徑 0.5 (單位: 公尺或對應座標)
    double xc = 2.0, yc = 2.0;
    double zoneRadius = 0.5;

    int lapCount = 0;     // 累計進入次數
    bool inZone = false;  // 紀錄目前是否在區域內
```

```
    for (int i = 0; i < numPositions; i++) {
        double x = positions[i][0];
        double y = positions[i][1];

        // 計算當前位置與圓形區域中心的距離
        double dist = distanceBetween(x, y, xc, yc);

        // 若距離 <= 半徑，表示在圓區內
        if (dist <= zoneRadius && !inZone) {
            lapCount++;
            inZone = true;
            printf("Lap detected at position (%.2f, %.2f), lapCount = %d\n",
                   x, y, lapCount);
        }
        // 若超過半徑且原本在區域內，表示離開了區域
        else if (dist > zoneRadius && inZone) {
            inZone = false;
        }
    }

    printf("Final lap count = %d\n", lapCount);

    return 0;
}
```

方法特點

- **適用場景：** 只要「進入某範圍」即可視為通過，判斷方式較單純。
- **需要注意：** 圓形範圍、半徑大小要與實際定位誤差搭配，否則易出現誤判或敏感度不足。

---

3. 兩種方法差異與比較

線段交叉法

- **適用場景**：有明確的「起終點線」或需要精準記錄「何時」跨越
- **設計重點**：通常需加入距離閾值、方向判斷來降低誤觸發可能

區域判定法

- **適用場景**：只要「進入特定區域」即視為通過，實作較直覺
- **設計重點**：區域大小（如圓形半徑）要配合定位精度

> 綜合而言，若專案中有清晰的線形範圍並需要精準記錄「何時」跨越，即可採用**線段交叉法**；若僅需知道「是否經過特定區域」或「進入一個目標範圍」，可採用**區域判定法**，實作較為簡易直覺。

以上即為整合所有內容的參考 **Markdown** 文件範本。可依實際情況進一步補充更詳細的硬體規格、韌體流程或演算法參數。祝專案開發順利！