STM32 RTOS ????

- STM32 RTOS 系列教學 - 第一篇: RTOS 簡介與 STM32CubeMX 設定
- STM32 RTOS 系列教學 - 第二篇: 多任務與任務切換 (雙 LED 閃爍)
- STM32 RTOS 系列教學 - 第三篇: RTOS 任務間通訊 (Queue 與 Semaphore 實作)
- STM32 RTOS 系列教學 - 第四篇: RTOS 計時器與非阻塞處理技巧
- STM32 RTOS 系列教學 - 第五篇: RTOS 訊號量 (Semaphore) 與資源同步應用
- STM32 RTOS 系列教學 - 第六篇: RTOS Queue 任務間資料傳輸實作
- 串口監控基礎框架使用說明
- 從 Goddard QuarterCar 到 FSAE Traction Control 模型
- 通訊測試 CODE解說
- 實車系統的潛在危險與 Failsafe (失效安全) 機制
- Arduino 紅外線單圈計時器使用說明書
- F042等無 BOOT0 Pin 的 MCU燒入後不執行處理
- Savaresi《 Active Braking Control Systems Design for Vehicles》Ch2 & Ch3 重點整理
- Time Step (步長) 的設定原則與語法
- Titania (AMB3626) 硬體設定與腳位控制指南
- Titania (AMB3626) UART 指令與回應值完整指南

STM32 RTOS ???? - ???? RTOS ??? STM32CubeMX ??

? ????

- 了解 RTOS 基本概念
- 使用 STM32CubeMX 設定 FreeRTOS
- 成功建立一個最小可運作的 RTOS 專案（Blink Task）

? RTOS ????

RTOS（Real-Time Operating System）是一種具備即時性、多工與排程能力的作業系統，常用於嵌入式系統中。例如：FreeRTOS 是 STM32 上廣泛使用的 RTOS。

RTOS 特點：

- 任務 (Task) 切換快
- 支援排程優先權
- 支援中斷與同步機制（Semaphore、Queue 等）
- 系統可預測性高

? STM32CubeMX ?? FreeRTOS

Step 1??????

- 開啟 STM32CubeMX
- 選擇你的晶片（如 STM32F103C8Tx）

Step 2??? RTOS

- 點選左邊選單 `Middleware > FREERTOS`
- Enable FreeRTOS

- Task Configuration → 建立一個名為 `defaultTask` 的任務，設為 `StartDefaultTask`
- 建議 stack size: `128`、priority: `Normal`

Step 3??? System Clock

- 確保 HSE 或 HSI 有啟用並正確設定頻率（建議使用 72MHz）

Step 4??????

- 點選 `Project Manager > Project`，輸入專案名稱
- Toolchain: 選擇你要的 IDE（如 STM32CubeIDE）
- Generate Code □

? ????? - Blink LED

在 `StartDefaultTask` 中加入簡單的 LED 閃爍測試：

```
void StartDefaultTask(void *argument)
{
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    osDelay(500);
  }
}
```

小提醒：GPIO 必須在 CubeMX 中設定為 `Output`，否則 `HAL_GPIO_TogglePin()` 無效。

? ??????

- 程式編譯成功
- 上電後 LED 以 0.5 秒頻率閃爍
- 無 HardFault、卡住情形

? ?????

□ 2 □□□□□□ Task □□□□□□□□□□□□□□ LED □□□

STM32 RTOS ???? - ????????????? LED ???

? ????

- 建立多個 RTOS Task
- 觀察 RTOS 的任務切換運作方式
- 實作雙 LED 閃爍功能

? RTOS ?????

RTOS 支援多任務（Multi-tasking）機制，每個任務都是獨立的執行單元，能透過 scheduler 依據優先權與排程策略輪流執行。

RTOS 排程器會:

- 根據任務優先順序分配 CPU 使用時間
- 任務之間可透過延遲（`osDelay()`）進行時間讓渡
- 支援同步與互斥（未來章節會提）

? STM32CubeMX ???????????

Step 1?????? Task

- 開啟你的專案（或從第一篇延續）
- `Middleware > FREERTOS > Tasks and Queues`
- 新增一個任務，例如 `blinkTask2`，對應函式: `StartBlinkTask2`
- 建議 stack size: `128`、priority: `Low`（低於 defaultTask）

Step 2?????? LED ? GPIO

- 假設第一顆 LED 使用 `PC13`
- 新增另一個 GPIO，例如 `PA5`，設定為 `GPIO_Output`

Step 3??????

- 點選 `Project > Generate Code`, 進入 IDE 編輯

? ??? Task ????

?? `StartDefaultTask()` ???

```
void StartDefaultTask(void *argument)
{
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13); // 第一顆 LED
    osDelay(500);
  }
}
```

??????? `StartBlinkTask2()`?

```
void StartBlinkTask2(void *argument)
{
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5); // 第二顆 LED
    osDelay(200);
  }
}
```

? ??????

- 程式編譯成功
- 上電後兩顆 LED 分別以不同頻率閃爍
- 無卡死或異常閃爍

? ?????

第 **3** 篇: **RTOS** 任務間通訊（ **Queue** 與 **Semaphore** 實作）

STM32 RTOS ???? - ???? RTOS ??????Queue ? Semaphore ???

STM32 RTOS ???? - ???? RTOS ??????Queue ? Semaphore ???

? ????

- 理解 RTOS 中 Queue（佇列）的用途與特性
- 學會在任務之間安全地傳遞資料
- 使用 Queue 實作一個簡單的生產者 -消費者範例
- 結合 Semaphore 控制 Queue 的存取

? Queue ????

Queue（佇列）是一種資料結構，可在 RTOS 中讓任務之間以 FIFO（先進先出）方式安全交換資料。每個任務可以是資料的「生產者」或「消費者」。

? ??

- 支援固定長度的訊息資料（ byte 或結構）
- 多任務可同時傳送（ Send）與接收（ Receive）資料
- 有 blocking / timeout 功能，等待 Queue 可用
- 適合用於任務間資料傳輸、命令交換、感測器讀值同步等

? Semaphore ????

Semaphore（信號量）是 RTOS 中的同步工具，用來控制對共享資源（如 Queue）的存取，避免資源競爭。

? ????

- Binary Semaphore: 常用於同步（如中斷觸發任務）
- Counting Semaphore: 常用於資源控制（如 Queue 空間或緩衝區）

? STM32CubeMX ?? Queue ? Semaphore

Step 1??? Queue

- 開啟 `Middleware > FreeRTOS > CMSIS RTOS v2`
- 新增 `Message Queue`，命名為 `myQueue01`
- 設定:
  - 資料長度: `sizeof(uint16_t)` 或自訂 struct
  - Queue 長度: 例如 10（最多可存放 10 筆資料）

Step 2??? Semaphore????

- 在 `FreeRTOS > CMSIS RTOS v2` 新增 `Semaphore`
- 命名為 `myBinarySem01`，選擇 Binary

Step 3????

```
// Queue 初始化
osMessageQueueId_t myQueue01Handle;
const osMessageQueueAttr_t myQueue01_attributes = {
  .name = "myQueue01"
};
myQueue01Handle = osMessageQueueNew(10, sizeof(uint16_t), &myQueue01_attributes);

// Semaphore 初始化
osSemaphoreId_t myBinarySem01Handle;
const osSemaphoreAttr_t myBinarySem01_attributes = {
  .name = "myBinarySem01"
};
```

```
myBinarySem01Handle = osSemaphoreNew(1, 1, &myBinarySem01_attributes); // 初始為可用
```

? ??????????

?? A????? - ?????????

```
void StartSenderTask(void *argument)
{
  uint16_t value = 0;
  for(;;)
  {
    osSemaphoreAcquire(myBinarySem01Handle, osWaitForever);  // 取得信號
    osMessageQueuePut(myQueue01Handle, &value, 0, 0);
    osSemaphoreRelease(myBinarySem01Handle);  // 釋放信號
    value++;
    osDelay(1000);
  }
}
```

?? B????? - ????????

```
void StartReceiverTask(void *argument)
{
  uint16_t recvVal;
  for(;;)
  {
    osSemaphoreAcquire(myBinarySem01Handle, osWaitForever);  // 取得信號
    if(osMessageQueueGet(myQueue01Handle, &recvVal, NULL, osWaitForever) == osOK)
    {
      printf("Received: %d\n", recvVal);
    }
    osSemaphoreRelease(myBinarySem01Handle);  // 釋放信號
  }
}
```

? ????

- 任務間傳送 sensor 資料
- 接收中斷服務程式的資料（需透過中斷內部觸發任務）
- 排程控制命令、狀態
- 結合 Semaphore 確保多任務對 Queue 的安全存取

? ??????

- 每秒有數值被傳送與接收
- 無資料遺失（ Queue 空間足夠）
- 接收順序正確（ FIFO）
- Semaphore 正確同步，無競爭狀況

第 **4** 篇: **RTOS** 計時器（ **Software Timer**）與非阻塞處理技巧

STM32 RTOS ???? - ???? RTOS ????????????

STM32 RTOS ???? - ???? RTOS ????????????

? ????

- 理解 RTOS 中軟體計時器 (Software Timer) 的用途與應用場景
- 實作非阻塞式任務執行邏輯
- 比較使用 osDelay 與計時器的異同

? RTOS Software Timer ??

RTOS 中的軟體計時器是一種非阻塞機制，讓你可以在不延遲任務執行的情況下，定時觸發事件。

? ??

- 背景執行，不佔用任務時間
- 適合用於輪詢檢查、定時觸發、 Timeout 等功能
- 可重複或單次觸發

? STM32CubeMX ?? Timer

Step 1??? Timer

- 開啟 `Middleware > FreeRTOS > Timers`
- 新增一個 Timer:
  - 名稱: `myTimer01`

- Period: `1000` (單位為 ms)
- Callback Function: `TimerCallback01`
- Auto-Reload: 啟用

Step 2??????

- 在 `StartDefaultTask` 或初始化任務中呼叫:

```
osTimerStart(myTimer01Handle, 0);
```

Step 3???????

```
void TimerCallback01(void *argument)
{
  HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
}
```

? ????????

傳統的 `osDelay()` 會讓任務進入 Blocked 狀態，若任務需同時處理多事件，建議改用計時器或時間差比較。

????? millis ???????? HAL_GetTick?

```
uint32_t last_tick = 0;
for(;;)
{
  if(HAL_GetTick() - last_tick >= 1000)
  {
    last_tick = HAL_GetTick();
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
  }
  // 其他可即時處理的邏輯
  osDelay(1); // 輕微延遲防止佔滿 CPU
}
```

?????? FreeRTOS ???????

使用 timer callback 執行週期性任務，不阻塞主任務邏輯。

? ??????

- Timer 可準時觸發 LED 閃爍（ GPIO 輸出）
- 任務在不使用 osDelay 下能維持反應速度與功能
- 系統穩定，無多餘延遲或卡死現象

? ?????

第 **5** 篇: **RTOS Memory Pool** 與動態資源管理

STM32 RTOS ???? - ???? RTOS ????Semaphore ????????

? ????

- 理解 RTOS 中 Semaphore 的概念與分類（ Binary / Counting）
- 學會在多任務間使用 Semaphore 實現資源同步
- 實作一個防止資源衝突的共享範例

? Semaphore ????

Semaphore（訊號量）是 RTOS 中用來控制任務間共享資源的一種同步工具，避免同時存取導致資源衝突。

? ??

- **Binary Semaphore**：僅有 0 和 1，用於任務間同步（類似旗標）
- **Counting Semaphore**：允許多個資源存取計數（像計數器），適用於多個資源管理

? STM32CubeMX ?? Semaphore

Step 1??? Semaphore

- 開啟 `Middleware > FreeRTOS > CMSIS RTOS v2`
- 新增 `Semaphore`，命名為 `myBinarySem01`
- 類型選擇 `Binary`

Step 2??? Handle ????

- 在 `main.c` 中取得 handle：

```
osSemaphoreId_t myBinarySem01Handle;
```

- 在初始化階段建立 semaphore:

```
const osSemaphoreAttr_t myBinarySem01_attributes = {
  .name = "myBinarySem01"
};
myBinarySem01Handle = osSemaphoreNew(1, 0, &myBinarySem01_attributes);
```

? ???????Button Trigger

?????

- 任務 A 持續等待 semaphore (類似被 block)
- 中斷服務程式 (EXTI) 發生時觸發 `osSemaphoreRelease()`

EXTI ?????

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if(GPIO_Pin == GPIO_PIN_13)
  {
    osSemaphoreRelease(myBinarySem01Handle);
  }
}
```

????????

```
void StartDefaultTask(void *argument)
{
  for(;;)
  {
    if(osSemaphoreAcquire(myBinarySem01Handle, osWaitForever) == osOK)
    {
      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
    }
  }
```

```
}
```

? ??????

- Button / IRQ 中斷喚醒任務
- 任務間事件同步（生產者 -消費者）
- 控制一次只允許一個任務進入共享區（ Critical Section）

? ??????

- 按下按鈕， LED 能夠閃爍一次，且不會重複觸發
- 任務能穩定接收中斷產生的 semaphore
- 無資源衝突或 crash 現象

? ?????

第 **6** 篇: **RTOS Queue** 任務間資料傳輸實作

STM32 RTOS ???? - ???? RTOS Queue ??????????

STM32 RTOS ???? - ???? RTOS Queue ??????????

? ????

- 理解 RTOS 中 Queue（佇列）的用途與特性
- 學會在任務之間安全地傳遞資料
- 使用 Queue 實作一個簡單的生產者 -消費者範例

? Queue ????

Queue（佇列）是一種資料結構，可在 RTOS 中讓任務之間以 FIFO（先進先出）方式安全交換資料。每個任務可以是資料的「生產者」或「消費者」。

? ??

- 支援固定長度的訊息資料（ byte 或結構）
- 多任務可同時傳送（ Send）與接收（ Receive）資料
- 有 blocking / timeout 功能，等待 Queue 可用

? STM32CubeMX ?? Queue

Step 1??? Queue

- 開啟 `Middleware > FreeRTOS > CMSIS RTOS v2`
- 新增 `Message Queue`，命名為 `myQueue01`

- 設定:
  - 資料長度: `sizeof(uint16_t)` 或自訂 struct
  - Queue 長度: 例如 10 (最多可存放 10 筆資料)

Step 2??? Handle ????

```
osMessageQueueId_t myQueue01Handle;


const osMessageQueueAttr_t myQueue01_attributes = {
  .name = "myQueue01"
};
myQueue01Handle = osMessageQueueNew(10, sizeof(uint16_t), &myQueue01_attributes);
```

? ??????????

?? A????? - ?????????

```
void StartSenderTask(void *argument)
{
  uint16_t value = 0;
  for(;;)
  {
    osMessageQueuePut(myQueue01Handle, &value, 0, 0);
    value++;
    osDelay(1000);
  }
}
```

?? B????? - ????????

```
void StartReceiverTask(void *argument)
{
  uint16_t recvVal;
  for(;;)
  {
    if(osMessageQueueGet(myQueue01Handle, &recvVal, NULL, osWaitForever) == osOK)
```

```
    {
      printf("Received: %d\n", recvVal);
    }
  }
}
```

? ????

- 任務間傳送 sensor 資料
- 接收中斷服務程式的資料（需透過中斷內部觸發任務）
- 排程控制命令、狀態

? ??????

- 每秒有數值被傳送與接收
- 無資料遺失（ Queue 空間足夠）
- 接收順序正確（ FIFO）

? ?????

第 **7** 篇: **RTOS Timer** 與軟體定時器應用

??????????

> 撰寫人　：范紹捷　/動力組　/5~9代

??

??

這是一個基於　ttkbootstrap 的 uart監控基礎框架，提供了基本的串口通信界面和日誌記錄功能。

????

```
pip install ttkbootstrap
pip install pyserial
```

????

?????

```
from base_monitor import BaseMonitor

app = BaseMonitor(title="監控程序")
app.run()
```

????

初始化時可設置的參數:

- title：窗口標題（默認: "監視器 "）
- size：窗口大小（默認: "500x600"）
- theme：界面主題（默認: "darkly"）

?????

```
darkly,cosmo,flatly,litera,minty,lumen,sandstone,yeti,pulse,united,morph
```

?????

?????

```
class CustomMonitor(BaseMonitor):
    def __init__(self):
        super().__init__(title="自定義監控")
```

??????

1. create_data_panel：自定義數據顯示面板

```
def create_data_panel(self):
    data_frame = ttk.LabelFrame(self.main_container, text="自定義數據", padding=10)
    data_frame.pack(fill=X, pady=10)
    # 添加自己的顯示元件
```

2. start_monitor：實現數據接收邏輯

```
def start_monitor(self):
    try:
        self.serial = serial.Serial(
            port=self.port_var.get(),
            baudrate=int(self.baud_var.get()),
            timeout=1
        )
        # 添加數據處理邏輯
```

```
    except Exception as e:
        self.log_message(f"啟動失敗: {str(e)}")
```

3. stop_monitor : 實現停止邏輯

```
def stop_monitor(self):
    if self.serial:
        self.serial.close()
        self.serial = None
```

????

使用 log_message 方法記錄日誌:

```
self.log_message("自定義消息")
```

????

1. 串口連接前請確保選擇了正確的端口和波特率
2. 程序退出時會自動關閉串口連接
3. 建議在子類中實現異常處理機制

????

```
import ttkbootstrap as ttk
from ttkbootstrap.constants import *
import threading
import serial
import serial.tools.list_ports

class BaseMonitor:
    def __init__(self, title="監視器", size="500x600", theme="darkly"):
        self.root = ttk.Window(title=title, themename=theme, resizable=(False, False))
        self.root.geometry(size)

        # 狀態變量
        self.is_running = False
```

```
        # 創建界面
        self.create_widgets()

    def create_widgets(self):
        # 主容器
        self.main_container = ttk.Frame(self.root, padding=10)
        self.main_container.pack(fill=BOTH, expand=YES)

        # 控制區域
        self.create_control_panel()

        # 數據顯示區域
        self.create_data_panel()

        # 日誌區域
        self.create_log_panel()

    def create_control_panel(self):
        """控制面板 - 可在子類中重寫"""
        control_frame = ttk.LabelFrame(self.main_container, text="控制面板", padding=10)
        control_frame.pack(fill=X, pady=5)

        # 端口設置
        port_frame = ttk.Frame(control_frame)
        port_frame.pack(fill=X, pady=5)
        ttk.Label(port_frame, text="端口:").pack(side=LEFT, padx=5)
        self.port_var = ttk.StringVar()
        self.port_combo = ttk.Combobox(port_frame, textvariable=self.port_var)
        self.port_combo.pack(side=LEFT, fill=X, expand=YES)

        # 刷新端口按鈕
        ttk.Button(
            port_frame,
            text="刷新",
            command=self.refresh_ports,
            style="info.TButton",
            width=8
        ).pack(side=LEFT, padx=5)

        # 波特率設置
```

```
        baud_frame = ttk.Frame(control_frame)
        baud_frame.pack(fill=X, pady=5)
        ttk.Label(baud_frame, text="波特率:").pack(side=LEFT, padx=5)
        self.baud_var = ttk.StringVar(value="9600")
        baud_choices = ['9600', '19200', '38400', '57600', '115200']
        ttk.Combobox(baud_frame, textvariable=self.baud_var,
values=baud_choices).pack(side=LEFT, fill=X, expand=YES)

        # 控制按鈕
        self.control_btn = ttk.Button(
            control_frame,
            text="連接",
            command=self.toggle_running,
            style="primary.TButton"
        )
        self.control_btn.pack(pady=10)

        # 初始化端口列表
        self.refresh_ports()

    def refresh_ports(self):
        """更新可用的串口列表"""
        ports = [port.device for port in serial.tools.list_ports.comports()]
        self.port_combo['values'] = ports
        if ports:
            self.port_var.set(ports[0])
        else:
            self.port_var.set('')
            self.log_message("未檢測到可用的串口")

    def toggle_running(self):
        """切換運行狀態"""
        self.is_running = not self.is_running
        if self.is_running:
            self.control_btn.configure(text="斷開", style="danger.TButton")
            self.status_label.configure(text="已連接", style="success.TLabel")
            self.log_message("串口連接成功")
            self.start_monitor()
        else:
            self.control_btn.configure(text="連接", style="primary.TButton")
```

```
            self.status_label.configure(text="未連接", style="danger.TLabel")
            self.log_message("串口已斷開")
            self.stop_monitor()

    def create_data_panel(self):
        """數據顯示面板 - 可在子類中重寫"""
        data_frame = ttk.LabelFrame(self.main_container, text="數據顯示", padding=10)
        data_frame.pack(fill=X, pady=10)

        # 狀態顯示
        self.status_label = ttk.Label(
            data_frame,
            text="未運行",
            style="danger.TLabel"
        )
        self.status_label.pack(pady=5)

    def create_log_panel(self):
        """日誌面板"""
        log_frame = ttk.LabelFrame(self.main_container, text="系統日誌", padding=10)
        log_frame.pack(fill=BOTH, expand=YES, pady=5)

        self.log_text = ttk.Text(log_frame, height=10, width=40)
        self.log_text.pack(fill=BOTH, expand=YES)

        scrollbar = ttk.Scrollbar(log_frame, orient="vertical", command=self.log_text.yview)
        scrollbar.pack(side=RIGHT, fill=Y)
        self.log_text.configure(yscrollcommand=scrollbar.set)

    def toggle_running(self):
        """切換運行狀態"""
        self.is_running = not self.is_running
        if self.is_running:
            self.control_btn.configure(text="停止", style="danger.TButton")
            self.status_label.configure(text="運行中", style="success.TLabel")
            self.log_message("系統啟動")
            self.start_monitor()
        else:
            self.control_btn.configure(text="開始", style="primary.TButton")
            self.status_label.configure(text="未運行", style="danger.TLabel")
```

```
            self.log_message("系統停止")
            self.stop_monitor()

    def start_monitor(self):
        """啟動監控 - 在子類中實現"""
        pass

    def stop_monitor(self):
        """停止監控 - 在子類中實現"""
        pass

    def log_message(self, message):
        """記錄日誌消息"""
        import time
        self.log_text.insert(END, f"{time.strftime('%H:%M:%S')} - {message}\n")
        self.log_text.see(END)

    def run(self):
        """運行程序"""
        self.root.mainloop()

if __name__ == "__main__":
    # 基礎使用示例
    app = BaseMonitor(title="基礎監視器")
    app.run()
```

? Goddard QuarterCar ? FSAE Traction Control ??

? Goddard QuarterCar ? FSAE Traction Control ??

撰寫人　：范紹捷　/動力組　/5~9代 基於　TTR9開發

目標:

- 以 Phil Goddard 的 *Slip Control of a Quarter Car Model* 為基礎, 改成符合 FSAE 後驅電動車的牽引力控制（ TC）實驗環境。 Goddard QuarterCar 模型介紹網頁、模型下載 Academia.edu - 詳細說明
- 在 Simulink 上:
  - 先驗證 QuarterCar plant 是否物理正確
  - 再以 PI 控制滑移率 λ, 限制輪上扭矩

---

1. ???????

1.1 FSAE ????

- 總重: $m_{total} = 300 ,\text{kg}$
- 前後配重（估）: 前 45%、後 55%
  - 單後輪垂直力 $$F_{z,\text{rear,single}} \approx \frac{m_{total} g \cdot 0.55}{2} \approx 809 ,\text{N}$$
- 單馬達後驅, 馬達接小齒輪、後輪大齒輪, 總傳動比: $i = 4.02$

1.2 ???????

- 馬達峰值扭矩: $T_{motor,max} = 220,\text{Nm}$
- 後軸總輪上扭矩: $$T_{wheel,total} = T_{motor} \cdot i \approx 220 \times 4.02 \approx 884.4 ,\text{Nm}$$

- 後驅兩輪分配 → 單後輪扭矩（QuarterCar 模型使用）：$$T_{wheel,single} \approx \frac{884.4}{2} \approx 442.2,\text{Nm}$$

在 Simulink 中，對 QuarterCar 單輪模型：

```
T_driver = 442.2;  % Nm, 單後輪全油門扭矩
```

---

2. ???????

2.1 ???????

- 輪胎: Goodyear D2773（FSAE 常用 20x7-13 slick）
- 名目直徑: 13 吋輪圈，滾動半徑近似: $$R_{wheel} \approx \frac{13}{2},\text{in} \times 0.0254 \approx 0.165,\text{m}$$

參數設定（`QuarterCarParams.m` 或 `tc_params.h`）：

```
R_REAR  = 0.165;   % m
R_FRONT = 0.165;   % m
```

---

3. ? ABS ???? TC ??

Goddard 原始模型是 ABS（煞車控制）：踏煞 → 調整煞車扭矩 Tb，維持 λ 在 0.2 左右。
這裡要改成 TC（驅動控制）：加速 → 限制驅動扭矩 T_{driver}，讓 λ 維持在最佳打滑附近。

3.1 Slip ?????

ABS 常用定義（煞車）：

$$\lambda_{ABS} = \frac{v - R \omega}{v}$$

TC 驅動時採用：

$$\lambda_{TC} = \frac{R \omega - v}{v}$$

在 Goddard 模型的 `Slip Calculation` block 中，確認 / 修改為：

```
lambda = (R * angVel - vel) / max(vel, 0.1);
```

> □□□□□□□□□□□□□□□□□□ `Abs` □□□□ TC
> □□□□□□□□□□□□□□□□ / □□□□

3.2 ???????

ABS → □□□ TC → □□□□□□□

- □□□□
  - ABS□□□□ $J \dot{\omega} = -T_b - R F_x$
  - TC□□□□□ $$J \dot{\omega} = T_{driver} - R F_x$$
- □□□□□
  - ABS□□□□ $m \dot{v} = -F_x$
  - TC□□□□□ $$m \dot{v} = F_x$$

□ QuarterCar □□□□□□□□

- □□□ `Sum` → □□ `1/J` □□□□□□□□ `+T_driver` □ `- R*Fx` □
- □□ `Sum` → □□ `1/m` □□□□□□□□ `+Fx` □□□ `-Fx` □□

□□□□□□□□□ □□□□□□ □□ □□□□□□ □

---

4. ????? Goodyear D2773 ??

Goddard □□□ Pacejka □□□

$$\mu_x = c \cdot \sin\bigl(b \cdot \arctan(a \lambda)\bigr)$$

`QuarterCarParams.m` □□□□□□

```
roadCoeffs = [1.28 23.99 0.52]; % [a, b, c], □□ μ_peak ≈ 0.52
```

□□□□□ `Fx` □□□□□ `λ≈0.05` □□□

4.1 ?????????

□□□□□□□□□ D2773□□□□□□

```
roadCoeffs = [1.28, 23.99, 0.52];
lambda_base = 0.05;  % □□□□□
```

- λ_peak 約 0.05
- 建議 `DesiredSlip` 的 Final value 也設在 0.05~ 0.08 之間

4.2 ?? FSAE Goodyear????

參考 FSAE 文獻與 μ 值, 可略調高峰值: $\mu_{peak} \approx 0.55$

```
% FSAE Goodyear 乾地近似
roadCoeffs = [1.67, 22.0, 0.55];   % [a, b, mu_peak]
lambda_base = 0.05;                % 峰值附近
```

> 最準確作法: 用實車量測的 ax、輪速與車速 log, 計算 Fx–λ 曲線, 透過 `lsqcurvefit` 或 Curve Fitting Tool 擬合 Pacejka 係數。

---

5. ??????????

5.1 ??????

依前述計算, 單後輪峰值扭矩約 442.2 Nm:

```
T_driver = 440;  % QuarterCar 單後輪輸入扭矩
```

測試時可先使用 Constant block, 或 Step block 0→450 Nm 模擬全油門起步。

5.2 ?? km/h ??

QuarterCar 輸出的 `vel` 單位是 m/s, 可加一個 Gain **3.6** 轉成 km/h:

```
vel (m/s) ── Gain(3.6) ──> v_kmh
```

接到 Scope 或 Display, 方便閱讀 0–50 / 0–100 km/h 表現。

5.3 Plant ?????????

1. `T_driver = 440 Nm` Constant, `DesiredSlip` 不用 (或 Final=0) 。
2. Scope 觀察: `v (km/h)`、`omega`、`lambda`、`Fx`。
3. 確認:

- v 隨時間單調上升。
- `R*omega > v` → λ 為正，表示有打滑。
- λ 峰值約在 0.05~ 0.1（取決於路面係數），Fx 對應峰值（輪胎最大抓地力）。

---

6. PI Traction Control ??

6.1 ????

在 Goddard 的控制器層級上，改成下列 TC 架構:

```
DesiredSlip (Step, 0 → lambda_base)
         |
         v
      [ + ] <——— slip (λ)
         |
         v
      PI(s)
         |
         v
TorqueLimit = min( PI_out, T_demand )
         |
         v
    MotorActuator / QuarterCar (T_driver = TorqueLimit)
```

- `DesiredSlip`: Step block，Initial=0，Final=`lambda_base`（例如 0.05），Sample time=0.005。
- `PI(s)`: 離散 PI；Sample time=0.005 s。
- `T_demand`: 駕駛扭矩需求，例如 Step 0→450 Nm。
- `TorqueLimit`: MinMax block，Mode=**min**，輸入 `[PI_out, T_demand]`。
  - 表示: $T_{driver} = \min(T_{demand}, T_{PI})$，即控制器只會「減扭」，不會要求超過駕駛需求。

6.2 PI ????

根據 QuarterCar 估算與 FSAE 經驗，可用以下範圍作起點:

```
Kp = 800;
Ki = 8000;
lambda_base = 0.05;  % 乾地 Goodyear 峰值附近
```

```
Ts = 0.005;            % 5 ms 控制週期
```

調參方向:

- λ 超調很大（例如先衝到 0.15 再回 0.05） → **Kp** 降低 （ 800 → 500）。
- λ 收斂太慢（ >1 s 才貼近 0.05） → **Kp** 提高 （ 800 → 1200–1500）。
- λ 有穩態誤差（例如停在 0.07） → **Ki** 提高 （ 8000 → 20000）。
- λ 出現明顯振盪 → **Ki** 降低 （ 8000 → 2000–4000）。

---

7. ??????????????

目標: 利用實車的 **CAN-logger** 與量測訊號, 把 Simulink 模型中的輪胎 /車輛參數 （尤其 roadCoeffs 、質量分佈、實際半徑）調整到貼近真實 FSAE 車。

7.1 ??????

建議至少量到下列訊號（你現有硬體已足夠）:

- 四輪輪速: ω_{FL} , ω_{FR}, ω_{RL}, ω_{RR}
- IMU: 縱向加速度 a_x、橫向加速度 a_y、偏航率 r
- 馬達: 轉速 ω_m、實際扭矩指令 / 回授 T_{motor}
- 其它: 油門開度、模式切換狀態等（方便之後分析）

所有訊號以 CAN logger 記錄成時間序列檔（例如 CSV / MDF）。

7.2 ??????

1. 直線全油門起步（乾地）
   - 場地: 平坦乾柏油。
   - 操作: 檔位固定、 TC 關閉, 從靜止全油門加速到約 50–60 km/h。
   - 目的: 取得高 μ 乾路下的 $\lambda–a_x$ 關係與最大加速度。
2. 中低扭矩掃描（低打滑）
   - 多次 run, 用 20%、40%、60% 油門限制。
   - 目的: 在低打滑區（ λ<0.05）估計「線性區」的剛度, 以修正 Pacejka 的 a、b 係數。
3. 低 **μ** 路面（如濕地 / 噴水）
   - 同樣的全油門起步, 在明顯較滑的路段。
   - 目的: 取得另一組 $\lambda–a_x$ 曲線, 用來驗證 μ 估測與多路面設定。

7.3 ??????? ? ? Fx

離線分析（ MATLAB）:

1. 由前輪輪速估車速（假設前輪未打滑）：

```
v_est = 0.5 * (omega_fl + omega_fr) * R_FRONT;  % m/s
```

2. 計算後輪滑移率 λ:

```
lambda_rl = (omega_rl * R_REAR - v_est) ./ max(v_est, 0.1);
lambda_rr = (omega_rr * R_REAR - v_est) ./ max(v_est, 0.1);
lambda_avg = 0.5 * (lambda_rl + lambda_rr);
```

3. 以 IMU 縱向加速度估計總縱向力:

```
Fx_total = m_total * ax_imu;   % N
Fx_single = Fx_total / 2;      % 假設主要由兩後輪提供
```

得到一組實車資料點 $(\lambda_{avg}, F_{x,single})$。

7.4 ?? Pacejka ??

使用非線性最小平方法（ lsqcurvefit）擬合 roadCoeffs = [a, b, c]:

```
lambda_data = lambda_avg(:);
Fx_data     = Fx_single(:);
Fz_single   = m_total * 9.81 * 0.55 / 2;

pacejka_fx = @(p, lam) Fz_single .* (p(3) .* sin(p(2) .* atan(p(1) .* lam)));

p0 = [1.3, 24, 0.55];  % 初始猜測 [a, b, mu_peak]
opts = optimoptions('lsqcurvefit', 'Display', 'iter');
[p_fit, ~] = lsqcurvefit(pacejka_fx, p0, lambda_data, Fx_data, [], [], opts);

a_fit = p_fit(1);
b_fit = p_fit(2);
mu_fit = p_fit(3);

roadCoeffs_new = [a_fit, b_fit, mu_fit];
```

將 `roadCoeffs_new` 帶回 Simulink 之後，再重跑 QuarterCar 模型， Comparing:

- 實車 vs 模擬的 `lambda(t)`、`v(t)`、`ax(t)` 曲線。
- 若誤差仍大，可根據特定區間再調整 a / b（斜率）與 c（峰值 μ）。

7.5 ?????? Kp / Ki

在更新後的輪胎模型下:

1. 用 Simulink 模型跑與實車相同的油門 / 路面情境（ T_demand, lambda_base 同步）。
2. 比較:
   - 模擬 λ 是否與實車趨勢相近（尤其峰值與收斂時間）。
   - 若模擬反應太快 → Kp/Ki 太大; 反之太慢 → Kp/Ki 太小。
3. 以「實車 λ 波形」為 benchmark, 把 PI 參數調到模擬與實車時間響應接近, 再將這組 Kp/Ki 寫回 STM32 Firmware。

---

8. ???????FSAE ???

參數	數值	說明
m_{total}	300 kg	FSAE 整車質量
重量分配	45:55	前 :後, 估計
單後輪 F_z	≈ 809 N	300·9.81·0.55/2
馬達峰扭 T_{motor}	220 Nm	80 kW 電機
傳動比 i	4.02	馬達小齒 →後輪大齒
單後輪峰扭 T_{wheel}	≈ 440 Nm	220·4.02/2
輪半徑 R	0.165 m	13 吋滾動半徑估計
roadCoeffs (原)	[1.28, 23.99, 0.52]	Goddard 乾地
roadCoeffs (FSAE 初猜)	[1.67, 22.0, 0.55]	Goodyear D2773 近似
λ_target	0.05	乾地峰值附近
Ts	0.005 s	控制週期 5 ms
Kp	800	PI 初始
Ki	8000	PI 初始

這份筆記對應的 Simulink 模型改動完成後, 就可以直接拿來:

- 在桌上跑 QuarterCar + PI/TC 算法調參
- 以實車數據反推輪胎模型, 持續修正 roadCoeffs
- 把同一套 PI/TC 參數下放到 STM32 上, 做實車迴圈驗證

????CODE??

STM32 CAN ??? + UART ????

專案名稱　　：TTR9.5_eco_VCU

> 本文檔針對提供的 STM32F4 韌體進行詳細分析，說明如何使用 CAN1、CAN2 和 UART 通訊模組。

? ??

?????

???????

```
/* main() 函數中的初始化流程 */
MX_GPIO_Init();             // GPIO 初始化
MX_CAN1_Init();            // CAN1 初始化
MX_CAN2_Init();            // CAN2 初始化
MX_USART1_UART_Init();     // UART1 初始化
MX_USART6_UART_Init();     // UART6 初始化
```

image

初始化順序重要性:

- GPIO 必須先初始化，為其他外設提供引腳支援
- CAN 和 UART 初始化順序可互換

CAN1 ????

1?? **CAN1 ?????** (`MX_CAN1_Init`)

```
static void MX_CAN1_Init(void)
{
  hcan1.Instance = CAN1;
  hcan1.Init.Prescaler = 5;               // 波特率預分頻: 決定 CAN 速度
  hcan1.Init.Mode = CAN_MODE_NORMAL;      // 正常模式
  hcan1.Init.SyncJumpWidth = CAN_SJW_1TQ;
  hcan1.Init.TimeSeg1 = CAN_BS1_15TQ;     // 位元時序段 1
  hcan1.Init.TimeSeg2 = CAN_BS2_2TQ;      // 位元時序段 2
  hcan1.Init.TimeTriggeredMode = DISABLE;
  hcan1.Init.AutoBusOff = DISABLE;
  hcan1.Init.AutoWakeUp = DISABLE;
  hcan1.Init.AutoRetransmission = DISABLE;
  hcan1.Init.ReceiveFifoLocked = DISABLE;
  hcan1.Init.TransmitFifoPriority = DISABLE;

  if (HAL_CAN_Init(&hcan1) != HAL_OK)
  {
    Error_Handler();
  }
}
```

關鍵參數:

參數	值	說明
`Prescaler`	5	影響 CAN 波特率計算
`TimeSeg1`	15TQ	時序段 1 = 15 時間量子
`TimeSeg2`	2TQ	時序段 2 = 2 時間量子
`SyncJumpWidth`	1TQ	同步跳躍寬度

波特率計算:

```
CAN_BaudRate = APB1_Clock / (Prescaler × (1 + TimeSeg1 + TimeSeg2))
             = 45MHz / (5 × (1 + 15 + 2))
             = 45MHz / (5 × 18)
             = 500 kbps
```

2?? CAN1 ?????

```
CAN_FilterTypeDef sFilterConfig;

// 配置 CAN1 過濾器
sFilterConfig.FilterBank = 0;                // 使用第 0 號過濾器
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
sFilterConfig.FilterActivation = ENABLE;
sFilterConfig.SlaveStartFilterBank = 14;   // CAN2 從第 14 號過濾器開始

HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig);
HAL_CAN_Start(&hcan1);
HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);
```

過濾器說明:

- **FilterBank 0-13**: CAN1 使用
- **FilterBank 14-27**: CAN2 使用（ Slave 模式）
- **FilterIdLow = 0x0000 + FilterMaskIdLow = 0x0000**: 接收所有訊息
- **CAN_RX_FIFO0**: 接收訊息存儲到 FIFO0
- **ActivateNotification**: 啟用 FIFO0 新訊息中斷

3?? CAN1 ???? (`CAN1_Send_Test`)

```
void CAN1_Send_Test(uint8_t counter)
{
    // 設定發送訊息頭
```

```
    TxHeader1.StdId = 0x101;           // CAN ID: 0x101
    TxHeader1.RTR = CAN_RTR_DATA;      // 資料幀（非遠端幀）
    TxHeader1.IDE = CAN_ID_STD;        // 標準 ID（11-bit）
    TxHeader1.DLC = 8;                 // 資料長度: 8 bytes
    TxHeader1.TransmitGlobalTime = DISABLE;

    // 準備 8 bytes 資料
    TxData1[0] = counter;              // byte 0: 計數器
    TxData1[1] = 0x22;
    TxData1[2] = 0x33;
    TxData1[3] = 0x00;
    TxData1[4] = 0x00;
    TxData1[5] = 0x00;
    TxData1[6] = 0x00;
    TxData1[7] = 0x00;

    // 檢查信箱有無空位並發送
    if (HAL_CAN_GetTxMailboxesFreeLevel(&hcan1) > 0)
    {
        if (HAL_CAN_AddTxMessage(&hcan1, &TxHeader1, TxData1, &TxMailbox1) != HAL_OK)
        {
            // 發送失敗處理
        }
    }
}
```

發送流程:

1. 配置 `TxHeader1` 結構體
2. 填充 `TxData1` 陣列（最多 8 bytes）
3. 檢查發送信箱是否有空位
4. 調用 `HAL_CAN_AddTxMessage()` 發送

在主循環中調用:

```
while(1)
{
    if (HAL_GetTick() - last_uart_tick >= 300)
    {
        Send_uart[86]++;
        HAL_UART_Transmit(&huart1, Send_uart, sendData_lenght + 3, 1000);
```

```
        CAN1_Send_Test(test_count++);  // ← 每 300ms 發送一次 CAN 訊息
        last_uart_tick = HAL_GetTick();
    }
}
```

CAN2 ????

1?? **CAN2 ?????** (`MX_CAN2_Init`)

```
static void MX_CAN2_Init(void)
{
  hcan2.Instance = CAN2;
  hcan2.Init.Prescaler = 5;
  hcan2.Init.Mode = CAN_MODE_NORMAL;
  hcan2.Init.SyncJumpWidth = CAN_SJW_1TQ;
  hcan2.Init.TimeSeg1 = CAN_BS1_15TQ;
  hcan2.Init.TimeSeg2 = CAN_BS2_2TQ;
  hcan2.Init.TimeTriggeredMode = DISABLE;
  hcan2.Init.AutoBusOff = DISABLE;
  hcan2.Init.AutoWakeUp = DISABLE;
  hcan2.Init.AutoRetransmission = DISABLE;
  hcan2.Init.ReceiveFifoLocked = DISABLE;
  hcan2.Init.TransmitFifoPriority = DISABLE;

  if (HAL_CAN_Init(&hcan2) != HAL_OK)
  {
    Error_Handler();
  }
}
```

與 **CAN1** 相同的配置參數

2?? **CAN2 ?????**

```
// 配置 CAN2 過濾器
sFilterConfig.FilterBank = 14;                  // ← 使用第 14 號過濾器 (Slave 區間起點)
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
```

```
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
sFilterConfig.FilterActivation = ENABLE;

HAL_CAN_ConfigFilter(&hcan2, &sFilterConfig);
HAL_CAN_Start(&hcan2);
HAL_CAN_ActivateNotification(&hcan2, CAN_IT_RX_FIFO0_MSG_PENDING);
```

CAN1 vs CAN2 過濾器分配:

```
CAN1: FilterBank 0-13   (14 個過濾器)
CAN2: FilterBank 14-27  (14 個過濾器, Slave 模式)
```

UART ????

1?? **UART1 ???** (Titania ????)

```
static void MX_USART1_UART_Init(void)
{
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 9600;              // ← 波特率: 9600 bps
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;       // 收發模式
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;

  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
}
```

UART1 用途:　　與 Titania 無線模組通訊

UART1 配置參數:

參數	值	說明
`BaudRate`	9600	低速通訊
`WordLength`	8B	8 位元資料
`StopBits`	1	1 個停止位元
`Parity`	NONE	無奇偶校驗

2?? **UART1 ????** (Titania ????)

```
// 全域變數定義
uint8_t sendData_lenght = 84;         //預傳送資料長度
uint8_t Send_uart[87];               // sendData_lenght + 3 (84 + 3)
uint32_t last_uart_tick = 0;         // 時間戳記
uint8_t Send_head[3] = {0x02, 0x00, 0x3A};  // 訊息標頭
uint8_t Send_checksum = 0;

// 主循環中的發送邏輯
while (1)
{
    if (HAL_GetTick() - last_uart_tick >= 300)
    {
        // 設定訊息頭
        Send_uart[0] = Send_head[0];   // 0x02
        Send_uart[1] = Send_head[1];   // 0x00
        Send_uart[2] = Send_head[2];   // 0x3A

        Send_uart[86]++;               // ← 計數器(最後一個 byte)

        // 發送 87 bytes(頭 3 bytes + 84 bytes 資料)
        HAL_UART_Transmit(&huart1, Send_uart, sendData_lenght + 3, 1000);

        last_uart_tick = HAL_GetTick();  // 更新時間戳記
    }
}
```

Titania 無線模組訊息格式:

| 頭 (3B) | 資料 (84B) |

發送間隔: 每 300ms 發送一次

3?? **UART6 ???** (??/????)

```
static void MX_USART6_UART_Init(void)
{
  huart6.Instance = USART6;
  huart6.Init.BaudRate = 115200;              // ← 波特率: 115200 bps (高速)
  huart6.Init.WordLength = UART_WORDLENGTH_8B;
  huart6.Init.StopBits = UART_STOPBITS_1;
  huart6.Init.Parity = UART_PARITY_NONE;
  huart6.Init.Mode = UART_MODE_TX_RX;
  huart6.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart6.Init.OverSampling = UART_OVERSAMPLING_16;

  if (HAL_UART_Init(&huart6) != HAL_OK)
  {
    Error_Handler();
  }
}
```

UART6 用途: IMU

UART6 配置參數:

參數	值	說明
`BaudRate`	115200	高速通訊
`WordLength`	8B	8 位元資料
`StopBits`	1	1 個停止位元
`Parity`	NONE	無奇偶校驗

???????

? **CAN ??????**

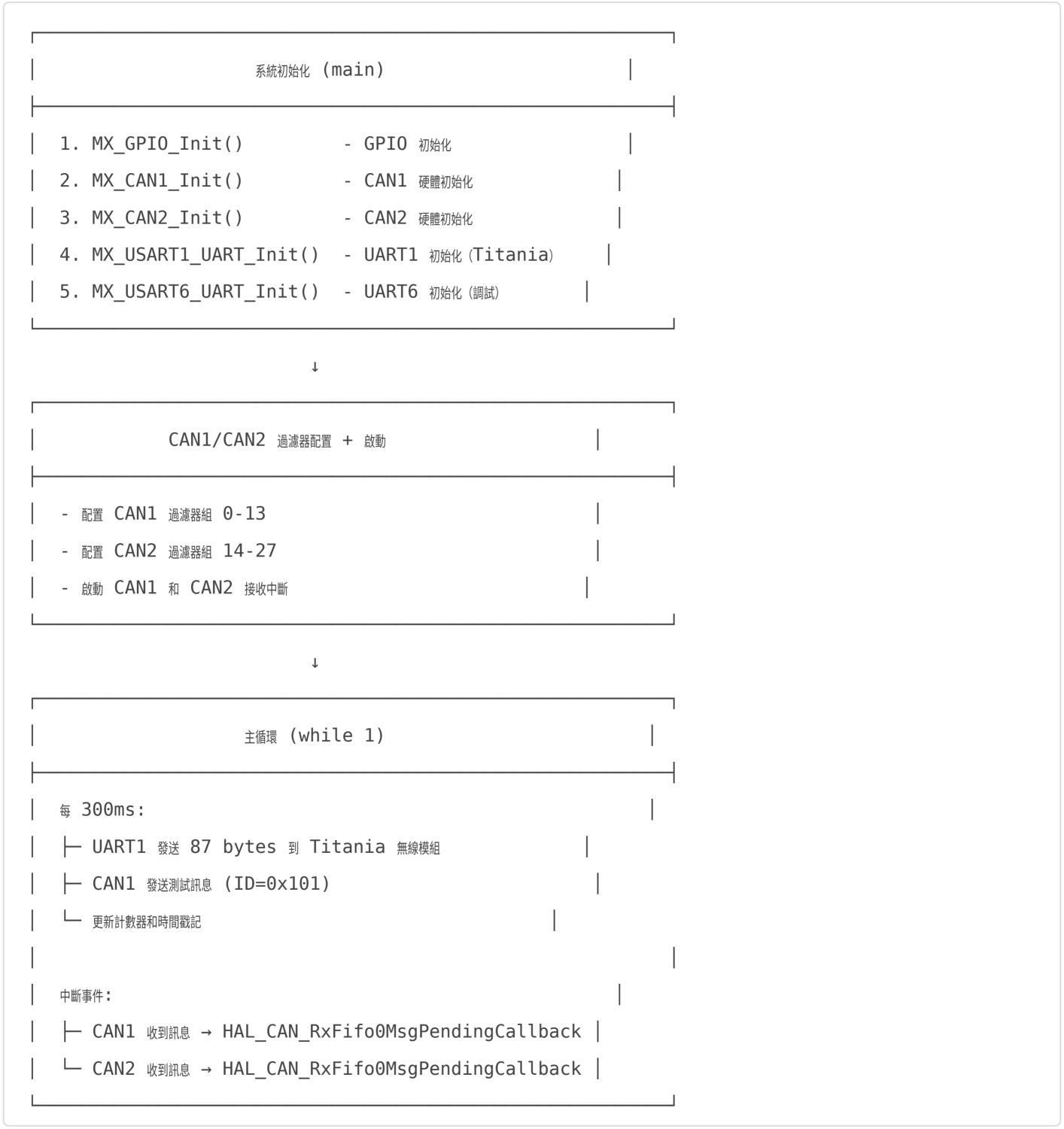
```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    CAN_Counter++;  // 計算接收訊息總數

    if (hcan->Instance == CAN1)
    {
        // CAN1 接收訊息處理
        if (HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader1, RxData1) == HAL_OK)
        {
            // 判斷訊息 ID
            if (RxHeader1.StdId == 0x101)
            {
                // 處理 CAN1 ID=0x101 的訊息
                // RxData1[0] ~ RxData1[7]: 8 bytes 資料
            }
        }
    }
    else if (hcan->Instance == CAN2)
    {
        // CAN2 接收訊息處理
        if (HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader2, RxData2) == HAL_OK)
        {
            // 從 RxData2 提取編碼器數值 (4 bytes)
            // 資料順序: RxData2[3] (LSB) → RxData2[6] (MSB)
            raw_encoder_value = (uint32_t)(
                (RxData2[6] << 24) |    // 最高位 byte
                (RxData2[5] << 16) |
                (RxData2[4] << 8)  |
                RxData2[3]              // 最低位 byte
            );
        }
    }
}
```

關鍵流程:

1. 中斷觸發 : CAN FIFO0 有新訊息時自動執行
2. 判斷來源 : 檢查 `hcan->Instance` 區分 CAN1 或 CAN2
3. 讀取訊息 : 調用 `HAL_CAN_GetRxMessage()` 讀取訊息和資料
4. 處理資料 : 根據訊息 ID 進行相應處理

????

? ???????


系統初始化 (main)
1. MX_GPIO_Init() - GPIO 初始化
2. MX_CAN1_Init() - CAN1 硬體初始化
3. MX_CAN2_Init() - CAN2 硬體初始化
4. MX_USART1_UART_Init() - UART1 初始化（Titania）
5. MX_USART6_UART_Init() - UART6 初始化（調試）
CAN1/CAN2 過濾器配置 + 啟動
- 配置 CAN1 過濾器組 0-13
- 配置 CAN2 過濾器組 14-27
- 啟動 CAN1 和 CAN2 接收中斷
主循環 (while 1)
每 300ms:
├─ UART1 發送 87 bytes 到 Titania 無線模組
├─ CAN1 發送測試訊息 (ID=0x101)
└─ 更新計數器和時間戳記
中斷事件:
├─ CAN1 收到訊息 → HAL_CAN_RxFifo0MsgPendingCallback
└─ CAN2 收到訊息 → HAL_CAN_RxFifo0MsgPendingCallback

? ??????

CAN1 ??

- [ ] `hcan1.Instance = CAN1`
- [ ] `Prescaler = 5` (波特率 500kbps)
- [ ] `FilterBank = 0` (使用第 0 號過濾器)
- [ ] `FilterFIFOAssignment = CAN_RX_FIFO0`
- [ ] 已調用 `HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING)`

CAN2 ??

- [ ] `hcan2.Instance = CAN2`
- [ ] `Prescaler = 5` (波特率 500kbps)
- [ ] `FilterBank = 14` (Slave 區間起點)
- [ ] `FilterFIFOAssignment = CAN_RX_FIFO0`
- [ ] 已調用 `HAL_CAN_ActivateNotification(&hcan2, CAN_IT_RX_FIFO0_MSG_PENDING)`

UART1 ???Titania?

- [ ] `BaudRate = 9600`
- [ ] 訊息格式: 頭 (3B) + 資料 (84B) = 87 bytes
- [ ] 發送間隔: 300ms
- [ ] 計數器在 `Send_uart[86]`

UART6 ??????

- [ ] `BaudRate = 115200`
- [ ] 用於高速調試輸出

? ?????

功能	CAN1	CAN2	UART1	UART6
實例	CAN1	CAN2	USART1	USART6
波特率	500kbps	500kbps	9600	115200
過濾器	Bank 0-13	Bank 14-27	-	-
用途	數據收發	編碼器 /感測器	無線模組	調試
中斷	FIFO0	FIFO0	-	-

功能	CAN1	CAN2	UART1	UART6
發送函數	`CAN1_Send_Test()`	-	`HAL_UART_Transmit()`	`HAL_UART_Transmit()`

? ????

????

Q1: 如何改變 **CAN** 波特率?

```
// 修改 Prescaler 計算
// CAN_BaudRate = 45MHz / (Prescaler × 18)
// 例如要 250kbps: Prescaler = 10
hcan1.Init.Prescaler = 10;  // 250 kbps
```

Q2: 如何接收特定 **CAN ID** 的訊息?

```
// 修改過濾器設定
sFilterConfig.FilterIdHigh = 0x2020;     // ID 高位
sFilterConfig.FilterIdLow = 0x0000;      // ID 低位
sFilterConfig.FilterMaskIdHigh = 0xFFFF; // 遮罩高位
sFilterConfig.FilterMaskIdLow = 0x0000;  // 遮罩低位
```

Q3: UART 傳送超時怎麼辦?

```
// 增加超時時間或使用非阻斷模式
HAL_UART_Transmit(&huart1, Send_uart, 87, 5000);  // 5秒超時


// 或使用 DMA 模式
HAL_UART_Transmit_DMA(&huart1, Send_uart, 87);
```

文檔版本: 1.0
最後更新: 2026年 3月 5日
適用裝置: STM32F4 系列

?????????? Failsafe????????

在 FSAE 賽車的極限操作環境下，沒有完整 Failsafe 機制的電控系統是非常危險的。牽引力控制（ TC ）若只考慮理想狀態，放到實車上很容易因為邊界條件（ Edge Cases ）引發失控或硬體損壞。以下是實車環境中最致命的 5 個危險區域及防護對策:

10.1 ?????????????

- 危險情境：計算滑移率的公式為 $\lambda = \frac{R \omega - v}{v}$。當車輛靜止或極低速時，車速 $v \approx 0$。此時若輪速感測器有一點雜訊，計算出的 λ 會爆衝至無限大，導致 TC 瞬間介入瘋狂砍扭矩，造成車輛無法起步或劇烈抖動。
- 解決方案（低速屏蔽）：在控制器中加入速度門檻。當車速低於 $5,\text{km/h}$ 時，強制關閉 TC（ Bypass PI 控制器），直接 100% 輸出駕駛的油門需求。

10.2 ????? (WSS) ?????

- 危險情境： FSAE 車輛震動極大，感測器容易受震動或電磁干擾（如馬達高壓線）產生突波。若感測器瞬間讀到 $200,\text{km/h}$ 的假訊號， TC 會誤判為嚴重打滑而瞬間切斷動力，雜訊消失後又瞬間給油。這種強烈的扭矩衝擊（ Torque Vectoring Shock）輕則扯斷傳動軸，重則導致失控甩尾。
- 解決方案（訊號濾波與合理性檢查）：
  1. 加入低通濾波器（ Low-pass Filter）平滑訊號。
  2. 合理性檢查（ **Plausibility Check**）：限制輪速變化的最大加速度（ Slew Rate ）。若輪速在 $5,\text{ms}$ 內暴增等同於 $5g$ 的加速度，即違反物理法則，系統應直接丟棄該數據，維持上一筆有效值。

10.3 ??????????Regen ???????

- 危險情境：若 PI 控制器計算結果為負數（要求馬達輸出負扭矩），且系統未做好下限箝位， TC 會突然命令馬達進行動能回收（ Regen Braking ）。若發生在彎中，後輪會瞬間被煞死，引發嚴重的瞬間過度轉向（ Snap Oversteer ），導致賽車原地打轉。
- 解決方案：確保 Anti-windup 與扭矩輸出的下限死死鎖在 $0,\text{Nm}$。TC 只能「減少前進動力」，絕對不能主動給予煞車力。

10.4 ??????????????

- 危險情境： Quarter Car 模型是基於「直線行駛」設計。過彎時外側輪路徑長、內側輪短（阿克曼轉向幾何）。若 TC

[illegible]

- [illegible] [illegible] Steering Angle Sensor[illegible] Yaw Rate [illegible] TC [illegible] λ [illegible]

10.5 ????????Trail Braking ???

- [illegible] [illegible] Trail Braking [illegible] v [illegible] TC [illegible]
- [illegible] [illegible] Brake Pressure Sensor [illegible] $10,\text{bar}$ [illegible] TC [illegible]

---

11. TC ??????State Machine???

[illegible] FSAE [illegible]

1. [illegible] [illegible] $\rightarrow$ [illegible] Failsafe [illegible] TC[illegible]
2. [illegible] [illegible] $5,\text{km/h}$[illegible]
3. [illegible] [illegible]
4. [illegible] [illegible]
   - [illegible] $\rightarrow$ [illegible] **PI** [illegible]
   - [illegible] $\rightarrow$ [illegible] **(Bypass TC)**

Arduino ??????????

1. ??

本計時器使用 Arduino UNO 及紅外線感測器，實現單圈計時功能，適用於賽車計時應用。計時結果透過 UART (Serial Monitor) 顯示，並支援 `START` / `STOP` 指令控制。

2. ????

- Arduino UNO
- 紅外線感測器 (數位輸出) E18-D80NK
- 電腦 (序列副工具)

3. ????

- 紅外線感測器數位輸出接至 Arduino 的 `D2` 腳位 (`IR_PIN = 2`)
- VCC 及 GND 正確接線
- Arduino 透過 USB 連接電腦

4. ????

1. 開啟 **Serial Monitor** 並設定鮑率為 `115200`。
2. 輸入 `START` 指令：
   - 計時器開始運作。
   - 進入等待第一圈計時狀態。
3. 觸發紅外線感測器 (例如模型車通過感測器)：
   - 記錄每一圈的時間。
   - 依序輸出 LAP 計時結果，格式如下：

     ```
     LAP X "HH:MM:SS.mmm"
     ```

   - LAP 計數從 0 開始。
4. 輸入 `STOP` 指令：
   - 停止計時。
   - 顯示 最快單圈 與 最慢單圈 時間 (排除 LAP 0)。
   - 停止後，即使感測器被觸發，計時器不會再記錄 LAP。

5. ??????

```
Lap Timer Ready. Type 'START' to begin.
Timer Started! Waiting for first lap...
LAP 0 "00:00:05.088"
LAP 1 "00:00:09.081"
LAP 2 "00:00:14.320"
LAP 3 "00:00:43.215"
---------------------------
Timer Stopped.
Fastest Lap "00:00:09.081"
Slowest Lap "00:00:43.215"
---------------------------

Lap Timer Ready. Type 'START' to begin.
```

6. ????

- 確保紅外線感測器安裝穩固，避免干擾計時。
- 只計算 `LAP 1` 之後的最快與最慢圈速 (不包含 `LAP 0`)。
- `STOP` 指令後計時器不會繼續運作，需重新輸入 `START`。

此計時器，透過 UART 輸出紀錄結果，便於數據記錄與分析。

F042??BOOT0_Pin ?MCU ????????

1. 開啟 STM32CubeProgrammer image
2. 點選左邊的「 OB (Option Bytes)」 image image
3. 依下表修改:

參數	建議值	說明
nBOOT_SEL	0	使用 BOOT0 腳決定啟動來源
nBOOT0	0	BOOT0=0 時啟動 Flash
RDP	0xAA	關閉讀取保護
BOOT_LOCK	0	不鎖定啟動

4. 點擊「 Apply」
5. 確保 BOOT0 接地（ 0）
6. 再燒錄一次程式，然後拔掉 ST-Link，重新上電測試。

Savaresi?Active Braking Control Systems Design for Vehicles? Ch2 & Ch3 ????

Savaresi?Active Braking Control Systems Design for Vehicles? Ch2 & Ch3 ????

> 著重與你 FSAE / QuarterCar / TC 專案直接相關的部分：輪胎模型、單 / 雙輪模型、線性化、以及連續作動器下的輪滑率控制架構與 PI 設計。

Ch2?Control-oriented Models of Braking Dynamics

2.1 ????

- 目的：建立「控制導向」的制動動力學模型，之後所有 ABS/TC/ 穩定控制的設計都建立在這些模型上。
- 使用兩種車輛模型：
  - **Single-corner model**：單一輪子 + 一半車身質量，適合 local ABS/TC 控制器設計與分析。
  - **Double-corner model**：前後兩輪 + 動態荷重轉移，適合研究前後軸耦合與 ABS/TC 性能邊界。

2.2 ??–??????Tyre–road Contact Forces?

- 接觸點受力分解:
  - 垂直力: F_z
  - 縱向力: F_x（驅動 /制動）
  - 橫向力: F_y（轉向）
- 一般寫成:
  - $F_x = F_x(F_z, \alpha_t, \gamma, \lambda)$
  - $F_y = F_y(F_z, \alpha_t, \gamma, \lambda)$
  - α_t: 輪胎側偏角; γ: 外傾角; λ: 縱向滑移率。

2.2.1 ???????

- 完整定義（含側偏）: $$ \lambda = \frac{v - \omega r \cos(\alpha_t)}{\max{v, \omega r \cos(\alpha_t)}} $$
- 只看直行縱向（$\alpha_t \approx 0$、煞車時 $v \ge \omega r$）: $$ \lambda = \frac{v - \omega r}{v}, \quad \lambda \in [0,1] $$
  - $\lambda = 0$: 純滾動
  - $\lambda = 1$: 輪鎖死（ABS 想避免的狀態）

2.2.2 ???????

- 為了讓模型與 F_z 分離，定義:
  - $\mu_x = F_x/F_z$
  - $\mu_y = F_y/F_z$
- 因此:
  - $F_x = F_z \mu_x(\alpha_t, \gamma, \lambda)$
  - $F_y = F_z \mu_y(\alpha_t, \gamma, \lambda)$
- 在控制設計中，多半假設 F_x, F_y 對 F_z 近似線性（大荷重才會明顯飽和）。

2.2.1 ?????Pacejka?Magic Formula?

- 書中先介紹完整 **Pacejka Magic Formula**（長版），說明真實輪胎在不同 F_z、α_t、γ 下的 Fx, Fy 如何用一大堆參數擬合。
- 對縱向力:
  - 形式類似: $F_x = D_x \sin{C_x \arctan[B_x \kappa - E_x(\cdots)]}$
  - κ 為歸一化 slip，D_x, C_x, B_x, E_x 等為由實驗擬合的常數。
- 對橫向力有類似結構，只是參數與輸入換成 α_t。
- 重點: **Pacejka** 非常準，但不適合做解析穩定性推導（太多參數）。因此書後面轉用簡化模型。

2.2.1 ???????Burckhardt

- 為了分析方便，書中採用 **Burckhardt** 模型 近似縱向摩擦曲線: $$ \mu(\lambda; \vartheta_r) = \vartheta_{r1}\bigl(1 - e^{-\lambda \vartheta_{r2}}\bigr) - \lambda \vartheta_{r3} $$

- 三個參數 $\vartheta_r = [\vartheta_{r1}, \vartheta_{r2}, \vartheta_{r3}]$ 可以描述各種路況（乾、溼、雪、石板路）。
- 書中給出典型值（例如乾柏油 $\mu_{peak} \approx 1.2$ 等），並畫出 $\mu(\lambda)$ 在不同路面的形狀：
  - 對所有路面：$\mu(\lambda)$ 在某個 λ_{peak} 有峰值，之後下降。
  - $\lambda = 0$：無縱向力；$\lambda = 1$：縱向力低於峰值且橫向力幾乎消失（車失去方向性）。

> 給你的應用：你之前用的 $\mu(\lambda) = c \sin(b \arctan(a\lambda))$ 其實就是 Burckhardt/簡化 Pacejka 的特例，書中分析完全可以直接套用到你的 Simulink Plant。

2.3 Single-corner Model??????

- 結構：一個輪胎 + 對應的四分之一或半車身質量（制動方向）。
- 動態方程（縱向）：
  - 輪子：$$ J \dot{\omega} = -T_b - r F_x $$
  - 車身：$$ m \dot{v} = F_x $$
  - 其中 $F_x = F_z \mu(\lambda)$，$\lambda = (v - \omega r)/v$。
- 這就是你 QuarterCar Plant 的原型，只是書中是煞車（$-T_b$），你現在換成驅動（$+T_{drive}$）。

2.5.1 ???????????

- 書中將車速 v 視為「慢變參數」，推導出只看 λ 的一階系統：$$ \dot{\lambda} = -\frac{1-\lambda}{J\omega}\bigl(\Psi(\lambda) - T_b\bigr) $$ 其中 $\Psi(\lambda) \approx r F_z \mu(\lambda)$。
- 結論：
  - 給定常制動扭矩 T_b，最多有兩個平衡點（與 $\Psi(\lambda)$ 曲線的交點）。
  - 小的那一個 λ_1：穩定（在 $\mu(\lambda)$ 峰值左邊）。
  - 大的那一個 λ_2：不穩定（在峰值右邊）。
  - 當 $T_b > \max_\lambda \Psi(\lambda)$ 時，無平衡點 → 系統會一直往 $\lambda \to 1$（輪鎖死）。

> 這段就是在數學上證明：若直接開迴路施加大煞車扭矩，輪胎運作在 μ 曲線右邊（滑移大），是天生不穩定，必須靠 ABS/TC 控制器把 λ 拉回峰值左側。

2.5.1 ???????????

- 在某個工作點 (λ, v, T_b) 作小訊號線性化，定義：

  - $\delta T_b = T_b - \bar T_b$
  - $\delta \lambda = \lambda - \bar \lambda$
- 使用一階 Taylor 展開:
  - $\mu(\lambda) \approx \mu(\bar\lambda) + \mu_1(\bar\lambda),\delta\lambda$
  - $\mu_1(\bar\lambda) = \partial \mu/\partial \lambda |_{\bar\lambda}$
    (摩擦曲線斜率)。
- 得到以 δT_b → $\delta \lambda$ 的傳遞函數: $$ G_\lambda(s) = \frac{\delta \lambda(s)}{\delta T_b(s)} = \frac{r}{J v},\frac{1}{s + a(\bar\lambda)} $$
  - $a(\bar\lambda)$ 是跟 $\mu(\bar\lambda), \mu_1(\bar\lambda), m, J, r, v$ 有關的常數, 符號會隨 linearisation 點在 μ 曲線的哪一邊而改變。
- 對 輪速 ω 與 歸一化減速度 $\eta = -\dot v/g$ 也有類似的 SISO 傳遞函數, 方便設計輪速 /減速度控制器。

> 結論: 在固定車速近似下, 單輪滑移率對制動扭矩的開迴路模型是一階系統, 很適合用 PI/PID 直接調參, 也解釋了為何你現在用 PI lambda-loop 感覺「像一階系統」。

2.4 / 2.5.2 Double-corner Model????? + ?????

- 引入:
  - 前輪、後輪各自有 λ_f, λ_r, 且 F_{zf}, F_{zr} 會隨制動動態改變 (荷重轉移)。
- 推導出 **2×2 MIMO** 線性系統 : $$ \begin{bmatrix} \delta \lambda_f \ \delta \lambda_r \end{bmatrix} = \begin{bmatrix} G_{ff}(s) & G_{fr}(s) \ G_{rf}(s) & G_{rr}(s) \end{bmatrix} \begin{bmatrix} \delta T_{bf} \ \delta T_{br} \end{bmatrix} $$
- 分析重點:
  - 對角項 G_{ff}, G_{rr}: 類似 single-corner 的一階行為。
  - 交叉項 G_{fr}, G_{rf}: 描述前輪制動對後輪滑移、及其反向影響。
  - 強調: 在合理慣量與距離參數下, 系統行為在高頻與單輪模型相似, 且可以視為「弱耦合」, 因此 實務上可用前 /後軸各自一個 **SISO ABS/TC** 控制器 。

> 對你 FSAE 後驅電車: 以後若要加入前輪制動、或四輪獨立 ABS, 可以參考這章分析前後輪 slip 耦合, 決定是否需要 MIMO 控制或相對 slip 控制 (書中建議後輪上用 relative slip 會比較 robust)。

Ch3?Braking Control Systems Design – Continuous Actuators

> 這章就是在單 /雙輪模型上疊控制器，重點是 連續可調的制動作動器 （ EHB / EMB ），等價到你的「馬達扭矩可連續控制」的 TC 情境。

3.1 ????

- 假設制動作動器可近似為一階系統 + delay: $$ G_{caliper}(s) = \frac{\omega_{act}}{s + \omega_{act}} e^{-s\tau} $$
  - 書中例子: $\omega_{act} \approx 70,\text{rad/s}, ; \tau \approx 10,\text{ms}$。
- 目標:
  - 設計 輪滑率控制器 讓 λ 追踪設定值 λ_{ref}（接近 μ 峰值）。
  - 提供啟用 /解除 ABS 的 supervisory 邏輯（因為不是隨時都要控制器介入）。

3.2 Wheel Slip Control???????

- 優缺點比較:
  - 輪減速度控制（ **$\dot \omega$** 或 **η**）：量測容易，但對路況敏感，常用 heuristic threshold 而不是純線性控制。
  - 輪滑率控制（ **λ**）：理論漂亮、動態 robust，易延伸到 TC / ESC; 缺點是 v 需要估測，低速 & 低 slip 區段難搞。
- 書中的設計方向: 主力放在 **slip control**，配合良好的車速估測與啟閉策略。

3.4 ??????????

- 使用 Ch2 線性化得到的 SISO 模型: $$ G_\lambda(s) = \frac{\delta \lambda(s)}{\delta T_b(s)} = \frac{k}{s + a} $$
- 控制器選擇:
  - 標準 PI: $C(s) = K_p + K_i/s$。
  - 目標閉迴路：一階或二階期望響應（指定頻寬、阻尼）。
- 調參考點:
  - 在 乾地、代表車速 的工作點做設計，確保此點 closed-loop 穩定且響應良好。
  - 再檢查其他路況（濕地、冰雪）與車速下的 robust 性能。
  - 若需要，加入 anti-windup 與輸出飽和處理（因為制動扭矩有限）。

>

> 對你的 TC: 等效就是讓 T_{drive} 經過 min() / saturator, 再由 PI 控 λ ，結構跟書的 ABS 幾乎一樣，只是扭矩作用方向相反。

3.6 ?????????

- 書中利用 Single-corner 模型 + EMB 作動器，跑一系列數值實驗:
  - 比較 open-loop 煞車 vs slip control 煞車的煞停距離與穩定性。
  - 改變 Kp, Ki, 看 λ 超調、 settling time 的差異。
  - 檢查不同路況（用 Burckhardt 參數換 μ 曲線）下控制器的穩健性。
- 報告中會畫:
  - $\lambda(t)$、$v(t)$、$T_b(t)$ 曲線。
  - 以此為根據，建議 Kp/Ki 的可接受範圍。

> 你可以把這一段流程直接套用到 FSAE TC: 先用 simulation 掃一圈 Kp/Ki, 在不同 μ 路況下看 $\lambda(t)$、$a_x(t)$、$wheel torque(t)$, 再把最穩定的一組下到 STM32 實車測試。

3.7 ?? / ?????Activation & Deactivation?

- ABS/TC 不應該一直啟用，必須符合某些條件才接管輪胎扭矩:
  - 警戒條件:
    - 車速高於門檻（例如 > 10 km/h）。
    - 踏板輸入達到一定強度（ ABS）或油門足夠大（ TC）。
    - 輪滑率或減速度超過閾值。
  - 解除條件:
    - 車速太低（ < 幾 km/h）。
    - λ 或 η 已回到安全範圍一段時間。
- 這部分書中給的是邏輯結構與典型閾值設定範例，可以直接翻成你的 traction mode 切換邏輯。

3.8 ?? Double-corner ??????

- 目的: 確認 Ch2 double-corner 耦合對前 /後軸 slip control 的影響。
- 結論:
  - 若各軸各有一個獨立的 slip controller, 且調參得當，系統可視為「弱耦合」，不一定需要 full MIMO 控制。
  - 後軸控制若改用 **relative slip**（如 $\lambda_r - \lambda_f$）會更 robust, 尤其在大制動＋轉彎情境下。

>

> □□
> λ vs □□ λ□□

????????????

1. □□□□□□ □
   - □□□□□□ Pacejka□ $\mu = c \sin(b \arctan(a\lambda))$□□□□□□□□ Burckhardt/□□□□□□□□ Ch2 □□□□□□□□□□□□□□
2. □□□□□□ □
   - □ Ch2 □□□□□□□ $G_\lambda(s)$□□□□□□□□ PI □□□□□□□□□□ μ □□□□□ robust □□□
3. □□□□ □
   - □ Ch3 □□□□ `lambda_ref -> PI -> torque_limit -> plant` □□□□□□□□□□□□□□□□□□□ STM32 + motor inverter TC □□□
4. □□□□ □
   - □□□□ FSAE TTC data □□□□ Magic Formula□ Ch2/Ch3 □□□□□□□□□ $\mu(\lambda)$ □□□□□□□□□□□□ roadCoeffs □□□□□□□□

Time Step (??) ????????

Time Step (??) ????????

在 MATLAB 腳本中呼叫 `sim()` 時，可以直接強制覆寫 Simulink 模型裡的步長設定。這能確保你每次跑腳本的結果都是一致的。

?? A?????? Variable-step (????) —— ???????

在 Variable-step 模式下，Simulink 會自動決定步長。但為了解決「畫圖時曲線不夠圓滑、有很多折角」的問題，我們通常只需要限制 `MaxStep`（最大步長）。

- 設定原則（經驗法則）：通常設定為「總模擬時間的 1/1000 到 1/5000」。例如：模擬 10 秒，`MaxStep` 可以設為 `0.01`。這樣可以保證每一秒至少有 100 個數據點，畫出來的圖就會非常平滑。
- **MATLAB** 語法:

```
% 限制最大步長為 0.01 秒 (其餘交給 Simulink 自動調整)
out = sim('my_model', 'MaxStep', '0.01');
```

?? B??? Fixed-step (????) —— ??????????????

如果你有用到 PWM（脈衝寬度調變）、數位控制器 (Discrete Controller)，或是未來要把模型轉成 C Code 燒進微控制器 (Arduino / DSP)，你就必須使用 Fixed-step。

- 設定原則（經驗法則）：必須比你系統中「變化最快」的訊號還要快上 10 到 20 倍（這與取樣定理有關）。
  - 例子 1（一般控制系統）：如果你的控制器迴圈是 100 Hz (0.01 秒執行一次)，你的 FixedStep 就設為 `0.01`。
  - 例子 2（電力電子/馬達 **PWM**）：如果你的 PWM 切換頻率是 10 kHz（週期 0.0001 秒），為了精準捕捉開關的瞬間，你的 FixedStep 通常要設為 `1e-5`（也就是 0.00001 秒）或更小。
- **MATLAB** 語法:

```
% 強制將 Solver 改為 Fixed-step，並將步長鎖死在 0.001 秒
out = sim('my_model', ...
```

```
            'SolverType', 'Fixed-step', ...
            'FixedStep', '0.001');
```

(註：語法中的 `...` 是 *MATLAB* 的換行符號，用來把很長的程式碼折成多行以方便閱讀)

Titania (AMB3626) ??????????

> 模組型號: Würth Titania 2607011111100x (AMB3626)
> 硬體相關手冊章節: Manual-um-titania-260701111100x (rev4.5) - 第 2 章、第 3 章、第 7 章
> 主要參考標準: EN 300 220 (歐盟無線設備規範)

? ??????????

- [ ] VCC 已穩定供電 (2.0–3.6 V, 建議 3.3 V)
- [ ] /RESET 腳位已釋放 (被 RC 電路或軟體拉高)
- [ ] **TRX_DISABLE** 接 **GND** (一定要)
- [ ] UART TX/RX 已接到 STM32
- [ ] /CONFIG 腳位接到 STM32 GPIO (選用, 但建議接)
- [ ] /RTS 腳位接到 STM32GPIO 或監測 (強烈建議)
- [ ] 沒有無線訊號、 UART 也空閒時, 才改設定
- [ ] 改完非揮發參數後, 務必 **CMD_RESET_REQ**

? ?????????

1. VCC ? /RESET ?

腳位定義:

腳位	標號	用途	電壓	說明
VCC	1	電源正	+3.3 V (範圍 2.0–3.6 V)	模組主電源
GND	2 (多個)	地	0 V	必須與 MCU 共地

腳位	標號	用途	電壓	說明
/RESET	某腳	模組重置	開漏輸出或 RC 延遲	上電時 必須先低、再高

上電時序（極其重要）：

時間線：

T=0 ms: VCC 開始上升

T=0~2 ms: VCC 還在上升，/RESET 必須保持 LOW（不要放高）

T=2 ms: VCC 穩定在 3.3 V

T=2~10 ms: /RESET 開始被 RC 電路或軟體拉高

T=10~50 ms: Titania 內部初始化進行中

T=50 ms+: 模組完成啟動，UART + /CONFIG 開始工作

⚠◌ 如果 /RESET 在 VCC 還不穩時就被拉高，模組會受損或無法啟動。

建議實作：

- 硬體 **RC** 延遲方案（最簡單）：

```
VCC ──[10kΩ]──┬──/RESET
              ├──[100nF]──┴──GND
              └──外部按鍵到 GND（選用）

效果：上電時，/RESET 會被 100nF 電容延遲 ~10~50 ms 才慢慢被拉高，
      自動保證 VCC 先穩定再放開 /RESET。
```

- 軟體方案（ **MCU** 控制）：

```
void Titania_PowerUp(void)
{
    // 上電時 /RESET 已經被拉低（可能通過外部電路或預設狀態）
    HAL_GPIO_WritePin(RESET_PORT, RESET_PIN, GPIO_PIN_RESET);  // 確保低
    HAL_Delay(10);   // 等 VCC 穩定
    HAL_GPIO_WritePin(RESET_PORT, RESET_PIN, GPIO_PIN_SET);    // 拉高
    HAL_Delay(100);  // 等模組啟動完成
}
```

---

2. TRX_DISABLE ??????????

功能： 當 TRX_DISABLE 被拉高時，模組無法傳送無線訊號（ RX 仍可用）。用來省功耗或避免干擾。

設定參數時的用途：雖然不是「必須」讓它 HIGH，但強烈建議在整個參數設定過程中保持 **LOW**（讓無線 **TX** 可用，但我們不會在設定時送訊號）。

狀態	說明
LOW（接 **GND**）	無線 TX/RX 都啟用（正常工作模式）
HIGH	無線 TX 被禁用，模組停止傳送（僅 RX）
浮接	不允許，會造成無法預知的行為

建議實作：

```
方案 1（最簡單，出廠推薦）：
  TRX_DISABLE ──→ GND
   （直接接地，永遠 LOW，模組正常工作）


方案 2（後期要動態禁用 TX 時）：
  TRX_DISABLE ──[10kΩ pull-down]──┬──STM32 GPIO（或保留浮接用軟體控制）
                                  └──GND


  HAL_GPIO_WritePin(TRX_DIS_PORT, TRX_DIS_PIN, GPIO_PIN_RESET);  // 允許 TX
  HAL_GPIO_WritePin(TRX_DIS_PORT, TRX_DIS_PIN, GPIO_PIN_SET);    // 禁用 TX
```

---

3. /CONFIG ??Mode Selector?Transparent ? Command?

功能：檢測「下降沿」（HIGH → LOW）來切換模式。

邊緣	狀態轉換
下降沿（HIGH → LOW）	進入或跳出 Command Mode
LOW 狀態保持	停留在 Command Mode
HIGH 狀態保持	Transparent Mode

時序要求（最重要）：

只有在「沒有無線收發、也沒有 UART 資料傳輸」的安全區間才能切模式。手冊建議看 **/RTS** 線：

- `/RTS = HIGH`（模組忙，有 UART 活動）→ 不要切模式
- `/RTS = LOW`（模組空閒）→ 可以安全地做下降沿切模式

```
void Titania_EnterCommandMode(GPIO_TypeDef *cfg_port, uint16_t cfg_pin,
                              GPIO_TypeDef *rts_port, uint16_t rts_pin)
{
    // 1. 等待 /RTS = LOW（模組空閒），timeout 100 ms
    uint32_t timeout = 100;
    while (HAL_GPIO_ReadPin(rts_port, rts_pin) == GPIO_PIN_SET && timeout--)
    {
        HAL_Delay(1);
    }

    // 2. 確認 /CONFIG 已經 HIGH（停留 Transparent Mode）
    HAL_GPIO_WritePin(cfg_port, cfg_pin, GPIO_PIN_SET);
    HAL_Delay(10);

    // 3. 做下降沿
    HAL_GPIO_WritePin(cfg_port, cfg_pin, GPIO_PIN_RESET);
    HAL_Delay(5);

    // 4. 保持 LOW 一段時間（表示「停留 Command Mode」）
    //  （不需要立即拉回 HIGH）

    HAL_Delay(50);  // 給模組時間完成模式切換
}

void Titania_ExitCommandMode(GPIO_TypeDef *cfg_port, uint16_t cfg_pin)
{
    // 1. /CONFIG 保持 LOW 一段時間（已在 Command Mode 中）
    // 2. 再做一次下降沿，跳出 Command Mode

    HAL_GPIO_WritePin(cfg_port, cfg_pin, GPIO_PIN_SET);   // 拉 HIGH
    HAL_Delay(10);
    HAL_GPIO_WritePin(cfg_port, cfg_pin, GPIO_PIN_RESET); // 拉 LOW（下降沿）
    HAL_Delay(5);

    // 3. 拉 HIGH 回 Transparent Mode
    HAL_GPIO_WritePin(cfg_port, cfg_pin, GPIO_PIN_SET);
    HAL_Delay(50);
}
```

4. UART ?????TX?RX?/RTS?/CTS?

最小硬體連接（必須）：

訊號	**Titania** 腳	**STM32** 腳	說明
UTXD	Titania TX	STM32 RXx	模組發送，MCU 接收
URXD	Titania RX	STM32 TXx	MCU 發送，模組接收
GND	GND	GND	必須共地

選用但強烈建議的流控腳位：

訊號	**Titania** 腳	**STM32** 腳	說明
/RTS	模組 RTS	STM32 GPIO 輸入	模組輸出，LOW = 可以發送，HIGH = 忙碌
/CTS	模組 CTS	STM32 GPIO 輸出	若不用，直接接 CTS flow control GND

為什麼要監測 **/RTS**？

- `/RTS = HIGH`：模組 UART buffer 滿或正在無線收發，MCU 不應再丟資料。
- `/RTS = LOW`：安全區間，可以發 UART 命令或改設定。

建議實作：

```
#define UART_RX_PIN     GPIO_PIN_10  // STM32 PA10 (USART1 RX)
#define UART_TX_PIN     GPIO_PIN_9   // STM32 PA9 (USART1 TX)
#define RTS_PIN         GPIO_PIN_11  // STM32 PA11 (輸入，讀取 Titania /RTS)
#define CTS_PIN         GPIO_PIN_12  // STM32 PA12 (輸出，若不用直接接 GND)

void Titania_UART_Init(void)
{
    // UART1 初始化為 9600 8N1
    huart1.Instance = USART1;
    huart1.Init.BaudRate = 9600;
    huart1.Init.WordLength = UART_WORDLENGTH_8B;
    huart1.Init.StopBits = UART_STOPBITS_1;
    huart1.Init.Parity = UART_PARITY_NONE;
    huart1.Init.Mode = UART_MODE_TX_RX;
    HAL_UART_Init(&huart1);

    // GPIO 設定: /RTS 為輸入（讀模組狀態）
```

```
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    GPIO_InitStruct.Pin = RTS_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    // /CTS 為輸出且拉低（若不用 CTS flow，直接 LOW）
    GPIO_InitStruct.Pin = CTS_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
    HAL_GPIO_WritePin(GPIOA, CTS_PIN, GPIO_PIN_RESET);
}

HAL_StatusTypeDef Titania_SendUARTSafe(const uint8_t *data, uint16_t len)
{
    // 先等 /RTS = LOW（模組空閒），timeout 100 ms
    uint32_t timeout = 100;
    while (HAL_GPIO_ReadPin(GPIOA, RTS_PIN) == GPIO_PIN_SET && timeout--)
    {
        HAL_Delay(1);
    }
    if (timeout == 0) return HAL_TIMEOUT;

    // 發送
    return HAL_UART_Transmit(&huart1, (uint8_t *)data, len, 100);
}
```

5. ANT??????

最小設置:　　不需要　　MCU 控制，但　　PCB layout 要遵守手冊規則:

- 天線線路　　旁邊要有「淨空區」（通常　　10~15 mm），不要拉數位線穿過。
- 天線匹配電路　　按手冊　　reference design 來，不要亂改。
- 若用外部天線，焊接要牢固，斷線會影響通訊距離。

? ??????????

?????

```
START
  ↓
[1] 上電 & /RESET 釋放
  └→ TRX_DISABLE = LOW (允許無線 TX/RX)
  └→ 等 100 ms 模組啟動完成
  ↓
[2] 初始化 STM32 UART (9600 8N1)
  ↓
[3] 監測 /RTS, 等它變 LOW (模組空閒)
  ↓
[4] 進入 Command Mode (/CONFIG 下降沿)
  └→ 等 /CONFIG 穩定在 LOW 50 ms
  ↓
[5] 透過 UART 發送 CMD_SET_REQ (改 baudrate/channel/NetID…)
  └→ 等 50~200 ms 回應
  ↓
[6] 檢查回應 Status (0x00 = OK)
  ├─ OK → 步驟 [7]
  └─ FAIL → 重試或 debug
  ↓
[7] 再監測 /RTS = LOW
  ↓
[8] 發送 CMD_RESET_REQ (軟重啟)
  └→ 等 500 ms 模組重啟完成
  ↓
[9] 改 STM32 UART 波特率 (新的 baudrate)
  ↓
[10] 退出 Command Mode (/CONFIG 再做下降沿)
  └→ 拉 HIGH, 再下降沿, 再拉 HIGH
  ↓
[11] 回到 Transparent Mode, 可以正常收發無線訊號
  ↓
DONE
```

? ??????????????

```c
#include "stm32f1xx_hal.h"

// □□□□
#define RESET_PORT      GPIOA
#define RESET_PIN       GPIO_PIN_0
#define TRX_DIS_PORT    GPIOA
#define TRX_DIS_PIN     GPIO_PIN_1
#define CONFIG_PORT     GPIOA
#define CONFIG_PIN      GPIO_PIN_2
#define RTS_PORT        GPIOA
#define RTS_PIN         GPIO_PIN_3

extern UART_HandleTypeDef huart1;

// □□ Checksum
static uint8_t CalcChecksum(uint8_t *buf, uint8_t len)
{
    uint8_t cs = 0;
    for (uint8_t i = 0; i < len; i++)
        cs ^= buf[i];
    return cs;
}

// □□ UART □□□□□ Checksum□
static HAL_StatusTypeDef Titania_SendCommand(uint8_t cmd,
                                             const uint8_t *data,
                                             uint8_t data_len)
{
    uint8_t frame[64];
    uint8_t idx = 0;

    frame[idx++] = 0x02;  // Start
    frame[idx++] = cmd;
    frame[idx++] = data_len;
    for (uint8_t i = 0; i < data_len; i++)
        frame[idx++] = data[i];

    uint8_t cs = CalcChecksum(frame, idx);
    frame[idx++] = cs;
```

```
    return HAL_UART_Transmit(&huart1, frame, idx, 100);
}

// 等待 /RTS = LOW（模組空閒）
static HAL_StatusTypeDef Titania_WaitReady(uint32_t timeout_ms)
{
    while (HAL_GPIO_ReadPin(RTS_PORT, RTS_PIN) == GPIO_PIN_SET && timeout_ms--)
    {
        HAL_Delay(1);
    }
    return (timeout_ms == 0) ? HAL_TIMEOUT : HAL_OK;
}

// 進 Command Mode
static void Titania_EnterCmdMode(void)
{
    Titania_WaitReady(100);
    HAL_GPIO_WritePin(CONFIG_PORT, CONFIG_PIN, GPIO_PIN_SET);   // HIGH
    HAL_Delay(10);
    HAL_GPIO_WritePin(CONFIG_PORT, CONFIG_PIN, GPIO_PIN_RESET); // 下降沿
    HAL_Delay(50);
}

// 退出 Command Mode
static void Titania_ExitCmdMode(void)
{
    Titania_WaitReady(100);
    HAL_GPIO_WritePin(CONFIG_PORT, CONFIG_PIN, GPIO_PIN_SET);   // HIGH
    HAL_Delay(10);
    HAL_GPIO_WritePin(CONFIG_PORT, CONFIG_PIN, GPIO_PIN_RESET); // 下降沿
    HAL_Delay(5);
    HAL_GPIO_WritePin(CONFIG_PORT, CONFIG_PIN, GPIO_PIN_SET);   // 回 HIGH
    HAL_Delay(50);
}

// 上電與初始化
void Titania_PowerUp(void)
{
    // 1. GPIO 初始化
    GPIO_InitTypeDef GPIO_InitStruct = {0};
```

```
    GPIO_InitStruct.Pin = RESET_PIN | TRX_DIS_PIN | CONFIG_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = RTS_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    // 2. □□ /RESET □□TRX_DISABLE □□/CONFIG □
    HAL_GPIO_WritePin(RESET_PORT, RESET_PIN, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(TRX_DIS_PORT, TRX_DIS_PIN, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(CONFIG_PORT, CONFIG_PIN, GPIO_PIN_SET);
    HAL_Delay(10);

    // 3. □□ /RESET
    HAL_GPIO_WritePin(RESET_PORT, RESET_PIN, GPIO_PIN_SET);
    HAL_Delay(100);  // □□□□□
}

// □□□□□□
typedef struct
{
    uint32_t baudrate;  // 1200~115200
    uint8_t  channel;   // 0~4
    uint16_t netid;     // NetID
    uint16_t addr;      // Addr LSB
} TitaniaSettings;

HAL_StatusTypeDef Titania_Configure(const TitaniaSettings *settings)
{
    uint8_t payload[10];
    uint8_t resp[10];
    HAL_StatusTypeDef st;

    // 1. □ Command Mode
    Titania_EnterCmdMode();
    HAL_Delay(100);
```

```
// 2. 改 UART Baudrate（非揮發）
payload[0] = 0x50;  // UART_Baudrate mem pos
payload[1] = 0x04;  // 4 bytes
payload[2] = (uint8_t)(settings->baudrate & 0xFF);
payload[3] = (uint8_t)((settings->baudrate >> 8) & 0xFF);
payload[4] = (uint8_t)((settings->baudrate >> 16) & 0xFF);
payload[5] = (uint8_t)((settings->baudrate >> 24) & 0xFF);

Titania_SendCommand(0x09, payload, 6);
HAL_Delay(100);
HAL_UART_Receive(&huart1, resp, 5, 200);

if (resp[3] != 0x00) return HAL_ERROR;  // Status check

// 3. 改頻道（非揮發）
payload[0] = 0x2A;  // PHY_DefaultChannel mem pos
payload[1] = 0x01;  // 1 byte
payload[2] = settings->channel;

Titania_SendCommand(0x09, payload, 3);
HAL_Delay(100);
HAL_UART_Receive(&huart1, resp, 5, 200);

if (resp[3] != 0x00) return HAL_ERROR;

// 4. 改 NetID（需要補上正確的 mem pos）
// payload[0] = 0xXX;  // MAC_DefaultDestNetID mem pos (TODO)
// payload[1] = 0x02;  // 2 bytes
// payload[2] = (uint8_t)(settings->netid & 0xFF);
// payload[3] = (uint8_t)(settings->netid >> 8);
// Titania_SendCommand(0x09, payload, 4);
// HAL_UART_Receive(&huart1, resp, 5, 200);

// 5. 軟重啟讓設定生效
Titania_SendCommand(0x05, NULL, 0);
HAL_Delay(100);
HAL_UART_Receive(&huart1, resp, 5, 500);

// 6. 改 STM32 UART 波特率
```

```
    huart1.Init.BaudRate = settings->baudrate;
    HAL_UART_Init(&huart1);
    HAL_Delay(100);

    // 7. 退出 Command Mode
    Titania_ExitCmdMode();

    return HAL_OK;
}

// 使用範例
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();  // 先 9600

    // 上電
    Titania_PowerUp();
    HAL_Delay(200);

    // 設定參數
    TitaniaSettings cfg = {
        .baudrate = 57600,
        .channel = 2,
        .netid = 0x0001,
        .addr = 0x0005
    };
    Titania_Configure(&cfg);

    while (1)
    {
        // 主應用程式...
    }
}
```

? ???? & ??

問題	原因	解決方案
上電後模組不回應	/RESET 在 VCC 未穩時被拉高，或 /RESET 沒被正確拉高	檢查 RC 延遲電路、或用示波器看 /RESET 時序
無法進 Command Mode	/CONFIG 下降沿時機不對，或被拉高時間不足	確保 /RTS = LOW 後再做下降沿，並保持 LOW 50 ms+
改參數後 UART 通訊失敗	沒有做 CMD_RESET_REQ、或 STM32 UART 波特率沒同步改	一定要 reset，再改 STM32 baud rate
頻道、NetID 沒生效	memory position 填錯，或 status byte 不是 0x00	對照手冊第 8 章確認 memory position；檢查 Checksum
/RTS 一直高	UART buffer 滿，或模組忙碌	等待、或檢查有沒有大量無線訊號干擾

? ??????????

設定項目	需進 **Command Mode**	需要 **Reset**	**UART** 切換	安全區間條件
UART_Baudrate	□	□	□ □□□ STM32 baud	/RTS=LOW
PHY_DefaultChannel	□	□	□ □□□	/RTS=LOW
MAC_DefaultDestNetID	□	□	□ □□□	/RTS=LOW
MAC_DefaultDestAddrLSB	□	□	□ □□□	/RTS=LOW
PHY_PAPower	□	□	□ □□□	/RTS=LOW
CMD_FACTORY_RESET	□	□	□ □□□ STM32 baud □ 9600	/RTS=LOW
CMD_SET_CHANNEL_REQ (揮發)	□	□	□ □□□	/RTS=LOW

? ?????????

- ☐ PCB 佈局按手冊 reference layout
- ☐ VCC 穩壓、旁路電容（ 100 nF + 1 µF）
- ☐ /RESET 有 RC 延遲或軟體控制
- ☐ TRX_DISABLE 接 GND（正常模式）
- ☐ UART TX/RX 正確連接且沒反接

- [ ] /RTS 接到 GPIO 或監測
- [ ] /CONFIG 接到 GPIO (可進 Command Mode)
- [ ] 天線線路淨空、不穿過數位線
- [ ] 第一次設定時用邏輯分析儀監看時序確認
- [ ] 改完參數後驗證設定有沒有真的生效（ CMD_GET_REQ 讀回來確認）

? ??

- **Würth Titania Manual:** Manual-um-titania-260701111100x (rev4.5) - 第 2、3、7、8 章
- **HAL Reference:** STM32 HAL Driver 官方文檔
- **EN 300 220:** 歐盟無線設備規範

最後更新: 2026-03-10

作者備註: 本硬體指南強調「時序」與「狀態機」，設定參數時務必遵守 /RTS、/CONFIG 的時序要求。

Titania (AMB3626) UART ??????????

> 模組型號: Würth Titania 2607011111100x (AMB3626)
> 通訊標準: 9600 8N1（出廠預設），可改為 1200~~115200 baud~~
> 頻段: ~~169.4125~~169.4625 MHz (Band D, EN 300 220)
> 文檔參考: Manual-um-titania-260701111100x (rev4.5)

? ?????

指令名稱	CMD Code	Request	Response	用途
軟體重置	0x05	`02 05 00 07`	`02 45 01 00 46`	軟重啟，讓新設定生效
設定頻道（揮發）	0x06	`02 06 01 [CH] [CS]`	`02 46 01 [CH] [CS]`	臨時改當前頻道，斷電失效
設定 **UserSetting**（非揮發）	0x09	`02 09 [LEN+2] [MEM] [BYTES] [DATA...] [CS]`	`02 49 [LEN] [STATUS] [CS]`	寫入 flash 參數（baudrate、NetID 等）
讀取 **UserSetting**	0x0B	`02 0B 01 [MEM] [CS]`	`02 4B [LEN] [STATUS] [DATA...] [CS]`	讀出 flash 參數現在的值
工廠重置	0x0D	`02 0D 00 4D`	`02 4C 00 4E`	清所有 UserSettings，回到出廠預設（UART 回 9600）
透明模式傳送（**MAC**）	0x80	`02 80 [LEN] [MAC DATA...] [CS]`	`02 C0 [LEN] [STATUS] [CS]`	透明模式下發送無線訊號
Command Mode 進入	—	—	—	接 /CONFIG 腳下降邊緣可進入

? ??????

1?? ?????CMD_RESET_REQ?

用途: 讓 Titania 重啟，新的 UserSettings（如 baudrate 變更）會在重啟後生效。

指令:

```
Start   CMD  LEN  Checksum
  02    05   00      07
```

回應:

```
Start   CMD  LEN  Status  Checksum
  02    45   01   00        46
```

- **Status = 0x00**: 成功，模組重啟中
- 回應出現延遲:　　模組需要　　100~500 ms 才會完全重啟

C 程式碼:

```
HAL_StatusTypeDef st = Titania_SendCommand(huart, 0x05, NULL, 0);
uint8_t resp[5];
HAL_UART_Receive(huart, resp, 5, 500);  // 等待重啟完成
```

2?? ??????????— CMD_SET_CHANNEL_REQ

用途:　　臨時改變無線頻道（　　runtime），重啟模組後會失效（回到預設頻道）。

頻道對應表:

頻道	頻率 **(MHz)**	指令值
0	169.4125	0x00
1	169.4250	0x01
2	169.4375	0x02
3	169.4500	0x03
4	169.4625	0x04

指令格式:

```
Start  CMD  LEN  Channel  Checksum
  02   06   01   [0~4]    [CS]
```

計算 **Checksum**: `0x02 ^ 0x06 ^ 0x01 ^ [Channel]`

範例 - 設定 **Channel 2**:

```
發送: 02 06 01 02 01
回應: 02 46 01 02 43

Checksum 計算:
  0x02 ^ 0x06 ^ 0x01 ^ 0x02 = 0x01 ✓
  0x02 ^ 0x46 ^ 0x01 ^ 0x02 = 0x43 ✓
```

回應解析:

```
02        Start
46        CMD_SET_CHANNEL_CNF
01        Length
[02]      新頻道 (0~4)
[43]      Checksum
```

C 程式碼:

```
uint8_t payload[1] = { channel };
Titania_SendCommand(huart, 0x06, payload, 1);
uint8_t resp[5];
HAL_UART_Receive(huart, resp, 5, 100);
```

3?? ?? UserSettings?????— CMD_SET_REQ ???

用途: 寫入 flash 中的永久設定（ UART baudrate、預設頻道、 NetID、Addr 等）。重啟後仍保持。

常用 **UserSettings** 參數表:

參數名	**Memory Position (Dec / Hex)**	長度	值範圍	預設值	說明
`PHY_DefaultChannel`	42 / 0x2A	1 byte	0~4	2	重啟後使用的頻道
`UART_Baudrate`	80 / 0x50	4 bytes	1200~115200	9600	UART 速率（小端序 LSB first）
`MAC_DefaultDestNetID`	TODO	2 bytes	0x0000~0xFFFF	0x0000	預設目的 NetID

參數名	**Memory Position (Dec / Hex)**	長度	值範圍	預設值	說明
`MAC_DefaultDestAddrLSB`	TODO	2 bytes	0x0000~0xFFFF	0x0000	預設目的位址 LSB
`UART_Timeout`	TODO	2 bytes	[見表]	[見表]	UART 無字元多久後自動送包
`PHY_PAPower`	TODO	1 byte	[見表]	[見表]	發射功率索引 (-11~+15 dBm)
`RF_ConfigIndex`	TODO	1 byte	[見表]	2	RF 設定（ data rate、調變）
`MAC_NumRetrys`	TODO	1 byte	0~N	[見表]	重送次數

指令格式:

```
Start  CMD  LEN    MPOS  NBYTES  DATA...  Checksum
  02   09  [2+N]  [MP]  [LEN]   [...]     [CS]
```

- **LEN = 2 +** 資料長度 (包括 MPOS + NBYTES 欄位)
- **MPOS = Memory Position** (十進位或十六進位)
- **NBYTES =** 要寫多少 **bytes**
- **DATA =** 實際資料
- **Checksum =** 從 **0x02** 到最後一個 **DATA** 的 **XOR**

範例 **1 -** 設定預設頻道為 **Channel 3** (非揮發):

```
發送: 02 09 03 2A 01 03 4B
      02 09 03 2A 01 03 4B
      ↑  ↑  ↑  ↑  ↑  ↑  ↑
     ST CMD LEN MP NB DC CS

回應: 02 49 00 00 4B
      ↑  ↑  ↑  ↑  ↑
     ST CMD LEN ST CS

Checksum 計算:
  發送: 0x02 ^ 0x09 ^ 0x03 ^ 0x2A ^ 0x01 ^ 0x03 = 0x4B ✓
  回應: 0x02 ^ 0x49 ^ 0x00 ^ 0x00 = 0x4B ✓
```

範例 **2 -** 設定 **UART Baudrate** 為 **57600** (非揮發):

57600 = 0x0000E100 (十六進位)
小端序 (LSB first): [0x00, 0xE1, 0x00, 0x00]

```
發送: 02 09 06 50 04 00 E1 00 00 CS
       ↑  ↑  ↑  ↑  ↑  ↑ ↑ ↑ ↑ ↑ ↑
      ST CMD LEN MP NB D0 D1 D2 D3 CS

MPOS = 0x50 (80 十進位)
NBYTES = 04
DATA = [00 E1 00 00]
LEN = 2 + 4 = 06
Checksum = 0x02 ^ 0x09 ^ 0x06 ^ 0x50 ^ 0x04 ^ 0x00 ^ 0xE1 ^ 0x00 ^ 0x00 = ?

回應: 02 49 00 00 4B
      (Status=0x00 表示成功)
```

回應解析:

```
02            Start
49            CMD_SET_CNF (確認)
00            Length
00            Status (0x00 = 成功, 非 0 = 失敗)
4B            Checksum
```

C 程式碼:

```
// 設定 UART Baudrate 為 57600
uint32_t baudrate = 57600;
uint8_t payload[6];
payload[0] = 0x50;                              // Memory Position (UART_Baudrate)
payload[1] = 0x04;                              // 資料長度 4 bytes
payload[2] = (uint8_t)(baudrate & 0xFF);
payload[3] = (uint8_t)((baudrate >> 8) & 0xFF);
payload[4] = (uint8_t)((baudrate >> 16) & 0xFF);
payload[5] = (uint8_t)((baudrate >> 24) & 0xFF);

Titania_SendCommand(huart, 0x09, payload, 6);
uint8_t resp[5];
HAL_UART_Receive(huart, resp, 5, 200);

// 檢查 resp[3] == 0x00 (Status OK)
```

4?? ?? UserSettings — CMD_GET_REQ

用途: 讀出 flash 中現在儲存的設定值（例如確認現在 baudrate 是多少）。

指令格式:

```
Start  CMD  LEN  MPOS   CS
  02   0B   01   [MP]  [CS]
```

範例 - 讀取 **PHY_DefaultChannel**:

```
發送: 02 0B 01 2A 21
       ↑  ↑  ↑  ↑  ↑
      ST CMD LEN MP CS

Checksum = 0x02 ^ 0x0B ^ 0x01 ^ 0x2A = 0x21 ✓

回應: 02 4B 02 00 03 4D
       ↑  ↑  ↑  ↑  ↑  ↑
      ST CMD LEN ST DA CS

解析:
  Status = 0x00 (成功)
  Data = 0x03 (目前預設頻道是 Channel 3)
  Checksum = 0x02 ^ 0x4B ^ 0x02 ^ 0x00 ^ 0x03 = 0x4D ✓
```

C 程式碼:

```
uint8_t mem_pos = 0x2A;  // PHY_DefaultChannel
Titania_SendCommand(huart, 0x0B, &mem_pos, 1);

uint8_t resp[6];  // Start + CMD + LEN + Status + Data + CS
HAL_UART_Receive(huart, resp, 6, 200);

uint8_t current_channel = resp[4];  // 第 5 byte 是資料
```

5?? ???? — CMD_FACTORY_RESET_REQ ? ????

用途: 清除所有 UserSettings, 回到出廠預設值。

項目	回復值
UART Baudrate	9600
PHY_DefaultChannel	2 (169.4375 MHz)
MAC_DefaultDestNetID	0x0000
MAC_DefaultDestAddrLSB	0x0000
所有其他設定	出廠預設

指令:

```
Start  CMD  LEN  CS
  02   0D   00   4D
```

Checksum = 0x02 ^ 0x0D ^ 0x00 = 0x4D ✓

回應:

```
Start  CMD  LEN  Status  CS
  02   4C   00   00      4E
```

Checksum = 0x02 ^ 0x4C ^ 0x00 ^ 0x00 = 0x4E ✓

C 程式碼:

```
Titania_SendCommand(huart, 0x0D, NULL, 0);
uint8_t resp[5];
HAL_UART_Receive(huart, resp, 5, 500);

// 模組會重啟, UART 變回 9600, 所有設定清零
```

6?? ?????? — CMD_DATA_REQ

用途: 在透明模式下, 透過 UART 送進來的資料會被打包成 MAC 訊號透過無線送出。

指令格式:

```
Start  CMD  LEN    MAC_DATA...  CS
  02   80   [N]    [...]        [CS]
```

- **LEN = MAC** 訊號的長度（ **bytes**）
- **MAC_DATA =** 你要發送的內容（ **payload**）
- **Checksum =** 從 **0x02** 到最後一個 **DATA** 的 **XOR**

範例 **-** 發送 **5 bytes** 資料 **"Hello"**:

```
發送: 02 80 05 48 65 6C 6C 6F CS
       ↑  ↑  ↑  ↑  ↑  ↑ ↑ ↑ ↑ ↑
      ST CMD LEN H  e  l l o CS

"Hello" 的 ASCII 值:
  H=0x48, e=0x65, l=0x6C, l=0x6C, o=0x6F

Checksum = 0x02 ^ 0x80 ^ 0x05 ^ 0x48 ^ 0x65 ^ 0x6C ^ 0x6C ^ 0x6F = ?

回應: 02 C0 00 00 C0
       ↑  ↑  ↑  ↑  ↑
      ST CMD LEN ST CS

Status = 0x00 (成功, 訊號已排隊發送)
```

C 程式碼:

```
uint8_t payload[5] = { 0x48, 0x65, 0x6C, 0x6C, 0x6F };  // "Hello"
Titania_SendCommand(huart, 0x80, payload, 5);

uint8_t resp[5];
HAL_UART_Receive(huart, resp, 5, 100);
// resp[3] = Status
```

? Checksum ????

XOR 從 **Start (0x02)** 一路算到最後一個 **Data**, 不含 **Checksum** 本身。

```
static uint8_t CalcChecksum(uint8_t *buf, uint8_t len)
{
    uint8_t cs = 0;
    for (uint8_t i = 0; i < len; i++)
        cs ^= buf[i];
```

```
    return cs;
}

// 使用方式:
uint8_t frame[10] = { 0x02, 0x09, 0x03, 0x2A, 0x01, 0x03, ... };
uint8_t cs = CalcChecksum(frame, 6);  // 不包括 Checksum 欄位本身
frame[6] = cs;
```

? ??????

A. ?????? Titania ??

1. 確認 **UART** 連線 → 用邏輯分析儀看波形確認 baudrate 是 9600
2. 讀取現在設定 → CMD_GET_REQ 讀 0x2A (PHY_DefaultChannel) 和 0x50 (UART_Baudrate)
3. 寫入新設定 → CMD_SET_REQ 改 UART_Baudrate、PHY_DefaultChannel、NetID、Addr 等
4. 軟重啟 → CMD_RESET_REQ, 等 100~500 ms
5. 切換 **STM32 UART** 波特率 → 改成新的 baudrate
6. 驗證 → 再讀一次設定確認有存進去

B. ?????? baudrate

1. 用邏輯分析儀 抓 Titania 上電時 TX 線的波形, 反推現在是多少 baudrate
2. 用那個 **baudrate** 連接 STM32
3. 發送 **CMD_FACTORY_RESET_REQ** 清所有設定（ UART 回 9600）
4. 切換 **STM32 UART** 回 **9600**
5. 重新用 **CMD_SET_REQ** 設定一次

C. ???????runtime ????

1. 發 **CMD_SET_CHANNEL_REQ** (0x06) 改當前頻道
2. 不需要 **reset**, 立即生效
3. 但重啟後會失效 （回到預設頻道）
4. 若要永久改, 用 CMD_SET_REQ (0x09) 改 PHY_DefaultChannel + CMD_RESET_REQ

D. ????? NetID / Addr?MAC ????

1. 用 **CMD_SET_REQ (0x09)** 寫入:
   - `MAC_DefaultDestNetID` (memory position TODO)
   - `MAC_DefaultDestAddrLSB` (memory position TODO)

2. 再 **CMD_RESET_REQ** 重啟讓設定生效
3. 之後所有透明模式的 MAC 訊號都會用這個 NetID/Addr 作為預設目的地

? ????????

?? 1???? + ??? 57600 + ?? 3 + NetID 0x0001

```
Step 1: CMD_SET_REQ UART_Baudrate = 57600
  → 發送: 02 09 06 50 04 00 E1 00 00 CS
  → 等待回應 02 49 00 00 4B

Step 2: CMD_SET_REQ PHY_DefaultChannel = 3
  → 發送: 02 09 03 2A 01 03 4B
  → 等待回應 02 49 00 00 4B

Step 3: CMD_SET_REQ MAC_DefaultDestNetID = 0x0001
  → 發送: 02 09 04 [MEM] 02 01 00 CS
  → 等待回應 02 49 00 00 4B

Step 4: CMD_RESET_REQ
  → 發送: 02 05 00 07
  → 等待 500ms 讓模組重啟
  → 接收回應 02 45 01 00 46

Step 5: STM32 UART 改 57600 baud
  → 重新初始化 UART

Step 6: 驗證設定（可選）
  → CMD_GET_REQ 讀各個參數確認
```

?? 2??????????

```
Step 1: CMD_FACTORY_RESET_REQ
  → 發送: 02 0D 00 4D
  → 等待 500ms，模組重啟
  → 接收回應 02 4C 00 00 4E
```

```
Step 2: STM32 UART 改回 9600 baud
  → 重新初始化 UART

Step 3: 確認恢復（可選）
  → CMD_GET_REQ 讀 0x50 (UART_Baudrate)
  → 應該讀到 9600
```

? ????

 UART 設定改變後必須 **reset**

- 改 UART_Baudrate 或其他非揮發參數後， 必須 CMD_RESET_REQ，否則設定寫進 flash 但模組沒用新值運作。

 Checksum 計算務必正確

- 差一個 bit 就會被模組拒絕或誤解。

 □□□□ **STM32 UART** 波特率

- 改完 Titania UART_Baudrate 後， STM32 也要改成一樣的速率，否則無法通訊。

 NetID / Addr 的 **memory position** 要對應手冊

- 上表中 TODO 的位置需要你查手冊第 8.13~8.16 章補上正確的十進位或十六進位位置。

 □□□□□□□□□□

- 包括 UART_Baudrate、頻道、 NetID、Addr、RF_ConfigIndex 等，全部回到出廠預設。

? STM32 ??????

```
#include "stm32f1xx_hal.h"

extern UART_HandleTypeDef huart1;

// 計算 Checksum
static uint8_t CalcChecksum(uint8_t *buf, uint8_t len)
{
```

```
    uint8_t cs = 0;
    for (uint8_t i = 0; i < len; i++)
        cs ^= buf[i];
    return cs;
}

// □□□□
void SendTitaniaCmd(uint8_t cmd, uint8_t *data, uint8_t len)
{
    uint8_t frame[64];
    uint8_t idx = 0;

    frame[idx++] = 0x02;  // Start
    frame[idx++] = cmd;
    frame[idx++] = len;
    for (uint8_t i = 0; i < len; i++)
        frame[idx++] = data[i];

    frame[idx] = CalcChecksum(frame, idx);
    idx++;

    HAL_UART_Transmit(&huart1, frame, idx, 100);
}

// □□□□ baudrate □ 57600 + □□ 3 + reset
void InitTitania(void)
{
    uint8_t payload[6];
    uint8_t resp[10];

    // 1. □ UART Baudrate 57600
    payload[0] = 0x50;      // Memory Pos (UART_Baudrate)
    payload[1] = 0x04;      // Length
    payload[2] = 0x00;      // 57600 = 0x0000E100 (LE)
    payload[3] = 0xE1;
    payload[4] = 0x00;
    payload[5] = 0x00;
    SendTitaniaCmd(0x09, payload, 6);
    HAL_UART_Receive(&huart1, resp, 5, 200);
```

```
    // 2. 改預設頻道為 3
    payload[0] = 0x2A;      // Memory Pos (PHY_DefaultChannel)
    payload[1] = 0x01;      // Length
    payload[2] = 0x03;      // Channel 3
    SendTitaniaCmd(0x09, payload, 3);
    HAL_UART_Receive(&huart1, resp, 5, 200);

    // 3. Reset 讓設定生效
    SendTitaniaCmd(0x05, NULL, 0);
    HAL_Delay(500);  // 等模組重啟
    HAL_UART_Receive(&huart1, resp, 5, 500);

    // 4. 改 STM32 UART 為 57600
    huart1.Init.BaudRate = 57600;
    HAL_UART_Init(&huart1);
}
```

? ????

- **Würth Titania Manual:** Manual-um-titania-260701111100x (rev4.5)
- **Titania** 規格表 **:** 2607011111100x datasheet
- **Band D Frequencies:** EN 300 220 (歐盟標準)

最後更新: 2026-03-10

作者備註: 本文檔整理自 Titania 官方手冊，所有指令與 memory position 須對應實際硬體版本。