FIRMWARE ENGINEERING SERIES

ACPI ASL

從零到實戰

完整教材：從 ACPI 架構、ASL 語言到 Linux Kernel 整合

From Zero to Real-World ACPI Firmware Engineering

`ASL → AML → DSDT/SSDT → ACPI Interpreter → Kernel → Driver`

12 Modules • 30 Chapters • 5 Projects • 12 週課程

2026 Edition • ACPI 6.5 • ACPICA iASL

教材資訊

版本	v1.0（2026 年 8 月）
對應規格	ACPI Specification 6.5
工具鏈	ACPICA iASL Compiler / Disassembler
測試平台	Linux（Ubuntu / Fedora）、QEMU + OVMF
排版	XeLaTeX、Songti TC、Menlo
用途	課程教學與自學（Firmware / BIOS / Embedded）

本教材所有 ASL 範例皆可使用 iASL 編譯驗證；
實機操作章節建議先在 QEMU 虛擬機中演練，再套用到實體機器。

前言

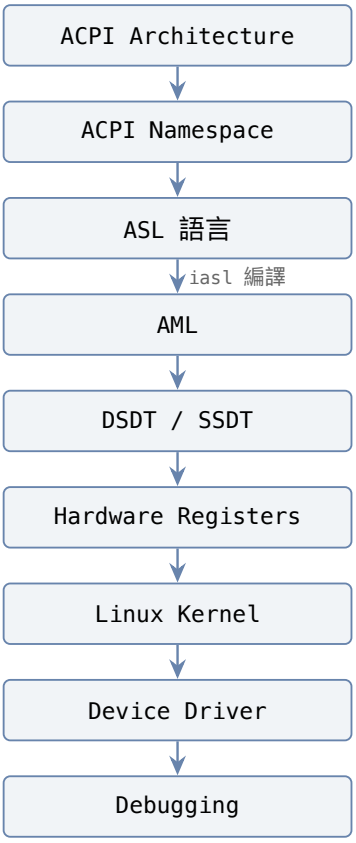
為什麼要學 ACPI ASL？

打開任何一台 x86 筆電或伺服器，作業系統之所以知道「這台機器有哪些電池、哪顆 GPIO 控制器、感測器接在哪條 I2C 匯流排、按下電源鍵該做什麼」，靠的不是魔法，而是韌體交給作業系統的一份「機器說明書」——**ACPI Tables**。而撰寫這份說明書的語言，就是 **ASL**（ACPI Source Language）。

ASL 是 BIOS/UEFI 韌體工程師的核心技能之一：無論是新平台 bring-up、電源管理調校、Windows/Linux 相容性除錯，或是 Embedded / AI SoC 平台開發，最終都會回到同一個問題——「DSDT/SSDT 裡到底寫了什麼？」本教材的目標，就是讓你有能力自己讀懂、寫出、除錯這份機器說明書。

本教材的學習路徑

真正有價值的學習路徑不是只背 ASL 語法，而是走完整條工程鏈：



教材結構總覽

全書共 12 個 Module、30 章、5 個實戰專案，難度由淺入深：

Module	主題	章節	難度
1	ACPI 基礎與架構	第 1 章	★☆☆☆☆
2	ASL、AML 與 ACPI Namespace	第 2–3 章	★★☆☆☆
3	ASL 基本語法	第 4–7 章	★★☆☆☆
4	Device 與硬體描述	第 8 章	★★★☆☆
5	OperationRegion 與 Field	第 9–11 章	★★★☆☆
6	Control Method	第 12–14 章	★★★☆☆
7	Power Management	第 15–17 章	★★★★☆
8	DSDT / SSDT 實戰	第 18–20 章	★★★★☆
9	iASL Toolchain	第 21 章	★★★★☆
10	Linux ACPI Debugging	第 22–24 章	★★★★☆
11	進階整合：ACPI、C/C++ 與 Linux Kernel	第 25 章	★★★★★
12	實戰專案（Projects 1–5）	第 26–30 章	★★★★★

12 週課程計畫

若作為一學期課程使用，建議按以下進度：

Week	學習內容	實作
1	ACPI Architecture（Module 1）	畫出 ACPI 架構圖
2	ASL / AML / Namespace（Module 2）	第一個 `.dsl` 檔
3	ASL Syntax（Module 3）	Variables / Operators
4	Device Objects（Module 4）	`_HID` / `_STA` / `_CRS`
5	OperationRegion（Module 5）	MMIO / I/O 存取
6	Field（Module 5）	Register Access
7	Method（Module 6）	Control Method
8	GPIO / I2C / SPI（Module 5、6）	Hardware Device 描述
9	Power Management（Module 7）	`_ON` / `_OFF` / `_STA`
10	DSDT / SSDT（Module 8）	Dump & Disassemble
11	iASL / Linux Debug（Module 9、10）	Real DSDT Analysis
12	Final Project（Module 11、12）	ACPI Device + Driver

如何使用本教材

每一章都遵循同樣的節奏：**觀念講解 → 完整範例 → 逐行拆解 → 動手練習**。強烈建議你「每章至少親手編譯一次範例」——ASL 是一門動手的語言，光用讀的學不會。

書中使用以下幾種提示框：

核心觀念

「核心觀念」框：本章最重要、必須內化的觀念，複習時先看這裡。

補充說明

「補充說明」框：延伸知識、規格細節、與實務現場的連結。

注意

「注意」框：常見誤解、容易踩到的坑、會讓機器不開機的操作。

動手練習

「動手練習」框：每章結尾的練習題，請務必完成再進入下一章。

環境需求

- 一台 Linux 機器（實體機或虛擬機皆可；Ubuntu 22.04+ 或 Fedora 皆適用）。
- `acpica-tools`（含 `iasl`、`acpidump`、`acpixtract`）。
- 建議另備 QEMU + OVMF，練習 DSDT/SSDT override 時安全許多。
- 文字編輯器（VS Code、Vim 皆可，建議安裝 ASL syntax highlighting）。

安裝方式與環境驗證步驟詳見附錄。現在，讓我們從「ACPI 到底解決了什麼問題」開始。

目錄

Module 1　ACPI 基礎與架構 **1**

1　ACPI Introduction **2**

1.1　ACPI 解決什麼問題？ 2
1.2　BIOS/UEFI 與 ACPI 的關係 3
1.3　OS 如何取得 Hardware Information？ 4
1.4　ACPI 與 Device Driver 的關係 5
1.5　ACPI 在 Windows / Linux 中的位置 5
1.6　ACPI 與 Power Management 的關係 6
1.7　完整資料流：從硬體規格到 Driver 6

Module 2　ASL、AML 與 ACPI Namespace **9**

2　ASL vs AML **10**

2.1　從 C 語言與機器碼談起 10
2.2　第一個.dsl 檔：DefinitionBlock 逐欄解剖 11
2.3　編譯：iasl test.dsl 12
2.4　直擊機器語言：解剖 test.aml 13
2.5　反組譯：iasl -d 與 roundtrip 14
2.6　完整旅程：從.dsl 到 ACPI Interpreter 16

3　ACPI Namespace **18**

3.1　Namespace：OS 眼中的整台機器 18
3.2　一棵樹的全貌與預定義根 Scope 19
3.3　命名規則：NameSeg 的四字元世界 20
3.4　逐行拆解：最小可編譯的 Namespace 20
3.5　`\_SB.DEV0` 到底代表什麼？ 21
3.6　路徑解析：絕對、相對與向上搜尋 22
3.7　Scope 與 Device 的差異 23
3.8　Alias：同一個物件的第二個名字 24

Module 3　ASL 基本語法 **26**

4 Data Types **27**

4.1 一切都是物件：型別總覽 . . . 27
4.2 Integer：64 位元的無號整數 . . . 28
4.3 String 與 Buffer . . . 29
4.4 Package：異質、可巢狀的容器 . . . 29
4.5 Object Reference 與 Field Unit . . . 30
4.6 Implicit Conversion：直譯器的自動轉換 . . . 31
4.7 Explicit Conversion：把轉換寫清楚 . . . 31
4.8 完整範例：型別實驗室 . . . 32

5 Variables & Constants **35**

5.1 資料的三個家：Name、Local、Arg . . . 35
5.2 Name：常駐 Namespace 的具名物件 . . . 36
5.3 Local0–Local7：方法內的暫存空間 . . . 36
5.4 Arg0–Arg6：呼叫者給的輸入 . . . 36
5.5 Store、指定運算子與 Return . . . 37
5.6 常數與數字字面值 . . . 37
5.7 生命週期：一次呼叫的一生 . . . 38
5.8 課綱範例逐行拆解 . . . 38

6 Operators **41**

6.1 一種位元組碼，兩種外衣 . . . 41
6.2 運算子速查總表 . . . 42
6.3 算術運算子與 target 參數 . . . 42
6.4 位元運算子 . . . 43
6.5 邏輯與比較運算子 . . . 44
6.6 複合指定、遞增與遞減 . . . 44
6.7 實戰：暫存器欄位的讀改寫 . . . 45

7 Control Flow **48**

7.1 If、ElseIf 與 Else . . . 48
7.2 條件式的真假值 . . . 49
7.3 While、Break 與 Continue . . . 49
7.4 實戰：走訪 Package . . . 51
7.5 Switch、Case 與 Default . . . 52

7.6 Sleep 與 Stall：兩種等待 . . . 53
7.7 綜合範例：輪詢一個裝置 . . . 53

Module 4 Device 與硬體描述 56

8 Device Object 57
8.1 Device 物件：向 OS 註冊一個裝置節點 . . . 57
8.2 _HID：裝置的主要身分證 . . . 58
8.3 _CID 與 _UID：相容身分與實例編號 . . . 59
8.4 _ADR：可列舉匯流排上的位址 . . . 60
8.5 _STA：裝置的健康申報 . . . 60
8.6 _CRS、_PRS 與 _SRS：資源申報三件套 . . . 61
8.7 _DSD 與 _INI：裝置屬性與初始化 . . . 62
8.8 綜合範例：一個完整的平台裝置 . . . 63

Module 5 OperationRegion 與 Field 65

9 OperationRegion 66
9.1 一行宣告，四個參數 . . . 66
9.2 九種 Address Space . . . 67
9.3 SystemMemory 與 PCI_Config 實例 . . . 68
9.4 Region 只是窗口：宣告不等於存取 . . . 68
9.5 綜合範例：Debug Port 裝置 . . . 70

10 Field 72
10.1 Field：把窗口切成具名暫存器 . . . 72
10.2 AccessType：一次交易敲多寬 . . . 73
10.3 LockRule：與韌體共搶硬體 . . . 74
10.4 UpdateRule：沒被寫到的位元怎麼辦 . . . 74
10.5 Offset()、位元欄位與跨 byte 欄位 . . . 75
10.6 IndexField 與 BankField . . . 76
10.7 綜合範例：MMIO 控制器 . . . 77

11 GPIO / I2C / SPI / PCI 實戰 80
11.1 Controller 與 Consumer：兩種角色 . . . 80
11.2 GPIO：從 Controller 到扳一根 Pin . . . 80

11.3 I2C：連線描述與 GenericSerialBus 存取 . . . 82
11.4 SPI 與 UART：其他序列匯流排 . . . 84
11.5 PCI：Root Bridge、_ADR 與 Config Space . . . 85
11.6 SystemMemory 與 SystemIO：完整裝置範例 . . . 86
11.7 綜合佈局：EC、I2C 感測器與 GPIO LED . . . 86

Module 6 Control Method **88**

12 Method **89**
12.1 誰會呼叫 Method？從靜態描述到可執行程式 . . . 89
12.2 Method 宣告：完整語法解剖 . . . 90
12.3 呼叫 Method 與使用回傳值 . . . 91
12.4 Serialized：方法重入與 AE_ALREADY_EXISTS . . . 92
12.5 方法呼叫方法、遞迴與方法內的 Namespace . . . 94
12.6 完整範例：感測器數值處理模組 . . . 94

13 Method Arguments **97**
13.1 一次呼叫的世界：Method 的四個組成 . . . 97
13.2 Arguments：Arg0 到 Arg6 . . . 98
13.3 Locals：Local0 到 Local7 . . . 99
13.4 參數個數不匹配：Interpreter 不會救你 . . . 99
13.5 傳參考：RefOf 與 DerefOf . . . 100
13.6 完整範例：Package 進、Package 出 . . . 101

14 Notify **104**
14.1 從同步到非同步：為什麼需要 Notify . . . 104
14.2 事件的源頭：GPE 與 SCI . . . 105
14.3 Level 與 Edge：_Lxx 與 _Exx . . . 105
14.4 Notify 語法與代碼表 . . . 106
14.5 完整範例：EC query 到電池事件鏈 . . . 107
14.6 在 Linux 上觀察 Notify . . . 109

Module 7 Power Management **112**

15 Power States **113**
15.1 電源狀態的全景地圖 . . . 113

15.2 S-states：S0 到 S5 . . . 114
15.3 D-states：D0 到 D3cold . . . 115
15.4 系統睡眠的完整流程 . . . 115
15.5 _Sx 物件與 _PTS / _WAK 實作 . . . 116

16 Power Resources **120**
16.1 PowerResource 物件：宣告與參數 . . . 120
16.2 三件套：_ON、_OFF、_STA . . . 121
16.3 OSPM 的 reference counting：何時真的關電 . . . 121
16.4 裝置端的掛勾：_PR0 到 _PR3 . . . 122
16.5 _PSx 與 PowerResource 的分工 . . . 122
16.6 喚醒能力：_PRW、_DSW 與 _PSW . . . 122
16.7 完整範例：GPIO 電源軌與兩顆相機 . . . 123

17 CPU Power Management **127**
17.1 兩條軸線：跑多快與睡多深 . . . 127
17.2 Processor 物件：兩代寫法 . . . 128
17.3 P-states：_PSS、_PCT 與 _PPC . . . 128
17.4 C-states：_CST 與它的歷史 . . . 130
17.5 多核協調：_PSD、_CSD 與 _TSD . . . 131
17.6 現代平台的現實：HWP 與 _CPC . . . 131
17.7 Thermal：溫度閉環 . . . 131

Module 8 DSDT / SSDT 實戰 **135**

18 DSDT **136**
18.1 DSDT 的定位：每系統唯一、開機必載 . . . 136
18.2 36 位元組表頭：逐欄位解讀 . . . 137
18.3 真實筆電的 DSDT 長什麼樣子 . . . 138
18.4 OS 如何用 DSDT 建立 Namespace . . . 141
18.5 幾千行裡找到你要的裝置：搜尋策略 . . . 141

19 SSDT **144**
19.1 為什麼要拆表：SSDT 存在的理由 . . . 144
19.2 合併語意：多張表、同一棵樹 . . . 145
19.3 External：跨表引用的橋樑 . . . 146
19.4 動態載入：Load、LoadTable 與 Unload . . . 147

19.5 在 Linux 上數 SSDT 149

20 DSDT/SSDT Override **151**

20.1 全景：七步流水線 151
20.2 Step 1：取出系統的表 152
20.3 Step 2：反組譯——為什麼需要 -e 153
20.4 Step 3：安全的小修改 153
20.5 Step 4：重編——面對 vendor DSDT 的現實 154
20.6 Step 5：打包 initrd 與 GRUB 設定 155
20.7 Step 6：重開機驗證——override 還是 install？ 156
20.8 QEMU：零風險的練習場 157
20.9 擋路的東西與退路 158

Module 9 iASL Toolchain **159**

21 iASL Compiler / Disassembler **160**

21.1 ACPICA：一套核心、整組工具 160
21.2 安裝與版本 161
21.3 第一次編譯：讀懂 iasl 的輸出 161
21.4 反組譯：-d 與 -e 163
21.5 常用旗標速查 163
21.6 acpidump 與 acpixtract 165
21.7 acpiexec：不開機就執行 AML 166
21.8 Roundtrip：反組譯再重編不是影印機 166

Module 10 Linux ACPI Debugging **168**

22 Linux ACPI **169**

22.1 Linux 上的 ACPI 觀測點 169
22.2 dmesg：開機時的 ACPI 足跡 171
22.3 acpidump 工具鏈：把韌體變回原始碼 172
22.4 閱讀 DSDT 的方法論：以筆電電池為例 173
22.5 acpiexec：在使用者空間執行 AML 175

23 ACPI Debug **177**

23.1 Debug 物件：一個只能寫的虛擬物件 177

23.2 一行 Debug 的旅程 . . . 179
23.3 打開輸出：兩道閘門 . . . 179
23.4 Lab：讓核心印出你的 Debug 訊息 . . . 181
23.5 更快的迴圈：acpiexec 直接印 . . . 182
23.6 進階：Method Tracing 簡介 . . . 183

24 ACPI Error Debugging **185**
24.1 解剖一行 ACPI Error . . . 185
24.2 常見錯誤碼速查表 . . . 187
24.3 Case Study 1：AE_ALREADY_EXISTS——重入的代價 . . . 187
24.4 Case Study 2：AE_NOT_FOUND——防禦性引用 . . . 188
24.5 Case Study 3：GPE Storm——用計數器抓兇手 . . . 189
24.6 通用定位流程與檢查清單 . . . 190
24.7 fwts：自動化健檢 . . . 191

Module 11 進階整合：ACPI、C/C++ 與 Linux Kernel **193**

25 ACPI 與 Linux Kernel 整合 **194**
25.1 全景圖：兩條路在核心會合 . . . 194
25.2 韌體工程師的 C 語言功力 . . . 195
25.3 Linux ACPI Subsystem：從 Namespace 到 device model . . . 198
25.4 實作：綁定 DEMO0001 的 platform driver . . . 199
25.5 各子系統的 ACPI 膠水 . . . 203
25.6 _DSD 與 device properties：韌體傳參數給驅動 . . . 205
25.7 韌體端：EDK II 的 ACPI 表生產線 . . . 206
25.8 端到端案例：一顆 LED 的旅程 . . . 208

Module 12 實戰專案 **210**

26 Project 1：Virtual ACPI Device **211**
26.1 專案目標與驗收標準 . . . 211
26.2 背景與設計 . . . 211
26.3 逐步實作 . . . 213
26.4 驗證 . . . 214
26.5 常見問題 . . . 216
26.6 加分挑戰 . . . 217

27 Project 2：GPIO Controller **218**
27.1 專案目標與驗收標準 . . . 218
27.2 背景與設計 . . . 218
27.3 逐步實作 . . . 219
27.4 驗證 . . . 222
27.5 常見問題 . . . 223
27.6 加分挑戰 . . . 224

28 Project 3：I2C Sensor **225**
28.1 專案目標與驗收標準 . . . 225
28.2 背景與設計 . . . 225
28.3 逐步實作 . . . 227
28.4 驗證 . . . 230
28.5 常見問題 . . . 231
28.6 加分挑戰 . . . 232

29 Project 4：Power Management **233**
29.1 專案目標與驗收標準 . . . 233
29.2 背景與設計 . . . 233
29.3 逐步實作 . . . 235
29.4 驗證 . . . 238
29.5 常見問題 . . . 239
29.6 加分挑戰 . . . 239

30 Project 5：DSDT Reverse Engineering（Final Project） **241**
30.1 專案目標與驗收標準 . . . 241
30.2 背景與設計 . . . 242
30.3 逐步實作 . . . 242
30.4 驗證與報告（階段五） . . . 247
30.5 常見問題 . . . 249
30.6 加分挑戰 . . . 249

A 開發環境安裝指南 **250**
A.1 Linux . . . 250
A.2 macOS . . . 250
A.3 驗證安裝 . . . 250
A.4 建議的專案目錄結構 . . . 251

B ACPI 預定義物件速查表 **252**

B.1 識別與列舉 252

B.2 狀態與資源 252

B.3 裝置電源管理 253

B.4 系統睡眠 253

B.5 電池、按鍵與熱管理 253

B.6 常用 Notify 代碼 254

C 工具指令速查 **255**

C.1 iASL 常用旗標 255

C.2 取表與執行 255

C.3 Linux ACPI 觀察點 256

D ACPI Error Code 對照表 **257**

參考資料 **259**

Module 1

ACPI 基礎與架構

難度：★☆☆☆☆

本模組回答三個問題：ACPI 是什麼？它為什麼存在？它在整個電腦系統中的位置在哪裡？讀完本模組，你將能畫出從硬體、韌體、ACPI Tables 一路到作業系統驅動程式的完整資料流——這張圖是之後所有章節的地基。

第 1 章

ACPI Introduction

在寫下第一行 ASL 之前，你必須先知道這門語言存在的理由。本章從歷史與架構切入，建立「韌體描述硬體、作業系統解讀並決策」這個 ACPI 的核心世界觀。

本章學習目標

- 說明 ACPI 解決了什麼問題，以及它取代了哪些舊機制。
- 描述 BIOS/UEFI 與 ACPI 的分工：誰產生資料、誰解讀資料。
- 說明 OS 如何透過 ACPI 取得硬體資訊並綁定 Device Driver。
- 指出 ACPI 在 Windows 與 Linux 軟體堆疊中的位置。
- 說明 ACPI 與 Power Management 的關係。
- 不看書畫出 Hardware 到 Device Driver 的完整資料流圖。

1.1 ACPI 解決什麼問題？

1990 年代中期以前，PC 的電源管理與硬體組態是韌體的私有領域：**APM**（Advanced Power Management）由 BIOS 全權決定何時降頻、何時休眠，作業系統只能透過少數 BIOS call 被動配合；主機板上有哪些裝置、用哪些資源，則由 PnP BIOS、MPS 表等各自為政的機制描述。這帶來三個問題：

- **OS 是瞎子**：BIOS 突然把機器休眠，OS 完全不知情，檔案系統與網路連線說斷就斷。
- **各家行為不一致**：每家 OEM 的 APM 實作不同，OS 無法寫出可攜的電源管理策略。
- **描述能力不足**：越來越多裝置（Embedded Controller、電池、感測器）不在 PCI 這類可自動列舉的匯流排上，舊機制無法告訴 OS 它們的存在。

1996 年，Intel、Microsoft、Toshiba 共同制定了 **ACPI**（Advanced Configuration and Power Interface），後來由 UEFI Forum 接手維護，目前版本為 ACPI 6.5。ACPI 的核心思想是**OSPM**（Operating System-directed configuration and Power Management）：

核心觀念：OSPM 的分工

韌體負責**描述**（describe）：把硬體組態、電源控制方法寫成 ACPI Tables 放進記憶體。作業系統負責**決策**（decide）：讀取這些描述，由 OS 統一決定何時開關裝置、何時睡眠。ACPI 本身不是驅動程式，而是韌體與 OS 之間的**介面與合約**。

兩代機制的對比如下：

	APM 時代	ACPI 時代
決策者	BIOS	作業系統（OSPM）
OS 的角色	被動配合	主動管理
硬體描述	PnP BIOS、MPS 等分散機制	統一的 ACPI Namespace
擴充方式	各家自訂 BIOS call	標準化的 Tables 與 AML
可攜性	低	高（跨 OS、跨平台）

1.2 BIOS/UEFI 與 ACPI 的關係

很多初學者以為「ACPI 是 BIOS 的一部分功能」，這只對了一半。正確的說法是：**BIOS/UEFI 在開機時「生產」ACPI Tables，之後便退居幕後；「消費」這些 Tables 的是作業系統。**

開機流程如下：韌體完成硬體初始化（POST）後，把一系列 ACPI Tables 放進系統記憶體，並在交棒給 OS 時留下一個入口點——**RSDP**（Root System Description Pointer）。UEFI 系統透過 EFI Configuration Table 提供 RSDP；傳統 BIOS 則讓 OS 在特定記憶體區段掃描簽章。OS 從 RSDP 出發，就能找到所有其他表：

ACPI Tables 的鏈狀結構

```
RSDP (Root System Description Pointer)
 └── XSDT (Extended System Description Table)
      ├── FADT (Fixed ACPI Description Table)
      │     └── DSDT  ← AML：本教材的主角
      ├── SSDT  ← AML：DSDT 的延伸（可有多個）
      ├── MADT  ← 中斷控制器與 CPU 拓撲
      ├── MCFG  ← PCIe ECAM 組態空間
      └── HPET / BGRT / ...
```

常見的表整理如下：

簽章	全名	用途
RSDP	Root System Description Pointer	一切的起點，指向 XSDT
XSDT	Extended System Description Table	其他表的位址目錄
FADT	Fixed ACPI Description Table	固定硬體暫存器資訊，並指向 DSDT
DSDT	Differentiated System Description Table	主要的硬體描述（AML），開機必載
SSDT	Secondary System Description Table	DSDT 的補充，可動態載入、可多個
MADT	Multiple APIC Description Table	CPU 與中斷控制器拓撲
MCFG	Memory-mapped Configuration	PCIe 組態空間基底位址
FACS	Firmware ACPI Control Structure	韌體與 OS 共享的控制區（喚醒向量）

補充說明：Tables 是「資料」，不是「程式」

BIOS/UEFI 把 ACPI Tables 放進記憶體後就功成身退；表中的 AML 位元組碼是由 **OS 內建的 ACPI Interpreter** 執行的，不是由韌體執行。這就是為什麼改壞 DSDT 通常導致 OS 開機當機，而不是 BIOS 畫面出不來。

1.3 OS 如何取得 Hardware Information ?

硬體分成兩類。第一類是可列舉（enumerable）的：PCI、USB 裝置插上去就能被匯流排協定自動發現，不太需要 ACPI 幫忙。第二類是不可列舉的：Embedded Controller、GPIO 控制器、I2C 感測器、電池、電源鍵、筆電上蓋開關……這些裝置沒有自我介紹的能力，**OS 完全依賴 ACPI 的描述才知道它們存在**。

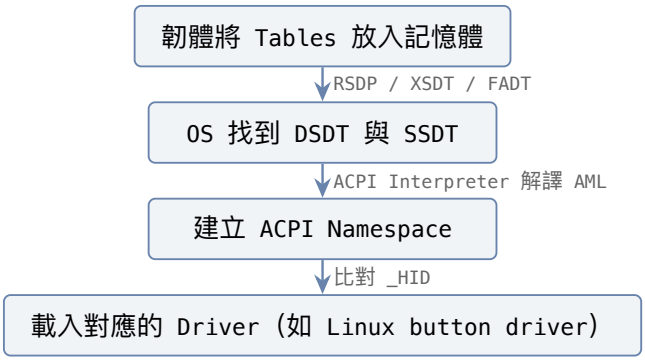
以「筆電上蓋開關」為例，DSDT 裡通常有這樣一段 ASL：

DSDT 片段：描述筆電上蓋開關

```
Scope (\_SB)
{
    Device (LID0)
    {
        Name (_HID, EisaId ("PNP0C0D"))  // Lid device 的標準 ID

        Method (_LID, 0, NotSerialized)  // Query: open or closed?
        {
            Return (LIDS)                // LIDS: EC field, 1 = open
        }
    }
}
```

OS 開機時的處理流程是：



之後只要上蓋開閤，韌體發出事件，OS 呼叫 `_LID` 方法讀取狀態，再決定要不要睡眠——描述來自韌體，決策屬於 OS，這正是 OSPM 的精神。

1.4 ACPI 與 Device Driver 的關係

ACPI 不取代 Driver，而是替 Driver 做三件事：

- **發現**（Discovery）：透過 `_HID` / `_CID` 識別碼，OS 知道該載入哪個 Driver。
- **資源**（Resources）：透過 `_CRS`，Driver 知道裝置用哪段 MMIO、哪個 IRQ、哪條 I2C 匯流排。
- **控制**（Control）：Driver 在執行期呼叫 ACPI Method （例如 `_PS0` 上電、`_PS3` 斷電、`_DSM` 裝置特定功能），由 AML 代為敲硬體暫存器。

核心觀念

ACPI 描述「裝置在哪裡、怎麼開關」；Driver 實作「裝置怎麼用」。兩者透過 Namespace 中的物件與 Method 互相銜接：Driver 是 Method 的呼叫者，AML 是 Method 的實作者。

1.5 ACPI 在 Windows / Linux 中的位置

兩大作業系統都內建 ACPI Interpreter：

- **Windows**：`acpi.sys` 是 ACPI bus driver，負責解譯 AML、建立裝置樹，再由 PnP Manager 載入各裝置驅動。
- **Linux**：核心的 `drivers/acpi/` 子系統內嵌了 **ACPICA**（ACPI Component Architecture，Intel 維護的參考實作），與 iASL 編譯器系出同門。

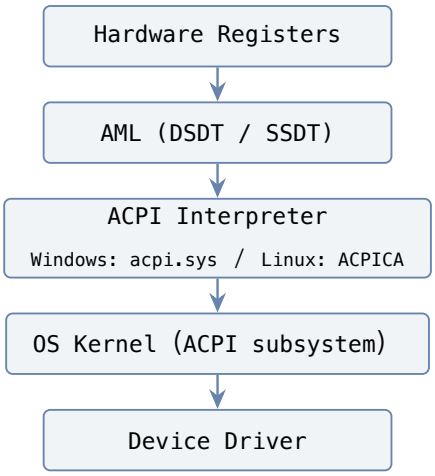
在 Linux 上，你隨時可以觀察 ACPI 的存在：

觀察 Linux 上的 ACPI

```
$ ls /sys/firmware/acpi/tables/
APIC  DSDT  FACP  FACS  HPET  MCFG  SSDT1  SSDT2  ...

$ dmesg | grep -i acpi | head -5
ACPI: Core revision 20230628
ACPI: Interpreter enabled
ACPI: PM: (supports S0 S3 S4 S5)
```

整體軟體堆疊如下（由下而上）：



1.6 ACPI 與 Power Management 的關係

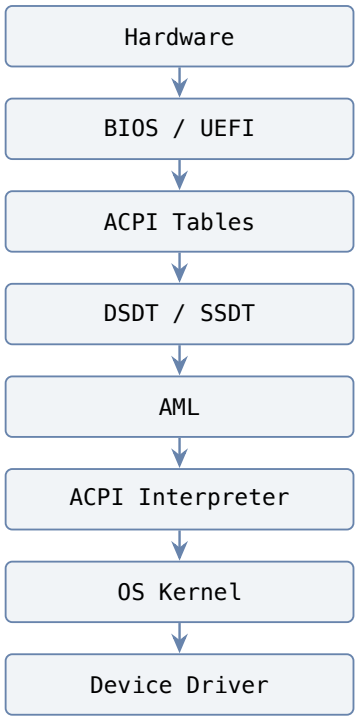
電源管理是 ACPI 最初誕生的理由，也是它最重要的應用。ACPI 定義了多層次的電源狀態（詳細內容在 Module 7）：

層級	狀態	意義
系統	S0–S5	S0 運作中、S3 睡眠、S4 休眠、S5 關機
裝置	D0–D3	D0 全速運作、D3 斷電
CPU（閒置）	C0–Cn	C0 執行中，數字越大越省電、喚醒越慢
CPU（效能）	P0–Pn	P0 最高頻率，數字越大頻率越低

按下睡眠鍵時，OS 會呼叫 DSDT 中的 _PTS（Prepare To Sleep），逐一將裝置降到低功耗 D-state，最後透過 FADT 描述的暫存器讓系統進入 S3——每一步都由 ASL 寫的 Method 參與。

1.7 完整資料流：從硬體規格到 Driver

把本章所有觀念串起來，就是這張你必須背下來的圖：



而站在開發者的角度，Intel 的 ASL Tutorial 描述的工作流程則是這張圖的「生產端」：



本教材接下來的所有內容，都是在把這條生產線的每一站拆開來練習：Module 2 先搞懂 ASL 與 AML 的關係和 Namespace，Module 3 學語法，Module 4–7 學會描述裝置與電源，Module 8–10 進入 DSDT/SSDT 實戰與除錯，最後在 Module 11–12 把 ACPI 接上 Linux Kernel 與 Driver。

注意：常見誤解

- 「ACPI 是 BIOS 設定畫面」——錯。設定畫面是韌體 UI；ACPI 是韌體交給 OS 的資料與位元組碼。
- 「AML 由 BIOS 執行」——錯。AML 由 OS 的 Interpreter 執行。
- 「PCI 裝置都靠 ACPI 才能被發現」——錯。PCI 可自動列舉，ACPI 主要補足不可列舉裝置與電源管理。

核心觀念：本章重點回顧

- ACPI 讓「描述」與「決策」分離：韌體描述硬體，OS 決策管理（OSPM）。
- BIOS/UEFI 生產 ACPI Tables；OS 透過 RSDP → XSDT → FADT 找到 DSDT，解譯 AML 建立 Namespace。
- 不可列舉的裝置（EC、GPIO、I2C 感測器、電池）完全依賴 ACPI 描述。
- Driver 透過 `_HID` 綁定、`_CRS` 取得資源、呼叫 Method 控制電源。
- Windows 用 `acpi.sys`，Linux 用內嵌的 ACPICA。

動手練習

1. 用自己的話解釋 OSPM：APM 時代與 ACPI 時代，「誰做決策」有什麼不同？
2. 為什麼 Embedded Controller 和 I2C 感測器一定需要 ACPI 描述，而 USB 隨身碟不需要？
3. 在一台 Linux 機器上執行 `ls /sys/firmware/acpi/tables/`，記錄你看到的表，並對照本章表格說出每一張的用途。
4. 不看書，畫出從 Hardware 到 Device Driver 的八層資料流圖，並標出「哪一層是韌體產生的、哪一層是 OS 執行的」。

Module 2

ASL、AML 與 ACPI Namespace

難度：★★☆☆☆

Module 1 建立了世界觀：韌體負責描述硬體，OS 負責解讀與決策。本模組走進生產線的最前段——開發者手上的 ASL 原始碼如何被 iasl 編譯成 AML 位元組碼，AML 又如何在開機時被 ACPI Interpreter 展開成一棵 Namespace 樹。讀完本模組，你將親手編譯第一張表、直視機器語言的位元組，並能準確說出 `\_SB.DEV0` 這種路徑在樹上代表什麼。

第 2 章

ASL vs AML

第 1 章的結尾出現過一條生產線：硬體規格經過 ASL、AML，最後抵達 OS 的 ACPI Interpreter。本章把鏡頭拉近到前兩站，回答一個看似簡單、卻是每個 BIOS 新人都踩過的問題：**ASL 和 AML 到底是不是同一個東西？**答案是否定的——它們的關係就像原始碼與機器碼，中間隔著一個編譯器。

本章你會第一次動手：寫 `.dsl`、用 iasl 編譯、用 xxd 直視位元組碼、再反組譯回文字——走完這一圈，「ASL ≠ AML」就是你親眼驗證過的事實。

本章學習目標

- 說明 ASL 與 AML 的差異，以及 iasl 在兩者之間扮演的角色。
- 逐欄解釋 `DefinitionBlock` 六個參數的意義與它們在表頭中的位置。
- 使用 iasl 編譯 `.dsl` 檔，並讀懂編譯器的每一行輸出訊息。
- 用 xxd 檢視 AML，辨認 ScopeOp、DeviceOp、MethodOp 等 opcode。
- 使用 `iasl -d` 反組譯 AML，並解釋為什麼結果與原始碼不完全相同。
- 不看書畫出從 `.dsl` 到 ACPI Interpreter 的完整流程圖。

2.1 從 C 語言與機器碼談起

如果你寫過 C，下面這張對照表就是理解 ASL 與 AML 最快的捷徑：

	C 的世界	ACPI 的世界
原始碼（給人讀寫）	`main.c`	`test.dsl`（ASL）
編譯器	gcc / clang	iasl
產物（給機器執行）	機器碼（x86 / Arm 指令）	AML 位元組碼
執行者	CPU 硬體直接執行	OS 內的 ACPI Interpreter

ASL（ACPI Source Language）是給開發者使用的原始語言：有註解、有縮排、有人類看得懂的關鍵字。**AML**（ACPI Machine Language）則是 ACPI Interpreter 執行的機器語言：一串緊湊的二進位 opcode，一個位元組都不浪費。兩者之間由 Intel 維護的編譯器 **iasl** 負責轉換。

但這個類比在「執行者」一列破功了：機器碼由 CPU 硬體直接執行，AML 卻是由軟體——住在 OS kernel 裡的直譯器——逐個 opcode 解譯（第 1 章：`acpi.sys` 與 ACPICA）。更貼切的類比是 Java：`javac` 編出 bytecode 交給 JVM，iasl 編出 AML 交給 ACPI Interpreter；差別是後者內建在每一份 OS kernel 之中。

核心觀念：一句話記住三者關係

ASL 是人寫的原始語言；AML 是 interpreter 執行的機器語言；iasl 是橋樑，能編譯也能反組譯。韌體映像與 ACPI Tables 裡 永遠只有 **AML**，不會有一行 ASL 文字。

2.2 第一個.dsl 檔：DefinitionBlock 逐欄解剖

用任何編輯器建立 `test.dsl`（副檔名 `.asl` 也通用，本教材統一用 `.dsl`）。先看全貌：

範例：test.dsl——你的第一張表

```
DefinitionBlock ("test.aml", "SSDT", 2, "DEMO", "TESTTBL", 0x00000001)
{
    Scope (\_SB)
    {
        Device (DEV0)
        {
            Name (_HID, "DEMO0001")

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)
            }
        }
    }
}
```

每一份 ASL 原始碼都以 `DefinitionBlock` 開頭——宣告「大括號裡是一張完整的 ACPI 表」，六個參數對應表頭欄位：

參數	範例值	表頭位置	說明
AMLFileName	`"test.aml"`	不進表頭	歷史遺留欄位；現代 iasl 以輸入檔名決定輸出檔名，此欄實際上被忽略
TableSignature	`"SSDT"`	0x00（4 B）	表的簽章，只能是 `DSDT` 或 `SSDT`
ComplianceRevision	2	0x08（1 B）	2 以上：Integer 為 64-bit；1：32-bit（見下方補充）
OEMID	`"DEMO"`	0x0A（6 B）	廠商代號，最多 6 字元（實機常見 LENOVO、DELL）
OEM Table ID	`"TESTTBL"`	0x10（8 B）	這張表自己的名字，最多 8 字元
OEM Revision	`0x00000001`	0x18（4 B）	表的版本號，每次改表就遞增

表頭剩下的欄位不用你填，iasl 自動算好寫入：**Length**（0x04，全表長度）、**Checksum**（0x09，讓全表位元組總和除以 256 餘 0，OS 載表前會驗算）、**Creator ID** ／ **Revision**（0x1C 起，編譯器簽名 `INTL` 與版本號）。

補充說明：ComplianceRevision 與 32 / 64-bit Integer

這個數字決定 AML 世界裡整數的寬度：1 是 32-bit，2 以上是 64-bit，影響算術溢位與位元反轉——同一行 ASL 在兩種模式下可能算出不同答案。實務上全平台的整數寬度由 **DSDT** 的 revision 決定，SSDT 跟隨之；revision 1 是古早 32-bit Windows 時代的遺物，現代平台一律寫 2。

大括號裡是表的本體，先快速導覽（細節是第 3 章的主題）：`Scope (_SB)` 切換到系統匯流排 scope；`Device (DEV0)` 宣告一台裝置；`Name (_HID, ...)` 給它 OS 配對驅動程式用的硬體 ID；`Method (_STA, ...)` 回報狀態，`0x0F` 表示「存在、啟用、可見、正常」（第 8 章詳談）。

2.3 編譯：iasl test.dsl

iasl 是 ACPICA 工具組的一員，各平台都能直接安裝：

安裝 iasl

```
$ sudo apt install acpica-tools    # Ubuntu / Debian
$ brew install acpica              # macOS
```

在 `test.dsl` 所在目錄執行編譯，完整輸出如下：

第一次編譯

```
$ iasl test.dsl

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20230628
Copyright (c) 2000 - 2023 Intel Corporation

ASL Input:     test.dsl -      15 lines, 287 bytes, 6 keywords
AML Output:    test.aml -      75 bytes, 3 named objects,
                               1 executable opcodes

Compilation successful. 0 Errors, 0 Warnings, 0 Remarks

$ ls
test.aml    test.dsl
```

逐行解讀這份輸出：

- **ASL Input**：來源統計——15 行、287 位元組的文字。
- **AML Output**：只剩 75 位元組，註解與排版全部消失；「3 named objects」是 `DEV0`、`_HID`、`_STA`，「1 executable opcodes」是那句 `Return`。
- **Compilation successful**：Errors 會讓編譯失敗；Warnings 與 Remarks 是提醒，bring-up 實務上要求三者全零。

目錄裡多出來的 `test.aml` 就是 AML——一個不折不扣的二進位檔，用文字編輯器打開只會看到亂碼。下一節我們用正確的工具直視它。

2.4 直擊機器語言：解剖 test.aml

`xxd` 能以十六進位傾印任何二進位檔。75 個位元組一頁看完：

AML 位元組碼的真面目

```
$ xxd test.aml
00000000: 5353 4454 4b00 0000 0246 4445 4d4f 0000  SSDTK....FDEMO..
00000010: 5445 5354 5442 4c00 0100 0000 494e 544c  TESTTBL.....INTL
00000020: 2806 2320 1026 5c5f 5342 5f5b 821e 4445  (.# .&\_SB_[..DE
00000030: 5630 085f 4849 440d 4445 4d4f 3030 3031  V0._HID.DEMO0001
00000040: 0014 095f 5354 4100 a40a 0f              ..._STA....
```

前 36（0x24）個位元組是表頭，對照上一節的欄位表逐一驗收：第一列有簽章 `SSDT`、little-endian 的總長度 `4b 00 00 00`（75）、revision `02`、checksum `46` 與 OEMID `DEMO`；第二列是 `TESTTBL`、OEM Revision，以及 iasl 自動填入的 `INTL` 與版本號。

從 offset 0x24 開始才是 AML 本體。先認識幾個本例用到的 opcode （完整清單在 ACPI 6.5 規格書的第 20 章「AML Specification」）：

位元組	名稱	意義
`0x10`	ScopeOp	開啟一個 Scope
`0x5B`	ExtOpPrefix	延伸 opcode 的前導位元組（ASCII 的 `[`）
`0x5B 0x82`	DeviceOp	定義 Device（雙位元組 opcode）
`0x14`	MethodOp	定義 Method
`0x08`	NameOp	定義 Name 物件
`0xA4`	ReturnOp	回傳值
`0x0A`	BytePrefix	後面跟著 1 位元組的整數常數
`0x0D`	StringPrefix	後面跟著以 `0x00` 結尾的字串
`0x5C`	RootChar	路徑的根前綴（就是 ASCII 的反斜線）

拿著這張表，把 0x24 之後的位元組逐段翻譯：

test.aml 位元組逐段解讀（offset 0x24 起）

```
10 26                           ← ScopeOp + PkgLength (0x26 = 38)
```

```
5C 5F 53 42 5F                ← RootChar + NameSeg "_SB_"
5B 82 1E                      ← DeviceOp + PkgLength (0x1E = 30)
44 45 56 30                   ← NameSeg "DEV0"
08                            ← NameOp
5F 48 49 44                   ← NameSeg "_HID"
0D 44 45 4D 4F 30 30 30 31 00 ← StringPrefix + "DEMO0001" + 結尾
14 09                         ← MethodOp + PkgLength (0x09 = 9)
5F 53 54 41                   ← NameSeg "_STA"
00                            ← MethodFlags (0 個參數、不序列化)
A4 0A 0F                      ← ReturnOp + BytePrefix + 0x0F
```

三個值得停下來咀嚼的觀察：

- **名稱就是 ASCII**：`5f 53 42 5f` 直接是 `_SB_`，`44 45 56 30` 就是 `DEV0`，所以 xxd 右側的 ASCII 欄看得到它們。名稱固定 4 字元、原樣內嵌（第 3 章正式介紹這條規則）。
- **PkgLength 是包裹的尺寸**：每個會「包住別人」的 opcode（Scope、Device、Method）後面都跟著 **PkgLength**，標示區塊到哪裡結束，數值含 PkgLength 本身；超過 63 時會追加 1 到 3 個位元組。
- **opcode 空間不夠就開第二層**：`0x5B` 是 ExtOpPrefix，後面再跟一個位元組決定操作——Device 的 `0x5B 0x82` 就是這樣來的。

這 75 個位元組就是 interpreter 眼中「機器語言」的全部——你已經親眼看過 AML 了。

2.5 反組譯：iasl -d 與 roundtrip

注意：別讓 -d 蓋掉你的原始碼

`iasl -d test.aml` 會把結果寫到 `test.dsl`——和你的原始碼**同名，直接覆蓋、不會詢問**。註解救不回來，先 `cp` 一份再反組譯是韌體工程師的肌肉記憶。

安全的反組譯流程

```
$ cp test.aml rt.aml
$ iasl -d rt.aml

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20230628
Copyright (c) 2000 - 2023 Intel Corporation

Input file rt.aml, Length 0x4B (75) bytes
Pass 1 parse of [SSDT]
Pass 2 parse of [SSDT]
Parsing Deferred Opcodes (Methods/Buffers/Packages/Regions)
```

```
Parsing completed
Disassembly completed
ASL Output:    rt.dsl - 1058 bytes
```

打開 `rt.dsl`，和原始碼比一比：

rt.dsl：反組譯的結果（節錄表頭註解）

```
/*
 * Intel ACPI Component Architecture
 * AML/ASL+ Disassembler version 20230628 (64-bit version)
 *
 * Original Table Header:
 *     Signature        "SSDT"
 *     Length           0x0000004B (75)
 *     Revision         0x02
 *     Checksum         0x46
 *     OEM ID           "DEMO"
 *     ...
 */
DefinitionBlock ("", "SSDT", 2, "DEMO", "TESTTBL", 0x00000001)
{
    Scope (\_SB)
    {
        Device (DEV0)
        {
            Name (_HID, "DEMO0001")  // _HID: Hardware ID
            Method (_STA, 0, NotSerialized)  // _STA: Status
            {
                Return (0x0F)
            }
        }
    }
}
```

語意一模一樣，但文字不是原樣。差異來自四件事：

- **你的註解與空行全數消失**：AML 沒儲存它們，反組譯器只能按固定風格重新排版。
- **AMLFileName 變成空字串**：檔名沒被編進 AML，反組譯器無從得知。
- **多出來的註解是反組譯器加的**：`// _HID: Hardware ID` 來自內建的預定義名稱表。
- **寫法被正規化**：數值統一成十六進位；legacy operator（如 `Store`）預設被改寫成 ASL 2.0 符號寫法，加 `-dl` 可要求 legacy 風格（Module 3 兩種都教）。

那語意到底有沒有損失？把反組譯結果重新編譯、與原本的 AML 逐位元組比對：

roundtrip 驗證：語意零損失

```
$ iasl rt.dsl

...
Compilation successful. 0 Errors, 0 Warnings, 0 Remarks

$ cmp rt.aml test.aml && echo BYTE-IDENTICAL
BYTE-IDENTICAL
```

結論值得背起來：**ASL 到 AML 丟掉的是外觀（註解、排版、檔名），留下的是完整語意**；所以 AML 到 ASL 再到 AML 可以位元組級重現，但 ASL 到 AML 再到 ASL 永遠回不到原來的文字。

2.6 完整旅程：從.dsl 到 ACPI Interpreter

把本章的實驗接回第 1 章的大圖，ASL 的一生如下：



為什麼韌體裡只放 AML？因為它緊湊（75 對 287 位元組）、免依賴（OS 只需 interpreter，不必內建編譯器）、又標準化（opcode 由規格書定義，任何 OS 讀到的都是同一套語言）。

注意：常見誤解

- 「AML 是 CPU 執行的機器碼」——錯。CPU 執行的是 kernel 裡 interpreter 的指令；AML 只是 interpreter 的輸入資料。
- 「`.dsl` 檔可以直接塞進韌體」——錯。韌體映像裡只有編譯後的 AML；沒有任何元件看得懂 ASL 文字。
- 「ASL 和 AML 只是副檔名不同」——錯。一個是文字原始碼、一個是二進位位元組碼，中間隔著一次編譯。

核心觀念：本章重點回顧

- ASL 是給人寫的原始語言；AML 是 ACPI Interpreter 執行的機器語言；iasl 負責編譯與反組譯，韌體裡永遠只有 AML。
- AML 由 OS 內的 interpreter 以軟體方式解譯執行，不是 CPU 指令——類比 Java bytecode 與 JVM。
- `DefinitionBlock` 六個參數對應表頭欄位；ComplianceRevision 2 代表 64-bit Integer。
- AML 等於 36 位元組表頭加 opcode 串流：Device 是 `0x5B 0x82`、Method 是 `0x14`，名稱以 4 字元 ASCII 內嵌，區塊長度靠 PkgLength 界定。
- 反組譯外觀有損、語意無損：註解與排版永遠消失，但重新編譯可得到位元組級相同的 AML。

動手練習

1. 用自己的話解釋：為什麼說 AML 是「機器語言」，卻又不是 CPU 執行的？那到底是誰在執行它？
2. `DefinitionBlock` 的六個參數中，哪一個不會出現在最終的 AML 表頭？ComplianceRevision 填 1 與 2 差在哪？
3. 紙上反組譯：不用電腦，把位元組序列 `14 08 4D 59 4D 31 00 A4 01` 翻譯回 ASL（提示：`0x01` 是 OneOp，代表常數 1）。
4. 動手做：安裝 iasl 並編譯本章的 `test.dsl`，用 xxd 找出 OEMID `DEMO` 的 offset；再把 OEM Revision 改成 2 重編，用 `cmp -l` 找出變動的位元組並解釋原因。
5. 動手做：對 `test.aml` 執行安全的反組譯流程，列出結果與原始碼的三個差異，再重編反組譯結果並用 `cmp` 驗證 AML 位元組級相同。

第 3 章

ACPI Namespace

上一章的結尾，`test.aml` 的 75 個位元組抵達了 ACPI Interpreter。但它做的第一件事不是「執行」——大部分 AML 描述的是結構，不是動作——而是把這串 opcode 展開成一棵活在記憶體裡的樹：**ACPI Namespace**。這棵樹是 OS 認識整台機器的地圖。本章教你讀懂它：長什麼樣、名字有什麼規則、路徑怎麼解析。這些是後續所有章節的地基——寫 ASL 的本質就是在對這棵樹做規劃。

本章學習目標

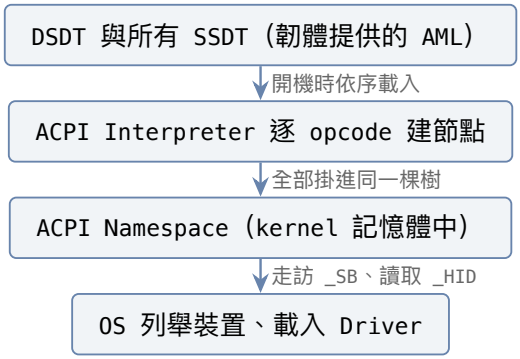
- 說明 Namespace 是什麼、由誰在什麼時機建立。
- 畫出含五個預定義根 scope 的 Namespace 樹並說出各自用途。
- 說明 NameSeg 的 4 字元規則與補底線行為。
- 寫出最小可編譯的 Namespace 範例，指出每一行在樹上做了什麼。
- 分辨絕對路徑、相對路徑、上層前綴與單段名稱的向上搜尋。
- 說明 Scope 與 Device 的差異、Alias 的用途。

3.1 Namespace：OS 眼中的整台機器

ACPI Namespace 是一棵具名物件的樹：每個節點有一個名字，節點上掛著物件——Device、Method、Name 這些你在第 2 章見過的東西。重點在於它的生成時機與材料：

核心觀念：Namespace 是開機時「蓋」出來的

Namespace 不存在於韌體中。開機時，OS 的 ACPI Interpreter 先載入 DSDT（永遠第一張），逐個 opcode 建立節點；接著依序載入每一張 SSDT，把內容**掛進同一棵樹**。多張表、一棵樹——韌體給的是藍圖（AML），樹是 OS 在自己的 kernel 記憶體裡蓋出來的房子。



「多張表合併成一棵樹」有一個直接的推論：**名字會撞**。如果 DSDT 和某張 SSDT 都宣告了 `\_SB.DEV0`，interpreter 載入第二份定義時會回報 `AE_ALREADY_EXISTS`，輕則該物件失效、重則整張表被拒載——實機除錯的常客，第 24 章詳談。

3.2 一棵樹的全貌與預定義根 Scope

樹的根是 **Root Scope**，寫作一個反斜線 `\`。根底下有幾個規格預先定義的 scope，各有各的住戶。一台典型筆電的樹長這樣：

典型筆電的 Namespace（節錄）

```
\                       ← Root Scope：一切路徑的起點
├── _SB_                系統匯流排：所有裝置物件的家
│   ├── PCI0            PCI Host Bridge
│   │   └── LPCB        LPC Bridge
│   ├── CPU0            處理器（現代寫法放這裡）
│   ├── USB0            USB Controller
│   └── EC0_            Embedded Controller
├── _PR_                Processor 物件（傳統擺放位置）
├── _TZ_                Thermal Zone（熱管理）
├── _SI_                系統指示燈
├── _GPE                General Purpose Event 處理方法
└── _OSI                預定義 Method：讓 AML 詢問 OS 身分
```

五個預定義根 scope 的分工如下：

Scope	名稱由來	住著什麼
`\_SB`	System Bus	所有 Device 物件；OS 列舉硬體的起點
`\_PR`	Processor	傳統的 Processor 物件（新平台改用 `\_SB` 下的 Device）
`\_TZ`	Thermal Zone	ThermalZone 物件與溫控 Method （第 17 章）
`\_SI`	System Indicators	系統狀態指示，如 `_SST`
`\_GPE`	General Purpose Events	GPE 事件處理 Method （`_L06`、`_E09` 這類命名）

補充說明：這五個 scope 不用你建、也輪不到你建

Interpreter 在載入任何一張表之前，就會先把 `\_SB`、`\_PR`、`\_TZ`、`\_SI`、`\_GPE` 連同 `_OSI`、`_REV`、`_GL` 等預定義物件建好，所以 DSDT 可以直接 `Scope (_SB)` 走進去掛東西。

如果你熟悉檔案系統，直接套用那個心智模型：根目錄 `/` 對應 Root Scope `\`，目錄對應 Device 這類可含子節點的物件，檔案對應 Name、Method 這類葉節點，「目前工作目錄」對應目前 scope，而 `..` 對應上層前綴 `^`。

3.3 命名規則：NameSeg 的四字元世界

樹上每個節點的名字叫 **NameSeg**，規則硬得像水泥：

- **固定 4 個字元**。在 AML 裡 NameSeg 佔恰好 4 個位元組，不多不少。
- **合法字元**：開頭是大寫字母或底線，其後可用大寫字母、數字、底線。ASL 名稱不分大小寫，一律視為大寫。
- **不足 4 字元，編譯器自動補尾端底線**：你寫 `_SB`，AML 裡是 `_SB_`；寫 `EC0`，存的是 `EC0_`；`CPU0` 剛好 4 字元，原樣保留。
- **底線開頭保留給規格**：`_HID`、`_STA`、`_SB_` 都是規格定義的名字；自家物件請避開底線開頭，未來規格新增名稱時才不會撞上。

這就是 `_SB_` 這個怪名字的由來：「`_SB` 補一條底線」。第 2 章的 xxd 傾印裡你已親眼看過證據——offset 0x27 起的 `5f 53 42 5f` 正是 `_SB_`，補的底線真實存在於 AML 中；Linux 的 `/sys` 與 acpidump 顯示的也是這個形式。

為什麼這麼死板？為了 interpreter 的效率：名稱定長 4 位元組，比對名字只需一次 32-bit 整數比較，路徑編碼也不需要長度欄位。

3.4 逐行拆解：最小可編譯的 Namespace

下面這個最小範例值得倒背如流——10 行原始碼、樹上兩個新節點：

最小 Namespace 範例（DSDT 身分）

```
DefinitionBlock ("test.aml", "DSDT", 2, "DEMO", "TEST", 1)
{
    Scope (\_SB)
    {
        Device (DEV0)
        {
            Name (_HID, "DEMO0001")
        }
    }
}
```

逐行拆解：

- **第 1 行 `DefinitionBlock`**：這次簽章是 `DSDT`，模擬平台主表的身分。revision 2 表示 64-bit Integer；OEM Table ID `"TEST"` 才 4 字元，上限是 8，沒問題。
- **`Scope (_SB)`**：`\` 開頭是絕對路徑——從根走到 `_SB_`。`Scope` **不建立任何東西**，只是把「目前位置」移過去。

- **`Device (DEV0)`**：在目前位置底下建立一個 Device 節點。`DEV0` 是相對名稱，完整路徑是 `\_SB.DEV0`。
- **`Name (_HID, "DEMO0001")`**：在 `DEV0` 底下建立 Name 物件，值是字串；完整路徑 `\_SB.DEV0._HID`。
- **三個右大括號**：依序關閉 Device、Scope、DefinitionBlock，「目前位置」隨之退回。

Interpreter 載完這張表之後，樹上多了什麼：

這張表建出來的樹

```
\
└── _SB_              ← 預先存在；Scope 只是走進去
    └── DEV0          ← 新建：Device 物件
        └── _HID      ← 新建：Name 物件，值為字串 DEMO0001
```

3.5 `\_SB.DEV0` 到底代表什麼？

這是本章的核心問題，值得用一整節、從三個層次把它回答到見骨。

層次一：文法上，它是一條絕對路徑——`\` 表示從根出發，`_SB`（實為 `_SB_`）與 `DEV0` 是兩個 NameSeg，句點是路徑分隔符。

層次二：Namespace 上，它指向樹上的一個節點：interpreter 讀到 `0x5B 0x82`（DeviceOp）時建立、掛在 `_SB_` 底下的 Device 物件。路徑只是「住址」，物件本身才是「住戶」。

層次三：OS 眼中，它就是一台裝置。OS 走訪 `\_SB` 子樹，遇到 Device 節點就讀取 `_HID`、配對驅動程式。如果這張表真的被載入一台 Linux 機器（Module 10 會教你怎麼做），你會看到：

Linux 眼中的 DEV0

```
$ ls /sys/bus/acpi/devices/ | grep DEMO
DEMO0001:00

$ cat /sys/bus/acpi/devices/DEMO0001:00/path
\_SB_.DEV0
```

注意 sysfs 印出的路徑是 `_SB_.DEV0`——補滿底線的四字元真身。

最後一個必須內化的心法：**它不是硬體本身**。`\_SB.DEV0` 只是韌體對 OS 的一句宣告——「這裡有台裝置，ID 是 DEMO0001」。即使主機板上根本沒焊那顆料，宣告依然成立、OS 照樣列舉出一台裝置（所以才需要 `_STA` 回報實際狀態，第 8 章詳談）。這正是第 1 章「描述與決策分離」的延伸。

核心觀念：一句話答案

`\_SB.DEV0` 是 Namespace 樹上的一條絕對路徑，指向一個名為 `DEV0` 的 Device 物件；對 OS 而言，它就是「一台裝置」的完整住址。

3.6 路徑解析：絕對、相對與向上搜尋

在 ASL 裡引用一個名字時，interpreter 依你的寫法套用四種規則：

- **絕對路徑**（`\` 開頭）：從根出發逐段往下走，找不到就報錯，絕不嘗試別處。
- **上層前綴**（`^` 開頭）：從目前 scope 往上剝一層再解析剩餘路徑；可以疊加，`^^` 就是上兩層。
- 單一 **NameSeg**（無前綴、無句點）：先在目前 scope 找，沒有就到上一層找，一路找到根——這叫 **search-to-root**（向上搜尋），最先找到者勝，「就近原則」。
- **多段相對路徑**（含句點但不以前綴開頭，如 `PCI0.DVAL`）：只從目前 scope 出發往下解析，不做向上搜尋。

用一張可編譯的表把四種規則一次演完：

路徑解析示範（Store 為「把前者存入後者」，Module 3 詳談）

```
DefinitionBlock ("", "SSDT", 2, "DEMO", "PATHS", 1)
{
    Name (ROOT, 0x11)               // \ROOT

    Scope (\_SB)
    {
        Name (SVAL, 0x22)           // \_SB.SVAL

        Device (PCI0)
        {
            Name (DVAL, 0x33)       // \_SB.PCI0.DVAL

            // Method scope node: \_SB.PCI0.M1
            Method (M1, 0, NotSerialized)
            {
                Store (DVAL, Local0)       // search up: 0x33
                Store (SVAL, Local1)       // search up: 0x22
                Store (ROOT, Local2)       // search up: 0x11
                Store (\_SB.SVAL, Local3)  // absolute
                Store (^DVAL, Local4)      // parent prefix
                Return (Local0)
            }
        }
```

```
    }
}
```

站在 M1 裡面，每一種寫法的解析過程如下。注意一個新手最常忽略的事實：**Method 本身也是一個 namespace 節點**，所以目前 scope 是 `\_SB.PCI0.M1`，不是 `\_SB.PCI0`：

寫法	解析過程	結果
`DVAL`	單段：M1 沒有，上一層 PCI0 命中	`0x33`
`SVAL`	單段：M1、PCI0 都沒有，`_SB_` 命中	`0x22`
`ROOT`	單段：一路搜到根才命中	`0x11`
`\_SB.SVAL`	絕對路徑直達，不搜尋	`0x22`
`^DVAL`	剝掉 M1 這層、落在 PCI0	`0x33`
`^^SVAL`	往上兩層到 `_SB_`	`0x22`
`PCI0.DVAL`	多段相對：只從 M1 往下找，不上搜	找不到

最後一列是經典陷阱：很多人以為相對路徑也會向上搜尋，於是在 Method 裡寫 `PCI0.DVAL`，執行期收到 `AE_NOT_FOUND`。記住：**向上搜尋只適用於單一 NameSeg**，路徑一旦含有句點或前綴，就只按字面解析。

3.7 Scope 與 Device 的差異

第 3.4 節說過 Scope 不建立東西，這一節把話說滿。兩者的差異可以濃縮成一句：**Device 蓋新房間，Scope 只是走進既有的房間繼續擺家具**。

	Scope	Device
建立新物件？	否——只切換「目前位置」	是，建立 Device 物件
目標必須已存在？	是，否則編譯錯誤	否，名稱本來就是新的
OS 列舉時可見？	Scope 本身不可見	可見（搭配 `_HID` 等）
典型用途	走進既有位置補充物件（SSDT 最常用）	宣告一台新裝置

「目標必須已存在」不是一句空話，iasl 會直接擋下來：

bad.dsl：故意示錯——對不存在的路徑開 Scope

```
DefinitionBlock ("", "SSDT", 2, "DEMO", "BAD", 1)
{
    Scope (\_SB.TSCP)       // TSCP was never created anywhere
    {
        Name (TST0, One)
    }
}
```

iasl 拒絕編譯

```
$ iasl bad.dsl
bad.dsl    3:     Scope (\_SB.TSCP)
Error    6084 -        Object does not exist ^  (\_SB.TSCP)

Compilation failed. 1 Errors, 0 Warnings, 0 Remarks
```

那 SSDT 想往「別張表定義的裝置」底下掛東西怎麼辦？答案是先用 `External` 宣告「這個名字存在於別處」——Module 8 詳談。

3.8 Alias：同一個物件的第二個名字

`Alias` 替既有物件取第二個名字——注意，是**同一個物件的另一個門牌**，不是複製一份：

Alias 示範：把深處的旗標開到淺處

```
DefinitionBlock ("", "SSDT", 2, "DEMO", "ALIAS", 1)
{
    Scope (\_SB)
    {
        Device (EC0)
        {
            Name (BATP, One)         // battery present flag
        }

        // Second doorplate for the same object
        Alias (\_SB.EC0.BATP, BPRE)  // creates \_SB.BPRE
    }
}
```

Alias 之後的樹

```
\
└── _SB_
    ├── EC0_
    │   └── BATP      ← 原名
    └── BPRE          ← Alias：與 BATP 是同一個物件
```

語法是 `Alias (來源物件, 新名字)`：來源用任何合法路徑指定，新名字建在目前 scope 底下；之後從 `\_SB.EC0.BATP` 或 `\_SB.BPRE` 讀寫，碰到的都是同一份資料。常見用途有二：把樹深處的物件開放到較好引用的淺處；平台改版後保留舊名字，讓期待舊路徑的 OS 或驅動程式繼續運作。

注意：常見錯誤

- **用 Scope 建新裝置**：`Scope (_SB.NEW0)` 在 `NEW0` 不存在時直接編譯錯誤；建裝置要用 `Device`。
- **期待多段相對路徑向上搜尋**：含句點的路徑只從目前 scope 出發；向上搜尋只屬於單一 NameSeg。
- **在 Method 裡用 ^ 忘了 Method 也是一層 scope**：`^` 先剝掉的是 Method 自己那一層。
- **兩張表建同名物件**：開機載表時 `AE_ALREADY_EXISTS`——第 24 章教你從 dmesg 抓錯。

核心觀念：本章重點回顧

- Namespace 是開機時 interpreter 讀取 DSDT 與所有 SSDT、合併建出的具名物件樹，活在 OS kernel 的記憶體裡。
- 五個預定義根 scope——`\_SB`（裝置）、`\_PR`、`\_TZ`、`\_SI`、`\_GPE`——由 interpreter 預建，直接 Scope 進去用。
- NameSeg 固定 4 字元，不足補尾端底線（`_SB` 實為 `_SB_`）；底線開頭的名字保留給規格。
- 路徑四規則：`\` 絕對、`^` 上層、單段名稱向上搜尋（就近原則）、多段相對路徑不向上搜尋。
- Scope 開門不蓋房、目標必須已存在；Device 蓋房；Alias 給同一物件第二個門牌。
- `\_SB.DEV0` 是指向一個 Device 物件的絕對住址，OS 眼中的一台裝置——但它是描述，不是硬體本身。

動手練習

1. 不看書說出五個預定義根 scope 各自住著什麼，以及哪一個是 OS 列舉裝置的起點。
2. 下列名稱哪些是合法的 NameSeg？寫出編譯器實際儲存的 4 字元形式：`EC`、`_BAT`、`TEMP1`、`3CPU`、`GPU_`。
3. 沿用 3.6 節的路徑範例：若把 `M1` 直接搬到 `Scope (_SB)` 底下，表中每種寫法各解析到哪裡？哪幾行會從此失敗？
4. 解釋：為什麼 `Scope (_SB.NEW0)` 會編譯失敗，`Device (NEW0)` 卻沒問題？兩者對樹各做了什麼？
5. 動手做：擴充本章的最小範例——在 `\_SB` 下新增 `DEV1`，並用 `Alias` 把 `\_SB.DEV0._HID` 開放成 `\_SB.HID0`；編譯、反組譯，畫出最終的樹。

Module 3

ASL 基本語法

難度：★★☆☆☆

前兩個模組建立了世界觀：ASL 原始碼被 iasl 編譯成 AML，由作業系統的直譯器執行，並掛進 ACPI Namespace 這棵樹。本模組進入語言本身——第 4 章打好型別地基，第 5 章認識變數與常數，第 6 章掌握新舊兩代運算子，第 7 章用流程控制把邏輯串起來。學完之後，你將能讀懂真實 DSDT 裡九成 Method 的本體，並寫出自己的第一批可編譯、可執行的 ASL 程式。

第 4 章

Data Types

第 3 章告訴你 Namespace 是一棵掛滿物件的樹；本章回答下一個問題：這些物件到底「是什麼」？ASL 沒有 `int`、`char` 這類型別宣告關鍵字——你宣告的是物件，型別由初始值決定，而且執行期還會換。這種動態型別的世界觀，讓你在 C 語言養成的直覺一半有用、一半會害你。

更麻煩的是，型別錯誤常常要到執行期才爆炸：iasl 願意讓你把 String 存進 Integer，因為規格書定義了一整套「隱式轉換」規則，直譯器會默默照做。規則沒背熟，Method 就會回傳「看起來像對、其實差很遠」的值——本章是整本書最值得反覆翻閱的基本功。

本章學習目標

- 列出 ASL 的主要物件型別，寫出各型別的字面值與典型用途。
- 說明 Integer 寬度由 DefinitionBlock 的 ComplianceRevision 決定，並解釋 32 / 64 位元差異造成的經典坑。
- 區分 String 與 Buffer，寫出 `Buffer (){}` 初始化與巢狀 `Package (){}`。
- 使用 `RefOf`、`DerefOf`、`Index` 操作 Object Reference，說明 Field Unit 的「代理」角色。
- 背出 implicit conversion 的方向與規則，需要時改用 `ToInteger` 等 explicit conversion 運算子。
- 用 `ObjectType` 與 `SizeOf` 在執行期檢查物件。

4.1 一切都是物件：型別總覽

在 ASL 裡，物件（object）是唯一的建材：整數是物件、字串是物件、一個 Device 是物件、一段可執行的 Method 也是物件。物件可以匿名存在（例如運算式中間的暫時值），也可以用 `Name` 掛上名字、放進 Namespace。型別大致分成三個家族：

ASL 物件型別的三大家族

```
ASL 的物件型別
├── 資料物件 (computational data)
│    ├── Integer   ← 64-bit 無號整數
│    ├── String    ← ASCII 字串，null 結尾
│    ├── Buffer    ← 原始位元組陣列
│    └── Package   ← 異質容器，可巢狀
├── 物件參考 (object reference)
```

```
│    └── 由 RefOf、Index、CondRefOf 產生
└── 命名空間物件（named objects）
    ├── Field Unit ← 硬體暫存器的代理人
    ├── Device、Method、Mutex、Event
    └── OperationRegion、ThermalZone、PowerResource 等
```

資料物件是 Method 裡搬來搬去的「值」；物件參考是指向其他物件的「箭頭」；命名空間物件則是掛在樹上的「結構」——它們描述硬體、提供行為，通常不會被當成值來運算。各型別的寫法與用途整理如下：

型別	建立／字面值寫法	典型用途
Integer	`42`、`0x2A`、`Zero`、`One`、`Ones`	暫存器值、旗標、計數器
String	`"PNP0C0D"`	`_HID` 識別字串、除錯訊息
Buffer	`Buffer () {0x01, 0x02}`	原始位元組、GUID、資源資料
Package	`Package () {...}`	結構化資料表（`_PRT`、`_PSS`）
Object Reference	`RefOf (X)`、`PKG0 [i]`	間接存取、走訪容器
Field Unit	在 `Field (...) {...}` 中宣告	硬體暫存器的代理（第 9 章）
Device	`Device (XXX) {...}`	裝置節點（第 8 章）
Method	`Method (XXX, n) {...}`	可執行的控制邏輯
Mutex、Event	`Mutex (MTX0, 0)`	同步物件（第 12 章）
OperationRegion	`OperationRegion (...)`	位址空間視窗（第 9 章）

核心觀念：型別跟著值走，不跟著名字走

ASL 是**動態型別**（dynamically typed）語言：名字只是掛勾，勾著什麼型別的物件、就是什麼型別；同一個 `Local0` 這行裝 Integer、下行可改裝 String。唯一的重要例外在 Store 的目標端——存進「已存在的具名資料物件」時，來源會被轉成目標的既有型別（4.6 節）。

4.2 Integer：64 位元的無號整數

ASL 的 Integer 沒有 `short` / `long` 之分，也沒有正負號與浮點數：一律是無號整數，運算溢位時默默繞回（wrap around），不會有任何警告。它的寬度由 `DefinitionBlock` 的第三個參數 **ComplianceRevision** 決定——就是本書每個範例開頭 `"SSDT", 2` 裡的那個 2：

注意：經典坑：DefinitionBlock 的第三個參數

- ComplianceRevision `>= 2`：Integer 是 64 位元，`Ones` 等於 `0xFFFFFFFFFFFFFFFF`。
- ComplianceRevision `= 1`：Integer 是 32 位元，`Ones` 等於 `0xFFFFFFFF`，超過 32 位元的常數與運算結果一律被截斷。
- 很多老 DSDT 為了相容早期 Windows 而寫 1，於是 `If (Local0 == Ones)` 這類判斷在新舊表之間行為不同，移植程式碼時務必先看這個參數。新專案一律寫 2。

由於沒有負數字面值，想表達「−1」時韌體工程師寫的是 `Ones`；想做減法防下溢，得自己先比大小（第 6 章實戰）。

4.3 String 與 Buffer

String 與 Buffer 都是「一串位元組」，初學者常混用，其實個性完全不同。String 是給人看的：內容限可列印 ASCII 字元，結尾隱含一個 null（`0x00`）；Buffer 是給機器看的：任意位元組都能放，長度明確宣告或由初始值推得。

Buffer 的三種初始化（片段）

```
Name (RAW0, Buffer () {0x10, 0x00, 0xFF})   // 3 bytes, as listed
Name (ZBF0, Buffer (8) {})                  // 8 bytes, all zero
Name (MIX0, Buffer (4) {0x01, 0x02})        // 01 02 00 00
```

`Buffer (N) {...}` 的括號內是「宣告長度」：初始值不足時自動補零（`MIX0`），完全省略初始值就是全零緩衝區（`ZBF0`）；省略長度時由初始值個數決定（`RAW0`）。兩者的差異整理如下：

比較項目	String	Buffer
內容	可列印 ASCII，null 結尾	任意位元組值
中途可含 `0x00`	不可（null 即結尾）	可以
`SizeOf` 回傳	字元數（不含結尾 null）	位元組數
典型用途	識別字串、訊息	暫存器資料、GUID、資源

一個好記的判斷法：資料若要跟硬體或二進位協定打交道，用 Buffer；若要給人（或 OS 的字串比對）讀，用 String。

4.4 Package：異質、可巢狀的容器

`Package` 是 ASL 唯一的複合容器：元素可以是任何資料物件，甚至是另一個 Package，型別不必一致。它是 ACPI「用資料表描述硬體」哲學的主要載體：中斷繞線表 `_PRT`、處理器效能狀態表 `_PSS`、裝置屬性表 `_DSD`，回傳的全是層層巢狀的 Package。

巢狀 Package：一筆簡化的 P-state 資料（片段）

```
Name (PST0, Package ()          // one CPU P-state entry
{
    2400,                       // core frequency (MHz)
    35000,                      // power (mW)
    Package () {0x10, 0x00}     // nested: control/status pair
})
```

Package 可以具名（掛在 `Name` 底下常駐）、也可以匿名（直接寫在 `Return` 裡，用完即丟）。單一 Package 上限 255 個元素；超過時 iasl 自動改用 VarPackage 位元組碼，語法不變。

補充說明：Package 元素的「延遲取值」

讀取 Package 元素拿到的不是值本身，而是一個 Object Reference，必須用 `DerefOf` 解參考才能取得值（下一節詳談）。這是 ASL 與 C 陣列手感差最多的地方：`PKG0 [1]` 不等於「第 1 號元素的值」，而是「指向第 1 號元素的箭頭」。

4.5 Object Reference 與 Field Unit

Object Reference 是 ASL 的指標。三個運算子會產生它：`RefOf (X)` 指向具名物件；`Index (PKG0, i)`——ASL 2.0 糖衣寫法 `PKG0 [i]`——指向 Package、Buffer 或 String 的第 `i` 個元素；`CondRefOf` 在物件可能不存在時安全試探。反向操作只有一個：`DerefOf` 把箭頭變回值。

Reference 的產生與解參考（片段）

```
Local0 = PKG0 [1]            // reference to element 1
Local1 = DerefOf (Local0)    // now the actual value
Local2 = DerefOf (BUF0 [0]) // one-liner: first byte of BUF0
Local3 = RefOf (NUM0)        // reference to a named object
PKG0 [0] = 0x5678            // store THROUGH the reference
```

最後一行示範了 reference 當「寫入目標」的行為：透過 `Index` 參考寫入，改的是容器裡的那個元素。這也是修改 Package 內容的標準做法。

Field Unit 則是 ASL 最獨特的型別：它看起來像變數，背後卻是硬體暫存器的代理人（proxy）。它在 `Field (...)` 區塊中宣告（第 9 章詳解）；例如把 EC 上的電池溫度暫存器宣告成 8 位元的 `BTMP` 之後，Method 裡一句 `Local0 = BTMP`，直譯器就會真的去 EC 讀回即時內容——語法上與讀一般變數毫無差別，這正是 ASL 好寫、也好踩雷的原因。

執行期想知道手上物件是什麼型別，用 `ObjectType`；它對 reference 會自動解參考、回傳目標物件的型別代碼：

代碼	型別	代碼	型別	代碼	型別
0	Uninitialized	6	Device	12	Processor（棄用）
1	Integer	7	Event	13	Thermal Zone
2	String	8	Method	14	Buffer Field
3	Buffer	9	Mutex	15	DDB Handle
4	Package	10	Operation Region	16	Debug Object
5	Field Unit	11	Power Resource		

搭配的 `SizeOf` 回傳「大小」：String 的字元數（不含 null）、Buffer 的位元組數、Package 的元素個數——第 7 章走訪 Package 就靠它決定終點。

4.6 Implicit Conversion：直譯器的自動轉換

當運算子拿到「型別不對」的運算元，直譯器不會報錯，而是依規格書的轉換表現場變出一個對的型別，這叫隱式轉換 （implicit conversion）。它發生在三種場合：

- **運算元轉換**：運算子要 Integer 你給了 String，先轉再算；雙運算元運算子（`Concatenate`、比較）以 第一個運算元的型別為準，把第二個轉過去。
- **Store 目標轉換**：目標是已存在的具名資料物件時，來源被轉成目標的既有型別。
- **Field 寫入轉換**：寫 Field Unit / Buffer Field 時轉成 Integer 或 Buffer，超出欄寬的位元直接捨棄。

Store 的目標不同，行為差很大，這張小表值得背下來：

Store 目標	行為
具名 Integer、String、Buffer	來源隱式轉換成目標的既有型別，目標型別不變
Local0–Local7、ArgX	不轉換：整個物件直接取代，型別跟著來源走
Field Unit、Buffer Field	轉成 Integer 或 Buffer 寫入，超寬位元捨棄
`Debug`	任何型別都收，轉成文字送到除錯輸出

三種資料型別之間的轉換規則如下（以 64 位元 Integer 為例）：

轉換方向	規則	範例
Integer → String	轉成 16 字元十六進位 ASCII （含前導零、無 `0x`）	`0x2A` → `"000000000000002A"`
Integer → Buffer	little-endian 逐位元組填入，長度等於 Integer 寬度	`0x1234` → `34 12 00 00 ...`
String → Integer	一律視為十六進位，遇非十六進位字元停止	`"1F"` → `0x1F`
String → Buffer	逐字元複製，含結尾 null	`"AB"` → `41 42 00`
Buffer → Integer	取前 8 bytes（little-endian），不足補零	`34 12` → `0x1234`
Buffer → String	逐位元組轉 `0x` 開頭十六進位文字，空白分隔	`01 02` → `"0x01 0x02"`

注意：十六進位還是十進位？兩套規則

隱式的 String → Integer 轉換一律當十六進位：`"10"` 會變成 `0x10`（十進位 16）。但顯式的 `ToInteger` 看前綴：`"0x10"` 是十六進位 16、`"10"` 是十進位 10。同一個字串走兩條路，結果差一截——處理 EC 或韌體吐出的數字字串時，永遠用 `ToInteger` 把意圖寫清楚，不要賭隱式轉換。

4.7 Explicit Conversion：把轉換寫清楚

隱式轉換是「直譯器幫你猜」；好的 ASL 應該用顯式運算子把意圖寫死。常用的一組如下：

運算子	結果	行為要點
`ToInteger (x)`	Integer	字串 `0x` 開頭當十六進位、否則十進位；Buffer 取前 8 bytes
`ToString (buf, len)`	String	Buffer 位元組當 ASCII 字元，取到 null 或 `len`（`Ones` 表示全部）
`ToBuffer (x)`	Buffer	Integer 給 8 bytes；String 連結尾 null 一起複製
`ToHexString (x)`	String	Buffer 逐位元組轉 `"0xNN"`，逗號分隔
`ToDecimalString (x)`	String	Buffer 逐位元組轉十進位文字，逗號分隔
`Concatenate (a, b)`	同 `a`	`b` 先轉成 `a` 的型別再串接；兩個 Integer 串成 16-byte Buffer
`ToUUID ("...")`	Buffer	編譯期巨集：GUID 字串轉 16-byte Buffer（`_DSM` 常客）

`Concatenate` 是隱式與顯式的交會點：它自己是顯式串接，第二個運算元卻走隱式轉換——把 Integer 接在提示字串後面，會自動變成 16 字元的十六進位文字（上一節表格的規則），這是韌體工程師組除錯訊息的日常招式。

4.8 完整範例：型別實驗室

把本章所有觀念裝進一個可編譯、可執行的 SSDT。先看程式：

完整範例：types.dsl

```
DefinitionBlock ("types.aml", "SSDT", 2, "ACME", "TYPES", 0x00000001)
{
    Name (NUM0, 0x2A)                   // Integer (decimal 42)
    Name (STR0, "ACPI")                 // String
    Name (BUF0, Buffer () {0x41, 0x43, 0x50, 0x49})   // "ACPI"
    Name (PKG0, Package ()
    {
        0x1234,                         // Integer element
        "battery",                      // String element
        Buffer () {0xAA, 0xBB},         // Buffer element
        Package () {1, 2}               // nested Package
    })

    Method (DEMO, 0, NotSerialized)
    {
        // explicit conversions
        Local0 = ToInteger ("0x2A")     // hex prefix  -> 0x2A
        Debug = Local0
        Local1 = ToInteger ("42")       // no prefix -> 42 = 0x2A
        Debug = Local1
        Local2 = ToHexString (BUF0)     // "0x41,0x43,0x50,0x49"
```

```
        Debug = Local2
        Local3 = ToString (BUF0, Ones)  // bytes as ASCII: "ACPI"
        Debug = Local3
        Local4 = ToBuffer (NUM0)        // 8-byte buffer, LSB first
        Debug = SizeOf (Local4)         // 8

        // implicit conversions
        Local5 = Concatenate ("NUM0 = ", NUM0)
        Debug = Local5                  // integer became hex text
        NUM0 = "1F"                     // store String -> Integer
        Debug = NUM0                    // 0x1F

        // runtime type inspection
        Debug = ObjectType (PKG0)       // 4 = Package
        Return (SizeOf (PKG0))          // 4 elements
    }
}
```

逐行拆解重點：

- 開頭四個 Name 宣告四種資料物件；BUF0 故意選 "ACPI" 四個字母的 ASCII 碼，讓 ToString 有戲唱。
- ToInteger ("0x2A") 與 ToInteger ("42") 殊途同歸都是 42——一個走十六進位、一個走十進位，印證顯式轉換「看前綴」的規則。
- ToHexString 與 ToString 對同一個 BUF0 給出完全不同的字串：前者是位元組的十六進位報表，後者把位元組當字元讀。
- NUM0 = "1F"：目標是具名 Integer，String 走隱式十六進位轉換，NUM0 變成 0x1F——型別不變、值變了；對照 Concatenate ("NUM0 = ", NUM0) 把 Integer 隱式轉成 16 字元十六進位文字，以及 Store 進 Local 完全不轉換。

用 iasl 編譯後，以使用者空間直譯器 acpiexec 執行（第 21 章詳解），每一步轉換的實際結果一覽無遺：

用 acpiexec 執行 DEMO

```
$ iasl types.dsl && acpiexec -b "execute DEMO" types.aml
ACPI Debug:  0x000000000000002A
ACPI Debug:  0x000000000000002A
ACPI Debug:  "0x41,0x43,0x50,0x49"
ACPI Debug:  "ACPI"
ACPI Debug:  0x0000000000000008
ACPI Debug:  "NUM0 = 000000000000002A"
ACPI Debug:  0x000000000000001F
ACPI Debug:  0x0000000000000004
```

```
Evaluation of \DEMO returned object
  [Integer] = 0000000000000004
```

補充說明：Debug 物件：ASL 的 printf

`Debug` 是特殊的內建物件：任何型別存進去都會轉成文字，送到直譯器的除錯輸出——acpiexec 的主控台，或 Linux 的 kernel log（Module 10 教你打開）。它是 ASL 最接近 `printf` 的東西。

核心觀念：本章重點回顧

- ASL 動態型別：型別跟著值走；但 Store 進具名資料物件時，來源會被轉成目標的既有型別，Local 與 Arg 則直接取代。
- Integer 一律無號；寬度由 ComplianceRevision 決定：`>= 2` 是 64 位元、`= 1` 是 32 位元，`Ones` 的值跟著變。
- String 是 null 結尾的 ASCII；Buffer 裝任意位元組；Package 是可巢狀的異質容器，元素以 reference 存取。
- `RefOf` / `Index` 造箭頭、`DerefOf` 取值；Field Unit 是硬體暫存器的代理型別。
- 隱式轉換的 String → Integer 一律十六進位；`ToInteger` 才看 `0x` 前綴。轉換規則表要背。
- `ObjectType` 回傳型別代碼，`SizeOf` 回傳字元數 / 位元組數 / 元素數。

動手練習

1. `SizeOf` 對 String、Buffer、Package 各代表什麼？`SizeOf ("ACPI")` 是多少？為什麼不是 5？
2. ComplianceRevision 為 1 與 2 時，`Ones` 分別是什麼值？寫出一個在兩種設定下結果不同的 `If` 判斷。
3. `"10"` 經過隱式轉換與 `ToInteger` 各得到多少？`"0x10"` 呢？把四個答案列成表。
4. 用 Package 描述一顆電池的三筆資料：設計容量（Integer）、電壓（Integer）、序號（String），並分別用 legacy 的 `Index + DerefOf` 與 ASL 2.0 的 `[]` 寫法取出電壓。
5. 動手做：輸入本章 types.dsl，用 iasl 編譯、acpiexec 執行；然後把 `NUM0 = "1F"` 改成 `NUM0 = "G1"`，重新執行並解釋輸出為什麼變成 0。

第 5 章

Variables & Constants

型別解決了「資料長什麼樣」，本章解決「資料放在哪裡」。ASL 給你三種容器：用 `Name` 建立、常駐 Namespace 的具名物件；隨 Method 呼叫而生滅的 `Local0` 到 `Local7`；以及呼叫者塞進來的 `Arg0` 到 `Arg6`。它們大致對應 C 語言的全域變數、區域變數與函式參數，但生命週期與重入行為藏著幾個 ASL 特有的地雷。

本章同時把「資料怎麼進出容器」講完：`Store` 與 ASL 2.0 的 `=` 完全等價，`Return` 把值送回呼叫者，再加上 `Zero`、`One`、`Ones` 這批內建常數。學完本章，你就具備讀寫完整 Method 的最小裝備。

本章學習目標

- 說明 Name、Local、Arg 三者的作用域、生命週期與適用場景。
- 寫出 `Store (a, b)` 與 `b = a` 兩種等價寫法，讀懂任一風格的既有程式碼。
- 使用 `Zero`、`One`、`Ones`、`Revision`，正確書寫十進位、十六進位與八進位字面值。
- 說明在 Method 內用 `Name ()` 建立物件為何在重入時觸發 `AE_ALREADY_EXISTS`。
- 畫出一次 Method 呼叫中變數的生命週期，逐行解釋課綱範例 `TEST` 的執行過程。

5.1 資料的三個家：Name、Local、Arg

先看全貌，再逐一深入：

項目	Name	Local0–Local7	Arg0–Arg6
建立者	你，用 `Name ()` 宣告	直譯器自動準備	呼叫者傳入
作用域	Namespace（依宣告位置）	單次 Method 執行	單次 Method 執行
生命週期	表載入起常駐（Method 內宣告者除外）	進入 Method 而生，Return 即滅	同 Local
數量	不限	固定 8 個	最多 7 個
初始值	宣告時給定	未初始化，讀取即錯	呼叫者給定
典型用途	組態、常數、跨呼叫狀態	暫存計算	輸入參數

選擇的原則很單純：值要活過一次呼叫、或要給別的 Method 看，用 Name；只在這次呼叫裡打轉的中間值，用 Local；呼叫者決定的輸入，自然是 Arg。

5.2 Name：常駐 Namespace 的具名物件

`Name (VAL0, 100)` 做兩件事：建立一個值為 100 的 Integer 物件，並在「目前 scope」掛上 `VAL0` 這個名字。寫在 `DefinitionBlock` 或 `Device` 底下時，物件在表載入時就建立，之後任何 Method 都能依第 3 章的名稱搜尋規則找到它——它就是 ASL 的全域變數，開機後一直活著，直到表被卸載。

真正的地雷在「把 `Name` 寫進 Method 裡」。此時物件改為**執行到那一行才建立**、Method 結束時移除，看起來像區域變數，其實名字仍然掛在共享的 Namespace 上：

反面教材：在 Method 內動態建立具名物件

```
Method (BAD0, 0, NotSerialized)     // do NOT copy this pattern
{
    Name (TMP0, 0)                  // created at run time
    TMP0 = 5
    Return (TMP0)
}   // TMP0 is removed when BAD0 exits ... usually
```

注意：Method 內 Name ＋重入 = AE_ALREADY_EXISTS

兩條執行緒同時跑 `BAD0`（或 Method 還沒結束又被事件處理重入）時，第二次執行的 `Name (TMP0, 0)` 會發現名字已存在，直譯器丟出 `AE_ALREADY_EXISTS`，整個 Method 中止——這是 Linux dmesg 裡出鏡率極高的 ACPI 錯誤。iasl 也會提醒你：編譯上面的程式會得到 Remark「Control Method should be made Serialized due to creation of named objects within」。解法有二：把 Method 宣告成 `Serialized`（第 12 章詳講），或者根本別在 Method 裡用 `Name`，改用 Local。

5.3 Local0–Local7：方法內的暫存空間

每次 Method 被呼叫，直譯器就發給這次執行八個全新的暫存槽 `Local0` 到 `Local7`。它們是每次呼叫獨立的一組，所以天生執行緒安全、遞迴安全，不會有上面 `Name` 的重入問題。

兩條規則要記牢。第一，Local 沒有宣告型別，第一次 Store 什麼型別就是什麼型別，之後還能換——它是純粹的「物件掛勾」，Store 進去不做任何隱式轉換（對照 4.6 節）。第二，Local 出生時是 Uninitialized 狀態，**初始化前就拿去讀是致命錯誤**：直譯器回報 `AE_AML_UNINITIALIZED_LOCAL`，Method 直接中止。C 語言讀未初始化變數頂多拿到垃圾值，ASL 直接翻桌，這反而是好事——錯誤藏不住。

5.4 Arg0–Arg6：呼叫者給的輸入

`Method (FOO, 3)` 的第二個參數宣告「我收三個引數」，Method 本體用 `Arg0`、`Arg1`、`Arg2` 取用，上限七個。呼叫端寫 `FOO (1, 2, 3)`，個數對不上時 iasl 在編譯期就會抓到表內的呼叫

錯誤；OS 驅動程式從外部呼叫時給錯個數，則是執行期錯誤。

補充說明：Arg 是傳值還是傳址？

Integer、String 這類單純資料是傳值：在 Method 裡對 `ArgX` 重新 Store 只改到自己的複本。但 Buffer 與 Package 傳進來的是物件參考——用 `Arg0 [i]` 這種 Index 寫法穿透去改，**改到的是呼叫者的原始資料**。第 7 章走訪 Package 的範例會利用這一點；不想被改到就先 `CopyObject` 一份。

5.5 Store、指定運算子與 Return

資料進容器只有一個動詞。ACPI 6.0 世代之前寫 `Store（來源，目標）`；ASL 2.0 引入 C 風格的 `目標 = 來源`。兩者編譯出一模一樣的 AML 位元組碼（StoreOp），純粹是原始碼層的糖衣：

同一件事的兩種寫法（片段）

```
Store (0x1F, Local0)        // legacy style
Local0 = 0x1F               // ASL 2.0 style, identical AML

Store (Local0, VAL0)        // legacy: source first!
VAL0 = Local0               // ASL 2.0: target first
```

注意方向：`Store` 是「來源在前」，`=` 是「目標在前」。反組譯舊機器的 DSDT、翻 BIOS 原始碼樹時滿眼都是 `Store`，新程式碼則建議一律用 `=`——兩種都要能秒讀（第 6 章會把這個對照擴大到所有運算子）。

`Return（x）` 結束 Method 並把 x 複製一份給呼叫者；沒有值可回就寫 `Return（Zero）` 或讓 Method 自然結束。規格書說沒執行到 `Return` 的 Method 回傳值未定義；ACPICA 為了相容老 Windows AML 有「隱式回傳最後運算結果」的怪癖，但那是相容性特技，**永遠明確寫 Return**。

5.6 常數與數字字面值

ASL 內建四個常數物件，各自有專屬的 AML opcode（比一般整數字面值省空間，編譯器會自動幫你選）：

- `Zero`、`One`：就是 0 與 1，常見於舊程式碼與反組譯輸出。
- `Ones`：所有位元為 1。64 位元下是 `0xFFFFFFFFFFFFFFFF`，32 位元下是 `0xFFFFFFFF`（回想第 4 章的 ComplianceRevision 坑）。慣用作「錯誤」「無效」「全選遮罩」的記號。
- `Revision`：直譯器實作的 AML 版本號，極少用到。

數字字面值支援三種進位。十進位直接寫（`100`）、十六進位加 `0x`（`0x64`）——以及一個容易忘記的老朋友：

三種進位的字面值（片段）

```
Name (CNT0, 100)        // decimal          -> 100
Name (CNT1, 0x64)       // hexadecimal      -> 100
Name (CNT2, 0144)       // OCTAL! leading 0 -> also 100
Name (EFLG, Ones)       // idiomatic error / all-bits marker
```

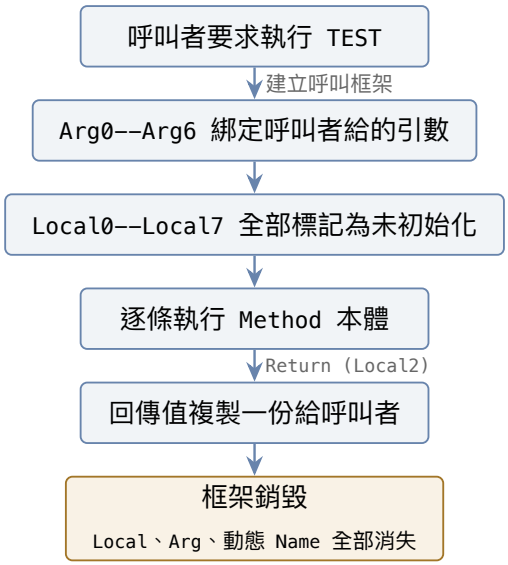
補充說明：前導零是八進位

ASL 文法繼承了 C 的八進位規則：`0144` 不是 144，是 $1 \times 64 + 4 \times 8 + 4 = 100$。手滑在十進位數字前面補零對齊排版，值就悄悄變了——用 iasl 反組譯回來會看到它變成 `0x64`。想排版對齊，請用十六進位加前導零（`0x0064`），那才是安全的。

常數物件是唯讀的：把 `Ones` 或字面值放在 Store 的目標端不會有意義，編譯器會直接指出錯誤。

5.7 生命週期：一次呼叫的一生

把前面幾節串成時間軸——以下一節的課綱範例 `TEST` 為例，一次呼叫從生到死長這樣：



對照之下，掛在 Method 之外的 `Name` 物件完全不參與這個循環：它們在表載入時出生，一直活到表卸載，是跨呼叫、跨 Method 共享的狀態。

5.8 課綱範例逐行拆解

把課綱指定的範例包進完整的 DefinitionBlock，先編譯起來再說：

完整範例：vars.dsl

```
DefinitionBlock ("vars.aml", "SSDT", 2, "ACME", "VARS", 0x00000001)
```

```
{
    Name (VAL0, 100)            // named object, lives in namespace

    Method (TEST, 0)
    {
        Local0 = 10             // first store initializes Local0
        Local1 = 20
        Local2 = Local0 + Local1
        Return (Local2)         // hand 30 (0x1E) back to caller
    }
}
```

逐行拆解：

- `Name (VAL0, 100)`：表載入時建立具名 Integer，值 100（十進位字面值，等於 `0x64`）。它掛在根 scope，與 `TEST` 無關，但任何 Method 都能讀寫它——故意放在這裡對照兩種生命週期。
- `Method (TEST, 0)`：宣告零參數 Method。沒寫 Serialized / NotSerialized 時預設 NotSerialized；因為本體只用 Local，不碰 `Name`，重入也安全。
- `Local0 = 10` 與 `Local1 = 20`：第一次 Store 同時完成「初始化」與「定型別」，兩個 Local 現在都是 Integer。
- `Local2 = Local0 + Local1`：右邊的加法產生匿名的 30，Store 進 `Local2`。三個 Local 都只活在這次呼叫。
- `Return (Local2)`：把 30 複製給呼叫者；隨後框架銷毀，`Local0` 到 `Local2` 蒸發，`VAL0` 依然是 100。

同一個 Method 的 legacy 寫法如下——AML 完全相同，只是外衣不同，拿去反組譯老 DSDT 時你會天天看到這種臉：

同一個 TEST 的 legacy 風格（片段）

```
Method (TSTL, 0)
{
    Store (10, Local0)
    Store (20, Local1)
    Add (Local0, Local1, Local2)
    Return (Local2)
}
```

核心觀念：本章重點回顧

- Name 常駐 Namespace（全域）；Local 是每次呼叫獨立的八個暫存槽；Arg 是呼叫者綁進來的最多七個輸入。
- `Store (a, b)` 與 `b = a` 等價、產生相同 AML；注意兩者來源 / 目標的方向相反。
- Local 未初始化就讀取是致命錯誤 `AE_AML_UNINITIALIZED_LOCAL`；Store 進 Local 不做隱式轉換。
- Method 內用 `Name ()` 動態建物件，重入時觸發 `AE_ALREADY_EXISTS`——用 Serialized（第 12 章）或改用 Local 避開。
- `Zero`、`One`、`Ones` 有專屬 opcode；`Ones` 的實際值取決於 Integer 寬度。
- 字面值三種進位：十進位、`0x` 十六進位、前導零八進位——第三種是陷阱。

動手練習

1. Name、Local、Arg 三者中，哪些在 Method 結束後仍然存在？用「表載入」「呼叫框架」兩個詞各造一句解釋。
2. `100`、`0x64`、`0144`、`064` 四個字面值，哪些相等？各是十進位多少？
3. 為什麼 Local 永遠不會觸發 `AE_ALREADY_EXISTS`，而 Method 內的 `Name` 會？從「名字掛在哪裡」回答。
4. 把 `TSTL` 反向改寫回 ASL 2.0 風格；再把 `TEST` 改成接受兩個引數、回傳兩者之和的 `SUM2`（提示：`Method (SUM2, 2)`）。
5. 動手做：編譯 vars.dsl，用 `acpiexec -b "execute TEST" vars.aml` 執行，確認回傳 `0x1E`；再故意在 `Local0 = 10` 之前插入 `Debug = Local0`，觀察未初始化錯誤長什麼樣子。

第 6 章

Operators

名詞（型別）與容器（變數）都有了，本章補上動詞。ASL 的運算子家族歷史上有兩套外衣：從 ACPI 1.0 用到今天的 legacy 函式式寫法（`Add (a, b, c)`），以及 ACPI 6.0 世代（2015 年前後）隨 ACPICA 引入的 **ASL 2.0**（又稱 ASL+）符號式寫法（`c = a + b`）。兩套語法編譯成完全相同的 AML，純粹是給人看的差別。

為什麼兩種都要學？因為你的日常工作是「讀舊碼、寫新碼」：OEM 與 BIOS 原始碼樹（AMI、Insyde、coreboot）裡的 DSDT 大量是 legacy 風格，舊版 iasl 反組譯的輸出也是；而新版 iasl 反組譯預設輸出 ASL 2.0 風格，新專案也該這樣寫。只會一種，等於半文盲。

本章學習目標

- 對照 legacy 與 ASL 2.0 寫法，任一風格都能讀、能互相改寫。
- 解釋 legacy 運算子的 target 參數，以及 `Divide` 「餘數在前、商在後」的雙 target 慣例。
- 使用算術、位元、邏輯與比較運算子，說明無號 64 位元運算與 `Ones` / `Zero` 回傳慣例。
- 區分位元運算 `Not` 與邏輯運算 `LNot`，避開最常見的條件式 bug。
- 用「讀取—遮罩—位移—組合」四步驟處理暫存器欄位。

6.1 一種位元組碼，兩種外衣

先建立一個關鍵事實：兩種寫法餵給 iasl，產出的 AML 一個位元組都不差。以加法為例，`AddOp` 的編碼是 `0x72`，後面跟著兩個運算元與一個目標的 NameString：

兩種寫法，同一串 AML

```
Add (LOCA, LOCB, RSLT)      ← legacy 寫法
RSLT = LOCA + LOCB          ← ASL 2.0 寫法
          |
          |  iasl（兩種寫法產出完全相同的位元組）
          ↓
72  4C 4F 43 41  4C 4F 43 42  52 53 4C 54
|   |           |           └ Target：RSLT
|   |           └ Operand 2：LOCB
|   └ Operand 1：LOCA
└ AddOp
```

所以「該用哪種」不是技術問題，是溝通問題：跟著你所在程式碼庫的既有風格走，反組譯輸出拿到什麼就要能讀什麼。

6.2 運算子速查總表

這張表是本章的地圖，之後各節逐一展開。建議影印貼在螢幕旁，直到兩種寫法都變成反射動作：

分類	Legacy 寫法	ASL 2.0 寫法	說明
算術	`Add (a, b, t)`	`t = a + b`	加法
	`Subtract (a, b, t)`	`t = a - b`	減法
	`Multiply (a, b, t)`	`t = a * b`	乘法
	`Divide (a, b, r, q)`	`q = a / b`	除法；r 收餘數
	`Mod (a, b, t)`	`t = a % b`	取餘數
	`Increment (a)`	`a++`	遞增 1
	`Decrement (a)`	`a--`	遞減 1
位元	`And (a, b, t)`	`t = a & b`	位元 AND
	`Or (a, b, t)`	`t = a | b`	位元 OR
	`Xor (a, b, t)`	`t = a ^ b`	位元 XOR
	`Not (a, t)`	`t = ~a`	位元反相
	`ShiftLeft (a, n, t)`	`t = a << n`	左移 n 位
	`ShiftRight (a, n, t)`	`t = a >> n`	右移 n 位
邏輯	`LAnd (a, b)`	`a && b`	邏輯 AND
	`LOr (a, b)`	`a || b`	邏輯 OR
	`LNot (a)`	`!a`	邏輯 NOT
比較	`LEqual (a, b)`	`a == b`	等於
	`LNotEqual (a, b)`	`a != b`	不等於
	`LGreater (a, b)`	`a > b`	大於
	`LGreaterEqual (a, b)`	`a >= b`	大於等於
	`LLess (a, b)`	`a < b`	小於
	`LLessEqual (a, b)`	`a <= b`	小於等於
指定	`Store (a, b)`	`b = a`	指定（方向相反 !）
	`Add (a, b, a)` 等	`a += b` 等	複合指定（6.6 節）

補充兩個沒有進表的實用位元運算子：`FindSetLeftBit` 與 `FindSetRightBit` 分別回傳最高、最低設定位元的位置（1 起算，全零回傳 0），掃描狀態暫存器時偶爾派得上用場。

6.3 算術運算子與 target 參數

legacy 運算子的最後一個參數叫 **target**：運算結果除了作為運算式的值傳出去，還會順手存一份到 target。target 可以省略，所以老程式碼有兩種等價的臉：

target 的兩種用法（片段）

```
Add (Arg0, Arg1, Local0)          // result into Local0
Store (Add (Arg0, Arg1), Local0)  // nested: same effect

Local0 = Arg0 + Arg1              // ASL 2.0, same AML idea
```

第二種「巢狀」寫法在反組譯輸出很常見：運算子本身是運算式，可以層層套疊。ASL 2.0 的中置寫法把這些巢狀攤平，可讀性高得多。

`Divide` 是唯一有兩個 target 的運算子，也是經典陷阱製造機：

Divide 的雙 target（片段）

```
Divide (17, 5, Local0, Local1)
// Local0 = 2  <- REMAINDER comes 3rd
// Local1 = 3  <- quotient comes 4th

Local1 = 17 / 5     // ASL 2.0: quotient only
Local0 = 17 % 5     // ASL 2.0: remainder via Mod
```

注意：Divide 的參數順序：餘數在前、商在後

十個人裡有九個直覺以為第三個參數是商。規格書的順序是 `Divide (被除數, 除數, 餘數, 商)`——餘數在前。拿到反組譯碼看到 `Divide` 時先默唸一次這句口訣再讀下去。另外，除數為零會讓直譯器以 `AE_AML_DIVIDE_BY_ZERO` 中止 Method，除法前先檢查除數是韌體的基本禮貌。

所有算術都是無號 64 位元（或 32 位元，看 ComplianceRevision）的模運算：`0 - 1` 會繞回 `Ones`，乘法溢位高位默默丟掉，沒有任何旗標或例外。需要「偵測下溢」時，先比較再相減。

6.4 位元運算子

韌體工程師跟位元的關係，比跟十進位親密得多——暫存器裡每個位元都有自己的名字與脾氣。ASL 的位元運算子與 C 一一對應：`&`、`|`、`^`（XOR）、`~`（反相）、`<<`、`>>`。

四個日常招式，先用片段熱身，下一節的完整範例會組合技：

位元運算四招（片段）

```
Local0 = STAT & 0x01            // test:  isolate bit 0
Local1 = CTRL | 0x80            // set:   force bit 7 on
Local2 = CTRL & ~0x08           // clear: force bit 3 off
Local3 = (STAT >> 4) & 0x0F     // field: extract bits [7:4]
```

第四招是重點：想取出暫存器中間的欄位，先右移到最低位、再用遮罩切掉高位。反過來組欄位就是先遮罩、再左移、最後 OR 起來。這一套「移位＋遮罩」的手感，練到不用想才算過關。

6.5 邏輯與比較運算子

邏輯家族的名字都以 L 開頭（Logical）：`LAnd`、`LOr`、`LNot`、`LEqual`……ASL 2.0 對應 `&&`、`||`、`!`、`==` 等符號。它們的回傳值不是 1 與 0，而是 `Ones` 與 `Zero`：

比較運算子回傳 Ones 或 Zero（片段）

```
Local0 = (5 > 3)        // Local0 = Ones (all bits set), NOT 1
Local1 = (5 < 3)        // Local1 = Zero
```

補充說明：為什麼是 Ones？另外：沒有短路求值

AML 規格把「真」定義成全位元為一，好處是它同時是萬用遮罩。因此**不要拿邏輯結果去比 1**：`(a > b) == 1` 永遠為假，正確寫法是直接 `If (a > b)`。另一個與 C 的重大差異是 ASL 的 `&&` 與 `||` **沒有短路求值**：兩側運算元一定都會被求值，所以「左邊先擋住、右邊才安全」的 C 慣用法在 ASL 行不通，得拆成巢狀 `If`。

比較運算子不只吃 Integer：兩個 String 或兩個 Buffer 也能比（逐位元組），而且第一運算元的型別決定第二運算元要被隱式轉換成什麼——又回到第 4 章的規則表。

注意：Not 還是 LNot？一字之差，天壤之別

`Not` 是位元反相、`LNot` 才是邏輯否定。想寫「旗標為 0 時」而誤用 `If (Not (FLG0))`：當 `FLG0 = 1`，`Not (1)` 是 `0xFFFFFFFFFFFFFFFE`——非零，條件照樣成立，bug 就此誕生且極難用肉眼看出。口訣：**條件式裡只用 L 家族**（或符號 `!`），`Not` / `~` 只拿來翻位元。

6.6 複合指定、遞增與遞減

ASL 2.0 把 C 的複合指定也搬了過來：`+=`、`-=`、`*=`、`/=`、`%=`、`<<=`、`>>=`、`&=`、`|=`、`^=`，以及 `++` 與 `--`。它們全是糖衣：`CNT0 += 4` 展開成 `Add (CNT0, 4, CNT0)`，`Local0++` 就是 `Increment (Local0)`：

複合指定與遞增遞減（片段）

```
CNT0 += 4           // Add (CNT0, 4, CNT0)
CTRL |= 0x01        // Or (CTRL, 0x01, CTRL)
MASK <<= 2          // ShiftLeft (MASK, 2, MASK)
Local0++            // Increment (Local0)
Local1--            // Decrement (Local1)
```

在迴圈計數器與「讀—改—寫」暫存器操作裡，這些糖衣能把三行縮成一行，下一節與第 7 章都會大量使用。

6.7 實戰：暫存器欄位的讀改寫

情境：某裝置有一個狀態暫存器 STAT 與一個控制暫存器 CTRL （本章先用 Name 模擬，第 9 章換成真正的 Field Unit，程式邏輯一行都不用改）。STAT 的位元配置如下：

模擬狀態暫存器 STAT 的位元配置

```
STAT = 0xB5 = 二進位 1011 0101
    bit 7     PWR    電源良好     = 1
    bit 6..4  FANS   風扇檔位     = 011（十進位 3）
    bit 3     ERR    錯誤旗標     = 0
    bit 2..1  RSVD   保留位元     = 10
    bit 0     READY  資料就緒     = 1
```

完整程式：

完整範例：bitops.dsl

```
DefinitionBlock ("bitops.aml", "SSDT", 2, "ACME", "BITOPS", 0x00000001)
{
    Name (STAT, 0xB5)   // simulated status register
    Name (CTRL, 0x00)   // simulated control register

    // Extract FANS field, bits [6:4] of STAT
    Method (GFAN, 0, NotSerialized)
    {
        Local0 = (STAT >> 4) & 0x07     // shift, then mask
        Return (Local0)                 // 0xB5 -> 0x03
    }

    // Read-modify-write: set ENABLE (bit 0), keep the rest
    Method (SETE, 0, NotSerialized)
    {
        Local0 = CTRL                   // read
        Local0 |= 0x01                  // modify: set bit 0
        CTRL = Local0                   // write back
    }

    // Same logic, legacy style (decompiled-DSDT flavor)
    Method (SETL, 0, NotSerialized)
    {
        Store (CTRL, Local0)
```

```
        Or (Local0, 0x01, Local0)
        Store (Local0, CTRL)
    }

    // Clear ERR (bit 3) without touching other bits
    Method (CLRE, 0, NotSerialized)
    {
        CTRL &= ~0x08                   // AND with 1111 0111b
    }

    // Combine two bytes into one 16-bit word
    Method (MK16, 2, NotSerialized)     // Arg0 = low, Arg1 = high
    {
        Return (((Arg1 & 0xFF) << 8) | (Arg0 & 0xFF))
    }
}
```

逐行拆解：

- `GFAN`：`STAT >> 4` 把 FANS 欄位移到最低三位（`0x0B`），再 `& 0x07` 切掉上面殘留的 PWR 位元，得到 `0x03`。「先移位、再遮罩」順序對調也行（`(STAT & 0x70) >> 4`），選一種風格貫徹到底。
- `SETE`：教科書級的讀—改—寫（read-modify-write）。直接 `CTRL = 0x01` 是錯的——會把其他位元全部歸零；先讀出、只改目標位元、再寫回，才不會誤傷鄰居。
- `SETL`：與 `SETE` 逐行對應的 legacy 版。拿反組譯輸出對照著讀，兩種外衣的對應關係就內化了。
- `CLRE`：清位元的慣用式「AND 上反相遮罩」。`~0x08` 是編譯期常數摺疊，iasl 會直接算好塞進 AML。
- `MK16`：低位元組 OR 上左移八位的高位元組，組出 16 位元值——讀感測器常見的「兩個 8-bit 暫存器拼一個讀數」場景。防禦性的 `& 0xFF` 確保呼叫者傳超過一個位元組寬的值時不會污染結果。

核心觀念：本章重點回顧

- legacy 與 ASL 2.0 編譯成相同 AML；讀舊碼要認得 `Add (a, b, t)`，寫新碼建議 `t = a + b`。
- legacy 的 target 是「順手存一份」；`Divide` 雙 target 口訣：餘數在前、商在後。
- 所有算術都是無號模運算：下溢繞回 `Ones`、溢位默默截斷、除零直接中止 Method。
- 邏輯 / 比較回傳 `Ones` / `Zero`；條件式只用 L 家族；ASL 的 `&&`、`||` 沒有短路求值。
- `Not` 翻位元、`LNot` 翻真假，用錯的 bug 肉眼幾乎抓不到。
- 暫存器欄位操作四招：測試、設定、清除、取欄位——以及永遠的讀—改—寫紀律。

動手練習

1. 把 `Store (Add (Local0, Multiply (Local1, 8)), Local2)` 改寫成 ASL 2.0；再把 `Local0 = (A - B) % 4` 改寫成 legacy。
2. `Divide (17, 5, Local0, Local1)` 之後，`Local0` 與 `Local1` 各是多少？
3. `If (Not (FLG0))` 想表達「旗標為 0 時成立」，錯在哪裡？寫出兩種正確寫法（legacy 與 ASL 2.0 各一）。
4. 寫一個 `GETB (val, pos, len)`：回傳 `val` 從第 `pos` 位起、寬 `len` 位的欄位值（提示：`(1 << len) - 1` 是現成的遮罩）。
5. 動手做：編譯 bitops.dsl，先手算 `GFAN` 與 `MK16 (0x34, 0x12)` 的結果，再用 `acpiexec` 執行驗證。

第 7 章

Control Flow

資料、容器、運算子都備齊了，最後一塊拼圖是「句型」。ASL 的流程控制刻意極簡：條件分支只有 `If` / `ElseIf` / `Else`，迴圈只有 `While`，加上 `Break`、`Continue`、`Return` 三個跳躍——沒有 `for`、沒有 `do-while`、沒有 `goto`。這不是偷懶，而是 AML 位元組碼本來就只定義了這幾個控制 opcode（第 2 章的精神：ASL 的每個句型都對得上 AML）。

工具少不代表能力弱：本章示範用 While 搭出 for 慣用式、走訪 Package、拆穿 Switch 語法糖的真面目，比較 Sleep 與 Stall 兩種等待，最後用一個輪詢範例把整個 Module 3 的武器全部用上。

本章學習目標

- 寫出 `If` / `ElseIf` / `Else` 與 `While` / `Break` / `Continue` 的完整結構。
- 說明條件式的真假值規則：非零即真、比較回傳 `Ones` / `Zero`。
- 用 While + 計數器實作 for 慣用式，走訪 Package 與 Buffer。
- 解釋 Switch 展開成 `While (One)` 的原理，以及它為何要求 Serialized。
- 區分 `Sleep` 與 `Stall` 的單位、行為與適用場景。
- 綜合條件、迴圈、位元運算寫出帶逾時的輪詢 Method。

7.1 If、ElseIf 與 Else

直接看課綱指定的範例——一個門檻判斷器：

課綱範例：門檻判斷（片段）

```
Method (TEST, 1)
{
    If (Arg0 > 100)
    {
        Return (1)
    }
    Else
    {
        Return (0)
    }
}
```

逐行拆解：

- `Method (TEST, 1)`：收一個引數。條件式 `Arg0 > 100` 是 ASL 2.0 寫法，legacy 等價於 `If (LGreater (Arg0, 100))`——反組譯舊表時常見。
- `If (...) {...}`：條件為真走這塊。注意 ASL 的 If 條件一定要括號，大括號就算只有一行也不能省。
- 兩個分支都以 `Return` 收尾，Method 不會「掉出去」；拿掉 `Else` 的話，`Arg0 <= 100` 時走完沒有 Return，回傳值未定義。

多路分支用 `ElseIf` 串接。它其實是 `Else { If {...} }` 的糖衣（AML 只有 IfOp 與 ElseOp 兩個 opcode），語意與 C 相同——由上而下，第一個成立的分支勝出：

ElseIf 階梯：溫度分級（片段）

```
Method (GRAD, 1)            // Arg0: temperature in Celsius
{
    If (Arg0 >= 90)     { Return (3) }      // critical
    ElseIf (Arg0 >= 60) { Return (2) }      // hot
    ElseIf (Arg0 >= 45) { Return (1) }      // warm
    Else                { Return (0) }      // normal
}
```

7.2 條件式的真假值

規則只有一條：**條件運算式最後會被當成 Integer 看，非零即真、零即假**。配合第 6 章的知識，就能解讀所有慣用形：

- `If (Arg0 > 100)`：比較運算子回傳 `Ones`（真）或 `Zero`（假），直接可用。
- `If (STA0 & 0x01)`：位元測試慣用式——遮罩後非零代表該位元為 1，不必再寫 `!= 0`。
- 千萬別寫 `If (Not (FLG0))`——回去複習第 6 章的 warnbox，這裡再踩一次就說不過去了。

7.3 While、Break 與 Continue

`While (條件) {...}` 是 ASL 唯一的迴圈。三個角色的關係一張圖講完：

While 迴圈的控制流

```
While (條件)
├── 條件為真 → 執行迴圈本體
│    ├── 遇到 Break    → 立刻離開迴圈
│    ├── 遇到 Continue → 跳回條件重新判斷
│    └── 本體跑完      → 跳回條件重新判斷
```

└─ 條件為假 → 離開迴圈，執行 While 後面的敘述

沒有 for，但「初始化—測試—更新」三件套用 While 一樣寫得出來，這是 ASL 最重要的迴圈慣用式：

for 迴圈的 ASL 慣用式（片段）

```
// C:  for (i = 0; i < 10; i++) { body }
Local0 = 0                  // init
While (Local0 < 10)         // test
{
    // ... body ...
    Local0++                // update — easy to forget!
}
```

Break 只跳出最內層迴圈；Continue 跳回條件判斷。用 Continue 時要特別小心計數器的位置：

Continue 的正確姿勢：先遞增再跳（片段）

```
// Sum only the non-zero elements of PKG0
Local0 = 0                          // index
Local1 = 0                          // running sum
While (Local0 < SizeOf (PKG0))
{
    Local2 = DerefOf (PKG0 [Local0])
    Local0++                        // increment BEFORE Continue
    If (Local2 == 0)
    {
        Continue                    // skip zero entries
    }
    Local1 += Local2
}
```

注意：Continue 前忘了遞增 = 死迴圈

把 Local0++ 放在迴圈尾端是 C 工程師的肌肉記憶——但只要中間有 Continue，遞增就被跳過，同一個元素永遠判斷不完。上面的範例把遞增提前到取值之後，就是為了讓每條路徑都會前進。寫 While 前先自問：「每一條路徑都會讓條件逼近假嗎？」

補充說明：直譯器的迴圈保險絲

真的寫出死迴圈也不會讓整台機器永遠卡死：ACPICA 內建 While 迴圈逾時（預設約 30 秒，版本可能不同），超過就以 AE_AML_LOOP_TIMEOUT 中止 Method 並在 kernel log 留案底。但被保險絲救回來不值得驕傲——輪詢迴圈永遠要自帶重試上限，章末範例示範標準寫法。

7.4 實戰：走訪 Package

第 4 章說過 Package 元素要用 reference 取值，第 5 章給了 SizeOf，現在全部串起來。情境：風扇控制表 TRIP 存放遞增的溫度觸發點，給定溫度，找出對應的風扇檔位：

完整範例：fanctl.dsl

```
DefinitionBlock ("fanctl.aml", "SSDT", 2, "ACME", "FANCTL", 0x00000001)
{
    // Fan trip points in Celsius, ascending order
    Name (TRIP, Package () {30, 45, 60, 75, 90})

    // Map a temperature (Arg0) to a fan level 0..5
    Method (FLVL, 1, NotSerialized)
    {
        Local0 = 0                      // index = fan level
        While (Local0 < SizeOf (TRIP))
        {
            // stop at the first trip point above Arg0
            If (Arg0 < DerefOf (TRIP [Local0]))
            {
                Break
            }
            Local0++
        }
        Return (Local0)
    }
}
```

逐行拆解：

- SizeOf (TRIP) 回傳元素個數 5，迴圈條件因此不必寫死長度——之後往 TRIP 加觸發點，FLVL 一行都不用改。
- DerefOf (TRIP [Local0])：Index 產生 reference、DerefOf 取值，一氣呵成。忘了 DerefOf 的話，比較運算子拿到的是 reference，行為不是你要的。

- Break 在第一個「高於目前溫度」的觸發點停下，此時 Local0 恰好是跨過的觸發點數，直接當檔位用：FLVL (25) 回傳 0、FLVL (47) 回傳 2、FLVL (95) 掃完全表回傳 5。
- 邊界行為：溫度剛好等於觸發點時（< 不成立）視為升檔；想改成「等於不升檔」，換 <= 即可。

7.5 Switch、Case 與 Default

多路等值分支寫成 ElseIf 階梯很囉唆，ASL 從 ACPI 2.0 起提供 Switch。它常見於 _DSM 這類「依功能編號分派」的 Method：

Switch 分派（片段）

```
Method (DSPT, 1, Serialized)        // Serialized: see below!
{
    Switch (ToInteger (Arg0))
    {
        Case (0)                  { Return (Buffer () {0x3F}) }
        Case (Package () {1, 2}) { Return (One) }  // multi-match
        Default                   { Return (Zero) }
    }
}
```

三個語法重點：Switch 運算元慣例上包一層 ToInteger 確保與 Case 值型別一致；Case (Package () {1, 2}) 一次匹配多個值；沒有 C 的 fall-through，每個 Case 執行完自動結束，不需要（也沒有）break 敘述。

但 Switch 其實是語法糖：AML 根本沒有 SwitchOp。iasl 把它展開成「只跑一圈的 While」加上一個隱藏的暫存物件：

Switch 的概念展開（示意片段）

```
While (One)                     // conceptual expansion
{
    Name (_T_0, 0)              // hidden temp, created by iasl
    _T_0 = ToInteger (Arg0)
    If (_T_0 == 0)                          { Return (Buffer () {0x3F}) }
    ElseIf ((_T_0 == 1) || (_T_0 == 2))     { Return (One) }
    Else                                    { Return (Zero) }
    Break                       // never loops twice
}
```

看到 Name (_T_0, ...) 想起什麼了嗎？第 5 章的地雷——在 Method 內動態建立具名物件。這就是含 Switch 的 Method 必須 Serialized 的原因：寫成 NotSerialized 時，iasl 會給 Remark「Control Method marked Serialized」並自作主張幫你改掉。

補充說明：反組譯時的 Switch

AML 裡沒有 Switch，反組譯器看到的是 While (One) 展開形。新版 iasl 認得這個模式（連同 `_T_0`）會還原成 Switch；模式稍有走樣或用舊版工具，你就會直接面對展開形——認得它，反組譯輸出才不會讀成天書。

7.6 Sleep 與 Stall：兩種等待

硬體常常要人等：EC 需要幾毫秒回應、電源軌拉起要時間。ASL 提供兩個等待原語，個性南轅北轍：

比較項目	`Sleep (ms)`	`Stall (us)`
單位	毫秒	微秒
行為	讓出 CPU，執行緒可被排程	busy-wait，整顆 CPU 空轉等你
時間保證	至少 ms 毫秒（可能更久）	約 us 微秒（實作相依）
規範限制	無上限	**單次不得超過 100 微秒**
適用場景	等 EC、裝置上電（毫秒級）	極短硬體時序（微秒級）

注意：Stall 不是拿來睡覺的

`Stall (5000)` 有兩宗罪：一是違反規格的 100 微秒上限；二是 AML 常在不可排程的環境執行，busy-wait 五毫秒等於讓整顆 CPU 抱著暫存器發呆，Windows 的驅動驗證器會直接抓包。要等毫秒級，用 `Sleep`；要等更久，用「`Sleep` + 重試上限」的輪詢迴圈——正是下一節的主角。

順帶認識 `Timer`：回傳單調遞增的 100 奈秒刻度計數，量測實際等待時間時可當碼表用（Module 7 的電源時序除錯會再遇到）。

7.7 綜合範例：輪詢一個裝置

期末考：把 Module 3 的所有武器裝進一個 Method。情境是韌體日常——向裝置要資料前，得先等它的 READY 位元亮起，而且不能無限等：

完整範例：poll.dsl

```
DefinitionBlock ("poll.aml", "SSDT", 2, "ACME", "POLL", 0x00000001)
{
    // Simulated registers (chapter 9 swaps in real Field Units)
    Name (DSTA, 0x81)       // status: bit7 = power, bit0 = ready
    Name (DDAT, 0x5A)       // data register

    // Poll until READY (bit 0) or timeout.
    // Return: masked data on success, Ones on timeout.
```

```
    Method (RDAT, 0, NotSerialized)
    {
        Local0 = 20                 // retry budget: 20 tries
        While (Local0 > 0)
        {
            If (DSTA & 0x01)        // ready bit set?
            {
                Local1 = DDAT & 0x7F    // strip parity bit
                Return (Local1)
            }
            Sleep (5)               // yield CPU for 5 ms
            Local0--
        }
        Return (Ones)               // timeout marker
    }
}
```

逐行拆解：

- `Local0 = 20` 是重試預算：搭配每輪 `Sleep (5)`，總逾時約 100 毫秒。輪詢迴圈的三要素——條件、讓出 CPU、遞減預算——一個都不能少，少一個就是潛在死迴圈。
- `If (DSTA & 0x01)`：位元測試慣用式當迴圈出口；就緒後 `DDAT & 0x7F` 剝掉同位位元，只留有效資料。成功路徑從迴圈**中間**直接 `Return`——ASL 允許隨處返回，比設旗標再 Break 乾淨。
- `Return (Ones)`：用第 5 章的慣用常數當逾時記號，呼叫者拿到 `Ones` 就知道別把它當資料用。
- 自己驗證兩條路徑：本例 `DSTA = 0x81`，acpiexec 執行 `RDAT` 立即回傳 `0x5A`；把 `DSTA` 改成 `0x80`（READY 熄滅）再跑，約 100 毫秒後回傳 `Ones`。

讀完 Module 3，你已經握有 ASL 的全部「語言核心」：型別、變數、運算子、流程控制。從 Module 4 起，戰場轉向真正的硬體——用 Device 描述裝置、用 OperationRegion 與 Field 接管暫存器，到時你會發現本模組的每一個慣用式都在現場等你。

核心觀念：本章重點回顧

- 分支：`If` / `ElseIf` / `Else`；ElseIf 是糖衣，AML 只有 IfOp 與 ElseOp。
- 真假值一條規則：條件轉成 Integer，非零即真；位元測試 `If (X & mask)` 是慣用式。
- While 是唯一迴圈；for 用「初始化—測試—更新」慣用式；Break 只出最內層，Continue 前記得遞增計數器。
- 走訪 Package：`SizeOf` 定終點、`DerefOf (P [i])` 取值。
- Switch 是 `While (One)` + 隱藏 `_T_0` 的語法糖，因此 Method 必須 Serialized。

- ▶ `Sleep` 讓出 CPU（毫秒）；`Stall` 忙等（微秒、單次上限 100）；長等待用 Sleep + 重試預算輪詢。

動手練習

1. ASL 為什麼不需要 `for`？寫出 `for (i = 9; i > 0; i--)` 的 While 對應片段。
2. 把本章開頭的 `TEST` 改寫成全 legacy 風格（`LGreater`、`Store`），並確認兩版對 `TEST (100)` 都回傳 0——邊界值算哪一邊？
3. 寫 `CSUM (buf)`：用 While 走訪 Buffer 的每個位元組加總，回傳總和的低 8 位元（提示：`DerefOf (Arg0 [i])` 對 Buffer 也有效）。
4. `Stall (5000)` 犯了哪兩條戒律？用 Sleep 與 While 改寫成等待 5 毫秒的合規寫法。
5. 動手做：編譯 fanctl.dsl，先在紙上追蹤 `FLVL (25)`、`FLVL (47)`、`FLVL (95)` 每輪迴圈的 `Local0` 值，再用 `acpiexec -b "execute FLVL 0x2F" fanctl.aml` 等指令驗證你的答案。

Module 4

Device 與硬體描述

難度：★★★☆☆

前三個模組打好了語法與 Namespace 的地基，本模組把這些零件組裝成 OS 真正關心的單位——Device。你將學會替裝置發身分證（`_HID`、`_CID`）、回報健康狀態（`_STA`）、申報硬體資源（`_CRS`），並完整走一遍 OS 從列舉、比對 ID 到綁定 Driver 的旅程。

第 8 章

Device Object

第 1 章說過，ACPI 存在的一大理由是替「不可列舉」的硬體向 OS 自我介紹，而自我介紹的載體，就是本章的主角——**Device** 物件：一個 Device 節點搭配一組名稱固定的標準子物件，就構成 OS 眼中「一個完整的裝置」。本章依 OS 列舉的順序講解它們：身分（_HID、_CID、_ADR、_UID）、狀態（_STA）、資源（_CRS、_PRS、_SRS）、屬性與初始化（_DSD、_INI）。

本章學習目標

- 寫出含 _HID、_UID、_STA 的最小 Device，並畫出 OS 列舉、比對 ID、綁 Driver 的流程圖。
- 說明 _HID 的 EisaId 與字串兩種格式及命名規則。
- 區分 _HID 與 _ADR 的分工，寫出 PCI 裝置的 _ADR 編碼。
- 逐位元解讀 _STA，診斷 0x0F、0x0B、0x00 的差異。
- 用 ResourceTemplate 寫出 _CRS，並說明 _PRS / _SRS 動態資源三件套。
- 說明 _DSD 裝置屬性與 _INI 的執行時機。

8.1 Device 物件：向 OS 註冊一個裝置節點

從 OS 的角度看，Device 關鍵字做的事只有一件：在 **ACPI Namespace** 掛出一個「裝置節點」，等著被列舉。節點本身沒有任何魔法，真正決定這個裝置「是誰、能不能用、用什麼資源」的，是掛在它底下、名稱以底線開頭的標準子物件。先看課綱指定的最小範例：

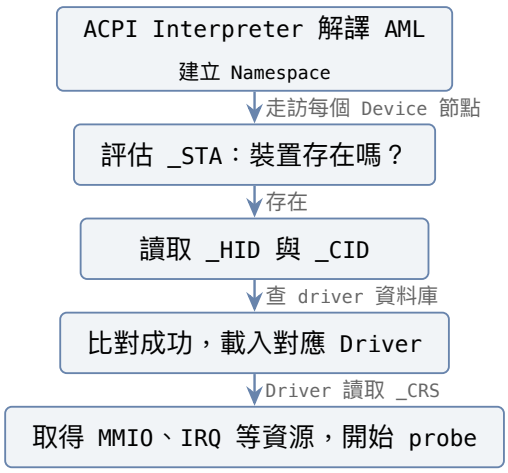
最小可用的 Device 宣告

```
Device (DEV0)
{
    Name (_HID, "ACME0001")     // who am I: vendor + product
    Name (_UID, 0)              // instance number 0
    Method (_STA, 0)            // am I present and working?
    {
        Return (0x0F)           // present + enabled + OK
    }
}
```

逐行拆解：

- `Device (DEV0)`：在目前 Scope 底下建立裝置節點，命名規則同一般物件（至多 4 碼、字母或底線開頭）。
- `Name (_HID, "ACME0001")`：硬體識別碼——裝置的身分證字號，OS 拿它比對 driver 資料庫。
- `Name (_UID, 0)`：實例編號，用來區分同 `_HID` 的多顆裝置。
- `Method (_STA, 0)`：回答「我在嗎」；`0x0F` 表示存在、已啟用、顯示於 UI、運作正常。

OS 開機列舉裝置的整條流水線如下——這張圖是本章的骨架：



寫 ASL 描述裝置，本質上是回答 OS 的三個問題：你是誰（`_HID`）、你好嗎（`_STA`）、你用什麼（`_CRS`）——接下來各節依序拆解每一環。

8.2 _HID：裝置的主要身分證

`_HID`（Hardware ID）是 OSPM 列舉裝置時的首要依據：OS 拿它去比對 driver 資料庫（Windows 的 INF、Linux 的 `acpi_device_id` 表），決定這個節點交給誰驅動。它有兩種合法格式：

_HID 的兩種格式

```
Name (_HID, EisaId ("PNP0C0D"))   // EISA form: 3 letters + 4 hex
Name (_HID, "ACME0001")           // ACPI form: 4 chars + 4 hex
```

兩種格式的規則：**PNP ID** 是「3 個大寫字母的 vendor 代碼 + 4 個十六進位數字」，可包在 `EisaId()` 裡，也可直接寫成 7 碼字串（如 `"INT34C5"`）；**ACPI ID** 是「4 碼 vendor（大寫字母或數字）+ 4 碼十六進位 product」，一律寫成 8 碼字串。`EisaId()` 是 PnP 時代的遺產：把 7 碼 ID 壓成 32-bit 整數放進 AML（`EisaId ("PNP0C0D")` 即整數 `0x0D0CD041`）。課綱範例 `"ACME0001"` 就是「向 UEFI Forum 註冊的 vendor ACME、其 0001 號產品」。

什麼時候必須有 `_HID`？一句話：**凡是要 OSPM 自己列舉的裝置（platform device），一定要有 `_HID`**；掛在可自我列舉匯流排上的裝置改用 `_ADR`（見稍後一節）。沒有身分的 Device 也合法，但 OS 只把它當容器，不會綁 driver。

補充說明：Vendor ID 是要註冊的

PNP 與 ACPI 的 vendor 代碼由 UEFI Forum 統一發放（uefi.org 可查清單）：`PNP` 開頭保留給歷史裝置類別、`ACPI` 開頭保留給規格定義的裝置。出貨韌體請用公司註冊的代碼；本書範例一律用虛構的 `ACME`。

實務上有一批「熟面孔」ID 值得先混個臉熟：`PNP0C0D`（上蓋開關）、`PNP0C0C`（電源鍵）、`PNP0C09`（Embedded Controller）、`PNP0303`（PS/2 鍵盤）、`PNP0A08`（PCIe root bridge）、`ACPI0003`（AC adapter）——本書後續章節會反覆遇到它們。

8.3 _CID 與 _UID：相容身分與實例編號

`_CID`（Compatible ID）回答的問題是：「如果找不到我的專屬 driver，我還相容於誰？」OS 的比對順序是先拿 `_HID` 找專屬 driver，找不到才依序嘗試 `_CID` 清單。最經典的例子是鍵盤：

DSDT 片段：鍵盤控制器的雙重身分

```
Device (KBD0)
{
    Name (_HID, "ACME0010")             // vendor-specific ID
    Name (_CID, EisaId ("PNP0303"))     // generic AT keyboard

    // _CID may also be a priority-ordered list:
    //   Name (_CID, Package (2) { "PNP0303", "PNP030B" })
}
```

廠商若有客製 driver（例如支援特殊快捷鍵），OS 用 `"ACME0010"` 比中它；沒有就退而求其次，用 `PNP0303` 綁內建的標準鍵盤 driver，鍵盤至少能打字。如註解所示，`_CID` 可以是單一 ID，也可以是候選清單（Package），順序由「最精確」排到「最通用」。現代版的同一招是 I2C 觸控板：vendor 專屬 `_HID` 加 `_CID "PNP0C50"`（HID over I2C），通用 HID driver 永遠接得住。`_UID`（Unique ID）則解決「同型號、多實例」：多個 Device 共用同一個 `_HID` 時，OS 靠它分辨誰是誰。

DSDT 片段：同 _HID 的兩個 UART

```
Device (UAR1) { Name (_HID, EisaId ("PNP0501"))  Name (_UID, 1) }
Device (UAR2) { Name (_HID, EisaId ("PNP0501"))  Name (_UID, 2) }
```

`_UID` 可以是整數或字串，規則有二：同 `_HID` 之間必須唯一、跨開機必須穩定——OS 拿它記住每個實例的設定，編號亂跳，設定就漂移到別顆裝置上。

8.4 _ADR：可列舉匯流排上的位址

PCI、USB 這類匯流排有自我列舉能力：裝置插上去，匯流排協定就能發現它、讀出它的 ID。這種裝置不需要 `_HID`——OS 已經知道硬體存在，ACPI 節點的任務只剩「對號入座」：用 **`_ADR`** 宣告「我描述的是匯流排上的哪個位址」，好讓 OS 把這個節點的 ACPI method（電源控制、喚醒能力等）附加到正確的硬體裝置上。

DSDT 片段：PCI 裝置用 _ADR 對號入座

```
Scope (\_SB.PCI0)
{
    Device (GFX0)
    {
        Name (_ADR, 0x00020000)    // device 0x02, function 0
    }
}
```

PCI 的 `_ADR` 編碼是固定公式：`(Device << 16) | Function`——device 0x1F 的 function 3 就編成 `0x001F0003`；`0xFFFF` 是萬用字元（該 device 的所有 function）。其他匯流排各有規則：USB 用 root hub port 編號、SATA 用 port 加 port multiplier 的組合、SD 用 slot 編號。

分工原則一句話：**OS 自己找得到的裝置用 `_ADR` 對號，找不到的用 `_HID` 自我介紹**。規格要求每個 Device 至少具備其中之一；兩者都放是舊時代遺風，新設計應避免。

8.5 _STA：裝置的健康申報

`_STA`（Status）是 OS 列舉時的第一道關卡：先問「你在嗎」，再決定要不要繼續讀 `_HID`、載 driver。之後裝置狀態改變（例如韌體發出 `Notify`）時，OS 也會重新評估它。回傳值是一個位元遮罩：

位元	名稱	意義
bit 0	Present	裝置實體存在
bit 1	Enabled	已啟用且正在 decode 資源
bit 2	Show in UI	應顯示在使用者介面中
bit 3	Functioning	運作正常（清 0 表示故障）
bit 4	Battery Present	電池槽裝置專用：電池在位

位元不能亂組：bit 0 清零（不存在）時 bit 1 必須跟著清零——不存在的裝置不可能在 decode 資源。三個必背的常見值：**0x0F**（bit 0–3 全設）是正常工作的裝置，OS 列舉、載 driver、顯示於 UI；**0x0B**（缺 bit 2）是「正常運作但別出現在 UI」，給使用者不該看到的內部裝置；**0x00** 是「不存在」，OS 直接跳過、不載 driver。電池槽裝置再加 bit 4，正常值成為 0x1F。

沒有 `_STA` 的裝置，OS 一律視同回傳 `0x0F`。所以「永遠存在」的裝置可以不寫；要寫就寫對（殺傷力見警告框）。它真正的價值在動態場景——同一份 DSDT 服務多個硬體版本時拿它當

開關：Method（_STA）裡讀一個對應硬體跳線的 Field（第 10 章的主角，例如 SKUV），高階 SKU 回 0x0F、其餘回 Zero——同一顆 AML 燒進三個機種，高階款看得到感測器，低階款彷彿它不存在。

注意：_STA 答錯話，裝置就人間蒸發

- 回傳 **0x00**：OS 當裝置不存在——Windows 裝置管理員看不到、Linux 連 platform device 都不建。常見事故：_STA 讀 EC 欄位，但 EC region 尚未就緒（第 9 章 _REG），讀到 0 就回 Zero，觸控板從此消失。
- **漏掉 bit 3 (Functioning)**：裝置看得到卻標成故障，Windows 顯示黃色驚嘆號、driver 不啟動；Linux 卻可能照常 probe——兩邊行為不一致，更難追。
- 非法組合：bit 0 清零卻設 bit 1，行為未定義，別賭任何 OS 的容錯。

排錯守則：懷疑裝置消失，第一步就是確認 _STA 的回傳值。

最後一個規格認可的特例：回傳 0x08（只設 bit 3）表示「本體不存在但功能正常」——OS 不執行 _INI、不載 driver，但仍會繼續列舉子節點；想讓橋接器隱形、卻保留底下子樹時用得到。

8.6 _CRS、_PRS 與 _SRS：資源申報三件套

_CRS（Current Resource Settings）回傳一個 Buffer，內容是一串二進位的 resource descriptor，描述裝置目前使用的 MMIO、IO port、中斷、DMA 等資源。手刻二進位太苦，ASL 提供 **ResourceTemplate** 巨集，在編譯期把可讀的巨集展開成 descriptor bytes。先看常用巨集的總覽：

資源巨集總覽（教學示範，實務依裝置取捨）

```
Name (_CRS, ResourceTemplate ()
{
    Memory32Fixed (ReadWrite, 0xFED90000, 0x1000)   // 4 KB MMIO
    IO (Decode16, 0x0300, 0x0300, 0x01, 0x10)       // 16 ports
    Interrupt (ResourceConsumer, Level, ActiveLow,
               Exclusive) { 24 }                    // GSI 24
    IRQNoFlags () { 5 }                             // ISA IRQ 5
    DMA (Compatibility, NotBusMaster, Transfer8) { 1 }
})
```

逐一拆解：

- Memory32Fixed（屬性，基底，長度）：位址固定的 32-bit MMIO；可重定位或 64-bit 改用 DWordMemory / QWordMemory。
- IO（解碼，最小，最大，對齊，長度）：legacy IO port；固定位址寫 min = max。

- `Interrupt (用途, 觸發, 極性, 共享) {n}`：編號是 **GSI**（Global System Interrupt），可超過 15；喚醒源用 `ExclusiveAndWake`（Module 7）。
- `IRQNoFlags () {n}` / `IRQ (...) {n}`：legacy PIC 中斷（0–15），只剩 SuperIO、PS/2 在用。
- `DMA (型別, 主控, 寬度) {通道}`：ISA DMA 通道（0–7）；現代裝置自帶 bus-master DMA，用不到它。

資源固定的裝置給 `_CRS` 就夠了。資源可調的裝置（歷史上是 ISA PnP 卡，今日剩 SuperIO 一族）要湊齊動態三件套：

- `_PRS`（Possible）：列出可接受的資源組合——在 ResourceTemplate 裡用 `StartDependentFn` 把「IO 0x3F8 + IRQ 4」「IO 0x2F8 + IRQ 3」一組組包起來。
- `_SRS`（Set）：OS 挑一組包成 buffer 當 Arg0 呼叫它；method 用 `CreateWordField` 挖出選中的位址與中斷，寫進 SuperIO 組態暫存器。
- `_CRS`（Current）：設定生效後回報新值——三件套形成「查詢、改定、確認」的循環。

配套的 `_DIS` 讓 OS 重新配置前先叫裝置停止 decode。現代平台多數裝置資源固定，你會寫到的幾乎只有 `_CRS`；但讀懂三件套，看到老平台的 DSDT 才不會迷路。

8.7 _DSD 與 _INI：裝置屬性與初始化

`_CRS` 能描述的東西有限：它說得出「哪段 MMIO、哪個 IRQ」，說不出「這顆感測器的取樣間隔是 250 ms」。**_DSD**（Device Specific Data）補上這塊：以 UUID 標記格式、用 key-value 形式傳遞任意裝置屬性：

DSDT 片段：用 _DSD 傳遞裝置屬性

```
Name (_DSD, Package ()
{
    ToUUID ("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
    Package ()
    {
        Package (2) { "compatible", "acme,wdt-100" },
        Package (2) { "timeout-sec", 30 }
    }
})
```

開頭的 UUID 是「Device Properties」格式的固定識別碼，後面每項是一組 `{名稱, 值}`。屬性命名慣例沿用 Linux Device Tree 的 binding；driver 端用 `device_property_read_u32()` 這類 API 直接讀取——第 25 章會寫出完整的 driver 對接。注意 ACPI 不驗證 key 的拼字與型別，韌體與 driver 得自己講好。

`_INI` 則是裝置的「開機初始化鉤子」，執行時機明確：OSPM 建立 Namespace、列舉裝置時執行，早於 **driver** 載入；執行條件是 `_STA` 回報 present（或裝置沒有 `_STA`；回傳 0x08 的特例不執行，但子樹照走）；順序由上而下，父節點先於子節點。典型用途是一次性的硬體預設值，例如 `Method (_INI, 0) { CTLR = Zero }` 在 driver 接手前先停掉 watchdog（`CTLR` 是對應控制暫存器的 Field）。切記 `_INI` 只跑一次，反覆執行的邏輯屬於 `_PS0` 或 driver。

8.8 綜合範例：一個完整的平台裝置

把本章所有元件組裝起來：一顆虛構的 ACME watchdog，MMIO 在 0xFED90000、中斷走 GSI 24。這是一份完整可編譯的 SSDT：

wdt0.dsl：完整可編譯的 SSDT

```
DefinitionBlock ("wdt0.aml", "SSDT", 2, "ACME", "WDTTABLE", 1)
{
    Scope (\_SB)
    {
        Device (WDT0)
        {
            Name (_HID, "ACME0001")     // ACME watchdog
            Name (_UID, 0)              // instance 0
            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)           // present and working
            }
            Name (_CRS, ResourceTemplate ()
            {
                Memory32Fixed (ReadWrite,
                    0xFED90000, 0x1000)     // 4 KB registers
                Interrupt (ResourceConsumer, Level,
                    ActiveLow, Exclusive) { 24 }    // GSI 24
            })
        }
    }
}
```

存成 `wdt0.dsl`、以 `iasl wdt0.dsl` 編譯即得 `wdt0.aml`（0 Errors、0 Warnings）。載入後，它在 Namespace 中的位置如下：

WDT0 在 Namespace 中的位置

```
\
└── _SB_
    ├── PCI0 ...                ← 平台原有的其他裝置
```

```
    └── WDT0                        ← 本例宣告的裝置
        ├── _HID = "ACME0001"       ← 身分：driver 比對用
        ├── _UID = 0                ← 第 0 號實例
        ├── _STA → 0x0F             ← 健康申報：一切正常
        └── _CRS → Buffer           ← MMIO 4 KB + GSI 24
```

逐行拆解：

- `Scope (\_SB)`：平台裝置一律掛在 `\_SB`（System Bus）底下，這是 OS 列舉的起點；SSDT 形式（Module 2）方便獨立編譯與載入。
- `_HID "ACME0001"`：Linux driver 會在 `acpi_device_id` 表裡放同一字串認領裝置（第 25 章實作）；`_UID 0` 讓日後加第二顆時不動既有描述。
- `_STA` 固定回 `0x0F`：裝置焊死在板上、永遠存在；省略 `_STA` 效果相同。
- `_CRS`：4 KB MMIO 加 level-triggered、active-low 的 GSI 24；driver probe 時據此映射位址、註冊中斷。

到這裡，OS 看得到裝置、找得到 driver、拿得到資源；唯一還做不到的是讓 **AML 自己讀寫 0xFED90000 的暫存器**——那正是下一個模組 OperationRegion 與 Field 的主場。

核心觀念：本章重點回顧

- Device 節點加標準子物件，回答 OS 三問：你是誰（`_HID`）、你好嗎（`_STA`）、你用什麼（`_CRS`）。
- `_HID` 兩種格式：EisaId 的 3+4 碼、字串的 4+4 碼；`_CID` 是相容身分備援；可列舉匯流排改用 `_ADR`（PCI：`(Device << 16) | Function`）。
- `_STA` 位元申報狀態：0x0F 正常、0x0B 隱藏、0x00 不存在；沒寫視同 0x0F。
- `_CRS` 申報現用資源，ResourceTemplate 巨集生成 descriptor；可調資源加 `_PRS` / `_SRS`。
- `_DSD` 傳遞 key-value 屬性給 driver；`_INI` 在列舉期做一次性初始化。

動手練習

1. `_HID`、`_UID`、`_STA` 分別回答 OS 的什麼問題？缺了其中一個各會發生什麼事？
2. 寫出一個鍵盤裝置：vendor 專屬 ID `"ACME0010"`、相容於 `PNP0303`，並說明 OS 的比對順序。
3. `_STA` 回傳 0x0B，裝置管理員會看到什麼？0x08 呢？0x07 是合法值嗎？為什麼？
4. 感測器暫存器在 0xFED92000（4 KB）、中斷走 GSI 25（level、active-low）：寫出它的 `_CRS`。
5. 動手做：把綜合範例改成雙實例（WDT0 / WDT1，`_UID` 0 / 1，MMIO 錯開 4 KB），以 `iasl` 編譯後用 `iasl -d` 反組譯，觀察 `_CRS` 的 bytes。

Module 5

OperationRegion 與 Field

難度：★★★☆☆

Module 4 教會你向 OS 申報「我用哪些資源」，但申報不等於存取——AML 真正讀寫硬體暫存器的機制，是 OperationRegion 與 Field。本模組先拆解 OperationRegion 的九種定址空間，再學會用 Field 把位址窗口切成具名暫存器，最後把整套機制實戰到 GPIO、I2C、SPI、UART 與 PCI，寫出完整的匯流排裝置描述。

第 9 章

OperationRegion

第 8 章的 _CRS 只是「申報」：告訴 OS 我的暫存器在哪、佔多大，請 OS 別把資源分給別人。但 AML 自己也常要動手碰硬體——_STA 要讀狀態暫存器、_INI 要寫預設值、EC 的電池電量要一個 byte 一個 byte 讀出來。ACPI 給 AML 的存取機制分成兩層：本章的 **OperationRegion** 負責「在某個位址空間開一扇窗」，第 10 章的 Field 再把窗內切成一顆顆具名暫存器。

先窗後刀。本章專心把「窗」講透：四個參數、九種位址空間、「宣告不碰硬體」的 lazy 語意，以及 _REG 就緒通知。

本章學習目標

- 說明 OperationRegion 四個參數的意義，寫出 SystemIO 與 SystemMemory 的 region 宣告。
- 列出九種 address space，說明各自的位址意義與典型用途。
- 寫出 PCI_Config region，並說明它與 _ADR 的搭配關係。
- 解釋「宣告不碰硬體、存取才碰硬體」與 region handler 的關係。
- 用 _REG 寫出 EC region 的就緒旗標，避免過早存取。
- 畫出硬體資源、OperationRegion、Field、暫存器的層次圖。

9.1 一行宣告，四個參數

先看課綱指定的範例——在 SystemIO 空間開一扇 16 byte 的窗：

課綱範例：SystemIO region

```
OperationRegion (REG0, SystemIO, 0x300, 0x10)
```

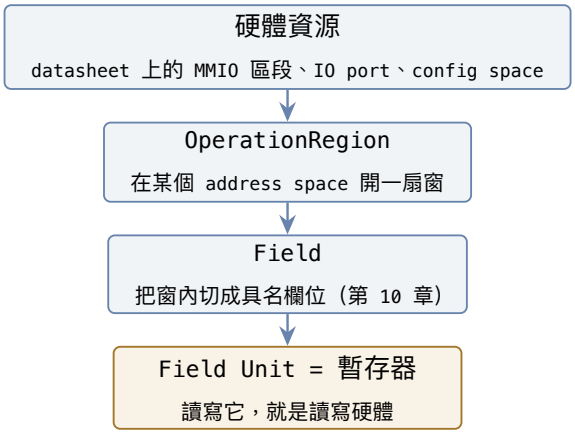
四個參數各司其職：

參數	本例	意義
RegionName	REG0	Namespace 物件名，命名規則同一般物件
RegionSpace	SystemIO	位址空間：決定「位址」怎麼解讀
Offset	0x300	窗口起點，意義隨空間而異（見下節）
Length	0x10	窗口長度，單位 byte

這行宣告的白話翻譯：「以後我會從 IO port 0x300 開始、長度 16 byte 的範圍讀寫，這扇窗叫 REG0。」值得一提的是，Offset 與 Length 不必是常數——它們是 TermArg，可以放運算式或

Named object。基底位址開機才知道時（例如先從別的暫存器讀出 BAR），就靠這個特性動態決定窗口位置。

OperationRegion 在整個存取體系中的位置如下，這張層次圖請刻在腦中：



9.2 九種 Address Space

第二個參數 RegionSpace 決定「0x300 這個數字要怎麼解讀」——是實體記憶體位址？IO port？還是 PCI config space 的偏移？ACPI 6.5 定義的標準空間如下：

RegionSpace	ID	Offset 的意義	典型用途
SystemMemory	0	實體記憶體位址（MMIO）	裝置暫存器、共享緩衝區
SystemIO	1	x86 IO port 編號	EC 0x62/0x66、CMOS 0x70、debug port 0x80
PCI_Config	2	該 PCI function 組態空間內的偏移	讀 Vendor ID、電源暫存器
EmbeddedControl	3	EC 位址空間偏移（0–255）	電池、風扇、溫度
SMBus	4	交易式：偏移對應 command	SMBus 電池、溫度晶片
GeneralPurposeIO	8	Connection pin list 的位元索引	直接讀寫 GPIO pin
GenericSerialBus	9	交易式：偏移 = command byte	I2C 裝置暫存器
PCC	0x0A	PCC subspace 內的偏移	與平台韌體信箱通訊
FFixedHW	0x7F	由 CPU 廠商定義	處理器功耗控制介面

九種空間可以分成三類來記：

- **位址直通型**（SystemMemory、SystemIO、PCI_Config、EmbeddedControl）：offset 就是某種「位址」，讀寫欄位對應一筆匯流排交易。
- **交易式**（SMBus、GenericSerialBus、PCC）：offset 其實是「命令編號」，一次欄位存取展開成一整筆協定交易，資料以 buffer 進出（第 10 章的 BufferAcc、第 11 章實戰）。
- **特殊型**（GeneralPurposeIO、FFixedHW）：GPIO 的 offset 是 pin 的位元索引，必須搭配 Connection（第 11 章）；FFixedHW 的解讀由 CPU 廠商自訂，多用於處理器電源介面。

規格裡還有 SystemCMOS（5）、PciBarTarget（6）、IPMI（7）等較少見的空間，以及留給平台自訂的區段；日常 bring-up 會遇到的，上表九種已經涵蓋。

9.3 SystemMemory 與 PCI_Config 實例

SystemIO 的例子已在上節出現（port 0x300），這節補齊另外兩個最常用的空間。先看 SystemMemory——課綱指定的 MMIO 範例，0xFED00000 是 Intel 平台 HPET 計時器的慣用基底：

DSDT 片段：HPET 的 MMIO 窗口

```
OperationRegion (HPTM, SystemMemory, 0xFED00000, 0x400)
Field (HPTM, DWordAcc, NoLock, Preserve)
{
    GCPL, 32,   // 0x00: general capabilities, low DWord
    GCPH, 32    // 0x04: general capabilities, high DWord
}
```

Field 的語法下一章才正式教，這裡先看形狀：宣告窗口之後，把開頭兩個 DWord 命名為 GCPL、GCPH，之後讀 GCPL 就是讀實體位址 0xFED00000。再看 PCI_Config：

DSDT 片段：PCI_Config 與 _ADR 的搭配

```
Device (SATA)
{
    Name (_ADR, 0x00170000)     // device 0x17, function 0
    OperationRegion (PCFG, PCI_Config, 0x00, 0x40)
    Field (PCFG, WordAcc, NoLock, Preserve)
    {
        VID, 16,    // config offset 0x00: Vendor ID
        DID, 16     // config offset 0x02: Device ID
    }
}
```

PCI_Config 的 offset 是「某一顆 **PCI function 的 config space** 內偏移」。哪一顆？由 region 所在的 **Device** 決定：interpreter 沿著 Namespace 往上找，用該裝置的 _ADR（配合上層 bridge 鏈與 _BBN / _SEG）定位到 bus/device/function。所以這段 region 必須宣告在 SATA 裝置底下才有意義——搬到別的 Scope，讀到的就是別顆裝置的 config space。

9.4 Region 只是窗口：宣告不等於存取

一個容易誤解的重點：**OperationRegion 的宣告不會產生任何匯流排交易**。interpreter 只是記下「有這麼一扇窗」；真正碰硬體的時刻，是第一次讀寫掛在窗上的 Field Unit。這個 lazy 語意有兩個實務後果：

- 宣告錯誤不會當場爆炸。位址寫錯的 region 可以安靜地躺在 DSDT 裡好幾年，直到某天有人存取它的欄位。
- 每種空間的存取都由對應的 **region handler** 代勞。SystemMemory 與 SystemIO 的 handler 由 OS 內建、永遠可用；PCI_Config 名義上也內建，但依賴 PCI 列舉完成；EmbeddedControl、GenericSerialBus、GeneralPurposeIO 的 handler 則由對應的 driver 註冊，開機初期不存在。

handler 什麼時候可用？OS 會呼叫 Scope 內的 `_REG` 通知：

DSDT 片段：_REG 守門 EC 存取

```
Scope (\_SB.PCI0.LPCB.EC0)
{
    Name (ECOK, Zero)
    Method (_REG, 2, NotSerialized)
    {
        If (Arg0 == 0x03)       // 3 = EmbeddedControl space
        {
            ECOK = Arg1         // 1 = handler ready, 0 = gone
        }
    }
    Method (SAFE, 0, NotSerialized)
    {
        If (ECOK == Zero)
        {
            Return (0xFF)       // EC not ready: bail out
        }
        Return (BTSC)           // read an EC field safely
    }
}
```

`_REG` 收兩個參數：Arg0 是空間 ID、Arg1 是可用性。上例把「EC 就緒」記在 `ECOK` 旗標裡，其他 method 存取 EC 欄位前先看旗標——第 8 章 warnbox 裡「觸控板消失」的事故，就是少了這道門。

補充說明：_REG 的參數就是空間 ID

`_REG` 的 Arg0 用的正是上一節表格的 ID 欄（3 = EC、9 = GenericSerialBus……），Arg1 為 1 表示 handler 就緒、0 表示即將移除。另外注意：SystemMemory 與 SystemIO 永遠可用，不會收到 `_REG`；PCI_Config 則要等 PCI 列舉完成——在開機極早期的 method（如 `_INI`）裡讀 config space，是經典的時序地雷。

注意：位址亂設，代價是當機甚至變磚

OperationRegion 沒有任何安全網：interpreter 不會、也無從檢查那段位址「屬不屬於你」。實務事故由輕到重：

- 讀到不存在的 MMIO：匯流排回 0xFF，或直接觸發 machine check 當機。
- SystemIO 誤寫 0xB2（SMI command port）：觸發非預期 SMI，韌體進入未知狀態。
- SystemMemory 蓋到 SPI flash controller 的暫存器並亂寫：可能改寫 flash 保護設定甚至內容，機器直接變磚。

宣告前逐一對照 chipset datasheet；並記住 lazy 語意的陰暗面——宣告當下不會爆，第一次存取才爆，出事的呼叫堆疊往往離元凶很遠。

9.5 綜合範例：Debug Port 裝置

把第 8 章的 Device 骨架與本章的 region 合體：一張插在 port 0x300 的 POST code debug 卡，完整可編譯：

dbg0.dsl：完整可編譯的 SSDT

```
DefinitionBlock ("dbg0.aml", "SSDT", 2, "ACME", "DBGPORT", 1)
{
    Scope (\_SB)
    {
        Device (DBG0)
        {
            Name (_HID, "ACME0100")     // debug port card
            Name (_UID, 0)
            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)
            }
            Name (_CRS, ResourceTemplate ()
            {
                IO (Decode16, 0x0300, 0x0300, 0x01, 0x10)
            })
            OperationRegion (REG0, SystemIO, 0x300, 0x10)
            Field (REG0, ByteAcc, NoLock, Preserve)
            {
                PSTC, 8     // offset 0x00: POST code register
            }
            Method (POST, 1, NotSerialized)
            {
                PSTC = Arg0     // write a code to the card
```

```
            }
        }
    }
}
```

逐行拆解：

- `_CRS` 用 `IO()` 巨集向 **OS** 申報「port 0x300 起 16 個 port 是我的」。
- `OperationRegion (REG0, SystemIO, 0x300, 0x10)` 向 **interpreter** 開窗——注意位址與 `_CRS` 申報的完全一致，這是鐵律：兩者各走各的，沒有人幫你核對。
- `Field` 把窗口第一個 byte 命名為 `PSTC`；`POST` method 寫它，等同對 port 0x300 做一次 `out` 指令。
- 之後 AML 任何地方呼叫 `\_SB.DBG0.POST`（例如在 `_INI` 裡打進度碼），debug 卡上的數字就會跳動——AML 版的 printf 除錯就是這麼做的。

核心觀念：本章重點回顧

- OperationRegion 四參數：名稱、address space、offset、length；offset 的意義由空間決定。
- 九種空間三類記憶法：位址直通（Memory / IO / PCI_Config / EC）、交易式（SMBus / GenericSerialBus / PCC）、特殊（GPIO / FFixedHW）。
- PCI_Config 的 offset 相對於「region 所在 Device」的 config space，靠 `_ADR` 對號。
- 宣告是 lazy 的：不碰硬體，第一次欄位存取才透過 region handler 碰硬體。
- EC、GenericSerialBus 等空間的 handler 由 driver 註冊，`_REG` 是就緒通知；存取前先檢查旗標。
- `_CRS` 說給 OS 聽、OperationRegion 說給 interpreter 聽，位址必須人工保持一致。

動手練習

1. 不看書寫出 OperationRegion 的四個參數，並各用一句話說明用途。
2. 替下列需求選 address space：讀 EC 的電池電量、讀 PCI 裝置的 Vendor ID、寫 0xFED90000 的 MMIO 暫存器、扳一根 GPIO pin、讀 I2C 溫度感測器。
3. 為什麼位址寫錯的 region 往往潛伏很久才出事？出事時的呼叫堆疊為什麼常常指不到元凶？
4. 寫出一個 `_REG` method 與旗標，讓 GenericSerialBus（空間 ID 9）的欄位存取有就緒保護。
5. 動手做：仿照綜合範例寫一份 SSDT，宣告 CMOS 的 IO 窗口（port 0x70 起 2 byte），以 `iasl` 編譯通過。（下一章會用 IndexField 真正操作它。）

第 10 章

Field

窗開好了，這一章動刀：**Field** 把 OperationRegion 切成一個個具名的 **Field Unit**——從此「讀寫一個變數」就是「讀寫一顆硬體暫存器」。這是 ASL 最漂亮的抽象：C driver 工程師眼裡的 `inl()` / `outl()` 與 MMIO 指標，在 ASL 裡全部化成一個等號。

但等號背後藏著三個必答題：一次交易敲多寬（AccessType）？要不要跟韌體搶鎖（LockRule）？沒被寫到的位元怎麼辦（UpdateRule）？本章逐一拆解，最後用 IndexField 與 BankField 收拾兩種「不走尋常路」的暫存器。

本章學習目標

- 用 Field 把 region 切成 Field Unit，寫出等號與 `Store` 兩種讀寫寫法。
- 依硬體匯流排寬度選擇 AccessType，說明 BufferAcc 的特殊角色。
- 說明 Lock 與 Global Lock 解決的並發問題。
- 畫出三種 UpdateRule 對「寫 4-bit 欄位進 8-bit 暫存器」的位元變化。
- 用 `Offset()` 與位元欄位對齊 datasheet，寫出跨 byte 的欄位。
- 寫出 IndexField 描述 index/data 暫存器對，說明 BankField 的適用場景。

10.1 Field：把窗口切成具名暫存器

先看課綱指定的範例——把上一章的 `REG0` 切成三顆暫存器：

課綱範例：Field 宣告

```
OperationRegion (REG0, SystemIO, 0x300, 0x10)
Field (REG0, ByteAcc, NoLock, Preserve)
{
    CTRL, 8,    // offset 0x00 -> port 0x300: control
    STAT, 8,    // offset 0x01 -> port 0x301: status
    DATA, 8     // offset 0x02 -> port 0x302: data
}
```

Field 的第一個參數是 region 名稱，後三個參數是「存取規則」，本章後面各給一節。大括號內是欄位清單：依宣告順序從 region 開頭排下去，每項給名稱與位元寬度。排出來的版圖如下：

Field 把 REG0 切成三顆具名暫存器

```
SystemIO port : 0x300     0x301     0x302     0x303 .. 0x30F
```

```
Field Unit    : CTRL(8)   STAT(8)   DATA(8)    (未切割區域)
                  │
                  └── CTRL = 1  ← 展開成一筆 out 0x300, 0x01
```

之後讀寫這些名字，interpreter 就替你產生對應的 IO 交易：

讀寫 Field Unit：新舊兩種寫法

```
Method (DEMO, 0, NotSerialized)
{
    CTRL = 1            // ASL 2.0 style: out 0x300, 0x01
    DATA = 0x55         // out 0x302, 0x55
    Store (0x55, DATA)  // legacy style, exactly the same
    Local0 = STAT       // read: in 0x301
}
```

課綱指定的 `CTRL = 1`、`DATA = 0x55` 是 ASL 2.0 的 symbolic 寫法；老 DSDT 裡等價的樣子是 `Store (1, CTRL)`。兩者編譯出的 AML 完全相同，讀舊碼、寫新碼都得認得。Field Unit 的物件型別是 `FieldUnitObj`：讀取時，寬度不超過 64 bit 的欄位轉成 Integer 回來，更寬的欄位回 Buffer （Module 3 的轉型規則在此適用）。

核心觀念：Field Unit 是「會碰硬體的變數」

對 Field Unit 的每一次讀寫都是真實的匯流排交易——它不是記憶體變數，沒有快取、沒有折疊。連續讀同一個欄位兩次，硬體就被敲兩次；對 read-to-clear 的暫存器，這個差別是天壤之別（見本章 warnbox）。

10.2 AccessType：一次交易敲多寬

Field 的第二個參數告訴 interpreter：對這個 region 的每筆交易應該用多大的存取單位。

AccessType	交易寬度	使用時機
`ByteAcc`	8 bit	IO port、EC、多數 legacy 裝置
`WordAcc`	16 bit	16-bit 暫存器組（老式 PCI 欄位）
`DWordAcc`	32 bit	絕大多數 MMIO 控制器
`QWordAcc`	64 bit	64-bit MMIO（HPET 計數器等）
`AnyAcc`	interpreter 自選	硬體不挑寬度時的方便選項
`BufferAcc`	一整包 buffer	交易式空間：SMBus、GenericSerialBus、IPMI

AccessType 本質上是**對硬體的承諾**。datasheet 常見這種話：「本暫存器僅支援 32-bit 對齊存取」——這時必須用 `DWordAcc`；若貪方便寫 `ByteAcc`，interpreter 會產生四筆 8-bit 交易，硬體可能直接忽略或行為錯亂，而 AML 端**不會有任何錯誤訊息**。反過來，欄位比存取寬度窄或跨越邊界時，interpreter 會自動用多筆交易讀出、組合、寫回（代價見下一節的 UpdateRule）。

`BufferAcc` 是完全不同的物種：欄位讀寫不再是「位址上的值」，而是一整包協定交易的資料緩衝區，用於 SMBus、GenericSerialBus 這類「一次存取等於一筆匯流排協定交易」的空間。第 11 章的 I2C 感測器會完整示範。順帶一提：EC 介面天生一次一個 byte，EmbeddedControl region 用 `ByteAcc` 就對了。

10.3 LockRule：與韌體共搶硬體

第三個參數只有兩個值：`NoLock` 與 `Lock`。寫 `Lock` 時，interpreter 在每次欄位存取前取得 **Global Lock**，結束後釋放。

為什麼需要它？歷史原因是 **OS 與韌體共用硬體**。EC 與 CMOS 這類資源，OS 的 AML 在存取，SMM 韌體（處理 SMI 的那段 BIOS 碼）也在存取；雙方各自 read-modify-write，不加鎖就會互相踩掉對方的修改。ACPI 因此定義了一把橫跨 OS 與韌體的鎖。

補充說明：Global Lock 的長相

Global Lock 本體是 FACS 表裡的一個 32-bit 欄位，OS 與韌體用規格定義的 compare-and-swap 協定競爭；搶不到的一方設下 pending 位元，持有者釋放時透過中斷或 SMI 通知對方再來。細節屬於韌體架構課，寫 ASL 只需記住：**datasheet 或平台文件說 SMM 也會碰的暫存器，Field 就宣告 `Lock`**；其餘一律 `NoLock`，因為拿鎖有實際的效能與死鎖風險成本。

10.4 UpdateRule：沒被寫到的位元怎麼辦

第四個參數回答一個微妙的問題。假設欄位只佔暫存器的一部分：

片段：只佔 8-bit 暫存器高 4 位的欄位

```
Field (REG0, ByteAcc, NoLock, Preserve)
{
    ,    4,        // bits 0-3: reserved, skip (anonymous)
    CTL4, 4        // bits 4-7: control nibble
}
```

寫 `CTL4 = 0x3` 時，交易單位是一整個 byte，但你只指定了高 4 位——低 4 位該寫什麼？三種 UpdateRule 三種答案：

對 8-bit 暫存器（現值 0xA6）寫 CTL4 = 0x3

```
位元編號        : 7 6 5 4   3 2 1 0
暫存器現值      : 1 0 1 0   0 1 1 0   (0xA6)
                  └─────┘   └─────┘
                  CTL4 欄位   未被寫到的位元

Preserve        : 0 0 1 1   0 1 1 0   (0x36)
```

```
                  ← 先讀 0xA6，改高 4 位，寫回（讀改寫）
WriteAsOnes     : 0 0 1 1   1 1 1 1   (0x3F)
                  ← 沒寫到的位元一律補 1
WriteAsZeros    : 0 0 1 1   0 0 0 0   (0x30)
                  ← 沒寫到的位元一律補 0
```

三者的成本與風險：

- `Preserve`：先讀後寫，兩筆交易。最直覺、最常用，鄰居位元保持原狀；但多出一次隱藏的讀取。
- `WriteAsOnes` / `WriteAsZeros`：不讀，直接寫，一筆交易。用在「其餘位元寫 1（或 0）無副作用」的硬體，或刻意避開隱藏讀取時。

注意：read-to-clear 與 write-1-to-clear 的雙重陷阱

狀態暫存器常有兩種特殊行為，跟 Field 的預設習慣正面相撞：

- **read-to-clear（RC）**：讀取即清零。`Preserve` 寫入鄰近欄位時的那次隱藏讀取，會把還沒被 driver 看到的狀態位元讀掉；AccessType 太寬「掃到」鄰居暫存器時也一樣。中斷旗標就這樣無聲消失，事件永遠等不到。
- **write-1-to-clear（W1C）**：寫 1 清零。`Preserve` 會把讀到的 1 原封不動寫回去——等於把所有 pending 位元一次清光。這種暫存器該用 `WriteAsZeros`（0 是無操作）。

對策：把 RC / W1C 暫存器獨立切成自己的 **Field**，選對 UpdateRule 與 AccessType，別讓它跟一般欄位共用一條交易路徑。datasheet 上的 access 欄（RO / RW / RC / W1C）就是你的切割地圖。

10.5 Offset()、位元欄位與跨 byte 欄位

真實 datasheet 的暫存器版圖不會乖乖連續。三個工具讓 Field 精準對齊：`Offset()` 跳到指定 byte、匿名欄位「`, n`」跳過 n 個 bit、任意 bit 寬度的欄位（可以只有 1 bit，也可以跨越 byte 邊界）。以筆電 EC 為例：

DSDT 片段：EC 暫存器版圖（節錄）

```
OperationRegion (ECRM, EmbeddedControl, 0x00, 0xFF)
Field (ECRM, ByteAcc, Lock, Preserve)
{
    Offset (0x20),
    BTSC, 8,         // 0x20: battery state of charge (%)
    Offset (0x23),
    LIDS, 1,         // 0x23 bit 0: lid open = 1
    ACIN, 1,         // 0x23 bit 1: AC adapter present
    ,     6,         // 0x23 bits 2-7: reserved
```

```
    Offset (0x30),
    TMP0, 16         // 0x30-0x31: temperature, 0.1 K units
}
```

逐行拆解：

- `Offset (0x20)`：游標直接跳到 region 內偏移 0x20，中間的 0x00–0x1F 不命名、也永遠不會被這個 Field 碰到。
- `LIDS, 1`：1-bit 欄位。第 1 章那個 `Return (LIDS)` 的謎底在此——讀它，interpreter 讀出 0x23 整個 byte，右移、遮罩，只回傳 bit 0。
- `, 6`：匿名跳過保留位元，讓後面的欄位對齊。
- `TMP0, 16`：跨兩個 byte 的欄位。`ByteAcc` 之下 interpreter 拆成兩筆交易再拼裝——不是 **atomic**，若硬體在兩筆交易之間更新數值，你會讀到「上半新、下半舊」的縫合怪。對會跳動的計數器，寧可分開讀兩個 byte 再自行判斷。
- 注意這個 Field 宣告了 `Lock`：EC 正是「OS 與韌體共搶」的經典案例。

10.6 IndexField 與 BankField

有些硬體暫存器太多、位址口太小，於是發明了「間接存取」：先把想存取的暫存器編號寫進 **index port**，再從 **data port** 讀寫值。CMOS RTC（port 0x70/0x71）、SuperIO（0x2E/0x2F）都是這一族。手寫兩步驟太囉唆，**IndexField** 把它自動化：

DSDT 片段：用 IndexField 讀 CMOS RTC

```
OperationRegion (CMOS, SystemIO, 0x70, 0x02)
Field (CMOS, ByteAcc, NoLock, Preserve)
{
    CIDX, 8,    // port 0x70: index register
    CDAT, 8     // port 0x71: data register
}
IndexField (CIDX, CDAT, ByteAcc, NoLock, Preserve)
{
    RSEC, 8,            // CMOS 0x00: RTC seconds
    Offset (0x02),
    RMIN, 8,            // CMOS 0x02: RTC minutes
    Offset (0x04),
    RHRS, 8,            // CMOS 0x04: RTC hours
    Offset (0x0E),
    DIAG, 8             // CMOS 0x0E: diagnostic status
}
Method (GSEC, 0, NotSerialized)
{
    Return (RSEC)       // = out 0x70, 0x00 then in 0x71
```

```
}
```

拆解：先用普通 Field 把 index / data 兩個 port 命名為 `CIDX`、`CDAT`；`IndexField (CIDX, CDAT, ...)` 再宣告一層虛擬 **region**——欄位的 offset 不是 IO 位址，而是「要寫進 index port 的編號」。讀 `RSEC` 時 interpreter 自動做兩步：把 0x00 寫進 `CIDX`，再從 `CDAT` 讀值。（小提醒：某些晶片組的 port 0x70 bit 7 兼任 NMI 遮罩，動 CMOS 前先讀平台文件。）

BankField 處理另一種怪招——「分頁」暫存器：同一段位址視 bank 暫存器的值切換內容。宣告時多帶 bank 暫存器與 bank 值兩個參數：

片段：BankField 描述分頁暫存器

```
Field (REG0, ByteAcc, NoLock, Preserve)
{
    Offset (0x0F),
    BSEL, 8             // 0x0F: bank select register
}
BankField (REG0, BSEL, 0, ByteAcc, NoLock, Preserve)
{
    R0A0, 8             // bank 0, offset 0x00
}
BankField (REG0, BSEL, 1, ByteAcc, NoLock, Preserve)
{
    R1A0, 8             // bank 1, offset 0x00
}
```

存取 `R1A0` 時，interpreter 先寫 `BSEL = 1` 切頁，再存取 offset 0x00。SuperIO 的 logical device 切換是最常見的應用場景。

10.7 綜合範例：MMIO 控制器

集大成：一顆虛構的 MMIO 控制器，同一個 region 疊兩層 Field （整暫存器視角與位元視角），加上輪詢邏輯，完整可編譯：

ctl0.dsl：完整可編譯的 SSDT

```
DefinitionBlock ("ctl0.aml", "SSDT", 2, "ACME", "MMIOCTRL", 1)
{
    Scope (\_SB)
    {
        Device (CTL0)
        {
            Name (_HID, "ACME0200")
            Name (_UID, 0)
            Name (_CRS, ResourceTemplate ()
```

```
            {
                Memory32Fixed (ReadWrite, 0xFED91000, 0x100)
            })
            OperationRegion (CREG, SystemMemory,
                             0xFED91000, 0x100)
            Field (CREG, DWordAcc, NoLock, Preserve)
            {
                CTRL, 32,       // 0x00: control register
                STAT, 32,       // 0x04: status register
                DATA, 32        // 0x08: data register
            }
            Field (CREG, DWordAcc, NoLock, Preserve)
            {
                ENAB, 1,        // 0x00 bit 0: enable
                RSTB, 1,        // 0x00 bit 1: soft reset
                ,     30,       // 0x00 bits 2-31: reserved
                RDYB, 1,        // 0x04 bit 0: ready
                ERRB, 1         // 0x04 bit 1: error
            }
            Method (INIT, 0, Serialized)
            {
                RSTB = One      // pulse soft reset
                RSTB = Zero
                ENAB = One      // enable the controller
            }
            Method (XFER, 1, Serialized)
            {
                Local0 = 100    // timeout: 100 polls
                While ((RDYB == Zero) && (Local0 > 0))
                {
                    Stall (10)  // wait 10 microseconds
                    Local0--
                }
                If (Local0 == Zero)
                {
                    Return (Zero)   // timed out
                }
                DATA = Arg0         // push the payload
                Return (One)
            }
        }
    }
}
```

逐行拆解：

- 同一個 CREG 疊了兩個 Field：第一層以 32-bit 為單位看整顆暫存器，第二層鑽進位元。兩層描述同一塊硬體，interpreter 不在乎重疊——這是對照 datasheet 寫碼的常用手法。
- INIT 對 RSTB 寫 1 再寫 0，做出 reset 脈衝；每次寫入在 Preserve 之下都是「讀 CTRL、改 bit 1、寫回」。
- XFER 輪詢 RDYB，每輪 Stall (10) 忙等 10 微秒；Stall 不會讓出 CPU，單次別超過 100 微秒，長等待改用 Sleep（Module 6 詳述）。
- Local0-- 與 While 條件裡的 && 都是 ASL 2.0 語法；legacy 等價寫法是 Decrement (Local0) 與 LAnd。
- 讀 RDYB 每次都是真實的 MMIO 讀取——輪詢會不斷敲硬體，這正是 Field Unit「會碰硬體的變數」本色。

核心觀念：本章重點回顧

- Field 把 region 依宣告順序切成 Field Unit；讀寫欄位就是真實的匯流排交易。
- AccessType 是對硬體的承諾：交易寬度要符合 datasheet；BufferAcc 專屬交易式空間。
- Lock 讓存取包在 Global Lock 裡，用於 OS 與 SMM 韌體共搶的硬體（EC、CMOS）。
- UpdateRule 決定沒被寫到的位元：Preserve 讀改寫、WriteAsOnes / WriteAsZeros 直接補 1 / 0；RC 與 W1C 暫存器是最大陷阱。
- Offset()、匿名欄位與位元欄位讓版圖精準對齊 datasheet；跨存取寬度的欄位不是 atomic。
- IndexField 自動化 index/data 間接存取；BankField 處理分頁暫存器。

動手練習

1. Field 的三個規則參數（AccessType、LockRule、UpdateRule）各自回答什麼問題？
2. 8-bit 暫存器現值 0xA6，在 WriteAsZeros 之下寫 CTL4 = 0x3（bits 4–7），暫存器變成多少？Preserve 之下呢？各發生幾筆匯流排交易？
3. 某中斷狀態暫存器是 W1C：為什麼 Preserve 會誤清 pending 位元？該選哪個 UpdateRule？
4. 用 IndexField 描述 SuperIO（index 0x2E、data 0x2F），並寫一個 method 讀出 offset 0x20 的 chip ID。
5. 動手做：在綜合範例中新增 16-bit 欄位 VERS（offset 0x0C），寫一個 method 回傳版本號，以 iasl 編譯驗證。

第 11 章

GPIO / I2C / SPI / PCI 實戰

現代機器的裝置掛在五花八門的匯流排上：觸控板在 I2C、感測器在 SPI、藍牙在 UART、LED 與開關在 GPIO、大傢伙在 PCI。本章把 Module 4 的 Device 骨架與前兩章的 region / field 機制組裝起來，逐一寫出這些匯流排的標準描述；貫穿全章的主軸只有一條：**controller 與 consumer 的分工**。

本章學習目標

- 區分 controller 與 consumer，說明 ResourceSource 的指向。
- 寫出 GPIO controller 與 GpioIo / GpioInt consumer，並用 GeneralPurposeIO region 直接讀寫 pin。
- 寫出 I2C 感測器的 I2cSerialBusV2 連線與 GenericSerialBus 存取 method；SPI 與 UART 比照辦理。
- 描述 PCI root bridge 與子裝置，讀取 config space。
- 佈局一份含 EC、I2C 感測器、GPIO LED 的 namespace 全景。

11.1 Controller 與 Consumer：兩種角色

一條 I2C 匯流排上有兩種東西：**controller**（主控端，SoC 裡的 I2C host）與 **consumer**（掛在線上的裝置，例如溫度感測器）。ACPI 描述它們的方式截然不同：

- **Controller 申報實體資源**：它是一顆平台裝置，`_CRS` 放 `Memory32Fixed` 與 `Interrupt`；PC 上常是 PCI 裝置（`_ADR`），SoC 平台常以 `_HID` 列舉。
- **Consumer 申報連線**：`_CRS` 放 `I2cSerialBusV2`、`GpioIo` 這類 **connection descriptor**——不佔位址，只申報「一條通往 controller 的線」，用 `ResourceSource` 路徑字串寫明「我掛在哪顆 controller、參數為何」。OS 讀到它，就把 consumer 掛到對應 controller 的匯流排底下 probe，driver 再經由 controller driver 存取 consumer。

11.2 GPIO：從 Controller 到扳一根 Pin

先立 controller。GPIO controller 是一顆平台裝置，用 `_HID` 列舉、申報自己的 MMIO 與中斷；**_AEI**（ACPI Event Interrupts）也掛在它身上，宣告「哪些 pin 觸發時請 OSPM 執行對應的 event method」：

DSDT 片段：GPIO controller 與 _AEI 事件

```
Device (GPI0)
{
    Name (_HID, "INT34C5")      // Intel TGL GPIO controller
    Name (_CRS, ResourceTemplate ()
    {
        Memory32Fixed (ReadWrite, 0xFD6E0000, 0x00010000)
        Interrupt (ResourceConsumer, Level, ActiveLow, Shared)
            { 14 }
    })
    Name (_AEI, ResourceTemplate ()
    {
        GpioInt (Level, ActiveLow, ExclusiveAndWake, PullUp, 0,
                 "\\_SB.GPI0", 0, ResourceConsumer, , ) { 43 }
    })
    Method (_L2B, 0, NotSerialized)  // lid pin 43 = 0x2B
    {
        Notify (\_SB.LID0, 0x80)     // status change event
    }
}
```

GpioInt 的參數依序是：觸發（Edge / Level）、極性（ActiveHigh / ActiveLow / ActiveBoth）、共享與喚醒（ExclusiveAndWake：此 pin 可在睡眠中喚醒系統）、上下拉、去彈跳時間（單位 0.01 ms，0 = 不指定），之後是所有 GPIO 巨集共通的尾巴：ResourceSource（controller 路徑）、index、usage、名稱、vendor data，大括號內是 pin list。事件 method 的命名規則：pin 編號轉十六進位接在 _L（level）或 _E（edge）後——pin 43 = 0x2B 就是 _L2B；編號超過 255 改用 _EVT。

consumer 端一次到位：_CRS 用 GpioIo 把 pin 申報給 driver；GeneralPurposeIO region（課綱指定寫法）加上 Connection()，讓 AML 自己扳 pin：

DSDT 片段：GPIO LED consumer

```
Device (LED0)
{
    Name (_HID, "ACME1103")
    Name (_CRS, ResourceTemplate ()
    {
        GpioIo (Exclusive, PullNone, 0, 0,
                IoRestrictionOutputOnly, "\\_SB.GPI0", 0,
                ResourceConsumer, , ) { 27 }   // LED drive pin
    })
    OperationRegion (GPIO, GeneralPurposeIO, 0, 1)
    Field (GPIO, ByteAcc, NoLock, Preserve)
    {
```

```
        Connection (GpioIo (Exclusive, PullNone, 0, 0,
            IoRestrictionOutputOnly, "\\_SB.GPI0", 0,
            ResourceConsumer, , ) { 27 }),
        LEDP, 1             // bit 0 = 1st pin in the list
    }                       // LEDP = One drives the pin high
}
```

GpioIo 逐參數解說（依序）：Exclusive（pin 是否可共用，幾乎都寫 Exclusive）、PullNone（內建上下拉：PullUp / PullDown / PullNone / PullDefault 沿用韌體設定）、0, 0（去彈跳與驅動強度，0 = 預設值）、IoRestrictionOutputOnly（方向限制：None / InputOnly / OutputOnly）、**ResourceSource**（controller 完整路徑，ASL 字串中反斜線要寫兩次），尾巴同 GpioInt：index、usage、名稱、vendor data，大括號內是 **pin list**（controller 相對編號）。

region 這一段三明治再拆一次：OperationRegion (GPIO, GeneralPurposeIO, 0, 1) 開 1 byte 的虛擬窗（容納 8 根 pin 的位元欄位）；Connection (...) 宣告「接下來的欄位走這條 GPIO 線」，欄位位元索引對應 pin list 順序，bit 0 是第一根；於是 LEDP = One 就把 pin 27 拉高——經 controller driver 的 region handler 完成，同樣受 _REG 約束。注意 _CRS 的 GpioIo 給 **driver** 看、Connection 給 **AML** 用；兩份指同一根 pin，慣例寫成一模一樣。

注意：三套 pin 編號，對錯一根全盤皆輸

pin list 裡的數字是**相對於 ResourceSource 那顆 controller** 的索引——不是 SoC datasheet 的 global GPIO 編號，也不是 Linux gpiochip 的編號；三套對不齊，輕則 LED 不亮，重則扳到電源樹訊號直接關機。且 ResourceSource 是 **不被編譯器驗證的字串**：路徑打錯 iasl 照樣編過，直到 OS probe 失敗才露餡。改 GPIO 前先對 schematic 與 controller 文件把 pin 索引算兩遍。

11.3 I2C：連線描述與 GenericSerialBus 存取

同一套劇本換到 I2C。controller（I2C1）與 GPI0 同一個模子——_HID 加 MMIO 加中斷，不再重列；直接看溫度感測器 consumer：

DSDT 片段：I2C 溫度感測器 consumer

```
Device (TMP1)
{
    Name (_HID, "ACME1102")     // I2C temperature sensor
    Name (_CRS, ResourceTemplate ()
    {
        I2cSerialBusV2 (0x0048, ControllerInitiated, 400000,
            AddressingMode7Bit, "\\_SB.I2C1", 0,
            ResourceConsumer, , Exclusive, )
    })
}
```

`I2cSerialBusV2` 逐參數解說（依序）：`0x0048` 是 slave address（右對齊，不含讀寫位元）；`ControllerInitiated` 表示交易由 controller 發起（幾乎永遠如此）；`400000` 是連線速度（Hz），即 Fast Mode 400 kHz；`AddressingMode7Bit` 是 7-bit 定址（另有 10Bit）；再來是 ResourceSource（指向 controller），其後依序為 index、usage、名稱（留空）、`Exclusive`（V2 新增的共享旗標）與 vendor data（留空）。

driver 有了這行就能透過 I2C subsystem 收發。但如果 **AML** 自己要跟感測器講話（例如 `_STA` 想確認它活著），就要動用第 9 章表格裡的交易式空間 `GenericSerialBus`：

DSDT 片段：GenericSerialBus 讀寫感測器

```
Scope (\_SB.TMP1)
{
    OperationRegion (SBUS, GenericSerialBus, 0x00, 0x100)
    Field (SBUS, BufferAcc, NoLock, Preserve)
    {
        Connection (I2cSerialBusV2 (0x0048,
            ControllerInitiated, 400000, AddressingMode7Bit,
            "\\_SB.I2C1", 0, ResourceConsumer, , Exclusive, )),
        AccessAs (BufferAcc, AttribByte),
        TEMP, 8,                // command 0x00: temperature
        CONF, 8                 // command 0x01: configuration
    }
    Method (RTMP, 0, Serialized)
    {
        Name (BUFF, Buffer (0x22) {})
        CreateByteField (BUFF, 0x00, STAT)
        CreateByteField (BUFF, 0x02, DATA)
        BUFF = TEMP             // issue an I2C read-byte
        If (STAT == 0x00)       // 0 = transfer OK
        {
            Return (DATA)
        }
        Return (0xFF)           // error marker
    }
}
```

這段是本章資訊密度最高的範例，逐層拆解：

- `OperationRegion (SBUS, GenericSerialBus, 0, 0x100)`：虛擬 region——offset 不是位址，而是 **command byte**（裝置內的暫存器編號）。`TEMP` 排在 offset 0 就代表 command 0x00，`CONF` 是 command 0x01。
- `AccessAs (BufferAcc, AttribByte)`：指定後續欄位的 存取協定為 read/write byte；其他常用協定有 `AttribWord`、`AttribBlock` 與 `AttribBytes (n)`（固定 n byte，I2C 常用）。

- BufferAcc 的資料不是純量而是一包 buffer，格式固定：offset 0 是 狀態（0 = 成功，非零 = 錯誤碼）、offset 1 是長度（block 類協定才有意義）、offset 2 起是資料。`CreateByteField` 就在這包 buffer 上開具名視窗。
- 讀寫方向由等號方向決定：`BUFF = TEMP` 觸發一筆 I2C read-byte 交易，結果落進 `BUFF`；反過來寫 `CONF = BUFF` 就是一筆 write-byte。任何交易後 先檢查 **`STAT`** 再用資料——I2C 是會 NAK 的。

11.4 SPI 與 UART：其他序列匯流排

SPI 與 UART 的 consumer 寫法與 I2C 同一個模子，換巨集與參數而已（每個 SerialBus 巨集都有不帶 V2 的舊版；V2 多了 Shared 旗標，新碼一律用 V2）：

DSDT 片段：SPI ADC 與 UART 藍牙模組

```
Device (ADC0)
{
    Name (_HID, "ACME1104")     // SPI analog-digital converter
    Name (_CRS, ResourceTemplate ()
    {
        SpiSerialBusV2 (0, PolarityLow, FourWireMode, 8,
            ControllerInitiated, 1000000, ClockPolarityLow,
            ClockPhaseFirst, "\\_SB.SPI1", 0,
            ResourceConsumer, , Exclusive, )
    })
}

Device (BTH0)
{
    Name (_HID, "ACME1105")     // Bluetooth module on UART
    Name (_CRS, ResourceTemplate ()
    {
        UartSerialBusV2 (115200, DataBitsEight, StopBitsOne,
            0x00, LittleEndian, ParityTypeNone,
            FlowControlNone, 64, 64, "\\_SB.UAR1", 0,
            ResourceConsumer, , Exclusive, )
    })
}
```

`SpiSerialBusV2` 參數重點（依序）：chip select 編號（0 = controller 的 CS0）、CS 極性（`PolarityLow` 為常態）、線制（FourWire / ThreeWire）、資料位寬（8）、發起端、時脈 1 MHz，然後是決定 **SPI mode** 的兩兄弟：Polarity 管 CPOL、Phase 管 CPHA，`ClockPolarityLow` + `ClockPhaseFirst` = mode 0，四種組合對應 mode 0–3，抄 datasheet 的時序圖就不會錯。

`UartSerialBusV2` 參數重點（依序）：鮑率 115200、資料位元、停止位元、LinesInUse（bitmask，宣告用到的 modem 控制線，0x00 = 只有 TX/RX）、位元組序、同位、流控（硬體流控寫 `FlowControlHardware` 並在 LinesInUse 設 RTS/CTS）、建議收發緩衝 64 byte，最後同樣以 ResourceSource 指向 controller。

11.5 PCI：Root Bridge、_ADR 與 Config Space

PCI 是「可自我列舉」匯流排的代表：子裝置不需要 `_HID`，用第 8 章的 `_ADR` 對號即可；但樹的根——PCI root bridge——OS 沒辦法自己發現，必須由 ACPI 以 `_HID` 立起來：

DSDT 片段：PCI root bridge 與子裝置（於 _SB 下）

```
Device (PCI0)
{
    Name (_HID, EisaId ("PNP0A08"))    // PCIe root bridge
    Name (_CID, EisaId ("PNP0A03"))    // PCI compatible

    Device (SATA)
    {
        Name (_ADR, 0x00170000)        // dev 0x17, func 0
        OperationRegion (PCFG, PCI_Config, 0x00, 0x40)
        Field (PCFG, WordAcc, NoLock, Preserve)
        {
            VID, 16,                   // 0x00: Vendor ID
            DID, 16                    // 0x02: Device ID
        }
        Method (PRSN, 0, NotSerialized)
        {
            Return (VID != 0xFFFF)     // all-ones = empty
        }
    }
}
```

拆解：

- `PNP0A08` 是 PCIe root bridge 的標準 ID，慣例搭配 `_CID PNP0A03`；完整的 root 還要以 `_SEG` / `_BBN` 宣告 segment 與 bus 編號、以 `_CRS` 申報 bus 範圍與 MMIO window（留給 Module 8）。
- `SATA` 以 `_ADR = 0x00170000` 對應 device 0x17、function 0；PCI_Config region 的 offset 相對這顆 function 的 config space。
- `PRSN` 讀 Vendor ID：PCI 規範空槽讀回全 1，所以 `VID != 0xFFFF` 是「裝置在不在」的慣用判斷。

11.6 SystemMemory 與 SystemIO：完整裝置範例

回頭補上兩個最樸素的空間，呼應第 9 章：HPET 計時器（MMIO 0xFED00000）與 debug 卡（port 0x300）：

DSDT 片段：HPET 與 debug 卡（於 _SB 下）

```
Device (TIM0)
{
    Name (_HID, EisaId ("PNP0103"))     // HPET
    Name (_CRS, ResourceTemplate ()
    {
        Memory32Fixed (ReadOnly, 0xFED00000, 0x400)
    })
    OperationRegion (MEM0, SystemMemory, 0xFED00000, 0x1000)
    Field (MEM0, DWordAcc, NoLock, Preserve)
    {
        CAPL, 32,       // 0x00: capabilities, low
        CAPH, 32        // 0x04: capabilities, high
    }
}
Device (DBG0)
{
    Name (_HID, "ACME0100")     // debug port card
    Name (_CRS, ResourceTemplate ()
    {
        IO (Decode16, 0x0300, 0x0300, 0x01, 0x10)
    })
    OperationRegion (REG0, SystemIO, 0x300, 0x10)
    Field (REG0, ByteAcc, NoLock, Preserve)
    {
        PSTC, 8         // 0x00: POST code register
    }                   // PSTC = 0x55 emits a POST code
}
```

兩顆裝置各自示範一種空間，共同示範同一條鐵律：_CRS 申報給 OS 的位址與 OperationRegion 開窗的位址必須人工保持一致。region 超出實際硬體範圍去存取，就是第 9 章 warnbox 的劇本。

11.7 綜合佈局：EC、I2C 感測器與 GPIO LED

最後把全章角色排進同一棵 Namespace，看清楚彼此的關係：

綜合佈局：EC + I2C 感測器 + GPIO LED

```
\
└── _SB_
    ├── GPI0            ← GPIO controller：MMIO + IRQ 14
    │   ├── _AEI  GpioInt {43}     ← lid 事件來源
    │   └── _L2B                   ← pin 43 的事件 method
    ├── I2C1            ← I2C controller：MMIO + IRQ 32
    ├── PCI0
    │   └── LPCB
    │       └── EC0_    ← EC：PNP0C09，IO 0x62/0x66，_REG
    ├── TMP1            ← I2cSerialBusV2 0x48 → 指向 I2C1
    ├── LED0            ← GpioIo {27}        → 指向 GPI0
    └── LID0            ← 上蓋開關，_LID（第 1 章的老朋友）
```

讀圖重點：controller 與 EC 各自申報實體資源；consumer 平行掛在 `\_SB` 下——樹的位置不決定連線關係，指向全靠 `ResourceSource`。事件鏈完整可循：lid 翻動、pin 43 觸發、OSPM 執行 `_L2B`、`Notify` 打向 LID0、OS 呼叫 `_LID` 讀狀態——第 1 章的場景，如今每一站你都寫得出來。

核心觀念：本章重點回顧

- Controller 申報實體資源（MMIO + 中斷）；consumer 用 connection descriptor 申報「線」，以 ResourceSource 指回 controller——路徑與 pin 編號編譯器不驗證，bring-up 前先算兩遍。
- GPIO：`GpioIo` / `GpioInt` 描述 pin；GeneralPurposeIO region 加 `Connection()` 讓 AML 直接扳 pin；`_AEI` 把 pin 變成事件。
- I2C：`I2cSerialBusV2` 給 driver 掛線；GenericSerialBus region 加 `AccessAs` 讓 AML 自己交易，buffer 格式是 status / length / data。
- SPI 用 chip select 與 CPOL / CPHA 定 mode；UART 申報鮑率、格式與流控——同一套 consumer 模板。
- PCI root bridge 以 `PNP0A08` 立根，子裝置用 `_ADR` 對號、PCI_Config region 讀組態空間。

動手練習

1. controller 與 consumer 的 `_CRS` 內容有什麼本質差異？OS 分別拿它們做什麼？
2. `GpioIo` pin list 裡的 27 屬於哪一套編號空間？為什麼對應的 Field Unit 只有 1 bit？
3. 寫出一顆 SPI 溫度感測器的 consumer 描述：掛在 `\_SB.SPI1` 的 CS1、mode 3、2 MHz、8-bit。
4. 動手做：把本章的 GPI0、TMP1、LED0 組成一份 SSDT，以 `iasl` 編譯通過，並手繪 namespace 樹與「誰指向誰」的連線圖。

Module 6

Control Method

難度：★★★☆☆

Module 5 教會你用 OperationRegion 與 Field 讀寫硬體暫存器，但那些終究是「資料」。本模組進入 ASL 真正的可執行單元——Control Method：如何宣告與呼叫方法、參數與區域變數如何運作、以及 AML 如何透過 Notify 主動向作業系統回報事件。學完本模組，你就具備撰寫「會動的」DSDT 所需的全部語言能力。

第 12 章

Method

Module 5 的 OperationRegion 與 Field 讓你能描述硬體暫存器，但 `Name` 與 Field 物件終究是「靜態資料」——它們安靜地躺在 Namespace 裡等人來讀。真正讓 DSDT「動起來」的是 **Control Method**：一段包在名字裡的 AML 程式，OS 在正確的時機呼叫它，它替 OS 敲硬體、算結果、回報狀態。從本章起，ASL 對你而言不再只是描述語言，而是一門真正的程式語言。

本章先建立 Method 的完整心智模型：誰呼叫它、何時呼叫、宣告語法每一欄的意義；接著深入 `Serialized` 這個最常被寫錯的選項；最後把多個 Method 組成一個小型計算模組，練習「方法呼叫方法」的完整寫法。

本章學習目標

- 說明 Method 與 Name 靜態物件的差別，以及 OS 在哪些時機呼叫 Method。
- 寫出完整的 Method 宣告，正確選擇參數個數與 SerializeRule。
- 說明 Serialized 解決的重入問題，並判斷何時必須使用。
- 用 ASL 2.0 與 legacy 兩種語法呼叫 Method 並使用回傳值。
- 說明方法呼叫方法時的名稱解析、遞迴限制與方法內物件的生命週期。
- 除錯 AE_ALREADY_EXISTS 這類典型的 Method 執行期錯誤。

12.1 誰會呼叫 Method？從靜態描述到可執行程式

先複習一個關鍵差異：`Name (X, 42)` 在表載入時求值一次，之後 `X` 就是一個躺在 Namespace 裡的整數；而 `Method` 的本體是一段 AML 位元組碼，載入時只登記名字，**每次被呼叫才逐條執行**。換句話說，Name 回答「是什麼」，Method 回答「怎麼做」。

那麼誰來呼叫？在 ACPI 的世界裡，Method 的呼叫者幾乎都是 OS——這正是 OSPM「韌體描述、OS 決策」精神的延伸：

呼叫者	時機	典型 Method
OS（列舉）	開機建立裝置樹時	`_STA`、`_CRS`、`_INI`
OS（電源）	睡眠、喚醒、D-state 轉換	`_PTS`、`_WAK`、`_PS0`–`_PS3`
OS（事件）	GPE 觸發、EC query	`_Lxx`、`_Exx`、`_Qxx`
Driver	執行期主動查詢與控制	`_BST`、`_LID`、`_DSM`
其他 Method	AML 內部呼叫	你自己寫的輔助方法

一次 Method 呼叫的完整旅程如下：



注意最後一站：Method 的執行者永遠是 OS 內建的 ACPI Interpreter（第 1 章講過），韌體只負責在開機前把這段位元組碼放進表裡。所以 Method 寫錯的症狀是「OS 執行期報錯」，而不是開機自檢失敗——這決定了 Module 10 教的除錯方式都繞著 OS 端的訊息打轉。

12.2 Method 宣告：完整語法解剖

Method 宣告最完整的形式有六個欄位，日常使用通常只寫前二到四個：

Method 宣告的完整語法（框架，非可編譯片段）

```
Method (MethodName, NumArgs, SerializeRule, SyncLevel,
        ReturnType, ParameterTypes)
{
    // TermList: statements executed on every call
}
```

欄位	可省略	意義
MethodName	否	4 字元名稱，一般 Namespace 命名規則
NumArgs	可（預設 0）	參數個數 0–7，決定能用幾個 `Arg`
SerializeRule	可	`NotSerialized`（預設）或 `Serialized`
SyncLevel	可（預設 0）	0–15，僅 Serialized 時有意義
ReturnType	可	給 iasl 靜態檢查用，如 `IntObj`
ParameterTypes	可	各參數型別清單，同樣僅供檢查

課綱的起手式範例如下——一個把兩數相加的方法，以及呼叫它的測試方法：

範例：add2.dsl——宣告與呼叫一個 Method

```
DefinitionBlock ("add2.aml", "SSDT", 2, "DEMO", "ADD2DEMO", 0x01)
{
    Scope (\_SB)
```

```
    {
        Method (ADD2, 2)               // 2 args, NotSerialized
        {
            Return (Arg0 + Arg1)
        }

        Method (TST0, 0)
        {
            Local0 = ADD2 (10, 20)         // ASL 2.0 style
            Store (ADD2 (30, 40), Local1)  // legacy style
            Return (ADD2 (Local0, Local1)) // methods compose
        }
    }
}
```

逐行看重點：

- `Method (ADD2, 2)`：宣告兩個參數，省略 SerializeRule 表示 `NotSerialized`。參數在方法內以 `Arg0`、`Arg1` 存取，最多到 `Arg6`——這是 AML 編碼層面的硬限制，第 13 章會再深入。
- `Return (Arg0 + Arg1)`：ASL 2.0 允許運算式直接出現在 `Return` 裡；legacy 寫法得先 `Add (Arg0, Arg1, Local0)` 再 `Return (Local0)`。
- `Local0 = ADD2 (10, 20)`：呼叫語法就是「名稱加括號」，回傳值可以直接指定給變數、當運算元、或丟進條件式。

補充說明：ReturnType 與跨表呼叫的檢查極限

最後兩個欄位（ReturnType、ParameterTypes）只給 iasl 做編譯期靜態檢查，不會編進 AML，市售 DSDT 幾乎沒人寫。另外要注意：iasl 只能檢查**同一張表內**的呼叫；跨表呼叫（例如 SSDT 呼叫 DSDT 的方法）得用 `External` 宣告對方，此時參數個數寫錯，編譯器無從發現，要到執行期才爆炸——第 13 章會示範爆炸現場。

12.3 呼叫 Method 與使用回傳值

呼叫語法在 ASL 2.0 與 legacy 兩代寫法中都存在，你在舊 DSDT 反組譯輸出裡兩種都會遇到，必須都看得懂：

片段：兩代呼叫語法對照

```
// ASL 2.0 (recommended for new code)
Local0 = ADD2 (10, 20)
Local1 = ADD2 (1, 2) + 3           // call inside expression
If (ADD2 (Local0, 5) > 10)         // call inside condition
```

```
{
    Local2 = 1
}

// Legacy (what you will see in shipping DSDTs)
Store (ADD2 (10, 20), Local0)
Store (Add (ADD2 (1, 2), 3), Local1)
If (LGreater (ADD2 (Local0, 5), 10))
{
    Store (1, Local2)
}
```

幾條實用規則：

- 不需要回傳值時，把呼叫單獨寫成一行即可，如 `TST0 ()`。
- `Return` 有三種姿態：`Return (值)` 帶值返回、`Return ()` 或 `Return (Zero)` 空手返回、以及執行到方法結尾自然結束。
- 一個方法可以有多個 `Return`，分散在不同的 If 分支——但要確保「每條路徑」的行為都是你要的。

注意：不要依賴沒有 Return 的回傳值

如果某條執行路徑沒有明確 `Return`，呼叫端拿到什麼是 **未定義行為**：有的 interpreter 隱含回傳最後一次運算的結果（所謂 implicit return，ACPICA 提供 slack mode 相容這種寬鬆行為），有的回傳零或直接報錯。同一份 DSDT 在 Windows 正常、在 Linux 出錯，不少案例正是踩了這條線。準則很簡單：**只要呼叫端會用到回傳值，每條路徑都要明確 `Return`**。iasl 會對「有些路徑有 Return、有些沒有」的方法提出警告，別忽略它。

12.4 Serialized：方法重入與 AE_ALREADY_EXISTS

`Serialized` 是宣告欄位裡最容易被忽略、也最常引發詭異錯誤的一個。要理解它，先看一個反面教材：

反面教材：會因重入而爆炸的方法（片段）

```
Method (GETB, 0, NotSerialized)        // BUG: not serialized
{
    Name (BUF0, Buffer (8) {})         // creates a named object
    CreateWordField (BUF0, 0, WRD0)    // another named object
    WRD0 = 0x1234
    Return (BUF0)
}
```

問題出在 Name 與 CreateWordField：方法內建立的具名物件會**真的掛進 Namespace**（掛在方法自己的 scope 底下），等方法返回時才移除。現在想像兩條執行緒幾乎同時呼叫 GETB——現代 OS 的 ACPI 子系統本來就是多執行緒的，兩個 driver 同時查詢、或 GPE 事件撞上 driver 呼叫，都稀鬆平常：

- 執行緒 A 進入方法，成功建立 BUF0。
- 執行緒 B 也進入方法，執行 Name (BUF0, ...) 時發現 Namespace 裡已經有同名物件——建立失敗。
- Interpreter 中止執行緒 B 的這次呼叫，回報 AE_ALREADY_EXISTS。

在 Linux 上，dmesg 會留下這樣的犯罪現場：

dmesg：典型的方法重入錯誤

```
$ dmesg | grep "ACPI Error"
ACPI Error: [BUF0] Namespace lookup failure, AE_ALREADY_EXISTS
            (20230628/dsfield-184)
ACPI Error: Aborting method \_SB.EC0.GETB due to previous error
            (AE_ALREADY_EXISTS) (20230628/psparse-529)
```

解法只要一個字：把宣告改成 Method (GETB, 0, Serialized)。Serialized 等於替方法配一把隱含的 Mutex：第二個呼叫者會排隊，等第一個離開後才進入，具名物件屆時早已移除，自然不再撞名。iasl 其實會主動提醒你——對「建立具名物件卻宣告 NotSerialized」的方法，編譯時會給一條 remark：

iasl 對未序列化方法的提醒

```
$ iasl getb.dsl
...
getb.dsl   14:  Method (GETB, 0, NotSerialized)
Remark 2120 -  ^ Control Method should be made Serialized
               (due to creation of named objects within)
```

核心觀念：Serialized 的實務準則

- 方法內用了 Name、CreateField 系列、OperationRegion 等會建立具名物件的敘述 → 一律 **Serialized**，沒有例外。
- 方法對硬體做「讀—改—寫」序列、或操作共享的暫存器組 → 用 Serialized 保護，避免兩次呼叫交錯。
- 純運算、只碰 Arg 與 Local → NotSerialized 即可，讓呼叫可以並行。
- 看到 iasl 的 Remark 2120，當成錯誤處理，不要放行。

至於第四欄 SyncLevel：進入 Serialized 方法等同於取得一把 SyncLevel 為 N 的隱含 Mutex，而 ACPI 規定持有 SyncLevel N 的鎖時，只能再取得 SyncLevel 大於等於 N 的鎖——這是防止死鎖的階層鎖設計。絕大多數韌體把它留在預設值 0；只有在方法之間有巢狀鎖需求時才需要精心安排，屆時記得「外層小、內層大」。

12.5 方法呼叫方法、遞迴與方法內的 Namespace

方法內呼叫另一個方法時，名稱解析走 Module 2 教過的規則：先在目前 scope 找，找不到就一層層往上。實務建議是：同一個 Device 內的輔助方法用相對名稱（如 `ADD2 (1, 2)`），跨裝置、跨 scope 一律寫絕對路徑（如 `\_SB.CALC.ADD2`），省得日後搬動程式碼時解析結果悄悄改變。

方法內建立的具名物件有三條性質，合在一起記：

- **動態建立**：執行到 `Name` 那行才建立，不是載表時建立。
- **外部不可見**：物件掛在方法 scope 下，方法外的程式碼與 OS 都查不到它。
- **返回即銷毀**：方法結束（或中止）時自動移除，下次呼叫重新建立。

那遞迴呢？規格允許方法呼叫自己，但有兩道現實的牆。第一，ACPICA 對同一方法的巢狀重入設了上限（255 層），超過就回報 `AE_AML_METHOD_LIMIT`；第二，含具名物件的方法一遞迴就撞 `AE_ALREADY_EXISTS` ——而且 **Serialized 也救不了**，因為隱含 Mutex 對同一條執行緒是可重入的，擋不住自己撞自己。結論：AML 是硬體控制的膠水語言，不是寫演算法的地方，需要重複就用 **While** 迴圈，不要用遞迴。

12.6 完整範例：感測器數值處理模組

把本章所學組裝起來：一個由三個 Method 構成的小型計算模組，模擬「把感測器原始讀值換算成毫度並夾在合理範圍內」的常見需求。

calc.dsl：多個 Method 組成的計算模組

```
DefinitionBlock ("calc.aml", "SSDT", 2, "DEMO", "CALCDEMO", 0x01)
{
    Scope (\_SB)
    {
        Device (CALC)
        {
            Name (_HID, "DEMO0001")

            // Larger of two integers
            Method (MAX2, 2)
            {
                If (Arg0 >= Arg1)
```

```
                {
                    Return (Arg0)
                }
                Return (Arg1)
            }

            // Clamp Arg0 into the range [Arg1, Arg2]
            Method (CLM3, 3)
            {
                Local0 = MAX2 (Arg0, Arg1)   // lower bound
                If (Local0 > Arg2)
                {
                    Return (Arg2)            // upper bound
                }
                Return (Local0)
            }

            // Raw sensor value -> milli-degrees, clamped.
            // Uses a named scratch buffer: must be Serialized
            Method (SCAL, 1, Serialized)
            {
                Name (TMPB, Buffer (4) {})
                CreateDWordField (TMPB, 0, VAL0)
                VAL0 = Arg0 * 125            // 1 LSB = 125 mC
                Return (CLM3 (VAL0, 0, 100000))
            }
        }
    }
}
```

逐行拆解：

- `MAX2` 與 `CLM3` 只碰 `Arg` 和 `Local`，不建立具名物件，所以放心用預設的 NotSerialized，讓多個呼叫者可以並行。
- `CLM3` 用相對名稱呼叫同 scope 的 `MAX2`：先套下界，再用 If 套上界。兩個 `Return` 分別覆蓋兩條路徑，沒有漏網之魚。
- `SCAL` 內有 `Name` 與 `CreateDWordField`，依準則宣告 `Serialized`。`VAL0` 是 buffer field，寫入時自動依 32 位元寬度截斷——這是拿 Buffer 當「固定寬度暫存器」用的慣用手法。
- `Return (CLM3 (VAL0, 0, 100000))`：把另一個方法的回傳值直接轉交，呼叫鏈 `SCAL` → `CLM3` → `MAX2` 深度只有 3，離 255 的重入上限遠得很。

用 iasl 編譯確認無誤後，你已經會寫「可執行」的 ASL 了。但眼尖的你應該注意到：參數與區域變數還有很多沒講清楚的細節——`Arg0` 到底是複本還是參考？`Local` 沒指定初值可以直

接讀嗎？這些正是下一章的主題。

核心觀念：本章重點回顧

- Name 載表時求值一次是「資料」；Method 每次呼叫才執行，是「程式」。呼叫者是 OS（列舉、電源、事件）、driver 與其他方法。
- 宣告六欄位：名稱、參數個數（0–7）、SerializeRule、SyncLevel、ReturnType、ParameterTypes；日常只寫前二到四個。
- 方法內建立具名物件就必須 `Serialized`，否則多執行緒重入會以 `AE_ALREADY_EXISTS` 中止。
- 呼叫語法兩代並存：`Local0 = ADD2 (10, 20)` 與 `Store (ADD2 (10, 20), Local0)`；會用到回傳值的方法，每條路徑都要明確 `Return`。
- 方法內物件動態建立、外部不可見、返回即銷毀；遞迴技術上可行但實務禁用，重複邏輯交給 While。

動手練習

1. 說出 Method 與 Name 物件的三個差異：求值時機、能否帶參數、對 Namespace 的影響。
2. 一個宣告為 NotSerialized 的方法內含 `Name (PKG0, ...)`，兩條執行緒同時呼叫它，逐步描述會發生什麼事、dmesg 會出現哪個錯誤碼。
3. 把 legacy 寫法 `Store (Add (MAX2 (Local0, 5), 1), Local1)` 改寫成 ASL 2.0 語法。
4. 為 12.6 的模組新增 `MIN2`（回傳較小值），並改寫 `CLM3` 讓它同時用 `MAX2` 與 `MIN2` 完成夾範圍，兩條路徑合併成一條。
5. 動手：把 calc.dsl 存檔並用 `iasl calc.dsl` 編譯；接著故意把 `SCAL` 改回 NotSerialized 再編譯一次，觀察並記下 iasl 的 remark 內容。

第 13 章

Method Arguments

上一章把 Method 當成一個會動的黑盒子；本章打開盒子。一次方法呼叫的內部世界由四個部分組成：**Arguments**（呼叫者給的）、**Locals**（自己用的）、**Namespace Objects**（動態建立的）、**Return Value**（交回去的）。搞清楚每一部分的生命週期與傳遞語意，是寫出正確 Method 的前提，也是讀懂一大類 dmesg 錯誤訊息的鑰匙。

這一章的知識點瑣碎但環環相扣，我們按「先畫全景圖，再逐格放大」的順序走：四個組成、Arg 的傳遞語意、Local 的初始化規則、參數個數不匹配的下場，最後是 RefOf 傳參考與 Package 進出的完整範例。

本章學習目標

- 畫出 Method 一次呼叫的四個組成，並說明各自的生命週期。
- 說明 Arg0 到 Arg6 的傳遞語意：何時拿到複本、何時拿到參考。
- 說明 Local0 到 Local7 的初始化規則，除錯 AE_AML_UNINITIALIZED_LOCAL。
- 解釋參數個數不匹配時 interpreter 的行為與對應錯誤碼。
- 用 RefOf 與 DerefOf 讓被呼叫的方法把結果寫回呼叫端的變數。
- 寫出傳遞 Package 與回傳 Package 的 Method。

13.1 一次呼叫的世界：Method 的四個組成

每次呼叫 Method，interpreter 都會替這次呼叫建立一個全新的執行環境，方法返回時整組拆除。它的內容物固定是四類：

Method 的四個組成（每次呼叫都是全新的一份）

```
Method (STAT, 2, Serialized) 的「一次呼叫」
├── Arguments          Arg0 ~ Arg6：呼叫者傳入，最多 7 個
├── Local Variables    Local0 ~ Local7：方法私有暫存，共 8 個
├── Namespace Objects  方法內 Name() 等動態建立，返回時銷毀
└── Return Value       Return (...) 交回呼叫者的唯一輸出
```

「每次呼叫都是全新的一份」這句話值得多想三秒：兩條執行緒同時執行同一個方法時，各自擁有獨立的 Arg 與 Local，互不干擾——所以上一章說「只碰 Arg 與 Local 的方法可以放心 NotSerialized」；真正共享的是 Namespace（第三格）與硬體，那才是需要 Serialized 與 Mutex 保護的東西。

13.2 Arguments：Arg0 到 Arg6

宣告 `Method (STAT, 2)` 代表呼叫者要給兩個參數，方法內用 `Arg0`、`Arg1` 取用。上限七個（`Arg0` 到 `Arg6`）是 AML 位元組碼格式的硬限制；真需要傳更多資料，就打包成一個 Package 傳進來（本章 13.6 示範）。

傳遞語意是本節的重頭戲。基本型別——Integer、String、Buffer——可以當成**逐值傳遞**來理解：方法拿到的是「這次呼叫自己的一份」，對 `Arg` 整個重新賦值，呼叫端完全無感：

片段：對 Arg 整個覆寫不會影響呼叫端

```
Method (NOPE, 1)
{
    Arg0 = 999              // rebinds the local slot only
}

Method (TRY0, 0)
{
    Local0 = 5
    NOPE (Local0)
    Return (Local0)         // still 5, not 999
}
```

想讓被呼叫者改到呼叫端的東西，正規做法是下一節的 `RefOf` 明確傳參考。先把語意整理成表：

呼叫者傳入	Arg 裡是什麼	callee 能改到呼叫端嗎
Integer 常數或變數	值的複本	不能
String、Buffer	該次呼叫的一份物件	整個覆寫不能
Package	物件的參考	改元素可能可以（見補充）
`RefOf (物件)`	明確的物件參考	能，這是正規寫回管道

補充說明：嚴謹版語意：String、Buffer、Package 其實是參考

嚴格說，AML 傳遞 String、Buffer、Package 時給的都是物件參考，所以在 callee 內用 `Index` 直接改 `Arg0` 的某個元素，ACPICA 上會改到呼叫端的物件；但「整個覆寫 `Arg0`」只是讓這個 slot 改指別的東西。這個邊角語意在不同 OS 的 interpreter 上有實作差異，是跨平台相容性的地雷區。實務準則：**要寫回就用 RefOf 明講**，不要依賴隱含共享；把「逐值」當成預設心智模型最安全。

13.3 Locals：Local0 到 Local7

每次呼叫有八個區域變數 `Local0` 到 `Local7`，型別是動態的：存整數它就是整數，存 Buffer 它就是 Buffer。關鍵規則只有一條：**Local 一開始是「未初始化」狀態，寫過才能讀**。讀一個沒寫過的 Local，interpreter 直接中止方法：

dmesg：讀取未初始化 Local 的下場

```
$ dmesg | grep "ACPI Error"
ACPI Error: AE_AML_UNINITIALIZED_LOCAL, While resolving operands
            (20230628/dswexec-431)
ACPI Error: Aborting method \_SB.DEMO.BADM due to previous error
            (AE_AML_UNINITIALIZED_LOCAL) (20230628/psparse-529)
```

這個錯誤最常見的長相不是「忘記初始化」這麼直白，而是**某條 If 分支忘了設值**：

片段：只有一條路徑初始化 Local 的典型 bug

```
Method (BADM, 1)
{
    If (Arg0 > 10)
    {
        Local0 = 1
    }
    Return (Local0)    // BUG: Arg0 <= 10 leaves Local0 unset
}
```

養成習慣：要在分支裡賦值的 Local，先在分支前給預設值（`Local0 = 0`），讓每條路徑都有明確定義。

13.4 參數個數不匹配：Interpreter 不會救你

呼叫端給的參數個數與宣告不符時會怎樣？分三種情況：

- **同一張表內**：iasl 看得到宣告，編譯期直接報錯（Too few / too many arguments），過不了關。
- **跨表呼叫、少給**：編譯期查不到，執行期 callee 一讀到沒人給值的 Arg，就以 `AE_AML_UNINITIALIZED_ARG` 中止——症狀跟未初始化 Local 一模一樣，只是錯誤碼不同。
- **跨表呼叫、多給**：多的參數被默默忽略，通常「看起來沒事」，直到有人改了方法簽名才一起爆發。

注意：AML 沒有執行期的參數個數檢查

AML 位元組碼裡的方法呼叫只是「名稱 + 一串運算元」，interpreter 不做跨表的編譯期檢查，也不做執行期的個數核對。SSDT 呼叫 DSDT 方法時，`External` 宣告寫錯參數個數、或日後有人改了 DSDT 端的 NumArgs 而忘了同步 SSDT，都只會在執行期以 `AE_AML_UNINITIALIZED_ARG` 收場。維護多張表時，把共用方法的簽名當成 ABI 看待：改簽名等於改介面，所有呼叫端都要盤點。

13.5 傳參考：RefOf 與 DerefOf

需要「被呼叫者把結果寫回呼叫端變數」時，用 `RefOf` 取得物件參考傳進去。callee 端的規則要背起來：

- 讀：`Arg0` 本身是參考，取值要用 `DerefOf (Arg0)`。
- 寫：對存著參考的 `Arg0` 執行 `Store (值, Arg0)`，會寫穿參考、改到被參考的物件——這是 ACPI 規格明定的行為。

片段：透過參考把結果寫回呼叫端

```
Name (CNT0, 5)

Method (INCR, 1)            // Arg0: reference to an Integer
{
    Local0 = DerefOf (Arg0) + 1
    Store (Local0, Arg0)    // writes through the reference
}

Method (TRY1, 0)
{
    INCR (RefOf (CNT0))     // CNT0: 5 -> 6
    Return (CNT0)
}
```

逐行拆解：

- `RefOf (CNT0)` 產生指向 `CNT0` 的參考物件，以它為實參呼叫 `INCR`。
- callee 內 `DerefOf (Arg0)` 取出目前值 5，加一後 `Store (Local0, Arg0)` 寫穿參考，`CNT0` 變成 6。
- 若 callee 想把參考存進 Local 再操作，注意下面補充框的差異。

補充說明：Arg 與 Local 對參考的待遇不同

同樣存著一個參考，`Store` 到 **Arg** 是「寫穿」（改到被參考的物件）；`Store` 到 **Local** 卻是「覆蓋」（Local 改存新值，參考被丟掉）。想透過 Local 裡的參考寫回目標，得寫 `Store (值, DerefOf (Local0))` 這類明確解參考的形式。這是 ACPI 規格裡最不對稱的角落之一，跨 interpreter 行為也最容易出入——寫回邏輯儘量放在 Arg 上，簡單、可攜。

13.6 完整範例：Package 進、Package 出

Package 是 ASL 裡的萬用容器，方法之間傳「一組資料」全靠它。下面的範例涵蓋三個常見動作：接收 Package 並走訪加總、建立 Package 回傳、以及呼叫端拆包取值。

args.dsl：Package 與參考的綜合演練

```
DefinitionBlock ("args.aml", "SSDT", 2, "DEMO", "ARGSDEMO", 0x01)
{
    Scope (\_SB)
    {
        Name (CNT0, 0)              // shared counter

        // Sum all integers inside a package
        Method (PSUM, 1)
        {
            Local0 = 0              // accumulator
            Local1 = 0              // index
            Local2 = SizeOf (Arg0)
            While (Local1 < Local2)
            {
                Local0 += DerefOf (Arg0 [Local1])
                Local1++
            }
            Return (Local0)
        }

        // Build a status package: [id, raw, scaled, flag]
        Method (STAT, 2, Serialized)
        {
            Name (PKG0, Package (4) {0, 0, 0, 0})
            PKG0 [0] = Arg0
            PKG0 [1] = Arg1
            PKG0 [2] = Arg1 * 10
            If (Arg1 > 100)
            {
                PKG0 [3] = 1
```

```
            }
            Return (PKG0)
        }

        // Increment an integer through a reference
        Method (INC2, 1)
        {
            Local0 = DerefOf (Arg0) + 1
            Store (Local0, Arg0)
        }

        Method (DEMO, 0, Serialized)
        {
            Name (VALS, Package () {3, 5, 8})
            Local0 = PSUM (VALS)            // 16
            INC2 (RefOf (CNT0))             // CNT0: 0 -> 1
            Local1 = STAT (1, 120)          // returned package
            Local2 = DerefOf (Local1 [2])  // scaled = 1200
            Return (Local0 + Local2)
        }
    }
}
```

逐行拆解：

- PSUM 用 SizeOf (Arg0) 取元素個數，DerefOf (Arg0 [Local1]) 是「索引再解參考」的標準連招——Index（此處是 [] 語法糖）回傳的是元素參考，要 DerefOf 才拿到值。
- STAT 在方法內 Name 出 PKG0，所以宣告 Serialized（第 12 章的準則）。PKG0 [0] = Arg0 是 ASL 2.0 的元素賦值，等價於 legacy 的 Store (Arg0, Index (PKG0, 0))。
- STAT 直接 Return (PKG0)：方法返回時 PKG0 這個具名物件會銷毀，但回傳的 Package 內容會交給呼叫端——interpreter 負責延長它的壽命。
- DEMO 依序演練三招：傳 Package 進去、傳參考讓 INC2 寫回 CNT0、接住 STAT 的回傳再用 DerefOf (Local1 [2]) 拆包。

參數超過七個怎麼辦？答案也在這個範例裡：把資料打包。呼叫 STAT (1, 120) 若改成傳一個 Package () {1, 120, ...}，一個 Arg 就能載任意多的欄位——這是 _DSM 這類標準方法的慣用介面設計。

核心觀念：本章重點回顧

- 一次呼叫四個組成：Arguments、Locals、Namespace Objects、Return Value；前兩者每次呼叫獨立，互不干擾。

- Arg 上限七個（`Arg0`–`Arg6`）；基本型別視為逐值，要寫回就用 `RefOf` 傳參考、`DerefOf` 取值，`Store` 到 Arg 會寫穿參考。
- Local 共八個，未初始化就讀會以 `AE_AML_UNINITIALIZED_LOCAL` 中止方法；分支賦值前先給預設值。
- 參數個數只有同表編譯期檢查；跨表錯配在執行期以 `AE_AML_UNINITIALIZED_ARG` 爆發。
- Package 是傳遞複合資料的標準容器：`DerefOf (Pkg [i])` 取元素、`Pkg [i] = x` 寫元素、超過七個參數就打包。

動手練習

1. 不看書畫出 Method 的四個組成，並各用一句話說明生命週期。
2. 為什麼兩條執行緒同時執行「只用 Arg 與 Local 的方法」不需要 Serialized？共享的東西到底在哪一格？
3. 寫一個 `SWAP` 方法，用兩個 RefOf 參數交換呼叫端兩個整數變數的值。
4. 下面的方法哪裡會出錯、錯誤碼是什麼：方法宣告兩個參數，呼叫端只給一個，且方法內第一行就是 `Local0 = Arg1`。
5. 動手：把 args.dsl 編譯通過後，故意把 `DEMO` 裡的 `PSUM (VALS)` 改成 `PSUM (VALS, 1)` 再編譯，記下 iasl 的錯誤訊息；再想想若這發生在跨表呼叫會怎樣。

第 14 章

Notify

到目前為止，所有互動都是單向的：OS 呼叫 AML，AML 回答。但電池拔了、上蓋闔了、溫度飆了——這些事 OS 不會自己知道，必須由韌體這一側主動開口。**Notify** 就是 AML 對 OS 說話的唯一正式管道：它把「某裝置發生了某類事件」丟進 OS 的事件佇列，喚起對應的 driver 來處理。ACPI 筆電上幾乎所有「有感」的功能——電量圖示更新、闔蓋睡眠、風扇轉速——背後都是一條 Notify 事件鏈。

本章沿著事件的旅程走一遍：先看事件的源頭 GPE 與 SCI，再看兩種 GPE handler 的差異、Notify 的語法與代碼表，最後用 EC 電池事件把整條鏈串起來，並在 Linux 上實際觀察。

本章學習目標

- 畫出從 GPE 硬體訊號到 OS driver 的完整事件鏈。
- 分辨 `_Lxx` 與 `_Exx` handler 的觸發與清除差異。
- 查表選用正確的 Notify 代碼，說明標準碼與裝置自訂碼的分界。
- 寫出 EC 的 `_Qxx` query 方法，用 Notify 通知電池裝置。
- 在 Linux 上用 acpi_listen 與 GPE 計數器觀察 Notify 事件。

14.1 從同步到非同步：為什麼需要 Notify

第 12 章的 Method 呼叫是**同步**的：OS 出題、AML 當場作答。但硬體事件天生是**非同步**的——使用者何時拔電源線，誰也料不到。ACPI 的解法分兩段：硬體側用中斷（GPE）把 OS 叫醒，AML 側用 Notify 告訴 OS「該找哪個裝置的 driver、發生哪類事」。

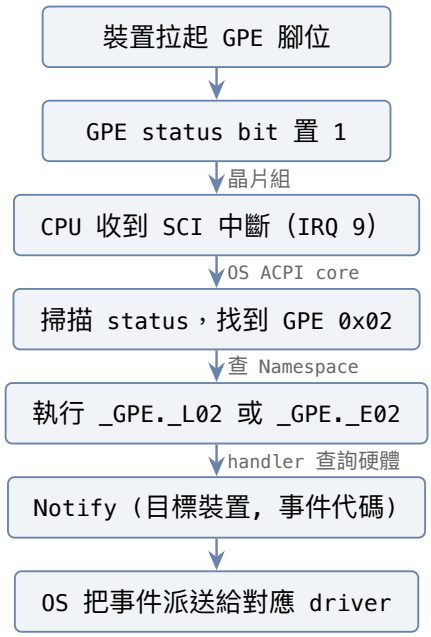
核心觀念：Notify 是 ACPI 世界的「中斷」

Notify 之於 ACPI，就像中斷之於 CPU：非同步、由下往上、只帶極少量資訊。`Notify(裝置, 代碼)` 只送出一個代碼，**不夾帶任何資料**——OS 收到後要自己回頭呼叫該裝置的 Method（如 `_BST`）查詳情。這是典型的「門鈴模型」：門鈴只告訴你有人來，開門看是誰是你的事。這個 pull 設計讓事件本身極輕量，也天然去重：狀態再怎麼連環變化，OS 查到的永遠是最新值。

14.2 事件的源頭：GPE 與 SCI

GPE（General Purpose Event）是晶片組提供的一組「通用事件腳位」，每個 GPE 對應狀態暫存器裡的一個 bit。FADT 描述了 GPE 暫存器區塊（GPE0_BLK 等）的位址，每個 bit 都有 status（發生了沒）與 enable（要不要理它）兩半。任何一個致能中的 GPE status 被硬體置 1，晶片組就對 CPU 發出 **SCI**（System Control Interrupt，傳統上是 IRQ 9）——這是 ACPI 專屬的中斷線。

OS 收到 SCI 後的標準動作是：掃描 GPE status 暫存器找出是哪一個 bit，然後到 Namespace 的 `\_GPE` scope 下找對應的 handler method 執行。整條鏈如下：



注意分工：GPE 只說「第 2 號事件線響了」，至於是哪個裝置、什麼事，由 handler method 裡的 AML 查詢硬體後用 Notify 補完。一條 GPE 線常常由多個裝置共用，handler 的第一件事就是逐一檢查誰在按門鈴。

14.3 Level 與 Edge：_Lxx 與 _Exx

GPE handler 的名字有嚴格格式：`_L` 或 `_E` 加上兩位十六進位的 GPE 編號，放在 `\_GPE` scope 下。字首決定這條 GPE 的觸發模式，而觸發模式決定了 status bit 由誰、在何時清除：

	`_Lxx`（Level）	`_Exx`（Edge）
觸發條件	訊號電位持續為高	訊號出現一次跳變
status 清除	handler 執行完後由 OS 清除	執行 handler 前先清除
適用事件源	可能持續 assert 的來源（EC、thermal）	一次性脈衝（按鈕類）
漏事件風險	低：電位還在就會再觸發	靠「先清再跑」接住執行中的新脈衝

片段：_GPE scope 中的兩種 handler

```
Scope (\_GPE)
{
    // GPE 0x02, level-triggered: PCIe hotplug slot
    Method (_L02, 0)
    {
        // query slot status registers here ...
        Notify (\_SB.PCI0.SLT1, 0x01)  // device check
    }

    // GPE 0x0D, edge-triggered: dock eject button
    Method (_E0D, 0)
    {
        Notify (\_SB.DOCK, 0x03)       // eject request
    }
}
```

同一個 GPE 編號只能有一種字首——`_L02` 與 `_E02` 二選一，由硬體接線與晶片組設定決定，不是你想選哪個就選哪個。韌體工程師的職責是把字首寫得跟硬體一致：把 level 來源寫成 `_Exx`，事件會在 handler 還沒處理完來源時就被清掉狀態，之後電位持續為高卻再也不觸發，症狀是「事件偶爾漏掉」，非常難查。

14.4 Notify 語法與代碼表

語法只有一行，但兩個參數都有講究：

片段：Notify 語法

```
Notify (\_SB.BAT0, 0x80)   // target object, notification value
```

目標必須是 `Device`、`ThermalZone` 或 `Processor` 物件；代碼是一個位元組，空間被規格切成三段：

代碼	名稱	意義
0x00	Bus Check	此點以下的匯流排結構可能變了，請重新列舉
0x01	Device Check	檢查此裝置是否出現 / 消失（熱插拔）
0x02	Device Wake	此裝置剛喚醒了系統（搭配 `_PRW`）
0x03	Eject Request	使用者請求退出此裝置（搭配 `_EJ0`）
0x04	Device Check Light	同 0x01，但不必重掃子樹
0x05–0x0D	（其餘標準碼）	頻率不符、電源故障、PLD 變更等
0x0E–0x7F	保留	規格保留，不可自用
0x80–0xBF	裝置自訂	意義由裝置類別的規格段落定義
0xC0–0xFF	OEM 自訂	硬體廠商自行定義

0x80 起的「裝置自訂」段是日常主力。同一個 0x80，掛在電池上和掛在熱區上意義不同——查表時要先看裝置類別：

裝置類別	代碼	意義（OS 的後續動作）
電池（PNP0C0A）	0x80	狀態變了，重讀 _BST
電池（PNP0C0A）	0x81	資訊變了，重讀 _BIF / _BIX
AC 變壓器（ACPI0003）	0x80	插拔了，重讀 _PSR
上蓋（PNP0C0D）	0x80	開闔了，重讀 _LID
熱區（ThermalZone）	0x80	溫度事件，重讀 _TMP
熱區（ThermalZone）	0x81	trip point 變了，重讀 _PSV 等
Processor	0x80	_PPC 變了，重新評估限速
Processor	0x81	_CST 變了，重讀 C-state 表

注意：兩個經典的 Notify 誤用

誤用保留碼：0x0E–0x7F 是規格保留區，拿來當自訂事件，現在沒事不代表下一版規格不會賦予它新意義，屆時 OS 的反應會完全出乎你意料。自訂事件一律從 0x80 起跳。**事件風暴**：handler 裡 Notify 之後，OS 會回頭呼叫 _BST 這類查詢方法；若查詢方法又碰了會觸發同一條 GPE 的硬體（例如寫 EC 暫存器導致 EC 再拉 SCI），就會形成 Notify → 查詢 → 再觸發的無窮迴圈，症狀是 CPU 一顆核心被 SCI 吃滿。查詢路徑要保持「唯讀」。

14.5 完整範例：EC query 到電池事件鏈

實務上最常見的 Notify 源頭是 **Embedded Controller**。EC 有自己的事件機制：它拉起 SCI 後，OS 的 EC driver 會送 query 命令問它「什麼事」，EC 回一個位元組（如 0x17），OS 便執行 EC 裝置底下名為 _Q17 的方法——_Q 加上 query 值的兩位十六進位，命名規則和 _Lxx 如出一轍。以下是「電池狀態改變」的完整劇本：

ecbat.dsl：EC query 事件到電池 Notify

```
DefinitionBlock ("ecbat.aml", "SSDT", 2, "DEMO", "ECBAT", 0x01)
{
    Scope (\_SB)
    {
        Device (BAT0)
        {
            Name (_HID, EisaId ("PNP0C0A"))
            Name (_UID, 0)

            Method (_STA, 0)
            {
                Return (0x1F)           // present + functioning
            }
```

```
            Method (_BST, 0, Serialized)
            {
                Name (PKG0, Package (4) {0, 0, 0, 0})
                PKG0 [0] = \_SB.EC0.BST0   // state bits
                PKG0 [1] = 0x500           // rate (demo value)
                PKG0 [2] = \_SB.EC0.BRC0   // remaining capacity
                PKG0 [3] = \_SB.EC0.BPV0   // present voltage
                Return (PKG0)
            }
        }

        Device (EC0)
        {
            Name (_HID, EisaId ("PNP0C09"))
            Name (_GPE, 0x17)          // EC SCI uses GPE 0x17

            OperationRegion (ECRM, EmbeddedControl, 0x00, 0xFF)
            Field (ECRM, ByteAcc, Lock, Preserve)
            {
                Offset (0x60),
                BST0, 8,               // battery state bits
                BRC0, 16,              // remaining capacity
                BPV0, 16               // present voltage
            }

            // EC query 0x17: battery status changed
            Method (_Q17, 0)
            {
                Notify (\_SB.BAT0, 0x80)   // status change
            }

            // EC query 0x18: battery inserted or removed
            Method (_Q18, 0)
            {
                Notify (\_SB.BAT0, 0x81)   // info change
            }
        }
    }
}
```

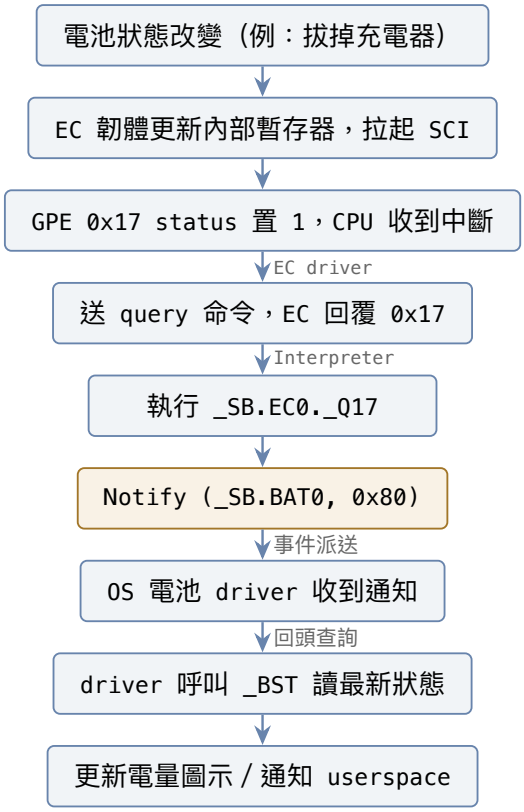
逐行拆解：

- `Name (_GPE, 0x17)`：EC 裝置用這個物件告訴 OS 「我的 SCI 走 GPE 0x17」。OS 的 EC

driver 會接管這條 GPE，所以你不需要也不應該再寫 _L17——query 分派是 EC driver 的內建行為。

- _Q17 是事件的 AML 入口：EC 韌體回報 query 值 0x17 時被執行。它只做一件事——Notify 電池裝置。保持 **_Qxx 輕薄是鐵則**：重活留給 OS 回頭呼叫的查詢方法。
- 兩個 query 用不同代碼：0x80（狀態變了，如電量、充放電）讓 OS 重讀 _BST；0x81（電池本體變了，如拔插電池）讓 OS 重讀 _BIF / _BIX 拿設計容量等靜態資訊。
- _BST 從 EC 的 Field 讀出目前狀態組成 Package 回傳（欄位順序是規格定死的：狀態、放電速率、剩餘容量、電壓）。真實韌體還會配合 _REG 確認 EC region 可用（Module 5），示範版從簡。

整條事件鏈的時序如下——從硬體到電量圖示更新，九站接力：



看清楚金色那站的位置：Notify 是整條鏈的「轉運站」，上游是硬體與 GPE 機制，下游是 OS 的事件派送。AML 在其中只負責兩小段（_Q17 與 _BST），但這兩小段寫錯，整條鏈就斷。

14.6 在 Linux 上觀察 Notify

除錯 Notify 最好的工具是「看得到事件流」。Linux 提供好幾個觀察點，從 GPE 計數器到 userspace 事件一應俱全：

觀察 GPE 觸發與 ACPI 事件

```
$ cat /sys/firmware/acpi/interrupts/sci
```

```
    1337
$ cat /sys/firmware/acpi/interrupts/gpe17
      42   enabled   unmasked

# acpid ships the acpi_listen tool: events in real time
$ acpi_listen
battery PNP0C0A:00 00000080 00000001
ac_adapter ACPI0003:00 00000080 00000000
button/lid PNP0C0D:00 00000080 00000001
```

`acpi_listen` 每行四欄：事件類別、裝置（`_HID` 加 instance）、Notify 代碼、事件資料。拔插電源線、開闔上蓋，就能看到對應的 0x80 事件即時滾出來；對照 `gpe17` 計數器的增長，整條鏈的每一站都可驗證。若懷疑「GPE 有觸發但沒有事件」，比對這兩個數字是第一步——計數器動了但 acpi_listen 沒動靜，問題就出在 AML 這段。

補充說明：acpid 與 netlink：事件怎麼到 userspace

核心把 ACPI 事件透過 **netlink** socket 廣播給 userspace，acpid 只是最常見的訂閱者（另一路是桌面環境經由 udev / D-Bus 接手）。古早的 `/proc/acpi/event` 介面已淘汰。除錯時記住分層：dmesg 看核心與 AML 層、`acpi_listen` 看 netlink 層、桌面行為（如自動睡眠）是最上層——一層層對，很快就能定位斷點。Module 10 會把這套除錯流程展開成完整章節。

核心觀念：本章重點回顧

- 事件鏈五大站：硬體 → GPE / SCI → handler method（`_Lxx` / `_Exx` / `_Qxx`）→ Notify → OS driver。
- `_Lxx` 是 level（跑完才清 status）、`_Exx` 是 edge（先清再跑），字首必須與硬體觸發特性一致。
- Notify 目標限 Device / ThermalZone / Processor；代碼 0x00–0x0D 是標準事件，0x80 起才是裝置自訂，同一代碼在不同裝置類別意義不同。
- Notify 不帶資料，OS 收到後回頭呼叫查詢方法（pull 模型）——所以 `_Qxx` 要輕薄，查詢方法要唯讀。
- Linux 觀察三件套：GPE 計數器、dmesg、`acpi_listen`。

動手練習

1. 依序說出電池事件鏈的九站，並標出哪兩站是 AML 負責的。
2. `_L17` 與 `_E17` 的差別是什麼？若把 EC 這種 level 事件源錯寫成 edge handler，會出現什麼症狀、為什麼難查？
3. 查表回答：電池狀態改變、電池被拔走、使用者按下退出鍵，分別該用哪個 Notify 代碼？

4. Notify 為什麼設計成不帶資料？如果讓它夾帶完整的電池狀態，會產生哪些新問題（提示：事件排隊時狀態又變了怎麼辦）？
5. 動手：在一台 Linux 筆電上執行 `acpi_listen`，拔插電源線並開闔上蓋各一次，記下每一行輸出；再用 `cat /sys/firmware/acpi/interrupts/sci` 前後對照，驗證 SCI 計數的變化。

Module 7

Power Management

難度：★★★★☆

電源管理是 ACPI 誕生的初衷，也是韌體工程師日常最常打交道的領域。本模組由大到小走三層：先攤開整機的 S-states 全景與睡眠喚醒流程，再深入裝置級的 D-states 與 PowerResource 電源軌控制，最後鑽進 CPU 的 P-states、C-states 與熱管理閉環。學完後，你能讀懂並親手撰寫筆電睡眠、裝置斷電、CPU 調頻背後的每一行 ASL。

第 15 章

Power States

第 1 章說過，電源管理是 ACPI 存在的理由；前幾章的 Method 與 Notify 也一直繞著電源事件打轉。本章正式把地圖攤開：G、S、D、C、P 五個字母家族各管什麼、彼此如何嵌套；一台筆電從闔上蓋子到再次喚醒，OS 與韌體之間完整的握手順序是什麼。這張地圖是 Module 7 後兩章、乃至日後所有電源除錯工作的座標系。

不用急著背數字。這一章的每個狀態都會回答同三個問題：**斷了哪些電、保住了什麼 context、醒來要多久**——抓住這三軸，所有表格都能自己推出來。

本章學習目標

- 畫出 G / S / D / C / P 五類電源狀態的階層關係。
- 列出 S0 到 S5 各自的硬體狀態、喚醒延遲與適用場景。
- 分辨 D3hot 與 D3cold，說明為何這是實務除錯的重點。
- 依正確順序排出進入 S3 與喚醒的完整流程（`_TTS`、`_PTS`、`_WAK` 的相對位置）。
- 讀懂 `_S3`、`_S5` 這類 `_Sx` Package 的內容與數值來源。

15.1 電源狀態的全景地圖

ACPI 把「省電」這件事切成五個正交的家族，各自作用在不同的層級：

家族	範圍	管理對象與一句話定位
G-states	G0–G3	整台系統，使用者視角：開著、睡著、關機、斷電
S-states	S0–S5	整台系統，OSPM 視角：G-state 的技術細分
D-states	D0–D3cold	個別裝置：每顆裝置獨立的電源深度
C-states	C0–Cn	CPU 閒置：不跑指令時睡多深
P-states	P0–Pn	CPU 執行：跑指令時用多少頻率與電壓

五個家族的嵌套關係，用一棵樹看最清楚：

電源狀態階層：G 包 S，S0 之下才有 D / C / P 的舞台

```
G-states（全域，使用者看得到的粗分類）
 ├── G0 ── S0：運作中
```

```
│          ├── CPU 閒置深度：C0 ~ Cn（C-states）
│          │     └── C0 執行中的頻率電壓：P0 ~ Pn（P-states）
│          └── 各裝置電源：D0 ~ D3cold（D-states，隨時獨立切換）
├── G1 ── S1 ~ S4：睡眠（S3 = 睡眠，S4 = 休眠）
├── G2 ── S5：軟關機（電源鍵可喚醒）
└── G3：機械斷電（拔插頭，只剩 RTC 電池）
```

兩個重點讀法。第一，**D ／ C ／ P 都是 S0 世界裡的事**：系統醒著時，個別裝置可以睡（D3）、CPU 可以打盹（C6）、頻率可以下探（Pn），這正是現代「待機時間」的主要來源。第二，S 與 G 的對應是固定的：G1 涵蓋 S1 到 S4，G2 就是 S5——講 G-state 給人聽，講 S-state 給程式看。

15.2 S-states：S0 到 S5

S-states 定義整機的睡眠深度。判準就是導言說的三軸——斷了哪些電、保住什麼、醒來多久：

狀態	CPU	RAM	喚醒延遲	說明
S0	執行中	供電	—	正常運作，靠 C ／ P ／ D 省電
S1	停止時脈	供電	近乎即時	Power on Suspend，已絕跡
S2	斷電	供電	極短	幾乎沒有平台實作
S3	斷電	self-refresh	1–3 秒	Suspend to RAM，傳統「睡眠」
S4	斷電	斷電（內容存磁碟）	5–30 秒	Suspend to Disk，「休眠」
S5	斷電	斷電	完整開機	Soft Off，「關機」

工程上真正活躍的只有四個：S0、S3、S4、S5。S3 斷掉 CPU 與大部分晶片組的電，只留 RAM 自我更新與少數喚醒電路，所以喚醒快但怕斷電；S4 把 RAM 內容寫進磁碟後全斷，喚醒慢但拔電池也不怕；S5 連 context 都不留，下次是全新開機。註：S4 與 S5 硬體狀態幾乎相同，差別在「磁碟裡有沒有一份待恢復的記憶體映像」——這解釋了為什麼兩者的喚醒電路與 `_Sx` 數值常常長得很像。

補充說明：S0ix ／ Modern Standby：S3 的接班人

近年輕薄筆電多改走 **S0ix**（Low Power S0 Idle，Windows 稱 Modern Standby，Linux 稱 s2idle 路徑之一）：系統名義上留在 S0，靠平台深度閒置狀態逼近 S3 的功耗，換來「闔蓋仍可收信、開蓋即亮」的體驗。韌體透過 FADT 的 LOW_POWER_S0_IDLE 旗標宣告支援。本書聚焦傳統 S3 流程——它仍是理解一切的基礎，S0ix 的除錯思路建立在同一套物件上。

15.3 D-states：D0 到 D3cold

D-states 是「單一裝置」的電源深度，與整機狀態獨立：S0 下你的相機可以在 D3，S3 下你的網卡可能還醒著等 Wake-on-LAN。

狀態	電源	Context	可喚醒系統	備註
D0	全開	保留	—	正常工作狀態
D1	略降	多半保留	類別定義	少有裝置實作
D2	更低	多半丟失	類別定義	少有裝置實作
D3hot	主電源仍在	丟失	視裝置而定	仍可被匯流排看見
D3cold	完全移除	丟失	需輔助電源	得靠平台重新上電

核心觀念：D3hot 與 D3cold：實務除錯的分水嶺

D3 其實是兩個狀態。**D3hot**：裝置自己深度省電，但主電源軌還在——以 PCIe 為例，config space 仍有回應，OS 隨時能把它叫回 D0。**D3cold**：電源軌整條拔掉，裝置從匯流排上「消失」，只能靠平台（下一章的 PowerResource，經由 `_PR3`）重新上電，再重新列舉。「裝置睡了就叫不醒」「resume 後裝置不見了」這類經典災情，八成出在 D3cold 的上電順序或時序——先記住這條分水嶺，下一章給你修它的工具。

15.4 系統睡眠的完整流程

現在把 S3 的入睡與喚醒流程走一遍。這條流程橫跨 OS、AML 與硬體，每一站都有固定的角色；三個根層級的 Method——`\_TTS`、`\_PTS`、`\_WAK`——是韌體參與的三個掛勾點：



三個掛勾點的時機差異是考點也是雷區：_TTS（Transition To State）在 **driver 還沒動作前**最早被叫，適合「趁裝置都醒著」先知會 EC；_PTS（Prepare To Sleep）在 driver 都收工、OS 即將儲存 CPU context 前執行，做最後的喚醒源設定；最後 OS 從 _S3 Package 取出晶片組定義的 SLP_TYP 值，連同 SLP_EN 位元一起寫進 FADT 描述的 PM1 Control 暫存器——這一寫下去，硬體立刻拉閘，程式碼不會有「下一行」。

喚醒是一條對稱的回程：



值得注意：**S4 與 S5 也走同一套**——關機前 OS 一樣會呼叫 _PTS (5)，所以「關機卡住」有時也是 ASL 的鍋；而喚醒路徑上 _WAK 執行時 driver 都還沒恢復，在裡面碰還沒上電的裝置就是自找麻煩。

15.5 _Sx 物件與 _PTS / _WAK 實作

OS 怎麼知道這台機器支援哪些 S-state、SLP_TYP 該填多少？答案是根層級的 _Sx Package：**宣告了哪個，OS 才提供哪個**。DSDT 沒有 _S3，睡眠選單就不會有睡眠。每個 Package 的前兩個元素是寫進 PM1a / PM1b 控制暫存器的 SLP_TYP 值，數值由晶片組 datasheet 定義，純屬平台常數：

sleep.dsl：_Sx 宣告與 _PTS / _WAK 骨架

```
DefinitionBlock ("sleep.aml", "SSDT", 2, "DEMO", "SLEEP", 0x01)
{
    // Which S-states this platform supports, and the
    // chipset-specific SLP_TYP values for each of them
    Name (\_S0, Package (4) {0x00, 0x00, 0x00, 0x00})
    Name (\_S3, Package (4) {0x05, 0x05, 0x00, 0x00})
    Name (\_S4, Package (4) {0x06, 0x06, 0x00, 0x00})
```

```
    Name (\_S5, Package (4) {0x07, 0x07, 0x00, 0x00})

    OperationRegion (DBG0, SystemIO, 0x80, 0x01)
    Field (DBG0, ByteAcc, NoLock, Preserve)
    {
        P80H, 8                     // POST code debug port
    }

    Method (_TTS, 1)                // earliest / latest hook
    {
        P80H = 0x54                 // log: TTS ran
    }

    Method (_PTS, 1)                // Prepare To Sleep
    {
        P80H = Arg0                 // log target S-state
        If (Arg0 == 3)
        {
            P80H = 0x53             // arm wake sources here
        }
    }

    Method (_WAK, 1)                // Wake handler
    {
        P80H = 0xA0                 // log: resumed
        Return (Package (2) {0x00, 0x00})
    }
}
```

逐行拆解：

- 四個 _Sx Package 宣告本平台支援 S0、S3、S4、S5。以 _S3 為例，0x05 就是這顆晶片組「進 S3」的 SLP_TYP 編碼——換一顆晶片組就是另一組數字，抄錯的下場見下面的警告框。
- _PTS 收到目標 S-state（Arg0），示範用 port 0x80 記 log——寫 POST code 除錯埠是 bring-up 階段追蹤睡眠流程的傳統手藝，邏輯分析儀或除錯卡上看得一清二楚。
- _WAK 規格要求回傳兩個整數的 Package：第一個是狀態碼（0 表示正常），第二個保留。忘記 Return 是新手最常犯的 _WAK 錯誤。
- _TTS 收到的參數同 _PTS；喚醒收尾時還會以 _TTS (0) 再被叫一次，需要區分方向時檢查 Arg0 是否為零。

在 Linux 上驗證平台宣告與實際行為：

觀察系統支援的睡眠狀態

```
$ dmesg | grep "ACPI: PM"
ACPI: PM: (supports S0 S3 S4 S5)
$ cat /sys/power/state
freeze mem disk
$ cat /sys/power/mem_sleep
s2idle [deep]
```

`mem_sleep` 顯示 `mem` 這個入口實際走哪條路：`deep` 就是傳統 S3，`s2idle` 則是前面補充框說的 S0ix 路線——同一台機器可能兩者都支援，以中括號標示目前選擇。

注意：S3 一睡不醒：韌體側的常見禍首

- `_S3` Package 的 SLP_TYP 抄錯 datasheet：一寫 SLP_EN 就直接關機、重開，或什麼都不發生。
- `_PTS` 對 EC 下錯命令、喚醒源沒 arm 好：睡下去就再也叫不醒，只能長按電源鍵。
- `_WAK` 忘記恢復 `_PTS` 改過的硬體狀態：醒來後鍵盤失靈、風扇亂轉、熱鍵失效。
- `_WAK` 沒有 Return Package：某些 OS 會回報 ACPI 錯誤，甚至影響後續睡眠週期。
- 喚醒後 GPE status 沒清乾淨：立刻再觸發喚醒事件，機器「秒醒」或陷入睡了又醒的迴圈。

除錯時先用 port 0x80 或 dmesg 定位「死在哪一站」，再對照本章兩張流程圖，十之八九能鎖定範圍。

核心觀念：本章重點回顧

- 五大家族各司其職：G（使用者視角）、S（整機睡眠深度）、D（單一裝置）、C（CPU 閒置）、P（CPU 效能）；D / C / P 都活在 S0 裡。
- S-state 三軸判讀：斷了哪些電、保住什麼 context、醒來要多久；實務上活躍的是 S0、S3、S4、S5。
- D3hot 主電源仍在、匯流排看得見；D3cold 電源全斷，要靠 PowerResource 重新上電——除錯分水嶺。
- 入睡順序：`_TTS` → driver 收工（`_PSx`）→ `_PTS` → 讀 `_Sx` Package → 寫 SLP_TYP + SLP_EN；喚醒對稱走回來，`_WAK` 在 driver 恢復之前執行。
- `_Sx` Package 宣告平台支援的睡眠狀態，內容是晶片組定義的 SLP_TYP 常數。

動手練習

1. 用「斷電範圍、context、喚醒延遲」三軸，各用一句話描述 S3、S4、S5 的差別。

2. 把下列事件排成正確的入睡順序：寫 SLP_EN、`_PTS (3)`、`_TTS (3)`、driver 儲存 context、讀 `_S3` Package。
3. `_S3` Package 裡的 0x05 是什麼？誰決定這個數字？如果這台板子換了晶片組，這個值會怎樣？
4. 說出 D3hot 與 D3cold 的三個差異，並解釋為什麼「resume 後裝置消失」通常指向 D3cold。
5. 動手：在一台 Linux 機器上執行本章的三個觀察指令，判斷它睡眠時走 S3 還是 s2idle；再執行 `sudo dmesg | grep -i "PM: suspend"` 觀察一次完整 suspend / resume 留下的訊息。

第 16 章

Power Resources

第 15 章的 D-states 定義了裝置「可以睡多深」，但誰去真的拉 GPIO、關穩壓器？答案是 **PowerResource**：ACPI 把每一條電源軌抽象成一個「可開、可關、可查詢」的物件，並讓裝置宣告「我在 D0 需要哪幾條軌」。OS 據此用 **reference counting** 決定何時真正開關——兩顆裝置共用一條軌時，最後一個離開的人才關燈。這是筆電裡攝影機、感測器、SSD 等周邊省電的實作核心，也是 D3cold 能不能正確醒來的關鍵。

本章從 PowerResource 的宣告語法開始，走過 `_ON` / `_OFF` / `_STA` 三件套、reference counting 的運作、裝置端的 `_PR0` 到 `_PR3` 掛勾，最後以「GPIO 控制的相機電源軌」完整範例收尾。

本章學習目標

- 寫出 PowerResource 宣告並解釋 SystemLevel 與 ResourceOrder 兩個參數。
- 實作 `_ON`、`_OFF`、`_STA` 三件套，遵守各自的回傳值規約。
- 說明 OSPM 的 reference counting 如何決定電源軌實際開關的時機。
- 用 `_PR0` 到 `_PR3` 把裝置需求掛上資源，說明 `_PSx` 與 PowerResource 的分工與執行順序。
- 寫出 `_PRW` 宣告喚醒能力，說明 `_DSW` 的角色。

16.1 PowerResource 物件：宣告與參數

PowerResource 的宣告長這樣（本體裡面放的是 Method，所以它其實是一種「帶電源語意的迷你 Device」）：

片段：PowerResource 宣告骨架

```
PowerResource (PWRC, 0x00, 0x0000)
{
    Method (_STA, 0) { Return (0x01) }   // is it on?
    Method (_ON,  0) { /* switch on  */ }
    Method (_OFF, 0) { /* switch off */ }
}
```

三個參數的意義：

參數	型別	意義
名稱	NameSeg	資源名稱，一般命名規則
SystemLevel	整數	此資源保持 ON 時，系統最深只能睡到的 S-state（0 表示只在 S0 有效）
ResourceOrder	16 位元整數	全系統唯一的順序編號，決定多條資源的開關次序

SystemLevel 把資源和 S-states 綁在一起：一條 SystemLevel 為 0 的軌，系統要進 S1 以下就必須先關掉它；宣告成 3，則它可以陪系統睡到 S3。ResourceOrder 解決相依問題——OSPM 開電時從小號碼開到大號碼，關電時反向。時脈源、上游穩壓器給小號碼，吃電的下游給大號碼，「先有電、再有時脈、再開裝置」的順序就有了保障。

補充說明：ResourceOrder 為什麼必須全域唯一

OSPM 面對「同時要開好幾條軌」的場景（例如喚醒時一口氣恢復多顆裝置）時，唯一的排序依據就是 ResourceOrder。兩條資源撞號，開關次序就成了未定義行為——今天能開機不代表下一版 OS 還照同樣順序。習慣做法：以 0x10 為間隔編號（0x00、0x10、0x20……），日後要在中間插入新資源不必全體改號。

16.2 三件套：_ON、_OFF、_STA

每個 PowerResource 必須提供三個零參數的 Method，規約如下：

- `_ON`：把資源打開。返回時資源必須**已可用**——電壓要穩、時脈要鎖定，必要的 ramp-up 延遲（`Sleep` / `Stall`）要做在這裡面。
- `_OFF`：把資源關閉。同樣要等關閉序列完成才返回。
- `_STA`：回報目前狀態，**只能回 0（關）或 1（開）**。OS 開機時靠它同步初始狀態，避免多此一舉的開關。

注意：三件套的三個經典錯誤

- **`_STA` 回傳 0x0F**：那是 Device 版 `_STA` 的 bitmask 格式。PowerResource 版只認 0 與 1，回 0x0F 在某些 OS 上會被解讀成非零即開，另一些則行為未定義——別賭。
- **`_ON` 立刻返回**：GPIO 拉高不等於電壓站穩。少了 ramp-up 等待，下游裝置在半路電壓下初始化，症狀是「偶爾」讀不到裝置——機率性 bug 最難查。
- **繞過 OSPM 私自開關**：在某個 `_PS3` 裡「順手」把共用軌關掉，等於無視 reference counting，另一顆還在用的裝置直接陪葬。開關永遠交給 OSPM 決定。

16.3 OSPM 的 reference counting：何時真的關電

裝置宣告需求（下一節的 `_PRx`），OSPM 替每條資源記帳：

- 有裝置進入「需要這條資源」的 D-state：計數加一；**0 變 1 的瞬間**呼叫 `_ON`。
- 裝置離開該 D-state：計數減一；**歸零的瞬間**呼叫 `_OFF`。
- 計數在中間變動（1 變 2、2 變 1）：什麼都不呼叫。

核心觀念：AML 寫「怎麼開關」，OSPM 決定「何時開關」

PowerResource 的 `_ON` / `_OFF` 是機制，reference counting 是政策——政策永遠在 OS 手上。你的 AML 不需要、也不應該自己判斷「還有沒有別人在用」：只要每顆裝置誠實申報 `_PRx`，共用軌的開關時機自然正確。這是 OSPM「描述與決策分離」哲學在電源領域的完整重演。

16.4 裝置端的掛勾：_PR0 到 _PR3

裝置用四個 Package 物件申報「各個 D-state 需要哪些資源」：

物件	對應狀態	內容
`_PR0`	D0	進入 / 停留 D0 所需的資源清單
`_PR1`	D1	同上，對應 D1
`_PR2`	D2	同上，對應 D2
`_PR3`	D3hot	D3hot 所需；**全部關掉即 D3cold**

`_PR3` 那行請畫線：D3hot 與 D3cold 的差別，在 ASL 層面就是「`_PR3` 列的資源開著還是關了」。OS 想把裝置送進 D3cold，就把 `_PR3` 資源的計數也釋放掉；資源歸零斷電，裝置從匯流排上消失——第 15 章那條除錯分水嶺，機關全在這個 Package 裡。

16.5 _PSx 與 PowerResource 的分工

裝置端還有一組 `_PS0` 到 `_PS3` Method（第 8 章出現過），和 PowerResource 是分工而非競爭的關係：

- **PowerResource**：管「可能共用」的平台資源——電源軌、時脈、reset 線。誰用都一樣，所以抽出來共管。
- **`_PSx`**：管「這顆裝置自己」的進出手續——儲存暫存器、通知 EC、設定內部省電模式。
- **順序是規格定死的**：進 D0 時先把資源開好（`_PR0` 全 ON）再叫 `_PS0`；進 D3 時先叫 `_PS3` 再放掉資源。口訣：**有電才辦事，辦完事才斷電**。

16.6 喚醒能力：_PRW、_DSW 與 _PSW

有些裝置睡了還要能叫醒系統（網卡的 Wake-on-LAN、USB 鍵盤喚醒）。裝置用 `_PRW` 申報喚醒能力：Package 的第一個元素是喚醒用的 GPE 編號（第 14 章的那些 GPE），第二個是「還能喚醒系統」的最深 S-state，之後可以再列喚醒期間必須維持的 PowerResource：

片段：宣告喚醒能力

```
Device (XHC0)                       // USB host controller
{
    Name (_HID, "DEMO0D10")
    Name (_PRW, Package (2) {0x0D, 3})   // GPE 0x0D, up to S3

    Method (_DSW, 3)
    {
        // Arg0: enable (1) / disable (0) wake
        // Arg1: target system S-state
        // Arg2: target device D-state
        // program EC / GPIO wake latches here
    }
}
```

搭配的動詞是 _DSW（Device Sleep Wake）：OS 在裝置入睡前呼叫它，告知「要不要武裝喚醒、系統要去哪、你會睡多深」，AML 據此設定硬體的喚醒鎖存。老式的 _PSW（一個參數，只管開關）已被 _DSW 取代，但舊表裡仍常見——兩者並存時 OS 用 _DSW。喚醒真的發生時，韌體或 handler 會對裝置 Notify 0x02（第 14 章的 Device Wake），告訴 OS 是誰按的鈴。

16.7 完整範例：GPIO 電源軌與兩顆相機

把整章組裝起來：一條由 memory-mapped GPIO 控制的 3.3V 軌 PCAM，供兩顆相機感測器共用。這是平板、筆電上如假包換的真實場景：

pwr.dsl：共用電源軌的完整宣告

```
DefinitionBlock ("pwr.aml", "SSDT", 2, "DEMO", "PWRRAIL", 0x01)
{
    Scope (\_SB)
    {
        // Memory-mapped GPIO controller; bit 3 of the
        // register at offset 0x40 drives the camera rail
        OperationRegion (GPIO, SystemMemory, 0xFED80000, 0x100)
        Field (GPIO, DWordAcc, NoLock, Preserve)
        {
            Offset (0x40),
            ,    3,             // bits 0-2: other pins
            GP03, 1             // bit 3: camera rail enable
        }

        PowerResource (PCAM, 0x00, 0x0010)
        {
            Method (_STA, 0)
```

```
            {
                Return (GP03)       // 1-bit field: 0 or 1
            }

            Method (_ON, 0)
            {
                GP03 = 1
                Sleep (10)          // rail ramp-up time
            }

            Method (_OFF, 0)
            {
                GP03 = 0
            }
        }

        Device (CAM0)               // world-facing camera
        {
            Name (_HID, "DEMO0C01")
            Name (_PR0, Package (1) {PCAM})

            Method (_PS0, 0)
            {
                Sleep (5)           // sensor init delay
            }

            Method (_PS3, 0)
            {
                // save sensor context here if needed
            }
        }

        Device (CAM1)               // user-facing camera
        {
            Name (_HID, "DEMO0C02")
            Name (_PR0, Package (1) {PCAM})
        }
    }
}
```

逐行拆解：

- GPIO 暫存器用 Module 5 的 OperationRegion + Field 描述，`GP03` 恰好 1 位元寬，讓 `_STA` 直接 `Return (GP03)` 就滿足「只回 0 或 1」的規約——順手的欄位切法能省掉一堆遮罩程

式碼。

- `_ON` 拉高 GPIO 後 `Sleep (10)` 等電壓站穩，對應穩壓器 datasheet 的 ramp-up 時間；`_OFF` 通常不必等。
- 兩顆相機的 `_PR0` 都列 `PCAM`：這一行就是 reference counting 的報名表。
- `CAM0` 另有 `_PS0` / `_PS3` 做感測器自己的手續——注意它們與 `_ON` / `_OFF` 的相對順序由 OSPM 保證，AML 不必自己協調。

兩顆相機先後使用時，OSPM 的記帳過程如下：



在 Linux 上可以直接觀察裝置目前的 D-state，驗證這套機制真的在動：

觀察 ACPI 裝置的電源狀態

```
$ ls /sys/bus/acpi/devices/ | grep DEMO
DEMO0C01:00
DEMO0C02:00
$ cat /sys/bus/acpi/devices/DEMO0C01:00/power_state
D0
```

核心觀念：本章重點回顧

- PowerResource 把電源軌抽象成物件；SystemLevel 定義「開著時系統最深睡到哪」，ResourceOrder 決定多資源開關次序（開小到大、關大到小，全域唯一）。
- 三件套規約：`_ON` 返回前資源必須可用、`_OFF` 完整關閉、`_STA` 只回 0 或 1。
- OSPM reference counting：0 變 1 才 `_ON`、歸零才 `_OFF`；AML 寫機制，OS 定政策。
- `_PRx` 申報各 D-state 的資源需求；`_PR3` 全關即 D3cold。順序口訣：有電才辦事（先資源後 `_PS0`），辦完事才斷電（先 `_PS3` 後資源）。

- 喚醒鏈：`_PRW` 申報 GPE 與最深 S-state，`_DSW` 武裝喚醒，事件發生後 Notify 0x02。

動手練習

1. 說明 PowerResource 三個參數的意義；SystemLevel 為 0 與為 3 的資源，行為差在哪？
2. 兩顆裝置共用一條軌，依序發生：A 進 D0、B 進 D0、A 進 D3、B 進 D3。寫出每一步的計數值與被呼叫的 `_ON` / `_OFF`。
3. 進入 D0 時，`_PR0` 資源的 `_ON` 和裝置的 `_PS0` 誰先執行？為什麼這個順序不能反過來？
4. `Name (_PRW, Package (2) {0x0D, 3})` 的兩個元素各是什麼意思？這顆裝置在 S4 下能喚醒系統嗎？
5. 動手：仿照 pwr.dsl 寫一條由 SystemIO GPIO 控制的軌，讓三顆裝置共用，其中一顆還宣告 `_PR3`；用 iasl 編譯通過，並畫出三顆裝置輪流進出 D0 時的計數時序圖。

第 17 章

CPU Power Management

就算整台系統穩穩待在 S0，電源管理也一刻沒停：CPU 每一毫秒都在「跑多快」與「睡多深」之間游走——前者是 **P-states**（Performance），後者是 **C-states**（Idle）。韌體用一組 ACPI 物件把「這顆 CPU 有哪些檔位、怎麼切換」告訴 OS；OS 的 governor 再依負載即時決策。

本章依序拆解 `_PSS` 效能表、`_CST` 閒置表、`_PSD` 多核協調；再看現代平台（HWP / `_CPC`）的衝擊；最後用 ThermalZone 接上溫度保護閉環——熱管理正是 P-state 限速最大的客戶。

本章學習目標

- 分辨 P-states、C-states 與 T-states 各自控制什麼。
- 寫出 legacy Processor 與現代 Device 兩種 CPU 物件宣告。
- 逐欄位解讀並撰寫 `_PSS`、`_PCT`、`_PPC`。
- 逐欄位解讀並撰寫 `_CST`，說明與 FADT P_LVL 的關係。
- 說明 `_PSD` 的 SW_ALL / SW_ANY / HW_ALL 協調模式。
- 寫出含 `_TMP`、`_PSV`、`_CRT`、`_AC0` 的 ThermalZone。

17.1 兩條軸線：跑多快與睡多深

先把三個字母釘在座標上：

家族	控制什麼	決策者與範圍
P-states	執行時的頻率與電壓	cpufreq governor；P0 最快，數字越大越慢越省
C-states	閒置時睡多深	cpuidle governor；C0 執行中，數字越大越省、喚醒越慢
T-states	時脈節流（強制打拍）	thermal 子系統；過熱急救用，現代平台罕見

P 與 C 的關係一句話講完：**P-states 只在 C0 有意義**——CPU 得先醒著才談跑多快。T-states 用「跳過時脈週期」硬降溫，效率遠差於降 P-state，帶過即可。

核心觀念：Race to idle：兩條軸的實戰哲學

現代省電策略常是「用高 P-state 火速做完，鑽進深 C-state 睡久一點」（race to idle），而不是慢吞吞跑低頻。所以 C-state 的品質（進得深、醒得快）往往比 P-state 檔位多寡更影響續

航——韌體把 _CST 寫壞時，使用者的第一個抱怨是「電池變差了」。

17.2 Processor 物件：兩代寫法

CPU 在 Namespace 裡也是裝置，但歷史留下兩種宣告。舊式用專屬的 Processor 關鍵字，參數依序是名稱、ProcID（對應 MADT 的編號）、P_BLK 埠位址與長度；ACPI 6.4 起這個關鍵字正式棄用，新式寫法就是普通的 Device 加上 _HID 為 ACPI0007：

片段：兩代 CPU 物件宣告（擇一使用）

```
// Legacy keyword, deprecated since ACPI 6.4 (iasl warns):
Processor (CPU0, 0x01, 0x00000410, 0x06) { }

// Modern declaration, a plain Device:
Device (CPU0)
{
    Name (_HID, "ACPI0007")    // processor device ID
    Name (_UID, 0)             // must match MADT entry
}
```

不論哪種殼，_PSS、_CST 這些電源物件都放在它的 scope 底下。_UID（或舊式的 ProcID）是 OS 把物件對應到實體核心的鑰匙；每顆邏輯 CPU 都要有自己的一份宣告，所以這類表格幾乎都由 BIOS 開機時動態生成 SSDT（Module 8 實戰）。

17.3 P-states：_PSS、_PCT 與 _PPC

三件套分工：_PSS 列出檔位選單、_PCT 指出換檔暫存器、_PPC 動態限制可用上限。_PSS 是「Package 的 Package」，每個內層六個欄位描述一個 P-state：

欄位	單位	意義
CoreFrequency	MHz	此檔位的核心頻率
Power	mW	此檔位的典型功耗
Latency	μs	切換到此檔位的最壞延遲
BusMasterLatency	μs	bus master 存取恢復的最壞延遲
Control	整數	寫入控制暫存器以「要求換檔」的值
Status	整數	從狀態暫存器讀回以「確認到檔」的值

完整範例——一顆有三個檔位的 CPU，換檔走原生 MSR：

cpupm.dsl：三個 P-state 的完整宣告

```
DefinitionBlock ("cpupm.aml", "SSDT", 2, "DEMO", "CPUPM", 0x01)
{
```

```
    Scope (\_SB)
    {
        Name (PPCV, 0)              // current _PPC limit

        Device (CPU0)
        {
            Name (_HID, "ACPI0007")
            Name (_UID, 0)

            // P-state control/status registers; FFixedHW
            // means: use the native MSR mechanism
            Name (_PCT, Package (2)
            {
                ResourceTemplate () { Register (FFixedHW, 0, 0, 0) },
                ResourceTemplate () { Register (FFixedHW, 0, 0, 0) }
            })

            // MHz    mW   Lat BMLat Control Status
            Name (_PSS, Package (3)
            {
                Package (6) {2400, 35000, 10, 10, 0x1800, 0x1800},
                Package (6) {1800, 22000, 10, 10, 0x1200, 0x1200},
                Package (6) {1200, 12000, 10, 10, 0x0C00, 0x0C00}
            })

            Method (_PPC, 0)
            {
                Return (PPCV)       // index of highest allowed
            }
        }
    }
}
```

逐行拆解：

- `_PCT` 的兩個 Register 分別是 PERF_CTL 與 PERF_STATUS。`FFixedHW` 意為「功能固定硬體」：OS 認得這家 CPU 的原生機制（x86 上是 MSR），不必給位址。
- `_PSS` 檔位由快到慢排列（規格要求 P0 在前）。Control 值是「寫進暫存器的位元樣式」，來自 CPU datasheet；Status 用來輪詢確認換檔完成。
- `_PPC` 回傳「目前允許的最高檔位索引」：0 表示全部開放，1 表示 P0 禁用，以此類推。

`_PPC` 的威力在「動態」。典型劇本是拔掉變壓器改用電池：EC 的 query 方法（例如 `_Q21`）把 `PPCV` 改成 1，再對 CPU 物件發 Notify 0x80（第 14 章的 Processor 專用碼），OS 便重讀 `_PPC`、

即刻禁用 P0——不改表、不重開機，限速就生效了。過熱降頻（thermal throttling）走的也是同一條路。

17.4 C-states：_CST 與它的歷史

C-state 的宣告走 `_CST`：外層第一個元素是狀態個數，之後每個內層 Package 四個欄位描述一個 C-state：

欄位	型別	意義
Register	ResourceTemplate	進入方式：FFixedHW（原生指令）或 SystemIO（讀 I/O 埠）
Type	1–3	語意等級：C1 / C2 / C3
Latency	μs	最壞喚醒延遲
Power	mW	停留此狀態的功耗

片段：_CST 宣告 C1 與 C3（放進 CPU 裝置內）

```
Name (_CST, Package (3)
{
    2,                          // number of C-states
    Package (4)
    {
        ResourceTemplate () { Register (FFixedHW, 1, 2, 0) },
        1,                      // type C1: native halt
        1,                      // latency: 1 us
        1000                    // power: 1000 mW
    },
    Package (4)
    {
        ResourceTemplate () { Register (SystemIO, 8, 0, 0x415) },
        3,                      // type C3
        200,                    // latency: 200 us
        250                     // power: 250 mW
    }
})
```

讀法：C1 用 `FFixedHW`——x86 上就是 HLT / MWAIT 原生指令；C2 / C3 用 `SystemIO`，OS 讀一下指定的 I/O 埠，晶片組就把 CPU 送入對應深度（範例列 C1 與 C3，C2 同理）。Type 告訴 OS 語意差異（C3 起快取不再 snoop），Latency 讓 governor 權衡值不值得睡這麼深。

一點歷史：`_CST` 之前，C-state 寫死在 FADT 的 P_BLK 裡——固定的 P_LVL2 / P_LVL3 埠、最多三個狀態、延遲是死數字。`_CST` 把這一切搬進 Namespace：數量不限、暫存器任意，還可以動態換——韌體發 Notify 0x81，OS 就重讀（接電源開放深睡、用電池收淺）。範例的 0x415 正是傳統 P_LVL3 的位址，兩代一脈相承。

17.5 多核協調：_PSD、_CSD 與 _TSD

兩顆核心共用電壓平面時，一顆想降頻、另一顆還在全速，誰說了算？_PSD（P-state Dependency）就是申報書：告訴 OS 哪些 CPU 同屬一個相依域、用哪種協調方式。Package 五欄位依序是 NumEntries（固定 5）、Revision（0）、Domain、CoordType、NumProcessors：

CoordType	名稱	意義
0xFC	SW_ALL	域內每一顆 CPU 都得執行切換動作，缺一不可
0xFD	SW_ANY	域內任一顆 CPU 執行切換，即對全域生效
0xFE	HW_ALL	硬體自行協調；各 CPU 只管申報自己的需求

片段：宣告兩顆核心共用一個 P-state 域

```
Name (_PSD, Package (1)
{
    Package (5) {5, 0, 0, 0xFC, 2}   // domain 0, SW_ALL, 2 CPUs
})
```

_CSD（C-state）與 _TSD（T-state）格式如出一轍。實務經驗：_PSD 寫錯域或協調型別，常見症狀是「頻率卡死最低檔」或「一顆核心降頻全機陪跑」——禍首在韌體表。

17.6 現代平台的現實：HWP 與 _CPC

讀到這裡你可能會問：我的筆電真的照 _PSS 換檔嗎？誠實的答案是——大概不是。

補充說明：intel_pstate、HWP 與韌體工程師的日常

現代 Intel 平台（Skylake 起）支援 **HWP**（Hardware P-states）：OS 只給效能區間提示，檔位由硬體自主微調，Linux 的 `intel_pstate` 走這條路時整份 **_PSS 基本被繞過**；AMD 與 ARM 則多用 _CPC（CPPC）連續效能控制。但別急著把本章丟掉：_CST 與 thermal 這半邊仍然活躍、大量既有平台與虛擬機仍靠 _PSS，而 BIOS 每天都在動態生成含這些物件的 CPU PM SSDT——讀懂寫對它們仍是韌體日常（Module 8 實戰）。

自己的平台走哪條路，在 Linux 上一查便知：`cpufreq` 目錄下的 `scaling_driver` 若顯示 `intel_pstate` 就是 HWP 路線，顯示 `acpi-cpufreq` 表示 OS 正在用你的 _PSS；而 `cpuidle` 的狀態名稱若帶 _ACPI 字尾（如 `C3_ACPI`），代表 _CST 被採用了。

17.7 Thermal：溫度閉環

CPU 電源管理的最後一塊拼圖是熱。ACPI 用 **ThermalZone** 物件描述一個溫度管理區，把感測、門檻與對策接成閉環。核心物件一覽：

物件	型別	意義
_TMP	Method	回傳目前溫度，單位 0.1 K
_PSV	溫度值	passive trip：超過就對 _PSL 清單裡的 CPU 降頻
_TC1 / _TC2 / _TSP	整數	passive 降頻演算法的係數與取樣週期
_AC0	溫度值	active trip：超過就開第 0 級風扇
_AL0	Package	第 0 級主動散熱裝置（風扇）清單
_CRT	溫度值	critical trip：立即關機保護

（表外還有 _HOT——進 S4 的門檻，與 _TZP ——輪詢週期，實務較少調整。）溫度單位是本節的頭號陷阱：**0.1 K（十分之一凱氏度）**。換算式：數值 = （攝氏 + 273.15）× 10，所以 3032 即 30.05 °C，95 °C 的 trip point 要寫 3682。完整範例——含被動與主動散熱的熱區，加上配套風扇：

thermal.dsl：完整的 ThermalZone 宣告

```
DefinitionBlock ("thermal.aml", "SSDT", 2, "DEMO", "THERMAL", 0x01)
{
    External (\_SB.CPU0, DeviceObj)    // CPU declared in DSDT

    Scope (\_SB)
    {
        PowerResource (FNP0, 0x00, 0x0020)
        {
            Name (FANS, 0)          // demo: rail state variable
            Method (_STA, 0) { Return (FANS) }
            Method (_ON,  0) { FANS = 1 }  // real: GPIO write
            Method (_OFF, 0) { FANS = 0 }
        }

        Device (FAN0)
        {
            Name (_HID, EisaId ("PNP0C0B"))
            Name (_UID, 0)
            Name (_PR0, Package (1) {FNP0})
        }
    }

    Scope (\_TZ)
    {
        ThermalZone (TZ00)
        {
            Name (TMPV, 3032)       // demo sensor: 303.2 K
                                    // real code reads EC field
```

```
            Method (_TMP, 0) { Return (TMPV) }

            Name (_PSL, Package (1) {\_SB.CPU0})
            Name (_TC1, 2)          // passive loop constants
            Name (_TC2, 3)
            Name (_TSP, 100)        // sample every 10 s
            Method (_PSV, 0) { Return (3682) }  // 95 C

            Method (_AC0, 0) { Return (3332) }  // 60 C: fan on
            Name (_AL0, Package (1) {\_SB.FAN0})

            Method (_CRT, 0) { Return (3782) }  // 105 C: off
        }
    }
}
```

逐行拆解：

- 風扇是「披著 Device 外皮的 PowerResource 客戶」：`FAN0`（標準 `_HID` 為 PNP0C0B）掛著 `_PR0`；OS 要開風扇就把它拉到 D0，reference counting 順勢呼叫 `FNP0._ON` ——上一章的機制無縫重用。
- `_PSV` + `_PSL`：過 95 °C 就對清單裡的 CPU 被動降頻，`_TC1` / `_TC2` / `_TSP` 是回授迴圈的係數與週期。
- `_AC0` + `_AL0`：60 °C 開風扇；多級風扇再加 `_AC1` / `_AL1`，數字越小越熱。`_CRT` 則是 105 °C 的保命關機線。
- 通知沿用第 14 章：EC 跨門檻時 `_Qxx` 對熱區 Notify 0x80，OS 重讀 `_TMP`；改 trip point 則 Notify 0x81。

事件驅動的完整閉環如下：



注意：熱區的三個經典災難

- 單位寫錯：把攝氏 95 直接填進 `_PSV` ——那表示 9.5 K，比絕對零度暖不了多少，OS 可能永遠 throttle 或直接忽略這個熱區。
- **`_CRT` 抄錯或設太低**：一到夏天就「莫名其妙關機」，dmesg 其實寫著 thermal critical，但沒人想到是 DSDT 的一個數字。
- **`_TMP` 讀了沒 `_REG` 保護的 EC 欄位**：開機早期讀值全錯，風扇全速或誤關機——Module 5 的 `_REG` 在這裡救命。

核心觀念：本章重點回顧

- 兩條軸線：P-states 管 C0 內跑多快，C-states 管閒置睡多深；race to idle 是現代策略主旋律。
- CPU 物件兩代寫法：legacy `Processor`（已棄用）與 `Device + ACPI0007`；`_UID` 對應 MADT。
- P-state 三件套：`_PSS` 檔位表、`_PCT` 換檔暫存器、`_PPC` 動態上限（配 Notify 0x80）。
- `_CST` 取代 FADT P_LVL 埠：數量任意、可動態更新（Notify 0x81）；`_PSD` 用 SW_ALL / SW_ANY / HW_ALL 宣告多核協調。
- ThermalZone 閉環：`_TMP` 感測（0.1 K !）、`_PSV` 降頻、`_ACx` 風扇、`_CRT` 關機。

動手練習

1. 用一句話分別說明 P-state、C-state、T-state 控制什麼；為什麼說「P-state 只在 C0 有意義」？
2. 寫出 `_PSS` 六個欄位的名稱與單位；並回答：`_CST` 裡 C1 為什麼用 `FFixedHW` 而 C2 / C3 用 `SystemIO`？0x415 這個位址歷史上叫什麼名字？
3. 四顆核心兩兩共用電壓平面，寫出兩個域的 `_PSD` 宣告（SW_ALL），並說明誤寫成一個域會發生什麼事。
4. 動手：把 thermal.dsl 的 `_PSV` 改成 45 °C （先換算 0.1 K），加上 `_AC1` / `_AL1` 第二級風扇（需第二個 PowerResource 與 Device），用 iasl 編譯通過；再在 Linux 上用 `cat /sys/class/thermal/thermal_zone*/type` 找出哪個 zone 是 `acpitz`。

Module 8

DSDT / SSDT 實戰

難度：★★★★☆

前面的模組把 ASL 語言與裝置描述練熟了；本模組把鏡頭拉遠，回到這些程式碼實際居住的兩張表——DSDT 與 SSDT。你將學會逐欄位讀懂表頭、拆解一份真實筆電 DSDT 的佈局、理解多張表如何合併成同一棵 Namespace，最後完成本書第一個完整實戰：取出自己機器的 DSDT、修改、重編、用 initrd 蓋回去並開機驗證。

第 18 章

DSDT

第 2、3 章說過：OS 開機時解譯 AML、建立 ACPI Namespace。那 AML 究竟住在哪裡？答案的主體就是 **DSDT**（Differentiated System Description Table）——每台機器都有、也只有一份的「硬體描述主檔」。你在前七個模組寫的每一個 Device、OperationRegion、Method，在真實產品中幾乎都躺在 DSDT （或它的延伸 SSDT）裡。

本章把 DSDT 當成一份文件來讀：它從哪裡來（FADT）、表頭說了什麼、內文如何組織、OS 什麼時候讀它。與其背規格，不如把你自己筆電的 DSDT 倒出來對照著讀——本章每個 shell 範例都能直接在 Linux 機器上重現。

本章學習目標

- 說明 OS 如何經由 FADT 的 `DSDT` 與 `X_DSDT` 欄位找到 DSDT，以及「每系統唯一、開機必載、不可卸載」的意義。
- 逐欄位解讀 36 位元組的 ACPI 表頭，並說明 Revision 欄位如何決定 Integer 寬度。
- 畫出一份典型筆電 DSDT 的頂層佈局（`\_SB.PCI0`、LPCB、EC0、`\_TZ`……）並說明 OEM 常見的組織方式。
- 說明 OS 建立 Namespace 時 DSDT 與 SSDT 的載入順序。
- 用 `_HID` 關鍵字策略，在幾千行的 DSDT 裡快速定位電池、EC 與 LID。

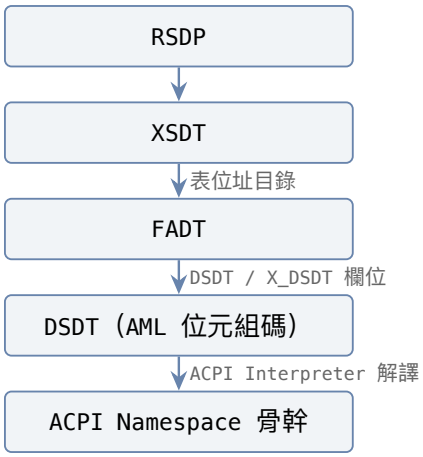
18.1 DSDT 的定位：每系統唯一、開機必載

ACPI 把「系統描述」分成兩塊。固定的一塊由 FADT 直接用欄位描述：PM timer 在哪個 I/O port、電源按鈕暫存器長什麼樣——這些是死的資料，不需要程式碼。可變的一塊則交給 AML：主機板上有哪些裝置、怎麼開關電源、EC 暫存器如何存取。DSDT 就是這塊可變描述的主體，規格把它歸類為 system fixed description 的一部分——意思是：它不是可選配件，而是與 FADT 綁定、開機時「必定存在、必定載入」的表。

OS 找到 DSDT 的路徑很短：RSDP 指向 XSDT，XSDT 列出 FADT，而 FADT 裡有兩個欄位直接指向 DSDT 的實體位址：

FADT 欄位	位移	寬度	說明
`DSDT`	40	4 bytes	DSDT 的 32-bit 實體位址（傳統欄位）
`X_DSDT`	140	8 bytes	DSDT 的 64-bit 實體位址；非零時以此為準、忽略 `DSDT` 欄位

整條尋找路徑與後續動作如下：



核心觀念：DSDT 的三個「一定」

一定只有一份：一個系統只有一張 DSDT，簽章固定為 `DSDT`；一定載入：OS 初始化 ACPI 時第一張解譯的 AML 表就是它，沒有它就沒有 Namespace；一定不卸載：DSDT 沒有卸載介面，從開機活到關機。相對地，SSDT（第 19 章）可以有多張、可以動態載入——它們是 DSDT 的延伸，而不是替代。

「不可卸載」有個實務推論：想「換掉」DSDT，唯一的機會是在 OS 讀它 之前動手——這正是第 20 章 override 實戰的原理。

18.2 36 位元組表頭：逐欄位解讀

所有標準 ACPI 表共用同一個 36 位元組表頭，DSDT 也不例外。把表頭讀熟，之後除錯、override、看 dmesg 都會反覆用到：

位移	欄位	長度	說明
0	Signature	4	固定為 `DSDT`
4	Length	4	整張表的長度（含表頭），單位 byte
8	Revision	1	決定 **Integer** 寬度：小於 2 為 32-bit，2 以上為 64-bit
9	Checksum	1	使全表位元組總和 mod 256 為 0；iasl 重編時自動計算
10	OEMID	6	OEM 名稱，如 `LENOVO`
16	OEM Table ID	8	平台／專案代號，如 `CB-01`
24	OEM Revision	4	OEM 自訂版本號——第 20 章 override 成敗的關鍵欄位
28	Creator ID	4	產生工具的 ID，iasl 填 `INTL`
32	Creator Revision	4	產生工具版本，如 `0x20240927`

直接看一份真實表頭。Linux 把每張表原封不動地曝露在 `/sys/firmware/acpi/tables/` 底下：

用 xxd 讀 DSDT 表頭（Linux）

```
$ sudo xxd -l 36 /sys/firmware/acpi/tables/DSDT
00000000: 4453 4454 3fcd 0200 02f4 4c45 4e4f 564f  DSDT?.....LENOVO
00000010: 4342 2d30 3120 2020 0200 0000 494e 544c  CB-01   ....INTL
00000020: 2709 2420                                '.$
```

逐段對照：開頭的 `44 53 44 54` 是簽章 `DSDT`。接著 `3f cd 02 00` 是 little-endian 的長度 `0x0002CD3F`（約 183 KB）；`02` 是 Revision 2（64-bit Integer）；`f4` 是 checksum；再來依序是 OEMID、OEM Table ID、OEM Revision `0x00000002`、Creator ID `INTL` 與 Creator Revision `0x20240927`（iasl 版本以日期命名）。

開機時 kernel 也會把同一份表頭印成一行 dmesg，欄位一一對得上：

dmesg 中的 DSDT 表頭（節錄）

```
$ sudo dmesg | grep 'ACPI: DSDT'
ACPI: DSDT 0xAA372000 02CD3F (v02 LENOVO CB-01 00000002 INTL 20240927)
```

括號內從左到右是 Revision（`v02`）、OEMID、OEM Table ID、OEM Revision、Creator ID、Creator Revision。之後只要一行 dmesg，你就能讀出整份表頭。

注意：Revision 欄位決定整數寬度

- DSDT 的 Revision **小於 2** 時，直譯器把所有 Integer 當成 32-bit；**2 以上**才是 64-bit。這是為了相容 ACPI 1.0 時代的表。
- ACPICA 以 **DSDT** 的 Revision 決定整個 Namespace 的整數寬度——SSDT 自己標多少都不會改變這個全域設定。
- 反組譯舊機器的 v1 DSDT 再重編時，不要順手把 Revision 改成 2：OEM 的算式可能依賴 32-bit 溢位截斷行為，改寬度等於改語意。

18.3 真實筆電的 DSDT 長什麼樣子

反組譯一台市售筆電的 DSDT，常會得到三萬行以上的 ASL。先別急著逐行讀——所有 OEM 的頂層佈局都大同小異，掌握骨架後你就知道每類東西住在哪一區：

典型筆電 DSDT 的頂層佈局

```
\  (root)
├── _SB_                    ← 所有實體裝置的家
│   ├── PCI0                ← PCIe root complex (_HID PNP0A08)
│   │   ├── GFX0            ← 內顯 (_ADR 0x00020000)
│   │   ├── I2C0 ── TPD0    ← I2C 控制器 → 觸控板
│   │   ├── GPP0 ── NVME    ← PCIe 橋 → SSD
│   │   └── LPCB            ← LPC/eSPI 橋 (_ADR 0x001F0000)
```

```
│   │       ├── EC0_        ← Embedded Controller (PNP0C09)
│   │       │   ├── BAT0    ← 電池 (PNP0C0A)
│   │       │   └── AC0_    ← 變壓器 (ACPI0003)
│   │       ├── HPET        ← 高精度計時器
│   │       └── RTC_        ← 即時時鐘
│   ├── LID0                ← 上蓋開關 (PNP0C0D)
│   └── PWRB                ← 電源鍵 (PNP0C0C)
├── _TZ_ ── THRM            ← 熱區 (_TMP、_PSV、_CRT)
├── _GPE                    ← GPE 事件處理 (_L17、_E02……)
└── _PR_                    ← 舊式 CPU 物件 (新平台多搬進 SSDT)
```

幾個觀察重點：可列舉的 PCI 裝置（GFX0、NVME）只放 _ADR 當「掛勾」，讓 ACPI 物件與 PCI 功能對上；不可列舉的裝置（EC、電池、LID）則必須給 _HID，OS 才知道它們存在。電池與變壓器慣例上掛在 EC 底下，因為它們的資料實際是透過 EC 暫存器讀的（第 9–11 章的 OperationRegion 就派上用場）。LID 有的 OEM 掛 _SB、有的掛 EC 底下，兩者都合法。

把這個骨架縮成一份可編譯的迷你 DSDT，就是下面這個範例——它去掉了 _CRS、EC Field 等細節（前幾個模組都教過），只留結構：

精簡示意（可編譯）：迷你筆電 DSDT

```
DefinitionBlock ("dsdt.aml", "DSDT", 2, "ACME", "NB100",
                 0x00000001)
{
    Scope (\_SB)
    {
        Device (PCI0)                   // PCIe root complex
        {
            Name (_HID, EisaId ("PNP0A08"))
            Name (_CID, EisaId ("PNP0A03"))
            Name (_UID, Zero)

            Device (LPCB)               // LPC/eSPI bridge
            {
                Name (_ADR, 0x001F0000)  // dev 31, func 0

                Device (EC0)            // Embedded Controller
                {
                    Name (_HID, EisaId ("PNP0C09"))
                    Name (_GPE, 0x17)    // EC uses GPE 0x17
                }
            }
        }

        Device (LID0)                   // lid switch
```

```
        {
            Name (_HID, EisaId ("PNP0C0D"))
        }

        Device (PWRB)                   // power button
        {
            Name (_HID, EisaId ("PNP0C0C"))
        }
    }

    Scope (\_TZ)
    {
        ThermalZone (THRM)
        {
            Method (_TMP, 0, Serialized)
            {
                Return (3232)           // 323.2 K = 50.05 degC
            }
        }
    }
}
```

逐行拆解：

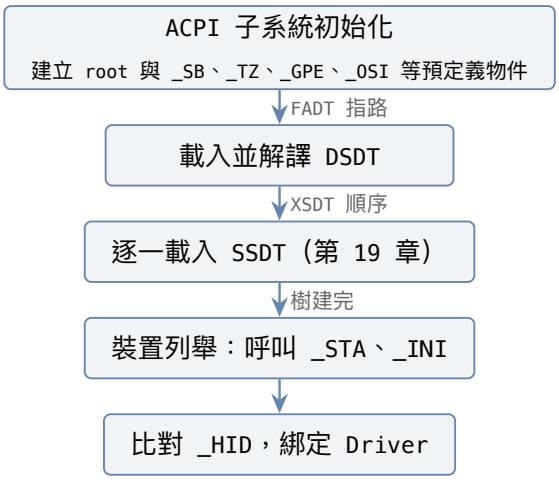
- `DefinitionBlock` 的第二個參數 `"DSDT"` 決定簽章；第三個參數 2 就是表頭的 Revision（64-bit Integer）；最後的 `0x00000001` 是 OEM Revision。
- `PCI0` 是「不可列舉世界」與「可列舉世界」的交界：root bridge 本身用 `_HID` 宣告，底下的 PCI 裝置改用 `_ADR`。
- `LPCB` 的 `_ADR` 是 `0x001F0000`，高 16 位是 device 31、低 16 位是 function 0——Intel 平台 LPC/eSPI 橋的慣用位置。
- `EC0` 的 `_GPE` 告訴 OS：這顆 EC 的事件走 GPE `0x17`，也就是 `\_GPE._L17`（Module 7 講過 GPE 與喚醒的關係）。
- `_TMP` 回傳十分之一 Kelvin：`3232` 代表 323.2 K，約攝氏 50 度。

補充說明：OEM 實際上怎麼維護這幾萬行

沒有人把三萬行寫在同一個檔案裡。OEM 的 BIOS 原始碼中，DSDT 是幾十個 `.asl` 檔（`ec.asl`、`battery.asl`、`thermal.asl`……）經由 include 與前置處理器拼出來的，build 時才組成單一 DefinitionBlock。你反組譯得到的 `dsdt.dsl` 是「攤平」的結果，行號與 OEM 原始檔完全無關——這就是為什麼溝通時要引用物件路徑（如 `\_SB.PCI0.LPCB.EC0.BAT0`），而不是行號。

18.4 OS 如何用 DSDT 建立 Namespace

把第 2 章的 Namespace 知識與本章的表串起來，OS 端的載入順序是固定的：



三個值得記住的細節：

- **DSDT 永遠第一**。SSDT 可以引用 DSDT 定義的物件（靠 External，第 19 章），反過來 DSDT 引用 SSDT 的物件就得小心載入時序。
- 解譯 DSDT 是「載入」而非「執行」：直譯器走過整份 AML，把 Device、Name、Method 掛上樹，但 Method 的本體此時不執行——要等 OS 或驅動呼叫才跑（第 12–14 章）。
- 載入完成後才開始裝置列舉：對每個 Device 呼叫 `_STA` 問存在與否、呼叫 `_INI` 做初始化，再用 `_HID` 找 Driver（第 1 章的資料流圖）。

18.5 幾千行裡找到你要的裝置：搜尋策略

拿到一份三萬行的 `dsdt.dsl`，工程師的第一個技能不是讀，而是 **搜**。最有效的進入點是 `_HID`：標準裝置的 ID 是固定的，背下常用幾個，等於拿到 DSDT 的目錄頁：

`_HID`	裝置	常見物件名
`PNP0A08` / `PNP0A03`	PCIe / PCI root bridge	PCI0
`PNP0C09`	Embedded Controller	EC0、EC、H_EC
`PNP0C0A`	電池（Control Method Battery）	BAT0、BAT1
`ACPI0003`	AC 變壓器	AC、AC0、ADP1
`PNP0C0D`	上蓋開關（Lid）	LID、LID0
`PNP0C0C`	電源鍵	PWRB、PBTN
`PNP0C0E`	睡眠鍵	SLPB
`PNP0103`	HPET 計時器	HPET
`PNP0B00`	RTC 即時時鐘	RTC
`MSFT0101`	TPM 2.0	TPM

配合 `grep -n` 拿行號、`sed -n` 看前後文，三分鐘就能定位目標：

在反組譯的 dsdt.dsl 中定位裝置

```
$ grep -n 'PNP0C09' dsdt.dsl            # find the EC
 1482:        Name (_HID, EisaId ("PNP0C09"))

$ grep -n 'PNP0C0A\|PNP0C0D' dsdt.dsl   # battery / lid
 3120:        Name (_HID, EisaId ("PNP0C0A"))
 4055:        Name (_HID, EisaId ("PNP0C0D"))

$ sed -n '3100,3125p' dsdt.dsl          # context around BAT0
    Device (BAT0)
    {
        Name (_HID, EisaId ("PNP0C0A"))
        Name (_UID, Zero)
        ...

$ grep -c 'Device (' dsdt.dsl           # how many devices?
84
```

反方向也行得通：如果你是從 Linux 這頭出發（例如 `upower` 回報電池異常），先問 kernel 這顆裝置的 Namespace 路徑，再回頭搜 `.dsl`：

從 sysfs 反查 Namespace 路徑

```
$ ls /sys/bus/acpi/devices/ | head -4
ACPI0003:00
LNXPOWER:00
PNP0C09:00
PNP0C0A:00

$ cat /sys/bus/acpi/devices/PNP0C0A:00/path
\_SB_.PCI0.LPCB.EC0_.BAT0
```

其他好用的搜尋錨點：搜 `EmbeddedControl` 找 EC 的 OperationRegion 與 Field 定義；搜 `_L17` 這類 GPE method 名找事件處理；搜 `Notify` 找誰在發事件。

核心觀念：本章重點回顧

- DSDT 是 system fixed description 的一部分：每系統唯一、開機必載、不可卸載；OS 經由 FADT 的 `DSDT` / `X_DSDT` 欄位找到它（`X_DSDT` 非零時優先）。
- 36 位元組表頭中，Revision 決定 Integer 是 32 還是 64 bit（以 DSDT 為準、全域生效）；OEM Revision 是第 20 章 override 的關鍵；checksum 由 iasl 自動計算。
- 典型佈局：`\_SB.PCI0` 底下用 `_ADR` 掛可列舉裝置，LPCB 底下住 EC，電池與變壓器掛在 EC 下，`\_TZ` 放熱區。

- OS 建 Namespace 的順序固定：預定義物件、DSDT、再來才是 SSDT；載入只掛物件，Method 等被呼叫才執行。
- 找東西先搜 `_HID`（`PNP0C09` EC、`PNP0C0A` 電池、`PNP0C0D` LID），或從 `/sys/bus/acpi/devices/*/path` 反查。

動手練習

1. FADT 的 `DSDT` 與 `X_DSDT` 欄位同時非零時，OS 用哪一個？規格為什麼要保留兩個欄位？
2. 給你一行 dmesg：`ACPI: DSDT 0x7A5F0000 01B3C2 (v01 DELL XPS13 00000005 MSFT 04000000)`。指出 Revision、OEMID、OEM Revision 與編譯器，並回答：這份表的 Integer 是幾 bit？
3. 修改 DSDT 之後為什麼不需要手動重算 checksum？請設計一個驗證方法（提示：比較重編前後 offset 9 的位元組）。
4. 動手做：在自己的 Linux 機器上倒出 DSDT 並反組譯（作法見第 20 章前兩節），回答四個問題——總共幾行？有幾個 `Device`？電池物件的完整路徑是什麼？EC 的 `_GPE` 值是多少？

第 19 章

SSDT

上一章說 DSDT 每系統唯一，但你在自己機器上執行 `ls /sys/firmware/acpi/tables/`，多半會看到七、八張 **SSDT**（Secondary System Description Table）。同一份硬體描述為什麼要拆成這麼多張表？答案與軟體工程的模組化一模一樣：DSDT 是穩定的主幹，SSDT 是可拆卸的擴充模組——不同 SKU、不同 CPU、有無獨顯，都靠換裝不同的 SSDT 解決，主幹不必重驗證。

本章先講「為什麼拆」，再講最重要的合併語意——多張表載入後仍是 **同一棵** Namespace——然後動手寫一張引用 DSDT 物件的 SSDT，最後認識動態載入的 Load 家族。

本章學習目標

- 說明平台把描述拆成 DSDT 加多張 SSDT 的工程理由，並舉出動態產生的實例。
- 描述多張表合併成單一 Namespace 的規則、載入順序與同名衝突的後果。
- 寫出一張用 External 引用 DSDT 既有裝置、可通過 iasl 編譯的擴充 SSDT。
- 說明 Load 與 LoadTable 的語意、使用情境，以及 Unload 被廢止的原因。
- 在 Linux 上列出系統的所有 SSDT 並反組譯其中一張。

19.1 為什麼要拆表：SSDT 存在的理由

想像你是 OEM 的 BIOS 工程師，同一代筆電有三個 SKU：內顯版、獨顯版、面板不同的教育版。如果所有描述都塞在 DSDT，你就得維護三份幾萬行的 DSDT，改一行熱區參數要重新驗證三份表。實務的做法是：

- **主機板共通描述進 DSDT**：EC、LPC、電池、LID 這些每台都有的東西。
- **變體差異進 SSDT**：獨顯版多載一張 dGPU 的 SSDT，內顯版根本不放；不同面板的背光表各自一張。
- **CPU 世代差異進 SSDT**：C-state、P-state 隨 CPU 不同而異，Intel 平台的 `PmRef`（`Cpu0Ist`、`Cpu0Cst`……）系列 SSDT 甚至是 BIOS 在 POST 階段依實際插的 CPU **動態產生**的，不是編譯期寫死的。
- **執行期才知道的東西動態載入**：例如接上 dock 之後才需要的裝置描述，可由 AML 用 Load 追加（見 19.4 節）。

在一台真實筆電上看 dmesg，就能數出這種拆法的痕跡：

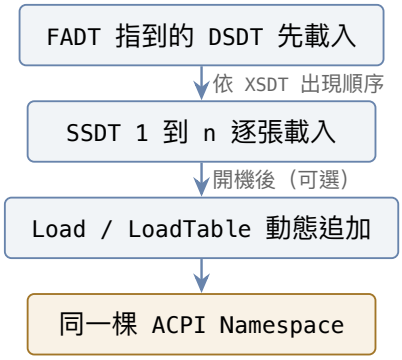
一台筆電的 SSDT 清單（dmesg 節錄）

```
$ sudo dmesg | grep 'ACPI: SSDT'
ACPI: SSDT 0xAA3A5000 0003A5 (v02 LENOVO Tpm2Tabl 00001000 ...)
ACPI: SSDT 0xAA3A0000 001B1C (v02 PmRef  Cpu0Ist  00003000 ...)
ACPI: SSDT 0xAA39E000 000394 (v02 PmRef  Cpu0Cst  00003001 ...)
ACPI: SSDT 0xAA390000 077B69 (v02 LENOVO SaSsdt   00003000 ...)
ACPI: SSDT 0xAA38C000 003657 (v02 LENOVO GppSsdt  00001000 ...)
```

OEM Table ID 一眼就能猜出分工：TPM 一張、CPU 電源兩張、北橋（System Agent）一張、PCIe 橋一張。

19.2 合併語意：多張表、同一棵樹

SSDT 的表頭格式與 DSDT 完全相同（簽章換成 SSDT），內容同樣是 AML。關鍵觀念是：**表只是「來源檔」，Namespace 才是「成品」**。直譯器載入每張表時，把表裡宣告的物件一一掛到同一棵樹上；載入完成後，樹上根本不記得哪個物件來自哪張表。



載入順序由規格保證：DSDT 最先，其餘 SSDT 依它們在 XSDT 裡出現的順序（ACPI 6.4 起規格特別澄清了這一點）。順序之所以重要，是因為**後載入的表可以引用先載入的物件，反之不行**；而如果兩張表宣告了同一個完整路徑的物件，就會撞名。

用第 19.3 節將完成的例子預先看合併的效果：

載入前後：SSDT 分支長到 DSDT 的樹上

```
載入前 ——— DSDT 建好的樹：
\
└── _SB_
    └── PCI0
        └── LPCB
            └── EC0_
                ├── BAT0
                └── LID0

SSDT 想追加的分支（用 External 宣告掛載點）：
Scope (\_SB.PCI0.LPCB.EC0)
```

```
    └── SEN0            ← 新裝置

載入後 ——— 仍是同一棵樹：
\
└── _SB_
    └── PCI0
        └── LPCB
            └── EC0_
                ├── BAT0
                ├── LID0
                └── SEN0    ← 來自 SSDT，與 BAT0 平起平坐
```

注意：同名衝突：AE_ALREADY_EXISTS

兩張表若在同一路徑宣告同名物件（例如都定義 `\_SB.PCI0.I2C1.TPL1`），先載入者建立成功，後載入者觸發 `AE_ALREADY_EXISTS`。kernel log 會出現 `ACPI BIOS Error (bug)` 開頭的錯誤，該物件建立失敗，依 ACPICA 版本與錯誤位置，整張表的載入可能就此中止。OEM 拆表時必須保證各表的路徑互斥；你在第 20 章做 override 或加自訂 SSDT 時，同樣要先確認路徑沒被用走。另外，內容完全相同的重複表會被 ACPICA 偵測並跳過，不會載入兩次。

19.3 External：跨表引用的橋樑

SSDT 要把 SEN0 掛到 `\_SB.PCI0.LPCB.EC0` 底下，就得在 `Scope` 裡寫出這條路徑。問題來了：編譯這張 SSDT 時，iasl 只看得到這一個檔案，根本不知道 EC0 是誰——直接編會報「Object does not exist」錯誤。**External** 就是為此而生：它向編譯器承諾「這個名字存在於別張表，放行吧」，本身不建立任何物件。

完整範例——在既有 EC 底下新增一顆示範感測器裝置：

ssdt-sen.dsl：在 DSDT 的 EC0 下新增裝置

```
DefinitionBlock ("ssdt-sen.aml", "SSDT", 2, "ACME", "ECSENSOR",
                 0x00000001)
{
    // Promise to the compiler: this device exists in the DSDT
    External (\_SB.PCI0.LPCB.EC0, DeviceObj)

    Scope (\_SB.PCI0.LPCB.EC0)
    {
        Device (SEN0)
        {
            Name (_HID, "ACME0001")       // vendor-specific HID
```

```
            Name (_UID, Zero)

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)           // present and enabled
            }
        }
    }
}
```

逐行拆解：

- `External (\_SB.PCI0.LPCB.EC0, DeviceObj)`：第一個參數是完整路徑，第二個參數宣告物件型別。編譯期它只是承諾；產出的 AML 中會留下 External 記錄，但不會建立 EC0。
- `Scope (...)`：切到既有裝置的範圍。Scope 本身也不建立物件，只是把後續宣告掛到那個節點下。
- `Device (SEN0)`：真正的新物件。載入這張 SSDT 時，直譯器在樹上找到 EC0，把 SEN0 掛上去。
- 用 `iasl ssdt-sen.dsl` 即可編譯，產出的 `ssdt-sen.aml` 可用第 20 章的 initrd 或 QEMU 掛載驗證。

External 的常見型別：`DeviceObj`、`MethodObj`、`IntObj`、`FieldUnitObj`、`MutexObj`。其中 `MethodObj` 最需要小心——編譯器得知道該 Method 收幾個參數才能正確解析呼叫點，宣告時可一併給出參數型別清單，例如 `External (\ECRD, MethodObj, IntObj, {IntObj})`。

三個典型的 External 陷阱：

- **路徑打錯照樣編譯過。**承諾不會被驗證，錯誤要到載入期才爆：直譯器回報 `AE_NOT_FOUND`，整段 Scope 掛載失敗。
- **型別或參數個數宣告錯**，AML 可能照樣產出，但呼叫端行為錯亂，或反組譯器解讀成錯誤的語法樹（第 20 章 –e 旗標的由來）。
- **把 External 當成「建立物件」。**它從不建立東西；如果目標表根本沒載入，引用它的程式碼全數失效。

19.4 動態載入：Load、LoadTable 與 Unload

到目前為止的表都是韌體開機前擺好、OS 開機時一次載入的。ASL 還提供執行期追加定義塊的能力，典型情境：筆電接上 dock 之後，AML 從 dock 的記憶體區把一張描述 dock 裝置的 SSDT 讀進來掛上樹。

現行規格（ACPI 6.5）的語法：`Load (Object, Result)`——`Object` 可以是 OperationRegion、Field 或 Buffer，內容必須是一張完整的表；載入成功時 `Result` 收到真值。

ssdt-load.dsl：執行期載入另一張表（示意）

```
DefinitionBlock ("ssdt-load.aml", "SSDT", 2, "ACME", "DYNLOAD",
                 0x00000001)
{
    Scope (\_SB)
    {
        // Example address only: table image exposed by firmware
        OperationRegion (TMEM, SystemMemory, 0xDF000000, 0x400)
        Field (TMEM, AnyAcc, NoLock, Preserve)
        {
            TBLD, 8192                  // 1024-byte table image
        }

        Name (LDST, Zero)               // load-state flag

        Method (LDTB, 0, Serialized)
        {
            If (LDST == Zero)
            {
                Load (TBLD, Local0)     // Local0: success flag
                If (Local0)
                {
                    LDST = One
                }
            }
        }
    }
}
```

拆解重點：TBLD 這個 Field 蓋住一張放在記憶體裡的完整表影像；LDTB 用旗標避免重複載入——載入兩次同一張表就會撞上一節說的 AE_ALREADY_EXISTS。

LoadTable 則不從記憶體讀，而是去 XSDT 找一張「已存在但尚未載入」的表，依簽章、OEMID、OEM Table ID 三個字串比對：

LoadTable 片段（節錄自 Method 內）

```
// Load a staged table: signature, OEM ID, OEM table ID
Local0 = LoadTable ("OEM1", "ACME", "DOCKTBL")
If (Local0)
{
    Store ("dock table loaded", Debug)
}
```

補充說明：Unload 已廢止

早年的 Load 第二個參數是 `DDBHandle`，搭配 `Unload (DDBHandle)` 可把表拆下來。實務上幾乎沒有韌體使用、OS 端也難以安全實作（拆表等於從活著的 Namespace 上鋸樹枝），因此 **ACPI 6.3 正式把 Unload 列為 deprecated**，新版 ACPICA 直譯器已不支援；Load 的第二個參數也改成單純回報成功與否的結果。讀舊 DSDT 看到 Unload 知道它是什麼即可，新程式碼一律不要用。

19.5 在 Linux 上數 SSDT

驗收本章觀念最快的方式，是打開自己的機器數一數：

觀察系統的 SSDT（Linux）

```
$ ls /sys/firmware/acpi/tables/ | grep SSDT
SSDT1
SSDT2
SSDT3
SSDT4
SSDT5

$ ls /sys/firmware/acpi/tables/dynamic/ 2>/dev/null
SSDT6                       # tables added by Load at runtime

$ sudo cp /sys/firmware/acpi/tables/SSDT2 /tmp/ssdt2.dat
$ cd /tmp && iasl -d ssdt2.dat && head -3 ssdt2.dsl
Intel ACPI Component Architecture
AML/ASL+ Disassembler version 20240927
...
```

注意兩件事：韌體給的 SSDT 直接以 `SSDT1`、`SSDT2`……命名；執行期由 Load 動態載入的表則收在 `dynamic/` 子目錄。把每張 SSDT 都反組譯開頭幾行、對照 OEM Table ID，就能拼出這台機器的描述拼圖——這正是本章練習的最後一題。

核心觀念：本章重點回顧

- SSDT 是 DSDT 的延伸：表頭格式相同、內容同為 AML，可以有多張；拆表動機是 SKU 變體、CPU 世代差異、選配硬體與動態產生（Intel PmRef 系列就是 POST 時生成的）。
- 合併語意：所有表載入同一棵 Namespace，DSDT 先、SSDT 依 XSDT 順序後；後載者可引用先載者。

- 同一路徑宣告兩次觸發 `AE_ALREADY_EXISTS`，該物件建立失敗、表可能整張載入中止。
- External 只是給編譯器的承諾，不建立物件；路徑錯誤要到載入期才以 `AE_NOT_FOUND` 爆出來。
- Load 從 OpRegion / Field / Buffer 載入表、LoadTable 從 XSDT 挑表載入；Unload 自 ACPI 6.3 起廢止。動態載入的表出現在 `/sys/firmware/acpi/tables/dynamic/`。

動手練習

1. 為什麼 CPU 的 P-state / C-state 描述特別適合做成 POST 階段動態產生的 SSDT，而不是寫死在 DSDT 裡？
2. 一張 SSDT 想在 `\_SB.PCI0` 底下加裝置，卻忘了寫 External 就直接 `Scope`。請問問題會在編譯期還是載入期爆出來？錯誤訊息大概長什麼樣？
3. 兩張 SSDT 都定義了 `\_SB.FOO`，載入結果由什麼決定？「贏家」是誰？輸家那張表會發生什麼事？
4. 承第 19.3 節範例：如果把 External 的型別誤寫成 `MethodObj`，iasl 還會編譯通過嗎？試著解釋為什麼 External 的錯誤這麼難在編譯期攔下來。
5. 動手做：數一數你機器上有幾張 SSDT（含 `dynamic/`），全部反組譯後列出各自的 OEM Table ID 與用途猜測，並找出其中一張 External 引用了哪些 DSDT 物件。

第 20 章

DSDT/SSDT Override

前兩章把表讀懂了，本章來真的：改。場景很常見——筆電閤蓋不會睡、電池閾值方法有 bug、或者你想驗證「如果 `_LID` 這樣寫就好了」的假說。BIOS 是 OEM 的地盤，你沒有原始碼也沒有簽章金鑰；但 Linux 提供了一條正規的實驗路徑：把修改過的 DSDT 放進 initrd，kernel 開機時用它取代韌體給的那份。韌體本身一個位元組都沒動，實驗失敗把 initrd 還原即可。

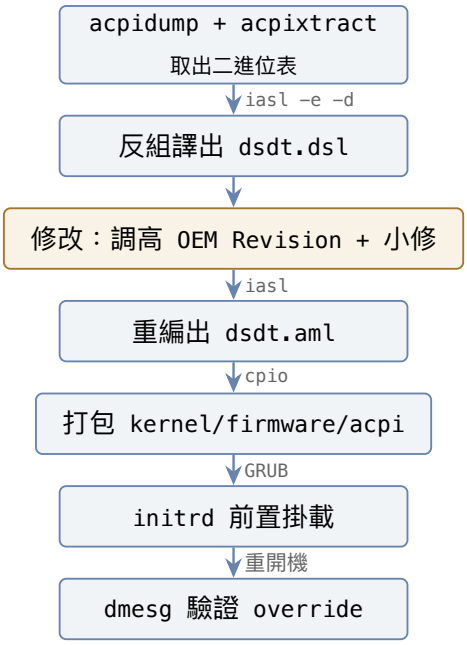
這條流水線是 BIOS / 驅動工程師的日常除錯手段，也是本書第一個端到端實戰。請找一台可重開機的 Linux 機器跟著做；想零風險練習，直接用 20.8 節的 QEMU 方案。

本章學習目標

- 完整操作 extract、disassemble、modify、recompile、override、boot、test 七步流程。
- 說明反組譯多表系統時 `-e` 旗標的作用與缺它的後果。
- 說明 initrd override 的比對規則，以及為什麼必須調高 OEM Revision。
- 在 dmesg 中辨認 override 成功與失敗（install）的訊息。
- 用 QEMU 的 `-acpitable` 安全地掛載自訂表。
- 說明 Secure Boot / lockdown 的限制與改壞後的還原方法。

20.1 全景：七步流水線

先看全貌，再逐步拆解：



Lab 實作：DSDT Override 一頁清單

以下是完整流程速查，細節見後續各節。全程需要 root 權限，建議在可實驗的機器（或 QEMU 虛擬機）上進行。

1. 取表：`sudo acpidump > acpi.out`，再 `acpixtract -a acpi.out`。
2. 反組譯：`iasl -e ssdt*.dat -d dsdt.dat`，得到 `dsdt.dsl`。
3. 修改：調高 DefinitionBlock 的 OEM Revision，再加入你要的變更。
4. 重編：`iasl dsdt.dsl`，把錯誤修到 0 Errors。
5. 打包：把 `dsdt.aml` 放進 `kernel/firmware/acpi/`，cpio 打包。
6. 掛載：設定 GRUB，讓它把這個 cpio 放在原本 initrd 之前。
7. 驗證：dmesg 出現 Table Upgrade: override 且 OEM Revision 已更新。

20.2 Step 1：取出系統的表

`acpidump` 把所有 ACPI 表連同位址倒成一個文字檔，`acpixtract` 再從中抽出二進位（兩個工具的完整用法見第 21 章）：

取出並抽出所有表

```
$ mkdir ~/acpi-lab && cd ~/acpi-lab
$ sudo acpidump > acpi.out
$ acpixtract -a acpi.out

Acpi table [DSDT] -  183615 bytes written to dsdt.dat
Acpi table [SSDT] -     933 bytes written to ssdt1.dat
Acpi table [SSDT] -    6940 bytes written to ssdt2.dat
```

```
Acpi table [FACP] -     276 bytes written to facp.dat
...
```

`-a` 表示抽出全部的表；不加的話預設只抽 DSDT 與 SSDT。把整組 `.dat` 留著當備份——之後不管改壞什麼，都能從這裡重來。

20.3 Step 2：反組譯——為什麼需要 -e

DSDT 常會呼叫定義在 SSDT 裡的 Method（反之亦然）。反組譯器遇到一個不認識的名字時，最大的困擾是：**它不知道那是幾個參數的 Method**。AML 的呼叫點沒有標參數個數，反組譯器只能靠已知的宣告去切後面的位元組流；猜錯參數個數，整段程式碼會被切歪，輕則產生錯誤的引用、重則反組譯出根本不能重編的垃圾。

`-e` 旗標把「其他表」餵給反組譯器當符號參考，讓它能正確解析跨表引用：

帶上所有 SSDT 反組譯 DSDT

```
$ iasl -e ssdt*.dat -d dsdt.dat

Intel ACPI Component Architecture
AML/ASL+ Disassembler version 20240927

External object resolution file ssdt1.dat
External object resolution file ssdt2.dat
Loading Acpi table from file   dsdt.dat
Pass 1 parse of [DSDT]
Pass 2 parse of [DSDT]
Disassembly completed
ASL Output:    dsdt.dsl - 1103894 bytes
```

反組譯完成後，`dsdt.dsl` 開頭會自動帶出一排 `External` 宣告，正是第 19 章教的機制。若你少帶了某張 SSDT，輕則重編時冒出一堆「Object does not exist」，重則反組譯結果本身就是歪的——遇到怪異的反組譯輸出，第一反應就是檢查 `-e` 有沒有帶齊。

20.4 Step 3：安全的小修改

第一次 override，不要急著修真正的 bug。先做兩個「安全改動」，把流程本身驗證通過：

dsdt.dsl 修改處（片段）：調高 OEM Revision 並留記號

```
// BEFORE (vendor original):
// DefinitionBlock ("dsdt.aml", "DSDT", 2, "LENOVO", "CB-01",
//                  0x00000002)

// AFTER (ours): bump OEM Revision 0x00000002 -> 0x00000003
```

```
DefinitionBlock ("dsdt.aml", "DSDT", 2, "LENOVO", "CB-01",
                 0x00000003)
{
    Name (OVRD, 0x20260829)      // harmless marker object
    ...                          // vendor content unchanged
```

- **調高 OEM Revision 不是儀式，是硬規則**：20.7 節會看到，只有 initrd 表的 OEM Revision 嚴格大於韌體表才會被採用。忘了調，整個 override 靜悄悄失效。
- `OVRD` 是無害的記號物件，足以證明「跑的是我這份表」。想留執行期腳印，可再往常被呼叫的 Method 裡 `Store` 字串到 Debug 物件——但那要開 kernel 除錯輸出才看得到（Module 10 詳述），本章先用 OEM Revision 驗證。

20.5 Step 4：重編——面對 vendor DSDT 的現實

OEM 的 DSDT 往往是多年前的舊版 iasl 甚至微軟 asl.exe 編的，語法寬鬆；新版 iasl 檢查嚴格，直接重編幾乎必有訊息：

重編與典型訊息

```
$ iasl dsdt.dsl

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20240927

dsdt.dsl   1482:    If (CondRefOf (\_OSI, Local0))
Warning  3144 -  Method Local is set but never used ^  (Local0)

dsdt.dsl   9021:    Return (Local3)
Error    6066 -  Method local variable is not initialized ^ (Local3)
```

常見訊息的分類與處置：

訊息（節錄）	等級	典型處置
Method Local is set but never used (3144)	Warning	多為 `CondRefOf` 舊寫法，可忽略；想消音就刪掉多餘的第二參數
Method local variable is not initialized (6066)	Error	舊編譯器放水的真 bug；補上初始化，或比照原 AML 邏輯改寫
Called method returns no value（6080）	Error	呼叫點期待回傳值；對照上下文補 `Return`
Called method may not always return a value（3122）	Warning	某些路徑沒 `Return`；補齊每條路徑
Not a parameter, used as local only (2087)	Remark	風格提醒，可忽略
Object does not exist	Error	多半是反組譯時 `-e` 沒帶齊；回 Step 2 重來

原則：**Error 必修（不出 AML）、Warning 逐條看懂再決定、Remark 與 Optimization 可放行**。訊息編號可配合 `-vw` 選擇性關閉、`-f` 可無視錯誤硬出檔——兩者的細節與風險第 21 章統一整理；在 override 情境裡，`-f` 等於把沒修好的炸彈裝進 initrd，強烈不建議。

20.6 Step 5：打包 initrd 與 GRUB 設定

kernel 在極早期會掃描 initrd 開頭是否有一個未壓縮的 cpio，裡面路徑固定為 `kernel/firmware/acpi/*.aml`。做法照抄即可：

打包 override 用的 cpio

```
$ cd ~/acpi-lab
$ mkdir -p kernel/firmware/acpi
$ cp dsdt.aml kernel/firmware/acpi/
$ find kernel | cpio -H newc --create > acpi-override.img
360 blocks
$ sudo cp acpi-override.img /boot/
```

接著讓 GRUB 把它放在既有 initrd 之前。Debian / Ubuntu 系有現成掛鉤；其他發行版用通用的串接法：

兩種 GRUB 掛載方式

```
# Way 1 (Debian/Ubuntu): early-initrd hook
#   add this line to /etc/default/grub:
#   GRUB_EARLY_INITRD_LINUX_CUSTOM="acpi-override.img"
$ sudo update-grub

# Way 2 (universal): prepend to the existing initrd
$ cat acpi-override.img /boot/initrd.img-$(uname -r) \
```

```
      > initrd-acpi.img
$ sudo cp initrd-acpi.img /boot/
# then point one GRUB menu entry's initrd at initrd-acpi.img
```

補充說明：kernel 端的前提條件

這套機制叫 ACPI table upgrade，需要 `CONFIG_ACPI_TABLE_UPGRADE=y`（主流發行版預設開啟）。initrd 裡最多放 64 張表；除了 RSDP 與 FACS，幾乎所有表都能換，換 SSDT、換 FADT 都是同一套玩法。載入非原廠表後 kernel 會標記 taint——這是提醒「除錯報告請註明表被改過」，不是故障。

20.7 Step 6：重開機驗證——override 還是 install？

重開機後，第一件事看 dmesg：

成功的樣子

```
$ sudo dmesg | grep -iE 'table upgrade|ACPI: DSDT'
ACPI: DSDT ACPI table found in initrd
      [kernel/firmware/acpi/dsdt.aml]
ACPI: Table Upgrade: override [DSDT-LENOVO-CB-01   ]
ACPI: DSDT 0xAA372000 Physical table override, new table:
      0xA9F1C000
ACPI: DSDT 0xA9F1C000 02CD41 (v02 LENOVO CB-01 00000003 ...)
```

重點有三：出現 `Table Upgrade: override` 字樣；最後一行的 OEM Revision 變成 `00000003`（我們調高後的值）；長度也從 `02CD3F` 變成 `02CD41`——多了記號物件的位元組級差異。不放心的話，再用第 18 章的 xxd 技巧讀 `/sys/firmware/acpi/tables/DSDT` 的 offset 24 起 4 個位元組，應該看到 `03 00 00 00`。

如果 dmesg 出現的是 `Table Upgrade: install` 而不是 override，代表你的表被當成新表附加（sysfs 裡出現 `DSDT1`、`DSDT2` 兩份），對 DSDT 而言等於沒生效。原因幾乎都是比對規則沒滿足：

核心觀念：initrd override 的比對規則

kernel 逐張比對 initrd 表與韌體表，全部滿足才 override：

- Signature 相同（例如都是 `DSDT`）；
- OEMID 相同、OEM Table ID 相同（所以反組譯後不要改動 DefinitionBlock 裡的這兩個字串）；
- initrd 表的 OEM Revision 嚴格大於韌體表——相等都不行。

不滿足時，簽章唯一的表會被 install 成附加表，其餘直接忽略。每張 initrd 表最多被使用一次；檔名不影響比對，放在 `kernel/firmware/acpi/` 目錄下即可。

20.8 QEMU：零風險的練習場

不想拿實體機冒險？QEMU 的 `-acpitable` 可以直接把一張自訂表附加給虛擬機。先寫一張迷你 SSDT：

qemu-lab.dsl：掛給 QEMU 的自訂表

```
DefinitionBlock ("qemu-lab.aml", "SSDT", 2, "CLASS", "LAB20",
                 0x00000001)
{
    Scope (\_SB)
    {
        Device (DEMO)
        {
            Name (_HID, "ACME0002")
            Name (_UID, Zero)
            Name (_STA, 0x0F)
            Name (MSG, "Hello from lab SSDT")
        }
    }
}
```

編譯後開機、進 guest 驗證：

掛表、開機、驗證

```
$ iasl qemu-lab.dsl
$ qemu-system-x86_64 -m 2G -enable-kvm \
      -drive file=lab.qcow2,if=virtio \
      -acpitable file=qemu-lab.aml

# inside the guest:
$ ls /sys/firmware/acpi/tables/ | grep SSDT
SSDT1
SSDT2
$ sudo cp /sys/firmware/acpi/tables/SSDT2 /tmp && cd /tmp
$ iasl -d SSDT2 && grep -n 'DEMO' SSDT2.dsl
   10:        Device (DEMO)
```

注意 `-acpitable` 是附加一張表（適合練 SSDT），不會替換 guest 的 DSDT；想練完整的 DSDT override，就在 guest 裡照 20.2 到 20.6 節原樣操作——虛擬機玩壞了，砍掉重建就好。

20.9 擋路的東西與退路

注意：Secure Boot、lockdown 與還原方法

- AML 以 kernel 權限執行，換表等於注入核心程式碼。因此開啟 Secure Boot 時，kernel 的 lockdown 機制會直接忽略 initrd 裡的 ACPI 表，dmesg 可看到 Lockdown 相關提示。實驗前請在韌體設定中關閉 Secure Boot，或改用 QEMU。
- 改壞了怎麼辦？韌體那份 **DSDT** 從頭到尾沒被動過，所以不存在「變磚」問題。開機選單選一個沒掛 override 的舊項目，或用救援模式把 `/boot` 裡的 `acpi-override.img` 移除、還原 GRUB 設定即可。
- 給自己留退路：永遠保留一個乾淨的 GRUB 開機項，不要把唯一的開機項指向掛了 override 的 initrd。

Windows 也有對應機制：用微軟工具把改過的表寫進 registry 供開機載入（需測試簽章模式）——原理同樣是「OS 端替換、韌體不動」，本書不展開。

核心觀念：本章重點回顧

- 七步流水線：acpidump 取表、`iasl -e -d` 反組譯、修改、重編、cpio 打包、GRUB 前置 initrd、開機 dmesg 驗證。
- 反組譯多表系統必帶 `-e`，否則跨表 Method 參數個數猜錯，輸出不可信。
- override 生效條件：Signature、OEMID、OEM Table ID 全同，且 OEM Revision 嚴格較大——所以每次改表都要調高它。
- dmesg 見 `override` 才是成功；見 `install` 代表被當新表附加，DSDT 沒被換掉。
- Secure Boot / lockdown 會使 initrd 表被忽略；還原只需移除 initrd 裡的表，韌體原表永遠完好。
- QEMU 的 `-acpitable file=...` 是零風險練習場。

動手練習

1. Override 生效期間，韌體 ROM 裡的 DSDT 變了嗎？這對「還原」意味著什麼？
2. 你重開機後 dmesg 出現 `Table Upgrade: install [DSDT-...]`，且 sysfs 出現 `DSDT1` 與 `DSDT2`。列出至少兩個可能原因與對應修法。
3. 為什麼 kernel 在 Secure Boot（lockdown）下要擋 ACPI table override？從「AML 由誰執行、以什麼權限執行」的角度說明。
4. 動手做（QEMU）：寫一張在 `\_SB` 下新增裝置的 SSDT，用 `-acpitable` 掛進虛擬機，再從 guest 的 sysfs 反組譯回來驗證。
5. 進階實作：在實體機完成整條 override 流水線（只調高 OEM Revision），用 dmesg 確認生效後，把系統還原乾淨。

Module 9

iASL Toolchain

難度：★★★★☆

工欲善其事：ACPICA 工具全家桶是 BIOS 與驅動工程師的日常兵器。本模組專講 iASL 編譯器 / 反組譯器與周邊工具——acpidump、acpixtract、acpiexec——的安裝、旗標與輸出解讀，並示範 mixed listing 這把理解 AML 的鑰匙。第 20 章的 override 流程與 Module 10 的除錯，都建立在這些工具之上。

第 21 章

iASL Compiler / Disassembler

第 20 章的流水線裡，`iasl`、`acpidump`、`acpixtract` 已經輪番上陣，但當時它們只是配角。本章把工具升格為主角：這些程式從哪裡來、彼此什麼關係、每個常用旗標解決什麼問題。你會發現它們全部出自同一個專案——**ACPICA**——而且與 Linux kernel 裡的 ACPI 直譯器系出同門。

把編譯器輸出讀懂、把反組譯極限摸清，是進入 Module 10 除錯實戰前的最後地基。本章操作全在使用者空間完成，不需要重開機，放心大膽地玩。

本章學習目標

- 說明 ACPICA 專案的角色與工具家族，列出官方文件及各自的適用時機。
- 安裝 acpica-tools，讀懂 `iasl` 的輸出統計與 Error / Warning / Remark 訊息等級。
- 活用 `-d`、`-e`、`-p`、`-tc`、`-l`、`-g`、`-vw` 等常用旗標。
- 用 `acpidump` 與 `acpixtract` 取得並抽出系統的 ACPI 表。
- 用 `acpiexec` 在使用者空間載入 AML、執行 Method。
- 說明 roundtrip 不保證 byte-identical 的原因與正確比對法。

21.1 ACPICA：一套核心、整組工具

ACPICA（ACPI Component Architecture）是 Intel 維護的 ACPI 參考實作，開放原始碼（GitHub 上的 `acpica/acpica`）。它的核心是一個與作業系統無關的 ACPI 直譯器——Linux kernel 的 `drivers/acpi/acpica/` 內嵌的就是這一套。同一份核心配上不同外殼，就變出整組工具：

ACPICA 專案的家族樹

```
ACPICA (github.com/acpica/acpica)
├── 核心直譯器 — 內嵌進 kernel，也是 acpiexec 的心臟
└── 使用者工具
    ├── iasl        ← 編譯器 + 反組譯器（本章主角）
    ├── acpidump    ← 從執行中系統倒出所有表
    ├── acpixtract  ← 從 dump 輸出抽出二進位表
    ├── acpihelp    ← 離線查 ASL 關鍵字與預定義名稱
    └── acpibin     ← AML 二進位層級比對 / 檢查
```

補充說明：kernel 與 iasl 系出同門

Linux 開機訊息裡的 `ACPI: Core revision 20240927` 與 `iasl -v` 的版本是同一種日期式版號，因為兩者共用同一份核心原始碼。實用推論：dmesg 裡的 ACPI 錯誤訊息（如 `AE_NOT_FOUND`）與 acpiexec 印的完全同格式——在使用者空間重現 bug，往往比重開機快十倍。

工具會用了，語意疑義還是要回到文件。官方文件就這五份，建議全部下載存檔(acpica.org 與 uefi.org 皆可取得)：

文件	內容	什麼時候查
ACPI Specification	語言與表格式的最終權威	語意有疑義時
ASL Tutorial	官方入門教學	新手起步
Introduction to ASL 2.0	ASL+ symbolic 語法總覽	讀寫新式語法
iASL Compiler User Guide	旗標與訊息的完整參考	查旗標與訊息
ACPICA User Guide and Programmer Reference	核心架構與 API	OS 端整合

21.2 安裝與版本

主流平台都有現成套件，套件名稱統一叫 `acpica-tools`（macOS 上是 `acpica`）：

安裝 acpica-tools

```
$ sudo apt install acpica-tools     # Debian / Ubuntu
$ sudo dnf install acpica-tools     # Fedora / RHEL
$ brew install acpica               # macOS
$ git clone https://github.com/acpica/acpica.git   # or: source
$ cd acpica && make -j8 && sudo make install

$ iasl -v
Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20240927
```

版本以日期命名。處理 vendor DSDT 盡量用新版；你用哪一版編譯，會被寫進表頭的 Creator Revision（第 18 章），將來查得到。

21.3 第一次編譯：讀懂 iasl 的輸出

寫一份最小但五臟俱全的測試檔：

test.dsl：本章的實驗樣本

```
DefinitionBlock ("test.aml", "SSDT", 2, "DEMO", "TOOLTEST",
```

```
                0x00000001)
{
    Name (MSG, "Hello from AML")

    Method (ADD2, 2, NotSerialized)
    {
        Return (Arg0 + Arg1)
    }

    Method (MAIN, 0, NotSerialized)
    {
        Debug = MSG
        Return (ADD2 (2, 3))
    }
}
```

基本指令就是 `iasl test.dsl`，成功時產生 `test.aml`：

編譯輸出（節錄，長行為排版折行）

```
$ iasl test.dsl

ASL+ Optimizing Compiler/Disassembler version 20240927

ASL Input:     test.dsl -   16 lines, 420 bytes, 6 keywords
AML Output:    test.aml -  100 bytes, 3 named objects,
               3 executable opcodes

Compilation successful. 0 Errors, 0 Warnings, 0 Remarks,
1 Optimizations
```

ASL Input 統計原始碼規模；AML Output 告訴你位元組碼多大、掛了幾個具名物件；最後一行是四種訊息的計數。等級的意義：

等級	編號	意義	產出.aml？
Error	6xxx	語法或語意錯誤	否
Warning	3xxx	很可能是問題，需逐條判讀	是
Remark	2xxx	風格與效率提醒	是
Optimization	——	編譯器做了等價改寫的通知	是

訊息格式固定：先印出檔名、行號與該行原文，再印等級、編號與說明，`^` 指向出錯位置。故意寫壞一份來看：

bad.dsl（片段）：兩個經典錯誤

```
Method (BAD0, 1, NotSerialized)
{
    Store (Arg0, Local0)      // Local0 never read back
    Return (Local3)           // Local3 never initialized
}
```

錯誤訊息的長相

```
$ iasl bad.dsl

bad.dsl      6:        Store (Arg0, Local0)
Warning  3144 -   Method Local is set but never used ^  (Local0)

bad.dsl      7:        Return (Local3)
Error    6066 -  Method local variable is not initialized ^
(Local3)
```

一個 Warning、一個 Error：前者放行，後者直接擋下 `.aml` 產出。訊息編號穩定，可拿去搜尋，也可餵給 `-vw` 選擇性消音（21.5 節）。

21.4 反組譯：-d 與 -e

iasl 的另一個身分是反組譯器。`iasl -d` 吃二進位表、吐回 `.dsl`：

反組譯剛才的成品

```
$ iasl -d -p round test.aml

AML/ASL+ Disassembler version 20240927

Loading Acpi table from file   test.aml
Disassembly completed
ASL Output:    round.dsl - 1207 bytes
```

這裡順手用了 `-p round` 指定輸出檔名前綴：不加的話輸出叫 `test.dsl`，會直接覆蓋你的原始碼——新手最常見的資料損失事故。反組譯真實系統的 DSDT 時，還必須帶 `-e` 把其他表餵進來當符號參考（`iasl -e ssdt*.dat -d dsdt.dat`），原因第 20 章拆解過。`round.dsl` 與 `test.dsl` 語意相同但不會一模一樣——深層原因與正確的比對方式留到 21.8 節。

21.5 常用旗標速查

先上總表，再逐一示範最重要的幾個：

旗標	作用	備註
`-d`	反組譯 `.dat` / `.aml` 成 `.dsl`	輸出同名檔，小心覆蓋
`-e`	反組譯時提供外部符號參考表	多表系統必帶（第 20 章）
`-da`	一次反組譯多張表，共用 namespace	`iasl -da dsdt.dat ssdt*.dat`
`-p NAME`	指定輸出檔名前綴	避免覆蓋原檔
`-tc`	額外輸出 C hex 陣列（`.hex`）	給 EDK II 嵌入韌體用
`-l`	產生 mixed listing（`.lst`）	ASL 對照 AML opcode
`-g`	從執行中 OS 取得系統表	僅 Windows；Linux 用 acpidump
`-vw ID`	抑制特定編號的訊息	如 `-vw 3144`
`-f`	忽略錯誤強制產出 AML	危險，除錯限定
`-h`	完整旗標說明	`-v` 顯示版本

21.5.1 -tc：把 AML 變成 C 陣列

韌體專案（EDK II）常把 AML 以 C 陣列形式連結進 BIOS 映像。`iasl -tc test.dsl` 除了 `test.aml` 還會生出 `test.hex`：

test.hex（節錄）

```
/*
 * Intel ACPI Component Architecture
 * Compilation of "test.dsl" - C source code output
 * AML code block contains 0x64 bytes
 */
unsigned char AmlCode[] =
{
    0x53,0x53,0x44,0x54,0x64,0x00,0x00,0x00,  /* "SSDTd..." */
    0x02,0x1B,0x44,0x45,0x4D,0x4F,0x00,0x00,  /* "..DEMO.." */
    ...
};
```

開頭四個位元組就是簽章 `SSDT`——第 18 章的表頭知識再次通用；EDK II 的平台程式碼引用這個陣列，把表安裝進記憶體。

21.5.2 -l：mixed listing，理解 AML 的利器

`iasl -l test.dsl` 產生 `test.lst`，把每行 ASL 與它編譯出的 AML 位元組並排列出：

test.lst（節錄）

```
      6:    Method (ADD2, 2, NotSerialized)
000000E1    14 0B 41 44 44 32 02
```

```
      8:        Return (Arg0 + Arg1)
000000E8    A4 72 68 69 00
```

逐位元組讀：`14` 是 MethodOp；`0B` 是 PkgLength（區塊共 11 位元組）；`41 44 44 32` 是 ASCII 的 `ADD2`；`02` 是旗標（2 個參數、NotSerialized）。下一段 `A4` 是 ReturnOp、`72` 是 AddOp、`68 69` 是 Arg0 與 Arg1、`00` 是空目標。想確認編譯器生了什麼碼、或手上只有 AML 片段要人肉解碼時，mixed listing 是最快的對照表。

21.5.3 -g 與 acpidump 的分工

`iasl -g` 會呼叫 OS 介面抓下執行中系統的表，但只在 **Windows** 上有實作（需管理員權限）；Linux 上由 `acpidump` 負責，直接讀 `/sys/firmware/acpi/tables/`。功能等價、工具不同，跨平台時記得換手。

21.6 acpidump 與 acpixtract

這對搭檔在第 20 章登場過，這裡補齊完整用法：

acpidump		acpixtract	
（無參數）	全部表倒成十六進位文字	（無參數）	只抽 DSDT 與 SSDT
`-o F`	輸出導向檔案 F	`-a`	抽出全部的表
`-b`	每張表直接存成二進位	`-l`	只列出有哪些表
`-n SIG`	只倒指定簽章	`-s SIG`	只抽指定簽章

組合技示範

```
$ sudo acpidump -o acpi.out        # dump everything
$ acpixtract -l acpi.out           # what is inside?
  DSDT - 183615 bytes
  SSDT -    933 bytes
  ...

$ acpixtract -s DSDT acpi.out      # extract one signature
Acpi table [DSDT] - 183615 bytes written to dsdt.dat

$ sudo acpidump -b && ls *.dat     # or: skip the text step
dsdt.dat  facp.dat  ssdt1.dat  ssdt2.dat
```

`acpidump` 需要 root；`-b` 直接產出二進位、省掉 acpixtract 一步，適合自動化腳本。文字模式的 `acpi.out` 則適合連同錯誤報告寄給別人——一個檔案裝下整台機器的 ACPI 狀態。

21.7 acpiexec：不開機就執行 AML

`acpiexec` 把 ACPICA 核心直譯器包成使用者空間程式：載入你的表、建出 Namespace、互動式執行 Method ——不必燒韌體、不必重開機：

第一次 acpiexec（輸出節錄）

```
$ acpiexec test.aml

Intel ACPI Component Architecture
AML Execution/Debug environment version 20240927

- methods
\ADD2      2 arguments
\MAIN      0 arguments

- execute MAIN
Executing \MAIN
[ACPI Debug]  String [0x0E] "Hello from AML"
Execution of \MAIN returned object, external buffer length 18
 [Integer] = 0000000000000005
```

提示符 `-` 之後是除錯器指令：`methods` 列出所有 Method、`execute` 呼叫指定 Method（可帶參數）、`namespace` 傾印整棵樹。Debug 物件的輸出直接印在畫面上——真機上要開除錯選項才看得到的東西，這裡免費；第 22、23 章將以 acpiexec 為主軸做整套 AML 除錯。

21.8 Roundtrip：反組譯再重編不是影印機

很多人以為 `.aml` 反組譯再編回去，會得到一模一樣的位元組——實際上不保證，原因：

- **反組譯是重建，不是還原**：註解、排版、include 結構在編譯時就消失了，永遠回不來。
- **ResourceTemplate 被巨集化**：反組譯器把 resource descriptor 重建成巨集寫法，重編後編碼細節可能與原表不同。
- **External 與最佳化**：反組譯器自動補 External；編譯器又會折疊常數、精簡 opcode（統計裡的 Optimizations 就是次數）。
- **版本差異**：不同版 iasl 產碼策略不同，Creator Revision 欄位本身就會變。

正確的驗證是比語意不比位元組：原始與重編的 `.aml` 各自反組譯，diff 兩份 `.dsl`——編譯器的重寫會被同樣的反組譯規則抹平，剩下的差異才是你真正改到的東西。

注意：反組譯工作的三條保命規則

- `iasl -d` 的輸出與輸入同名（只換副檔名），會無預警覆蓋既有 `.dsl`；用 `-p` 防呆，並永遠保留原始 `.dat` 備份。
- 別拿「重編後 byte 數不同」判斷自己改壞了東西；用雙反組譯 diff 法。
- 重編 vendor 表用新版 iasl 並記錄版本；跨版本混用會讓 diff 全是雜訊。

Lab 實作：三連發：編譯、反組譯、對照 listing

用本章的 `test.dsl` 完成三個回合，全程不需要 root：

1. 編譯：`iasl test.dsl`，確認 0 Errors，記下 AML 大小與 named objects 數量。
2. 反組譯：`iasl -d -p round test.aml`；與原始檔對照，找出至少三處差異（註解消失、寫法互換、排版重排）。
3. 對照 listing：`iasl -l test.dsl`，在 `test.lst` 找出 Method 的 `0x14`、Return 的 `0xA4` 與 Add 的 `0x72`，親手把一行 ASL 對應到它的位元組。

收尾加碼：把同一套流程套在你機器的某張小 SSDT 上。

核心觀念：本章重點回顧

- ACPICA 是 Intel 維護的參考實作；kernel 內嵌同一套核心，全家桶工具全部同源。
- iasl 身兼編譯器與反組譯器：Error（6xxx）擋下產出，Warning（3xxx）與 Remark（2xxx）放行但要判讀。
- 反組譯用 `-d`、多表加 `-e`、`-p` 防覆蓋；`-tc` 出 C 陣列給 EDK II；`-l` 的 mixed listing 是 ASL 對照 AML 的最佳教材。
- `-g` 抓系統表僅限 Windows；Linux 由 acpidump 負責，`-b` 直接出二進位。
- acpiexec 不開機就能載表、跑 Method、看 Debug 輸出——第 22、23 章的主戰場。
- roundtrip 不保證 byte-identical；比對改動用「雙反組譯再 diff」法。

動手練習

1. Errors / Warnings / Remarks / Optimizations 各代表什麼？哪一種會擋下 `.aml` 產出？
2. 為什麼反組譯 DSDT 時常要用 `-e` 帶上所有 SSDT？沒帶齊時會出現哪兩類症狀？
3. mixed listing 裡 Method 那行的第一個位元組是 `0x14`，後面的 `0x0B` 是什麼？對照第 3 章的 AML 知識回答。
4. 解釋 roundtrip 為何不保證 byte-identical，並寫出驗證「只改到我想改的地方」的三條指令。
5. 動手做：完成 labbox 三連發後，用 `acpiexec` 執行 `ADD2` 帶兩個自選參數。

Module 10

Linux ACPI Debugging

難度：★★★★☆

前九個模組教你把 ASL 寫出來；本模組教你在一台真實的 Linux 機器上觀測、驗證、除錯它。你將學會逛遍 sysfs 裡的 ACPI 觀測點、把韌體中的 DSDT 反組譯回 ASL、用 Debug 物件替 AML 加上 printf 式的輸出，最後看懂 kernel log 裡的 ACPI Error，一路追到硬體。這套流程是韌體工程師日常 debug 的核心武器庫。

第 22 章

Linux ACPI

寫了九個模組的 ASL，你可能還沒有真正「摸」過一台機器裡活著的 ACPI。Linux 是觀察 ACPI 最友善的平台：核心把 ACPI 的一舉一動攤在 `/sys` 與 kernel log 裡，工具鏈又與第 21 章的 iASL 系出同門。本章建立一條你之後會反覆走的分析路線：從 sysfs 找到裝置、反組譯 DSDT、讀懂 Method、對應到硬體——第 23、24 章的 Debug 與錯誤排除都蓋在這條路線上。

本章學習目標

- 說明 `/sys/firmware/acpi/` 與 `/sys/bus/acpi/devices` 之下各目錄與檔案的意義。
- 使用 acpidump、acpixtract 與 `iasl -d`，把韌體中的表反組譯回可讀的 ASL。
- 解讀 dmesg 中開機階段的 ACPI 表載入訊息。
- 依「Device → Method → OperationRegion → Hardware」的方法論閱讀 DSDT，並以筆電電池完整走一遍。
- 使用 acpiexec 在使用者空間安全地執行韌體的 AML。

22.1 Linux 上的 ACPI 觀測點

Linux 核心解譯完 AML、建好 Namespace 之後，把大量內部狀態匯出到 **sysfs**。你最常拜訪的是三個地方：原始表、列舉出的裝置、以及事件計數器。先看地圖：

Linux ACPI 檔案系統地圖

```
/sys/firmware/acpi/
 ├── tables/            ← 原始 ACPI Tables（二進位）
 │    ├── DSDT          ← 主要硬體描述（AML）
 │    ├── FACP          ← 即 FADT（指向 DSDT）
 │    └── SSDT1 SSDT2   ← 每張 SSDT 一個檔案
 └── interrupts/        ← SCI 與各 GPE 的觸發計數
      ├── gpe17         ← 單一 GPE：計數 狀態 遮罩
      ├── gpe_all       ← 所有 GPE 的總計數
      └── sci           ← SCI 中斷總數

/sys/bus/acpi/devices/
 └── PNP0C0A:00/        ← 每個 acpi_device 一個目錄
      └── path hid uid status ...  ← 對應 ASL 物件（見下表）
```

22.1.1 tables/：原始表的家

`/sys/firmware/acpi/tables/` 之下的每個檔案，就是一張完整的 ACPI Table，內容與韌體放進記憶體的二進位一模一樣：

原始表就躺在這裡

```
$ ls /sys/firmware/acpi/tables/
APIC  BGRT  DSDT  FACP  FACS  HPET  MCFG  SSDT1  SSDT2  data

$ sudo xxd -l 36 /sys/firmware/acpi/tables/DSDT
00000000: 4453 4454 a5f2 0100 028d 4c45 4e4f 564f  DSDT......LENOVO
00000010: 5450 2d4e 3332 2020 0010 0000 494e 544c  TP-N32  ....INTL
```

前 36 個 byte 正是標準表頭：signature、length、OEM ID，以及編譯器簽名 `INTL`（iASL）——**這些檔案就是韌體的原貨**。第 22.3 節的工具鏈，本質上就是把它們打包、還原成可讀的 ASL。

22.1.2 devices/：列舉出來的裝置

核心每從 Namespace 解譯出一個 `Device`，就建立一個 `struct acpi_device`，並在 `/sys/bus/acpi/devices/` 產生一個「`_HID:` 序號」目錄：

逛一遍列舉出的裝置

```
$ ls /sys/bus/acpi/devices/
ACPI0003:00  LNXPWRBN:00  PNP0A08:00  PNP0C0C:00  PNP0C0F:00
LNXCPU:00    LNXSYSTM:00  PNP0C02:00  PNP0C0A:00  PNP0C0D:00  ...

$ cd /sys/bus/acpi/devices/PNP0C0A:00
$ cat path hid uid status
\_SB_.PCI0.LPCB.EC0_.BAT0
PNP0C0A
1
31
```

每個目錄裡的檔案，都對應到你在前面模組寫過的 ASL 物件：

檔案	對應物件	內容
`path`	節點位置	Namespace 完整路徑（含底線補滿的 4 字元名）
`hid`	`_HID`	Hardware ID 字串，driver 比對的依據
`uid`	`_UID`	同型多實例時的區分編號
`status`	`_STA`	裝置狀態（十進位），開機時評估的結果
`physical_node`	—	symlink，指向綁定的 platform / i2c 等實體裝置

上例的 `status` 是 31，即 0x1F：bit 0 存在、bit 1 已啟用、bit 2 顯示於 UI、bit 3 功能正常、bit 4 電池存在——正是第 8 章 `_STA` 各 bit 的意義。`LNX` 開頭（`LNXSYSTM`、`LNXCPU`）是核心合成的節點，不是 DSDT 寫的。

22.1.3 interrupts/：抓事件的計數器

韌體要主動通知 OS 事件時，發出 **SCI**（System Control Interrupt）；SCI 的主要來源是 **GPE**（General Purpose Event），每個 GPE bit 觸發後核心執行 `_GPE` scope 下對應的 `_Lxx` / `_Exx` handler。這個目錄把每一路的觸發次數攤開給你看：

GPE 與 SCI 計數

```
$ cat /sys/firmware/acpi/interrupts/sci
8241

$ grep -H . /sys/firmware/acpi/interrupts/gpe1? | sort -t: -k2 -rn
/sys/firmware/acpi/interrupts/gpe17:    8117  enabled  unmasked
/sys/firmware/acpi/interrupts/gpe10:     121  enabled  unmasked
/sys/firmware/acpi/interrupts/gpe13:       3  enabled  unmasked
/sys/firmware/acpi/interrupts/gpe11:       0  invalid  unmasked
```

檔名與 `_Lxx` 的編號都是十六進位：`gpe17` 對應 `_L17` 或 `_E17`。除了計數，檔案還顯示該 GPE 是否啟用（`enabled` / `invalid`，後者表示沒有對應 handler）與是否被遮罩。這個目錄是抓 **GPE storm**（某路 GPE 瘋狂觸發、吃光 CPU）的神器，第 24 章的 Case Study 會實際用它辦案。

補充說明：固定事件也有計數

同目錄 `ff_` 開頭（如 `ff_pwr_btn` 電源鍵）是 Fixed Event 計數，`error` 記錄 GPE 處理出錯次數。`gpeXX` 檔案還可寫入 `echo disable` 暫時關閉——第 24 章的急救手段。

22.2 dmesg：開機時的 ACPI 足跡

核心開機的前幾毫秒，就把它找到的每一張表逐一列在 kernel log 裡。這是你確認「韌體到底交了什麼給 OS」最快的方法：

開機時的表載入訊息

```
$ sudo dmesg | grep "ACPI:" | head -8
ACPI: RSDP 0x00000000BFEFE014 000024 (v02 LENOVO)
ACPI: XSDT 0x00000000BFEFD0E8 0000B4 (v01 LENOVO TP-N32   00001000)
ACPI: FACP 0x00000000BFEF9000 0000F4 (v05 LENOVO TP-N32   00001000)
ACPI: DSDT 0x00000000BFEC7000 01F2A5 (v02 LENOVO TP-N32   00001000
      INTL 20230628)
```

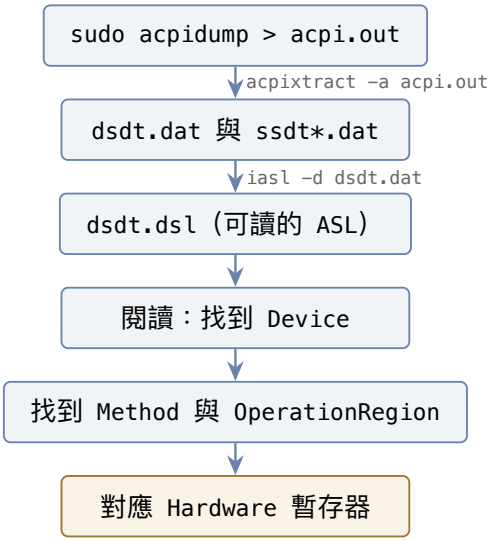
```
ACPI: SSDT 0x00000000BFEF8000 0003A2 (v02 LENOVO TP-SSDT  00001000
      INTL 20230628)
ACPI: Interpreter enabled
ACPI: PM: (supports S0 S3 S4 S5)
ACPI: EC: interrupt mode used
```

以 DSDT 那一行為例，欄位由左到右：signature 之後是表的實體記憶體位址與長度（`01F2A5`，十六進位 byte 數，約 127 KB）；`v02` 是表格式 revision（DSDT v2 代表 64 位元整數語意）；`LENOVO / TP-N32` 是 OEM ID 與 OEM Table ID——這張表來自哪家、哪個平台；結尾 `INTL 20230628` 是編譯器 ID 與版本，`INTL` 就是 iASL，日期即 ACPICA 釋出版本——看得出 OEM 用了多舊的工具鏈。

接著的 `Interpreter enabled` 表示 ACPICA 已完成 Namespace 建立；`PM: (supports S0 S3 S4 S5)` 是從 DSDT 掃出的 `_Sx` 睡眠狀態。開完機先跑一次 `sudo dmesg | grep -iE "acpi (bios )?error"` 是好習慣：沒輸出最好，有輸出也別慌——第 24 章教你逐字讀懂。

22.3 acpidump 工具鏈：把韌體變回原始碼

要讀 DSDT 的內容，得先把它從韌體手上拿回來、再反組譯成 ASL。標準流程三步，全部來自 `acpica-tools`（Debian / Ubuntu 套件名；Fedora 為 `acpica-tools`，Arch 為 `acpica`）：



標準三步：dump、extract、disassemble

```
$ sudo acpidump > acpi.out

$ acpixtract -a acpi.out
  DSDT -  127653 bytes written to dsdt.dat
  SSDT -     930 bytes written to ssdt1.dat
  ...
```

```
$ iasl -d dsdt.dat
Disassembly completed
ASL Output:    dsdt.dsl - 733725 bytes
```

`acpidump` 把記憶體中所有表以十六進位傾印成一個文字檔；`acpixtract -a` 把每張表還原成二進位 `.dat`；`iasl -d` 把 AML 反組譯回 `.dsl`。這份兩萬多行的 `dsdt.dsl` 就是這台筆電的完整硬體描述。

補充說明：反組譯的兩個實用技巧

- DSDT 與 SSDT 常互相引用，單獨反組譯會缺外部符號、導致方法參數個數判讀錯誤：用 `iasl -e ssdt*.dat -d dsdt.dat` 讓 SSDT 提供符號。
- 只要 DSDT 一張表時可直接 `sudo cp /sys/firmware/acpi/tables/DSDT dsdt.dat`；acpidump 的優點是打包全部、適合附在 bug report。

22.4 閱讀 DSDT 的方法論：以筆電電池為例

兩萬行的 dsl 檔不能從第一行讀起。韌體工程師讀 DSDT 永遠帶著問題來，沿著固定路徑走：**Device** → **Method** → **OperationRegion** → **Hardware**——先用 `_HID` 找到 Device，再看 OS 會呼叫的 Method，Method 讀寫的 Field 屬於哪個 OperationRegion，最後由 region 的空間型別與位址對應到真正的硬體。每一步都把答案縮小一圈。以最生活化的「筆電電池」完整走一遍（標準 `_HID` 是 `PNP0C0A`，Control Method Battery，第 8 章）。

第一步：從 sysfs 找到 Device。22.1 節已查出電池的路徑是 `\_SB_.PCI0.LPCB.EC0_.BAT0`——光看路徑就知道它是 Embedded Controller 的下游裝置。到 dsl 裡找到它：

Step 1: locate the device

```
$ grep -n "PNP0C0A" dsdt.dsl
9482:            Name (_HID, EisaId ("PNP0C0A"))
```

第二步：找到 Method。跳到 dsl 第 9482 行附近，看 `Device (BAT0)` 的內容（節錄並簡化）：

dsdt.dsl 節錄：電池裝置與 _BST

```
// dsdt.dsl excerpt (simplified for teaching)
Scope (\_SB.PCI0.LPCB.EC0)
{
    Device (BAT0)
    {
        Name (_HID, EisaId ("PNP0C0A"))  // Control Method Battery
        Name (_UID, One)

        Name (PBST, Package (0x04)
        {
```

```
            Zero,   // [0] state: bit0 discharging, bit1 charging
            Ones,   // [1] present rate (mA or mW)
            Ones,   // [2] remaining capacity
            Ones    // [3] present voltage (mV)
        })

        Method (_BST, 0, Serialized)
        {
            PBST [Zero] = BST0     // EC field: state
            PBST [One]  = BRTE     // EC field: rate
            PBST [0x02] = BRCP     // EC field: capacity
            PBST [0x03] = BVLT     // EC field: voltage
            Return (PBST)
        }
    }
}
```

逐行拆解：PBST 是規格定義的 4 元素 Package（狀態、瞬時電流或功率、剩餘電量、電壓），OS 每次查電量都呼叫 _BST （Battery Status），把四個 **Field** 名字（BST0、BRTE……）的值填進 Package 後回傳——_BST 本身不碰硬體位址，真正的存取藏在 Field 宣告裡。靜態資訊（設計容量、序號等）在兄弟方法 _BIF / _BIX，讀法完全相同。

第三步：找到 **OperationRegion**。在 dsl 裡搜尋 BST0 的宣告處：

dsdt.dsl 節錄：EC 的 OperationRegion 與 Field

```
// dsdt.dsl excerpt (simplified for teaching)
Scope (\_SB.PCI0.LPCB.EC0)
{
    OperationRegion (ERAM, EmbeddedControl, Zero, 0xFF)
    Field (ERAM, ByteAcc, Lock, Preserve)
    {
        Offset (0xA0),
        BST0,   8,     // battery state
        BRTE,   16,    // present rate
        BRCP,   16,    // remaining capacity
        BVLT,   16     // present voltage
    }
}
```

第四步：對應 **Hardware**。region 空間是 EmbeddedControl：EC 內部 256 byte 的暫存器窗口（第 9–11 章）。「讀電池電壓」的完整真相：OS 呼叫 _BST → interpreter 讀 Field BVLT → EC 協定讀暫存器 0xA5..0xA6 → EC 韌體透過 SMBus 問電池的 gas gauge 晶片。ASL 的角色，就是把 EC 暫存器配置翻譯成 OS 聽得懂的標準介面。

最後用 Linux 的電池驅動驗證這條鏈真的在動：

Step 4: verify via the battery driver

```
$ cat /sys/class/power_supply/BAT0/status
Discharging
$ cat /sys/class/power_supply/BAT0/voltage_now
11460000
```

`voltage_now` 的單位是 μV：battery driver 呼叫 `_BST` 拿到 mV 值 11460，乘以 1000 匯出；state、rate、capacity 欄位則對應 `status`、`current_now`（或 `power_now`）與 `charge_now` / `energy_now`。「EC 暫存器 → Field → `_BST` → sysfs」這條翻譯鏈至此完整對上。

Lab 實作：在自己的筆電重走一遍

1. 安裝工具：`sudo apt install acpica-tools`。
2. 反　組　譯：`sudo acpidump > acpi.out`、`acpixtract -a acpi.out`、`iasl -e ssdt*.dat -d dsdt.dat`。
3. 用 `cat /sys/bus/acpi/devices/PNP0C0A:00/path` 找出電池的 Namespace 路徑。
4. 在 `dsdt.dsl` 找到 `_BST`，畫出「Method → Field → EC offset」對照表。
5. 對照 `/sys/class/power_supply/BAT0/` 驗證單位換算；桌機改追 `PNP0C0D`（Lid）或 `PNP0C0C`（電源鍵）。

22.5 acpiexec：在使用者空間執行 AML

讀懂了 ASL，下一個願望自然是「跑跑看」，但改一行 DSDT 就重開機太昂貴。**acpiexec** 把 ACPICA interpreter 包成使用者程式：載入你給的表、建好 Namespace，讓你像用 gdb 一樣呼叫任何 Method。

acpiexec 互動 session：載入、瀏覽、執行

```
$ acpiexec dsdt.dat ssdt1.dat ssdt2.dat
Table [DSDT] - 2143 Objects with 87 Devices, 612 Methods

- find BAT0
\_SB_.PCI0.LPCB.EC0_.BAT0 Device

- evaluate \_SB.PCI0.LPCB.EC0.BAT0._STA
Evaluation returned object 0x55e3c2a4, buffer length 18
  [Integer] = 000000000000000F

- execute \_SB.PCI0.LPCB.EC0.BAT0._BST
Execution returned object 0x55e3c4f0, buffer length 78
  [Package] Contains 4 Elements:
```

```
    [Integer] = 0000000000000000
    ...(all four elements are zero)
```

除了示範中的 `find`（用 4 字元名反查完整路徑）與 `evaluate`（執行 Method，`execute` 是同義字），常用的還有 `namespace`（傾印整棵或某子樹的 Namespace）、`methods`（列出所有 Method）與 `help`。帶參數執行時以空白分隔、數字用 `0x` 前綴、字串加引號——第 23 章會實際用到。

注意上面的執行結果：機器明明插著電池，`_STA` 卻回 0x0F（bit 4 電池存在沒亮）、`_BST` 全是 0。這不是 bug：

注意：acpiexec 跑的是模擬硬體

acpiexec 在使用者空間執行，不能碰真正的 EC、MMIO 或 I/O Port：所有 OperationRegion 讀寫都由模擬 handler 接住，讀取通常回 0。它適合驗證控制流程、型別、**Namespace** 解析（第 24 章會用它重現錯誤）；要看真實硬體值，回到真機的 `/sys/class/power_supply/`。

即使讀值是假的，acpiexec 仍讓「改 ASL → 編譯 → 執行」的迴圈從一次重開機縮短到一秒鐘——第 23 章的 Debug 實驗會先在這裡跑通，再放上真機。

核心觀念：本章重點回顧

- `/sys` 三大觀測點：`tables/`（韌體原貨）、`devices/`（列舉結果：path / hid / uid / status）、`interrupts/`（GPE / SCI 計數器）。
- dmesg 開機訊息列出每張表的位址、大小、OEM 與編譯器版本，是確認「韌體交了什麼」的第一手資料。
- 標準工具鏈：`sudo acpidump > acpi.out` → `acpixtract -a` → `iasl -d`（記得 `-e` 帶上 SSDT）。
- 讀 DSDT 的固定路徑：Device → Method → OperationRegion → Hardware。
- acpiexec 讓你在使用者空間執行 AML；region 讀寫是模擬的，驗流程不驗真值。

動手練習

1. `tables/`、`devices/`、`interrupts/` 三個目錄各自回答什麼問題？除錯時分別在哪一步派上用場？
2. 某裝置的 `status` 檔案顯示 11（十進位）。換算成 `_STA` 的 bit，這個裝置處於什麼狀態？OS 會載入它的 driver 嗎？
3. 為什麼反組譯 DSDT 時建議加 `-e ssdt*.dat`？少了它會出什麼類型的問題？
4. 動手：完成 22.4 節的 Lab，並找出上蓋開關（`PNP0C0D`）的 `_LID` 讀哪個 Field、屬於哪個 OperationRegion。
5. 動手：用 acpiexec 載入反組譯出的表，`find` 電池（或 Lid），`evaluate` 其 `_STA` 與 `_BST`，解釋你看到的回傳值。

第 23 章

ACPI Debug

C 程式不會動了，你會加 printf；AML 在核心深處執行，既沒有終端機也沒有除錯器，怎麼辦？ASL 其實內建了自己的 printf——**Debug 物件**。往它寫任何東西，interpreter 就替你印出來。問題是：印去哪裡？要開哪些開關才看得到？這正是本章要打通的環節。

本章先講 Debug 物件的語法與原理，接著沿著「ASL → AML → Interpreter → kernel log」把輸出的旅程走一遍，最後動手做一個完整的 Lab：寫一張會說話的 SSDT，讓它在你的 dmesg 裡留言。

本章學習目標

- 使用 `Store (..., Debug)` 與 `Debug = ...` 兩種寫法輸出除錯訊息。
- 畫出 Debug 輸出從 ASL 原始碼到 kernel log 的完整路徑。
- 說明 `CONFIG_ACPI_DEBUG`、`acpi.debug_layer` 與 `acpi.debug_level` 三個開關的分工，並寫出常用組合。
- 於執行期透過 `/sys/module/acpi/parameters/` 調整輸出。
- 用 SSDT 加 initrd（或 acpiexec）實際看到自己的 Debug 輸出。

23.1 Debug 物件：一個只能寫的虛擬物件

`Debug` 是 ASL 預先定義的特殊物件：它不存在於 Namespace、沒有型別、**只能當 Store 的目標**，不能讀。每次有值寫入，interpreter 就把該值連同型別印到除錯輸出。兩種寫法完全等價：

片段：Debug 物件的兩種寫法

```
Store ("Hello ACPI", Debug)     // legacy operator style
Debug = "Hello ACPI"            // ASL 2.0 symbolic style

Store (Local0, Debug)           // print a local variable
Debug = Arg0                    // print a method argument
Debug = CNT0                    // print a named object
```

舊韌體與反組譯輸出多為 `Store` 寫法，新專案建議用 `Debug =`——這也是本書一貫「兩種語法都要認得」的原則。各型別的輸出樣式如下（以 kernel log 為例）：

寫入型別	輸出樣式（示意）
String	`ACPI Debug: String [0x0A] "Hello ACPI"`
Integer	`ACPI Debug: Integer 0x000000000000002A`
Buffer	`ACPI Debug: Buffer [0x04] = 0xDE 0xAD 0xBE 0xEF`
Package	`ACPI Debug: Package [0x02]`，並逐一遞迴列出元素

方括號內是長度；Integer 一律以 64 位元十六進位列印。把這些樣式記個大概，之後在幾千行 dmesg 中一眼就能撈出自己的訊息。

下面是本章的主角：一張完整、可編譯的 SSDT，開機時會自我介紹。（`_INI` 會在核心列舉到該裝置、且 `_STA` 表示存在時自動執行一次——這讓我們不需要任何 driver 就能觸發 Debug 輸出。）

dbgdemo.dsl：會在開機時說話的 SSDT

```
DefinitionBlock ("dbgdemo.aml", "SSDT", 2, "DEMO", "DBGDEMO",
                 0x00000001)
{
    Scope (\_SB)
    {
        Device (DBG0)
        {
            Name (_HID, "DEMO0002")
            Name (MSG0, "Hello ACPI from DBG0")

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)   // present, enabled, functioning
            }

            Method (_INI, 0, Serialized)
            {
                Debug = "DBG0._INI: SSDT is alive"
                Debug = MSG0
                Local0 = 0x2A
                Debug = Local0
                Debug = Buffer (0x04) { 0xDE, 0xAD, 0xBE, 0xEF }
                Store ("legacy style still works", Debug)
            }

            Method (DBLE, 1, Serialized)
            {
                Debug = Arg0    // echo the argument
                Return (Arg0 * 2)
            }
```

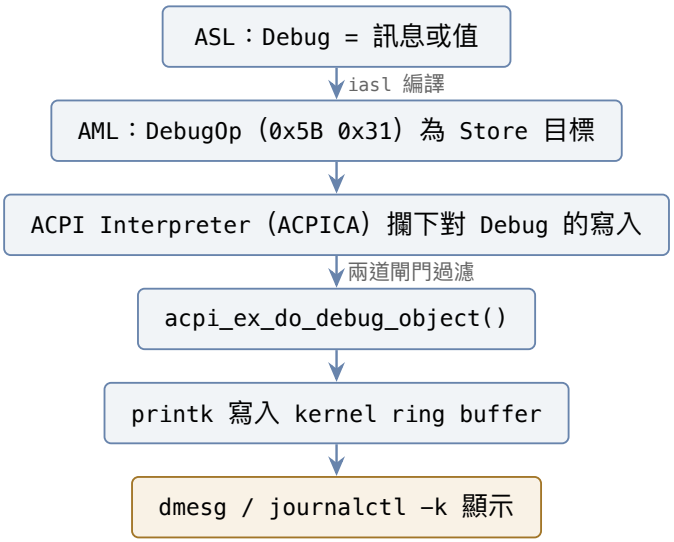
```
        }
    }
}
```

逐段拆解：

- `DefinitionBlock` 的簽章是 `SSDT`：我們是去 補充 DSDT，不是取代它（第 18–20 章的做法）。
- `_STA` 回 0x0F，確保核心認為裝置存在，才會呼叫 `_INI`。
- `_INI` 依序印出 String、Named object、Local、Buffer，以及一行 legacy 寫法——正好把上表的樣式各示範一次。
- `DBLE` 帶一個參數並回傳兩倍，留給 acpiexec 練習參數傳遞。

23.2 一行 Debug 的旅程

「寫入 Debug 就會印出來」的背後是一條完整的管線。理解它，之後任何一站沒動靜你都知道去哪查：



編譯期，`Debug` 變成 AML 的 `DebugOp`（extended opcode `0x5B 0x31`）。執行期，interpreter 發現 Store 的目標是 Debug，就呼叫 ACPICA 的 `acpi_ex_do_debug_object()` 把值格式化後丟給 OS 的列印服務——Linux 上就是 `printk`。但訊息要真的出現在 dmesg，必須通過兩道閘門，這是下一節的主題。

23.3 打開輸出：兩道閘門

閘門一：編譯期選項 `CONFIG_ACPI_DEBUG`。 ACPICA 的所有除錯列印（包括 Debug 物件）都包在條件編譯裡；核心沒開這個選項，程式碼根本不存在。先確認你的核心：

檢查開門一

```
$ grep CONFIG_ACPI_DEBUG /boot/config-$(uname -r)
CONFIG_ACPI_DEBUG=y
```

多數發行版的預設核心有開此選項（例如 Ubuntu），但若顯示 `is not set`，就得改用發行版的 debug 核心或自行編譯。

開門二：執行期的 **`debug_layer`** 與 **`debug_level`**。兩個 32 位元的 bitmask，共同決定哪些訊息放行：

- `debug_layer`——誰可以說話：ACPICA 依元件劃分（interpreter、namespace、事件……），每個元件一個 bit。
- `debug_level`——說哪一類話：訊息依類型劃分（初始化、資訊、Debug 物件輸出……），每類一個 bit。

兩者都過才會印。常用的名稱與值（節錄自 `include/acpi/acoutput.h`）：

名稱	值	意義
ACPI_ALL_COMPONENTS	0xFFFFFFFF	layer：全部元件
ACPI_EVENTS	0x00000004	layer：GPE 與 Notify 路徑
ACPI_NAMESPACE	0x00000010	layer：Namespace 操作
ACPI_EXECUTER	0x00000080	layer：AML 執行核心
ACPI_LV_INIT	0x00000001	level：初始化訊息
ACPI_LV_DEBUG_OBJECT	0x00000002	level：**Debug 物件輸出**
ACPI_LV_INFO	0x00000004	level：一般資訊

實務上你只需要背一個組合——「只看 Debug 物件輸出」：layer 全開、level 只開 `ACPI_LV_DEBUG_OBJECT`：

目的	debug_layer	debug_level
只看 Debug 物件輸出（標準組合）	0xFFFFFFFF	0x00000002
Debug 物件 + 初始化訊息	0xFFFFFFFF	0x00000003
一般資訊全開	0xFFFFFFFF	0x00000007
全部訊息（洪水，慎用）	0xFFFFFFFF	0xFFFFFFFF

設定方式有二。開機參數（對 `_INI` 這類開機期執行的方法是唯一選擇）：

開機參數（編輯 GRUB）

```
$ sudo nano /etc/default/grub
GRUB_CMDLINE_LINUX="acpi.debug_layer=0xFFFFFFFF acpi.debug_level=0x2"
$ sudo update-grub && sudo reboot
```

或執行期動態調整（對之後才執行的方法生效，例如 EC query、driver 呼叫的方法）：

執行期調整

```
# echo 0xFFFFFFFF > /sys/module/acpi/parameters/debug_layer
# echo 0x00000002 > /sys/module/acpi/parameters/debug_level
```

注意：別把 log 開成洪水

`debug_level=0xFFFFFFFF` 會讓每個 AML opcode、每次 namespace 查找都印一行，每秒數千行，瞬間淹掉 ring buffer，機器也會明顯變慢。需要全開時搭配 `log_buf_len=8M` 加大緩衝區，抓完立刻關掉。日常除錯永遠從 `0x2`（只看 Debug 物件）開始，不夠再加。

補充說明：新版核心的預設值

較新的 ACPICA 已把 `ACPI_LV_DEBUG_OBJECT` 納入預設 `debug_level`，因此有些機器只開 `CONFIG_ACPI_DEBUG` 就能看到 Debug 輸出。但預設值隨版本變動，顯式設定才不吃版本運氣——寫進 SOP 的永遠是上面那組參數。

23.4 Lab：讓核心印出你的 Debug 訊息

把 23.1 的 `dbgdemo.dsl` 放上真機。Linux 提供 **Table Upgrade** 機制：把 `.aml` 打包成 cpio 檔接在 initrd 前面，核心開機時會安裝這張 SSDT（同第 20 章載入 SSDT 的思路，這裡走 initrd 版本）。

Lab 實作：SSDT + initrd + dmesg 完整流程

1. 編譯：`iasl dbgdemo.dsl` 產生 `dbgdemo.aml`。
2. 打包並串接：aml 放進 cpio 的 `kernel/firmware/acpi/` 路徑，接在現有 initrd 前面（見下方指令）。
3. 開機：GRUB 選單按 `e`，`initrd` 行改指向新檔、`linux` 行尾加上 `acpi.debug_layer=0xFFFFFFFF acpi.debug_level=0x2`，`Ctrl-x` 開機（只影響本次）。
4. 驗證：`sudo dmesg | grep -E "Table Upgrade|ACPI Debug"`。

步驟 1–3 的實際指令

```
$ iasl dbgdemo.dsl
$ mkdir -p kernel/firmware/acpi
$ cp dbgdemo.aml kernel/firmware/acpi/
$ find kernel | cpio -H newc --create > ssdt.cpio
$ cat ssdt.cpio /boot/initrd.img-$(uname -r) > initrd-dbg.img
$ sudo cp initrd-dbg.img /boot/
```

dmesg 中的成果

```
$ sudo dmesg | grep -E "Table Upgrade|ACPI Debug"
[    0.008112] ACPI: Table Upgrade: install [SSDT-DEMO -DBGDEMO]
[    2.310547] ACPI Debug:  String [0x18] "DBG0._INI: SSDT is alive"
[    2.310551] ACPI Debug:  String [0x14] "Hello ACPI from DBG0"
[    2.310554] ACPI Debug:  Integer 0x000000000000002A
[    2.310558] ACPI Debug:  Buffer [0x04] = 0xDE 0xAD 0xBE 0xEF
[    2.310561] ACPI Debug:  String [0x18] "legacy style still works"
```

第一行是核心宣告安裝了我們的 SSDT；接著五行正是 `_INI` 的五個 Debug 寫入，型別與長度都對得上 23.1 的樣式表。若核心開了 `CONFIG_ACPI_CONFIGFS`，還可免重開機：`modprobe acpi_configfs` 後在 `/sys/kernel/config/acpi/table/` 下 `mkdir` 一個目錄、把 aml `cat` 進其中的 `aml` 檔即可（多數發行版預設沒開，故本 Lab 以 initrd 為主）。

23.5 更快的迴圈：acpiexec 直接印

上真機前，先讓 SSDT 在 acpiexec 裡跑通。acpiexec 不需要任何開關，Debug 輸出直接印在終端機：

秒級開發迴圈

```
$ iasl dbgdemo.dsl && acpiexec dbgdemo.aml

- execute \_SB.DBG0._INI
Executing \_SB.DBG0._INI
[ACPI Debug]  String [0x18] "DBG0._INI: SSDT is alive"
[ACPI Debug]  String [0x14] "Hello ACPI from DBG0"
[ACPI Debug]  Integer 0x000000000000002A
[ACPI Debug]  Buffer [0x04] = 0xDE 0xAD 0xBE 0xEF
[ACPI Debug]  String [0x18] "legacy style still works"

- execute \_SB.DBG0.DBLE 0x15
Executing \_SB.DBG0.DBLE
[ACPI Debug]  Integer 0x0000000000000015
Execution returned object 0x5601a8c2, buffer length 18
  [Integer] = 000000000000002A
```

第二段示範參數傳遞：`0x15`（21）傳入 `Arg0`，Debug 印出它，方法回傳 0x2A（42）。建議的工作節奏：**acpiexec 迭代到邏輯正確，再用 initrd 上真機驗證與硬體互動的部分**。

補充說明：輸出格式因版本而異

kernel 前綴為 `ACPI Debug:`，acpiexec 為 `[ACPI Debug]`，長度欄位與縮排在不同 ACPICA 版本間略有出入。寫解析腳本時抓語意（型別關鍵字與引號內容），不要寫死整行格式。

23.6 進階：Method Tracing 簡介

Debug 物件要你事先在原始碼裡佈點；如果拿到的是**別人的**韌體，想看某個 Method 的完整執行軌跡呢？ACPICA 提供 method tracing：指定方法名，interpreter 在進入該方法時自動打開你指定的 layer / level，離開時關閉——等於「只對這個方法開洪水」。

追蹤單一 Method（執行期設定）

```
# cd /sys/module/acpi/parameters
# echo "\_SB.DBG0.DBLE" > trace_method_name
# echo method > trace_state
```

`trace_state` 可為 `method`（每次都追）、`method-once`（只追第一次）、`disable`（關閉）；搭配 `trace_debug_layer` 與 `trace_debug_level` 指定追蹤期間放行的訊息。同名的開機參數（`acpi.trace_method_name=` 等）可以從第一次執行就開追。完整說明見核心文件 `Documentation/firmware-guide/acpi/method-tracing.rst`。

核心觀念：本章重點回顧

- Debug 是只能寫的虛擬物件：寫入即列印，型別(String / Integer / Buffer / Package)自動標註。
- 路徑：ASL → AML DebugOp（0x5B 0x31）→ interpreter → printk → dmesg。
- 兩道閘門：編譯期 `CONFIG_ACPI_DEBUG=y`；執行期 `debug_layer`（誰說話）與 `debug_level`（說哪類話）。
- 標準組合：`acpi.debug_layer=0xFFFFFFFF acpi.debug_level=0x2`；執行期可透過 `/sys/module/acpi/parameters/` 調整。
- 開發節奏：acpiexec 秒級迭代邏輯，initrd Table Upgrade 上真機驗證。
- method tracing 能只對單一方法開啟完整追蹤，適合分析別人的韌體。

動手練習

1. 為什麼 Debug 物件被設計成「只能寫」？如果允許讀取，會對 interpreter 與韌體行為造成什麼問題？
2. `Debug = Package (0x02) {0x01, "AB"}` 會印出幾行、各是什麼樣式？對照 23.1 的樣式表回答。
3. 想同時看到 Debug 物件輸出與初始化訊息，`debug_level` 應設多少？寫出計算過程。

4. 動手：修改 `dbgdemo.dsl`，在 `_INI` 中用 `While` 迴圈印出 1 到 5，用 acpiexec 驗證輸出。
5. 動手（真機）：完成 23.4 的 initrd Lab，把你的名字放進 `MSG0`，截圖 dmesg 的輸出。

第 24 章

ACPI Error Debugging

打開任何一台量產筆電的 dmesg，十之八九能撈到幾行 `ACPI Error`。有些是無害噪音，有些正是「休眠喚不醒」「風扇狂轉」「電量顯示卡死」的元兇。分辨兩者、並把有害的那種一路追到 ASL 原始碼，是韌體工程師最值錢的技能之一。

本章先解剖錯誤訊息的每個欄位，建立十個常見錯誤碼的直覺，再用三個擬真 Case Study 完整演練「Kernel Log → ACPI Error → Method → ASL Source → Namespace → Hardware」這條定位鏈。

本章學習目標

- 拆解一行 ACPI Error：方法路徑、錯誤碼、ACPICA 版本與原始碼位置。
- 說出十個常見 `AE_*` 錯誤碼的意義、典型成因與修法。
- 依定位鏈從 kernel log 一路追到 ASL 與硬體。
- 診斷並修復三類實戰問題：`AE_ALREADY_EXISTS`、`AE_NOT_FOUND` 與 GPE storm。
- 使用 fwts 對韌體做自動化健檢。

24.1 解剖一行 ACPI Error

先看一組典型的錯誤（實際各為單行，此處為排版換行；下同）：

kernel log 中的典型 ACPI 錯誤串

```
$ sudo dmesg | grep -B1 "Aborting method"
[  312.480211] ACPI BIOS Error (bug): Could not resolve symbol
    [\_SB.PCI0.LPCB.EC0.XDAT], AE_NOT_FOUND (20230628/psargs-330)
[  312.480230] ACPI Error: Aborting method \_SB.PCI0.LPCB.EC0._Q66
    due to previous error (AE_NOT_FOUND) (20230628/psparse-529)
```

逐段解讀第二行：

片段	意義
`ACPI Error:`	訊息來自 ACPICA（核心的 ACPI 元件）
`Aborting method ..._Q66`	被中止的 Method 的完整 Namespace 路徑——這台機器的 EC query 66 號
`(AE_NOT_FOUND)`	ACPICA 錯誤碼：找不到引用的名稱
`20230628`	ACPICA 版本（釋出日期），回報 bug 必附
`psparse-529`	拋錯位置：ACPICA 原始碼 `psparse.c` 第 529 行——拿去搜尋核心原始碼可直達現場

兩個關鍵閱讀習慣：

- **找第一行。**錯誤常成串出現：第一行（本例的 `Could not resolve symbol [...XDAT]`）才是根因，後面的 `Aborting method` 是連鎖反應。從最早的那行開始讀。
- **看前綴。**`ACPI BIOS Error (bug)` 是 ACPICA 明確判定「錯在韌體交來的表，不在核心」；純 `ACPI Error` 則兩種可能都有。看到 BIOS Error，方向直接轉向 DSDT / SSDT。

補充說明：錯誤碼定義在哪裡

所有 `AE_` 開頭的碼定義於 ACPICA 的 `acexcep.h`（核心原始碼 `include/acpi/acexcep.h`），`AE` 即 ACPI Exception。其中 `AE_AML_` 子類代表「執行 AML 期間」的錯誤——十之八九可以在 ASL 原始碼裡找到對應的病灶。

24.2 常見錯誤碼速查表

錯誤碼	意義	典型成因	修法
AE_ALREADY_EXISTS	要建立的名稱已存在	NotSerialized 方法內用 Name() 且被併發重入；同一張 SSDT 載入兩次	方法改 Serialized；檢查重複載表
AE_NOT_FOUND	找不到引用的名稱	韌體版本差異；External 騙過編譯器但執行期不存在；路徑拼錯	CondRefOf 防禦；用 acpiexec 核對 Namespace
AE_AML_OPERAND_TYPE	運算元型別不符	Method 某路徑沒回傳值卻被當值用；把 Package 當 Integer 運算	檢查所有 Return 路徑與 ObjectType
AE_AML_UNINITIALIZED_ARG	使用了未傳入的 ArgX	呼叫端參數個數不足（_DSM 轉呼叫是慣犯）	對齊宣告與呼叫的參數個數
AE_AML_UNINITIALIZED_LOCAL	使用未初始化的 LocalX	某條 If 分支忘了先賦值就使用	進方法先初始化再分支
AE_AML_PACKAGE_LIMIT	Package 索引越界	Index 超出元素個數；迴圈上限寫錯	以 SizeOf 界定索引範圍
AE_AML_BUFFER_LIMIT	Buffer 索引越界	CreateField 系列的位移或長度算錯	核對位移、長度與 Buffer 大小
AE_AML_INVALID_RESOURCE_TYPE	資源描述子型別錯誤	_CRS / _SRS 回傳的 Buffer 不是合法的 resource template	用 ResourceTemplate 重寫並以 iasl 驗證
AE_TIME	等待逾時	Acquire / Wait 逾時；EC 交易沒回應	查 Mutex 死鎖順序與 EC 狀態
AE_AML_LOOP_TIMEOUT	While 迴圈超過時限	輪詢的硬體 bit 永遠不變（預設保險絲約 30 秒）	迴圈加計數上限；查硬體為何不回應

注意：不是每行 ACPI Error 都要修

量產機器的 log 常有「無害噪音」：例如 _DSM 查詢不支援的功能、_OSI 分支導致某平台本來就沒有的物件被引用一次。判斷準則是**有沒有伴隨功能異常**（睡眠、電池、熱管理、熱鍵……）。先確認症狀，再決定投入；別看到紅字就急著改 DSDT 或刷 BIOS。

24.3 Case Study 1：AE_ALREADY_EXISTS——重入的代價

症狀：使用者回報「連按亮度熱鍵，螢幕亮度偶爾卡住」。dmesg 出現：

Case 1 的錯誤訊息

```
[  201.118402] ACPI Error: [TBUF] Namespace lookup failure,
    AE_ALREADY_EXISTS (20230628/dswload2-326)
[  201.118420] ACPI Error: Aborting method
    \_SB.PCI0.LPCB.EC0._Q11 due to previous error
    (AE_ALREADY_EXISTS) (20230628/psparse-529)
```

定位：根因行說「建立 TBUF 時發現名稱已存在」；被中止的是 EC query _Q11（亮度熱鍵的事件 handler，第 12–14 章）。反組譯 DSDT，找到：

dsdt.dsl 節錄：有 bug 的 _Q11

```
// dsdt.dsl excerpt: brightness hotkey EC query (buggy)
Method (_Q11, 0, NotSerialized)     // BUG: NotSerialized
{
    Name (TBUF, Package (0x02) { Zero, Zero })
    TBUF [Zero] = BRTL              // EC field: current level
    TBUF [One]  = BRTM              // EC field: max level
    \_SB.PCI0.GFX0.UPBL (TBUF)      // ask GFX to update backlight
}
```

機轉：方法內的 Name() 在執行期建立節點、方法結束時拆除。使用者連按熱鍵，EC 連續送出兩個 query，第二個 _Q11 在第一個還沒退出時進場——第二次 Name (TBUF, ...) 發現節點還在，拋出 AE_ALREADY_EXISTS，方法中止，亮度就「卡住」了。

修正：把宣告改成 Method (_Q11, 0, Serialized)——一字之差，interpreter 會替方法加上隱式 Mutex，重入的執行緒排隊等前一個完成，Name() 不再撞名。這正是第 12 章的鐵律「方法內有 Name() 就宣告 Serialized」的實戰版。現代 iasl 對 NotSerialized 方法內的 Name() 會直接給出 remark 警告——編譯器的嘮叨要當真。

24.4 Case Study 2：AE_NOT_FOUND——防禦性引用

症狀：某批機器升級 EC 韌體後，dmesg 在熱事件時報錯：

Case 2 的錯誤訊息

```
[   88.130077] ACPI BIOS Error (bug): Could not resolve symbol
    [\_SB.PCI0.LPCB.EC0.SEN3], AE_NOT_FOUND (20230628/psargs-330)
[   88.130095] ACPI Error: Aborting method
    \_SB.PCI0.LPCB.EC0._QD5 due to previous error
    (AE_NOT_FOUND) (20230628/psparse-529)
```

定位：_QD5 引用了 EC field SEN3，但這台機器的 DSDT 沒宣告它——這張 SSDT 是兩個機種共用，另一機種的 EC 多一顆感測器。編譯時的 External 宣告（第 19 章）只是「我保證執行期會有」的承諾，兌現不了就是 AE_NOT_FOUND：

ssdt 節錄：有 bug 的 _QD5

```
// ssdt excerpt: thermal sensor query (buggy)
External (\_SB.PCI0.LPCB.EC0.SEN3, FieldUnitObj)

Method (_QD5, 0, Serialized)
{
```

```
    Local0 = \_SB.PCI0.LPCB.EC0.SEN3   // may not exist!
    If (Local0 > 0x5F)
    {
        Notify (\_TZ.THM0, 0x80)       // re-evaluate temperature
    }
}
```

修正：用 CondRefOf 先確認物件存在。它回傳真假而不會因為找不到而拋錯，是跨韌體版本的標準防禦寫法：

修正後的 _QD5：CondRefOf 防禦

```
Method (_QD5, 0, Serialized)
{
    If (CondRefOf (\_SB.PCI0.LPCB.EC0.SEN3, Local0))
    {
        If (DerefOf (Local0) > 0x5F)
        {
            Notify (\_TZ.THM0, 0x80)
        }
    }
    Else
    {
        Debug = "_QD5: SEN3 absent on this EC firmware"
    }
}
```

CondRefOf 成功時把參照放進 Local0，用 DerefOf 取值才會真正去讀 EC。Else 分支順手放一行 Debug（第 23 章），日後遇到同型問題 dmesg 直接告訴你原因。在 acpiexec 裡把兩個版本各跑一次，就能離線驗證修正。

24.5 Case Study 3：GPE Storm——用計數器抓兇手

症狀：機器待機也風扇狂轉，top 顯示 kworker 執行緒吃掉一顆核心。這是 GPE storm 的經典長相：某路 GPE 不停觸發，核心疲於執行它的 handler。用第 22 章的計數器辦案：

Step 1：確認 SCI 暴增，鎖定兇手 GPE

```
$ cat /sys/firmware/acpi/interrupts/sci; sleep 5; \
  cat /sys/firmware/acpi/interrupts/sci
418772
601233

$ grep -H . /sys/firmware/acpi/interrupts/gpe?? | sort -t: -k2 -rn \
  | head -3
```

```
/sys/firmware/acpi/interrupts/gpe17:  600911  enabled  unmasked
/sys/firmware/acpi/interrupts/gpe09:    3120  enabled  unmasked
/sys/firmware/acpi/interrupts/gpe02:     411  enabled  unmasked
```

五秒漲了 18 萬次，幾乎全記在 gpe17 頭上。GPE 編號是十六進位，對應 handler 就是 _GPE scope 下的 _L17（level）或 _E17（edge）。到 dsl 裡看：

dsdt.dsl 節錄：有 bug 的 _L17

```
// dsdt.dsl excerpt: thermal GPE handler (buggy)
Scope (\_GPE)
{
    Method (_L17, 0, NotSerialized)
    {
        If (\_SB.PCI0.LPCB.EC0.TSTS)   // thermal status bit
        {
            Notify (\_TZ.THM0, 0x80)   // re-evaluate temperature
        }
        // BUG: TSTS is never cleared. The level-triggered
        // GPE line stays asserted and fires again immediately.
    }
}
```

機轉：_L17 是 level-triggered handler：只要事件源沒被清掉，GPE 線持續有效，處理完馬上又觸發。這個 handler 讀了狀態 bit TSTS 卻忘了清除（該 bit 規格為 write-1-to-clear），於是無限循環。

修正：在 handler 內先清事件源再做事——於 If 區塊開頭、Notify 之前加上一行 _SB.PCI0.LPCB.EC0.TSTS = One（write-1-to-clear），GPE 線放掉，storm 即止。

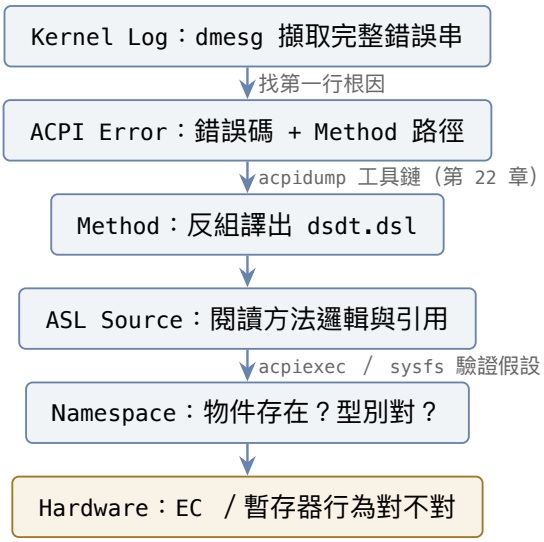
救急 **workaround**（改不了韌體時）：執行期把 disable 寫進 interrupts/gpe17（echo disable > 該檔案），或用開機參數 acpi_mask_gpe=0x17 永久遮罩——代價是犧牲該 GPE 承載的功能（本例為熱事件通知），只能當止血帶。

Lab 實作：體檢你的機器有沒有隱性 storm

1. 記下 gpe_all 的值，等 60 秒再記一次，相減得出每分鐘 GPE 次數（安靜的機器通常是個位數到數十）。
2. 若異常，用本節的 grep + sort 找出最活躍的 GPE。
3. 在你反組譯的 dsdt.dsl 中找到對應的 _Lxx / _Exx，讀懂它服務的裝置，判斷觸發頻率是否合理。

24.6 通用定位流程與檢查清單

三個 Case 走的其實是同一條鏈——課綱那條你必須內化的定位流程：



核心觀念：ACPI 錯誤除錯 checklist

1. 收證據：完整 dmesg（含錯誤前後各十行）、BIOS 版本、ACPICA 版本（錯誤行裡那個日期）。
2. 看前綴：`ACPI BIOS Error (bug)` 直接朝韌體方向查。
3. 找第一行根因，記下 Method 路徑與 `AE_` 碼，對照 24.2 速查表。
4. 反組譯 DSDT 與所有 SSDT（記得 `iasl -e`），讀該 Method 與它引用的每個名稱。
5. 用 `/sys/bus/acpi/devices` 與 acpiexec 驗證 Namespace 假設；GPE 問題加看 `interrupts/` 計數。
6. 修 ASL（或先上 workaround），用 initrd Table Upgrade 驗證，確認 log 乾淨、功能恢復。

24.7 fwts：自動化健檢

手動找完個案，出貨前還需要地毯式掃描。Canonical 維護的 **fwts**（FirmWare Test Suite）內建數百項韌體檢查，其中 ACPI 相關測項會靜態驗證每張表的欄位、並實際執行 Method 檢查回傳型別是否符合規格：

用 fwts 掃一遍 ACPI

```
$ sudo apt install fwts
$ sudo fwts acpi klog method
$ grep -A3 "Test Failure Summary" results.log
Test Failure Summary
====================
Critical failures: NONE
High failures: 1
 method: BAT0._BIF returned package of 12 elements, expected 13
```

`acpi` 測項驗表、`klog` 掃 kernel log、`method` 逐一執行 Method 驗回傳型別；結果依嚴重度寫進 `results.log`。bring-up 階段每改一版韌體跑一輪 fwts，能在使用者看到之前攔下大多數本章等級的問題。

核心觀念：本章重點回顧

- 讀錯誤先找第一行根因；`ACPI BIOS Error (bug)` 前綴代表 ACPICA 判定錯在韌體。錯誤行給四樣線索：Method 路徑、`AE_` 碼、ACPICA 版本、原始碼位置。
- `AE_ALREADY_EXISTS`：方法內 `Name()` 被併發重入——用 `Serialized` 修。
- `AE_NOT_FOUND`：引用不存在的名稱（韌體版本差異）——用 `CondRefOf` 防禦。
- GPE storm 用 `interrupts/` 計數器抓兇手，根修是在 `_Lxx` 內清事件源；`acpi_mask_gpe` 只是止血帶。
- 定位鏈：Kernel Log → ACPI Error → Method → ASL Source → Namespace → Hardware；出貨前用 fwts 地毯式掃描。

動手練習

1. 逐段解讀：`ACPI Error: Aborting method \_SB.PCI0.SBRG.EC0._Q3F due to previous error (AE_AML_PACKAGE_LIMIT) (20240322/psparse-529)`。你會先檢查 ASL 裡的什麼？
2. `Serialized` 為什麼能修 `AE_ALREADY_EXISTS`？它的代價是什麼？在什麼情況下你反而要小心它造成 `AE_TIME`？
3. `External` 與 `CondRefOf` 都能處理「別的表定義的物件」，兩者的職責差在哪？為什麼只用前者不安全？
4. 動手：寫一張引用不存在物件的 SSDT，在 acpiexec 中執行，觀察 `AE_NOT_FOUND` 的完整輸出；再加上 `CondRefOf` 驗證錯誤消失。
5. 動手：在你的機器上做 24.5 的 GPE 體檢 Lab，找出最活躍的 GPE，並在 dsdt.dsl 中讀懂它的 handler 在服務誰。

Module 11

進階整合：ACPI、C/C++ 與 Linux Kernel

難度：★★★★★

ASL 只是故事的一半：寫進 DSDT 的每個 Device，最終要由 Linux kernel 的驅動程式接手才會真正動起來。本模組把 C 語言的暫存器功力、Linux device model 與 UEFI 韌體整合串成一條完整的生產線，讓你能從一份硬體暫存器規格，一路整合到使用者空間的 sysfs 介面。這是 Firmware、BIOS、Embedded 與 AI SoC 工程師的核心整合能力。

第 25 章

ACPI 與 Linux Kernel 整合

前面 24 章，你一直站在「韌體端」寫 ASL、除錯 ASL。本章換到桌子的另一側：OS 工程師拿到你的 DSDT，是怎麼把 `Device` 變成 `struct device`、把 `_CRS` 變成 `ioremap` 的參數、把 Method 變成一個 C 函式呼叫的？這是全書最長的一章，也是把你從「會寫 ASL 的人」升級成「能做系統整合的韌體工程師」的一章——第 26 章 Project 1 的 `DEMO0001` 裝置正是本章的主角。

本章學習目標

- 用 C 正確操作硬體：`volatile`、`__packed`、位元遮罩、MMIO 與 port I/O，並對照同一件事的 ASL 寫法。
- 說明 ACPI 列舉如何把 `acpi_device` 變成 `platform_device`，並寫出綁定 `_HID` 的 driver：取 `_CRS` 資源、呼叫 Method、匯出 sysfs。
- 寫出 GPIO / I2C / SPI / PCI 的最小 ASL 描述與 C 端取用方式，並用 `_DSD` 在韌體與驅動之間傳遞 key-value 參數。
- 描述 EDK II 中 ACPI 表從 `.asl` 到開機安裝的完整流程，以及 FADT / MADT / MCFG 在 bring-up 時要填什麼。

25.1 全景圖：兩條路在核心會合

先把課綱的中心圖攤開。ACPI 這門技術有兩條產線：開發端把人寫的 ASL 編譯成 AML、包進 DSDT / SSDT 燒入韌體；執行端由 OS 內建的 interpreter 把 AML 解譯成 Namespace。兩條路在 Linux kernel 會合，往下才是 driver 與真正的硬體：

本章的全景圖：從 ACPI 規格到硬體

```
ACPI 規格（韌體與 OS 之間的合約）
 ├── 開發端：ASL 原始碼（人寫的，第 3 章起）
 │        ↓  iASL Compiler（第 21 章）
 │    AML 位元組碼，打包成 DSDT / SSDT
 │        ↓  整合進韌體（BIOS / UEFI / EDK II，25.7 節）
 └── 執行端：ACPI Interpreter（ACPICA，OS 內建）
          ↓  開機解譯 AML、建立 Namespace
     Linux Kernel：ACPI subsystem 列舉裝置（25.3 節）
          ↓  _HID 比對、_CRS 轉資源
     ACPI Driver（platform / i2c / spi / pci，25.4–25.6 節）
          ↓  readl / writel、inb / outb、evaluate Method
```

```
      Hardware（MMIO 暫存器、I/O Port、EC、GPIO）
```

本章路線：25.2 先練 C 的暫存器基本功（圖的最底層）；25.3 看核心怎麼列舉；25.4 寫出完整的 platform driver；25.5 與 25.6 補齊各子系統與參數傳遞的膠水；25.7 回到韌體端看 EDK II 生產線；25.8 用一顆 LED 把全鏈路串一遍。

25.2 韌體工程師的 C 語言功力

為什麼講 ASL 的書要教 C？因為 ACPI 的兩端都是 C：韌體端的 EDK II 與 OS 端的 kernel / ACPICA 都是 C 專案，ASL 只是中間那層描述語言。同一件事——讀一個暫存器的某幾個 bit——你必須會用兩種語言寫，並且知道它們最後落到同一次匯流排存取。

25.2.1 pointer 與 volatile：跟編譯器搶控制權

C 的指標讓你把「位址」當一般變數操作，這是直接戳硬體的基礎。但編譯器不知道某個位址背後是硬體：它看到迴圈裡重複讀同一個位址，會「好心」只讀一次、把值快取在 CPU 暫存器——對一般記憶體是最佳化，對硬體暫存器是災難。`volatile` 就是告訴編譯器：**每次讀寫都必須真的執行，不准合併、不准刪除。**

bare-metal 風格：輪詢一個 ready bit

```c
/* Poll a ready bit at a fixed MMIO address (bare metal). */
#define STAT_REG  ((volatile unsigned int *)0xFEDF0004UL)
#define STAT_RDY  (1u << 0)

static int wait_ready(void)
{
    int timeout = 1000;

    while ((*STAT_REG & STAT_RDY) == 0) {
        if (--timeout == 0)
            return -1;      /* never spin forever */
    }
    return 0;
}
```

拿掉 `volatile`，開最佳化（`-O2`）後這個迴圈會變成死迴圈：`*STAT_REG` 只被讀一次。另外注意 `timeout` ——這正是第 24 章 `AE_AML_LOOP_TIMEOUT` 的教訓在 C 端的版本：**輪詢硬體永遠要有停損。**

同一件事在 ASL 怎麼寫？宣告 OperationRegion 與 Field，在 `While` 迴圈裡讀那個 Field（見 25.2.3 與第 9–11 章）。關鍵差異是 ASL 不需要 volatile：Field 存取每次都由 interpreter 真的執行一次匯流排存取，天生 volatile。還要記得 `volatile` 只約束編譯器，不提供跨 CPU 的同步保

證；kernel 裡也幾乎不直接解參考指標讀硬體，而是用 25.2.4 的 `readl` / `writel`——它們把 volatile 語意、必要的記憶體排序與可攜性一次包好。

25.2.2 struct 對齊與 __packed：描述線上格式

ACPI Table 是「線上格式」（wire format）：每個欄位的 byte 位置由規格釘死。用 C struct 描述它時要小心對齊：編譯器為了讓 u32 落在 4 的倍數位址，會在成員之間塞看不見的 padding，一塞整個表就對不上了。`__packed` 屬性禁止 padding：

ACPI 表頭的 C 定義

```c
#include <linux/types.h>

struct acpi_table_header {
    char signature[4];        /* e.g. "DSDT"                */
    u32  length;              /* whole table, in bytes      */
    u8   revision;
    u8   checksum;            /* all bytes must sum to 0    */
    char oem_id[6];
    char oem_table_id[8];
    u32  oem_revision;
    char asl_compiler_id[4];  /* "INTL" for iASL            */
    u32  asl_compiler_revision;
} __packed;                   /* exactly 36 bytes, no padding */
```

這個 36 byte 的表頭你已經在第 22 章用 `xxd` 親眼看過。實務備註：這個 layout 恰好天然對齊（欄位排列本來就沒有洞），kernel 內建定義因此沒加 `__packed`，但描述韌體資料結構時防禦性地加上是業界慣例；checksum 規則是「全表 byte 總和 mod 256 為 0」——第 21 章 iasl 幫你算，自己拼表（25.7 節）就得自己算。對照 ASL：`DefinitionBlock` 的參數填的正是這些欄位，length 與 checksum 由 iasl 代勞。

25.2.3 位元操作：mask 與 shift 的肌肉記憶

硬體暫存器慣用「一個 32 bit 暫存器塞好幾個欄位」。C 端的標準動作是 read-modify-write（RMW）：讀出、清掉目標欄位、填新值、寫回。kernel 提供 `BIT()`、`GENMASK()`、`FIELD_PREP()` 讓意圖一目瞭然：

RMW：設定 LED 亮度欄位（bits 7..4）

```c
/* from <linux/bits.h>, <linux/bitfield.h>, <linux/io.h> */
#define LED_CTRL        0x00            /* register offset  */
#define LED_CTRL_EN     BIT(0)          /* bit 0: enable    */
#define LED_CTRL_BRT    GENMASK(7, 4)   /* bits 7..4: level */

```

```
static void led_set(void __iomem *base, unsigned int level)
{
    u32 v;

    v = readl(base + LED_CTRL);             /* read       */
    v &= ~LED_CTRL_BRT;                     /* clear field */
    v |= FIELD_PREP(LED_CTRL_BRT, level);   /* set field   */
    v |= LED_CTRL_EN;
    writel(v, base + LED_CTRL);             /* write back  */
}
```

同一顆暫存器在 ASL 裡怎麼描述？Field 直接把 bit 欄位「切」出名字，RMW 由 interpreter 代勞（`Preserve` 正是「寫欄位時保留其他 bit」的意思，第 10 章）：

同一顆暫存器的 ASL 版：Field 切 bit

```
OperationRegion (LEDR, SystemMemory, 0xFEDF0000, 0x10)
Field (LEDR, DWordAcc, NoLock, Preserve)
{
    LEDE,   1,      // bit 0: enable
        ,   3,      // bits 1..3: reserved (skipped)
    LEDB,   4       // bits 7..4: brightness
}

Method (LSET, 1, Serialized)
{
    LEDB = Arg0     // interpreter does the RMW for you
    LEDE = One
}
```

兩相對照就看出分工：ASL 把「欄位在哪」宣告成資料，C 把「欄位在哪」寫成程式。韌體工程師的日常，就是拿著同一份暫存器規格表，在兩邊寫出彼此一致的描述——欄位位置抄錯一位，兩邊就開始互相說謊。

25.2.4 MMIO 與 Port I/O：兩種戳硬體的匯流排

x86 平台有兩套定址空間。**MMIO**（memory-mapped I/O）把裝置暫存器映進實體位址空間，kernel 中先 `ioremap` 取得虛擬位址，再用 `readl` / `writel` 存取；**port I/O** 是獨立的 64 KB I/O 空間，用 `inb` / `outb` 指令存取：

片段：kernel 中的 MMIO 存取（probe 內）

```
void __iomem *base;

/* map the MMIO range that came from _CRS (see 25.4) */
```

```
base = devm_platform_ioremap_resource(pdev, 0);
if (IS_ERR(base))
    return PTR_ERR(base);

writel(LED_CTRL_EN, base + LED_CTRL);   /* enable the LED */
```

`ioremap` 回傳的是帶 `__iomem` 註記的 cookie，不可直接解參考（靜態檢查工具 sparse 會抓），一律透過 `readl` / `writel` 系列；port I/O 則是 `outb(0x55, 0x80)` 這樣的一行（0x80 是經典的 POST code 埠）。ASL 端用 OperationRegion 的空間型別區分這兩種匯流排：`SystemMemory` 對應 MMIO（上一小節的 LEDR），`SystemIO` 對應 port I/O——宣告 `OperationRegion (DBGP, SystemIO, 0x80, 0x01)` 加一個 8 bit Field 後，對該 Field 賦值就等於 `outb`。

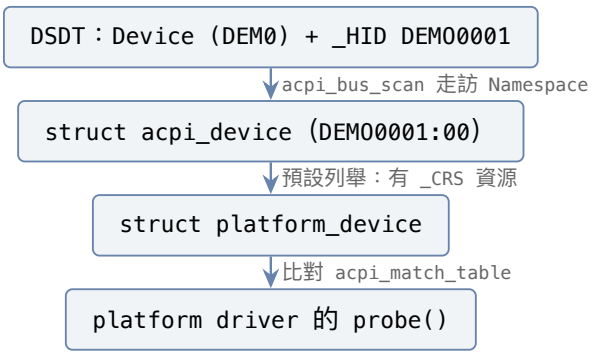
四塊拼圖到此湊齊：指標與 volatile、struct 與對齊、bit 操作、MMIO / port I/O。接下來看核心怎麼把 ASL 描述的裝置送到你的 C 程式手上。

25.3 Linux ACPI Subsystem：從 Namespace 到 device model

Linux 的 ACPI 程式碼分兩層，位置要背起來：

- `drivers/acpi/acpica/`——**ACPICA core**：interpreter、Namespace、事件處理。這份程式碼與 iasl、acpiexec 同一個上游專案，OS 無關。
- `drivers/acpi/`——Linux 膠水層：`scan.c`（列舉）、`ec.c`、`battery.c`……把 ACPI 物件接上 Linux 自己的裝置模型。
- `include/linux/acpi.h`——驅動工程師的 API 入口。

開機時，膠水層走訪 Namespace，替每個 Device 建立一個 `struct acpi_device`——第 22 章在 `/sys/bus/acpi/devices` 看到的每個目錄就是它。這個結構打包了韌體節點的一切：成員 `pnp` 存 `_HID` / `_UID` 等身分、`power` 記錄 D-state、`fwnode` 是 25.6 節屬性 API 的掛載點，最後內嵌一個 `struct device` 讓它加入統一裝置模型。但 `acpi_device` 只是「韌體節點」的化身，多數 driver 並不直接綁它。對於有 `_CRS` 資源、又不屬於特定匯流排的裝置，膠水層會再生一個 `platform_device`——Linux 用來承載「不可列舉裝置」的通用容器：



兩個世界在 sysfs 互相指認——ACPI 裝置目錄下的 `physical_node` symlink 指向 platform 裝置目錄，後者的 `firmware_node` 又指回來，用 `readlink` 即可驗證。兩個例外先記著，25.5

節會展開：_CRS 裡若是 I2C / SPI 的 serial bus 描述子，裝置改由對應的 controller driver 建立 i2c_client / spi_device；PCI 裝置本來就自我列舉，ACPI 節點只是「伴侶」（companion），driver 可用 ACPI_COMPANION() 取回 acpi_device 呼叫 Method。

25.4 實作：綁定 DEMO0001 的 platform driver

理論夠了，動手寫。目標：一個完整可編譯的 kernel module，綁定 _HID 為 DEMO0001 的裝置：probe 時讀出 _CRS 的 MMIO 資源、呼叫 ASL 的 TEST、把結果匯出成 sysfs attribute。它正是第 26 章 Project 1 的主角——先看 ASL 端：

demo.dsl：DEMO0001 裝置的 SSDT

```
DefinitionBlock ("demo.aml", "SSDT", 2, "DEMO", "DEMODEV",
                 0x00000001)
{
    Scope (\_SB)
    {
        Device (DEM0)
        {
            Name (_HID, "DEMO0001")     // matched by the driver
            Name (_CRS, ResourceTemplate ()
            {
                Memory32Fixed (ReadWrite, 0xFEDF0000, 0x00000100)
            })

            Method (TEST, 1, Serialized)
            {
                Debug = "DEM0.TEST called from the driver"
                Return (Arg0 * 2)
            }
        }
    }
}
```

注意：示範位址是佔位符

0xFEDF0000 是教學用的假位址：本章 driver 只讀出資源不映射它，在真機上載入無害。若要進一步 ioremap 實際讀寫，務必先查 /proc/iomem 確認該區段沒有別人——對著別家裝置的暫存器亂寫，輕則當機重則損壞硬體。安全的完整實驗環境（QEMU 加自訂硬體模型）是第 26 章 Project 1 的內容。

沒寫 _STA 時核心視同「存在且啟用」（第 8 章）。driver 全文約 75 行，分兩段列出，第一段是私有資料與 sysfs attribute：

demo_acpi.c（上）：sysfs attribute

```
// demo_acpi.c — ACPI platform driver for the DEMO0001 device
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/acpi.h>

struct demo_priv {
    u64 test_result;
};

/* cat /sys/devices/platform/DEMO0001:00/test_result */
static ssize_t test_result_show(struct device *dev,
                                struct device_attribute *attr,
                                char *buf)
{
    struct demo_priv *priv = dev_get_drvdata(dev);

    return sysfs_emit(buf, "%llu\n", priv->test_result);
}
static DEVICE_ATTR_RO(test_result);
```

第二段是主體：probe 取資源、呼叫 ASL 的 `TEST`、掛上 attribute，加上比對表與模組樣板：

demo_acpi.c（下）：probe 與比對表

```
static int demo_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct acpi_object_list args;
    union acpi_object arg;
    unsigned long long value;
    struct demo_priv *priv;
    struct resource *res;
    acpi_status status;

    priv = devm_kzalloc(dev, sizeof(*priv), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;
    dev_set_drvdata(dev, priv);

    /* the resource was parsed from _CRS by the ACPI core */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
        return dev_err_probe(dev, -ENODEV, "no MMIO in _CRS\n");
    dev_info(dev, "MMIO from _CRS: %pR\n", res);
```

```
    /* evaluate the ASL method TEST(21) under this device */
    arg.type = ACPI_TYPE_INTEGER;
    arg.integer.value = 21;
    args.count = 1;
    args.pointer = &arg;
    status = acpi_evaluate_integer(ACPI_HANDLE(dev), "TEST",
                                   &args, &value);
    if (ACPI_FAILURE(status))
        return dev_err_probe(dev, -EIO, "TEST failed\n");
    priv->test_result = value;
    dev_info(dev, "TEST(21) = %llu\n", value);

    return device_create_file(dev, &dev_attr_test_result);
}

static void demo_remove(struct platform_device *pdev)
{
    device_remove_file(&pdev->dev, &dev_attr_test_result);
}

static const struct acpi_device_id demo_acpi_ids[] = {
    { "DEMO0001", 0 },
    { }
};
MODULE_DEVICE_TABLE(acpi, demo_acpi_ids);

static struct platform_driver demo_driver = {
    .probe  = demo_probe,
    .remove = demo_remove,
    .driver = {
        .name             = "demo_acpi",
        .acpi_match_table = demo_acpi_ids,
    },
};
module_platform_driver(demo_driver);

MODULE_DESCRIPTION("Demo ACPI platform driver (DEMO0001)");
MODULE_LICENSE("GPL");
```

逐段拆解：`platform_get_resource()` 拿到的 `struct resource` 是 ACPI core 解析 `_CRS` 的成果；`ACPI_HANDLE(dev)` 取回裝置的 Namespace 節點，`acpi_evaluate_integer()` 以它為起點用**相對路徑** `"TEST"` 執行方法，參數包成 `acpi_object_list` 送進 `Arg0`——kernel 電池驅動呼叫 `_BST`（第 22 章）用的是同一套機制。綁定則靠三件套：`acpi_device_id` 表、

`.acpi_match_table`，以及把表匯出成 modalias 別名（`acpi:DEMO0001:`）讓 udev 自動 `modprobe` 的 `MODULE_DEVICE_TABLE(acpi, ...)`。Makefile 與操作步驟：

Makefile（recipe 行開頭必須是 Tab）

```
obj-m += demo_acpi.o
KDIR ?= /lib/modules/$(shell uname -r)/build

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```

Lab 實作：編譯、載入、驗證

1. `iasl demo.dsl` 產生 `demo.aml`，依第 23 章的 initrd Table Upgrade 載入，重開機後確認 `/sys/bus/acpi/devices/DEMO0001:00` 存在。
2. 安裝 kernel headers 後執行 `make`，再 `sudo insmod demo_acpi.ko`——driver 立即與裝置配對並執行 probe。
3. `sudo dmesg -t | tail` 檢查 `_CRS` 資源與 `TEST(21) = 42`；`cat` 讀 sysfs 的 `test_result` 應輸出 42。
4. 收尾：`sudo rmmod demo_acpi`。

驗證輸出

```
$ sudo insmod demo_acpi.ko
$ sudo dmesg -t | tail -3
demo_acpi DEMO0001:00: MMIO from _CRS: [mem 0xfedf0000-0xfedf00ff]
ACPI Debug:  String [0x20] "DEM0.TEST called from the driver"
demo_acpi DEMO0001:00: TEST(21) = 42

$ cat /sys/devices/platform/DEMO0001:00/test_result
42
```

整條鏈在這一刻閉合：`Memory32Fixed` 變成 dmesg 那行 MMIO 資源範圍；C 端傳入的 21 走進 `Arg0` 乘以 2 回來；夾在中間的 `ACPI Debug` 是 ASL 方法執行的現場證明（需要第 23 章的兩道閘門打開才看得到）。

補充說明：kernel API 會演進

本章以 kernel 6.x 為準：platform driver 的 `remove` 自 6.11 起回傳 `void`；I2C 的 `probe` 自 6.3 起只帶一個參數。移植程式碼時，以你手上核心樹的標頭為最終依據。

25.5 各子系統的 ACPI 膠水

DEMO0001 走的是 platform bus；真實平台上更多裝置掛在 GPIO、I2C、SPI、PCI 上。ACPI 對每種匯流排都定義了資源描述子，kernel 的對應子系統負責翻譯：`GpioIo` 配上 `_DSD` 命名，C 端就能用 `gpiod_get()` 要腳位；`I2cSerialBusV2` / `SpiSerialBusV2` 會自動生出 `i2c_client` / `spi_device`；PCI 裝置則以 `_ADR` 與 `pci_dev` 配對。以下逐一給最小對照。

25.5.1 GPIO：GpioIo 加 _DSD 命名

ASL 端在 `_CRS` 放 `GpioIo`（或 `GpioInt`）描述腳位，再用 `_DSD` 給每組 GPIO 一個名字——C 端就是用這個名字來要腳位的：

片段：帶一根 reset 腳的感測器

```
Device (SNS0)
{
    Name (_HID, "DEMO0003")
    Name (_CRS, ResourceTemplate ()
    {
        GpioIo (Exclusive, PullNone, 0, 0,
                IoRestrictionOutputOnly,
                "\\_SB.GPI0", 0, ResourceConsumer) { 42 }
    })
    Name (_DSD, Package ()
    {
        ToUUID ("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package ()
        {
            Package () { "reset-gpios",
                         Package () { ^SNS0, Zero, Zero, One } }
        }
    })
}
```

`_DSD` 條目的值是四元素 Package：指回持有 `GpioIo` 的裝置、`_CRS` 中第幾個 GPIO 資源、資源內第幾根腳、以及 active-low 旗標（此處 One 表示低電位有效）。C 端把 `reset-gpios` 去掉字尾變成名字 `"reset"`：

C 端：用名字要 GPIO

```
#include <linux/gpio/consumer.h>

static int sns_reset(struct device *dev)
{
    struct gpio_desc *reset;
```

```
    /* "reset" matches the _DSD entry "reset-gpios" */
    reset = devm_gpiod_get(dev, "reset", GPIOD_OUT_HIGH);
    if (IS_ERR(reset))
        return PTR_ERR(reset);

    gpiod_set_value_cansleep(reset, 0);  /* deassert reset */
    return 0;
}
```

注意 gpiod API 談的是邏輯值：_DSD 已宣告 active-low，邏輯 1（assert reset）在硬體上輸出低電位。極性由韌體描述、C 程式碼保持中立——這正是把硬體細節放進 ACPI 的價值。

25.5.2 I2C 與 SPI：描述即列舉

I2C 裝置的位址、速度、掛在哪個 controller，全部寫在 I2cSerialBusV2 裡。當 I2C controller driver 註冊 adapter 時，核心掃描 Namespace，替每個引用它的裝置自動建立 i2c_client——driver 完全不用自己 instantiate：

片段：位址 0x48 的 I2C 溫度感測器

```
Device (TMP1)
{
    Name (_HID, "DEMO0004")
    Name (_CRS, ResourceTemplate ()
    {
        I2cSerialBusV2 (0x0048, ControllerInitiated, 400000,
            AddressingMode7Bit, "\\_SB.PCI0.I2C1",
            0x00, ResourceConsumer, , Exclusive)
    })
}
```

C 端：i2c_driver 的 probe

```
static int tmp_probe(struct i2c_client *client)
{
    s32 raw;

    /* client->addr == 0x48, from I2cSerialBusV2 */
    raw = i2c_smbus_read_word_swapped(client, 0x00);
    if (raw < 0)
        return raw;

    dev_info(&client->dev, "temp raw = 0x%04x\n", raw);
```

```
    return 0;
}
```

註冊樣板與 25.4 同構，不再重列：`acpi_device_id` 表填 `DEMO0004`，掛進 `i2c_driver` 的比對表欄位，最後用 `module_i2c_driver()` 收尾。SPI 的邏輯也完全同構：`SpiSerialBusV2` 描述 chip select、線模式、速度與時脈極性，controller 註冊時自動生出 `spi_device`，由 `spi_driver` 以同樣的比對表認領：

片段：SPI 裝置的資源描述

```
SpiSerialBusV2 (0x0000, PolarityLow, FourWireMode, 0x08,
    ControllerInitiated, 1000000, ClockPolarityLow,
    ClockPhaseFirst, "\\_SB.PCI0.SPI1",
    0x00, ResourceConsumer, , Exclusive)
```

25.5.3 PCI：_ADR 配對與 _PRT 一句話

PCI 裝置能自我列舉（第 1 章），所以 ACPI 節點不負責「發現」，只負責「補充」：以 `_ADR`（高 16 bit 是 device、低 16 bit 是 function）宣告自己對應哪個 PCI 位址，核心據此把 ACPI 節點與 `pci_dev` 配成 companion，電源方法（`_PS0` / `_PS3`）與 `_DSM` 才有處落地：

片段：PCI 裝置的 ACPI 伴侶節點

```
Device (SAT0)
{
    Name (_ADR, 0x00170000)   // device 0x17, function 0
}
```

至於中斷：PCI root bridge 下的 `_PRT`（PCI Routing Table）負責把每個裝置的 INTA–INTD 腳對應到中斷控制器輸入——一句話記住它：**PCI 傳統中斷的路由表由 ACPI 提供，MSI 則不經過它。**

25.6 _DSD 與 device properties：韌體傳參數給驅動

GPIO 小節已經偷用過 `_DSD`（Device Specific Data）；它的一般形式是「UUID 加 key-value Package」，等於韌體塞給 driver 的組態檔。搭配的 C 端 API 是 `device_property_*` 家族：

片段：用 _DSD 傳兩個屬性

```
Name (_DSD, Package ()
{
    ToUUID ("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
    Package ()
    {
```

```
        Package () { "sample-rate-hz", 48000 },
        Package () { "channel-map", Package () { 0, 1, 2, 3 } }
    }
})
```

C 端：device_property_* 讀取

```
static int demo_read_props(struct device *dev)
{
    u32 map[4];
    u32 rate;
    int ret;

    ret = device_property_read_u32(dev, "sample-rate-hz", &rate);
    if (ret)
        return ret;               /* mandatory property */

    ret = device_property_read_u32_array(dev, "channel-map",
                                         map, ARRAY_SIZE(map));
    if (ret)
        return ret;

    dev_info(dev, "rate=%u ch0=%u\n", rate, map[0]);
    return 0;
}
```

字串與布林屬性各有對應 API（`..._read_string()`；布林以 `device_property_present()` 查「存在即為真」）。命名慣例：屬性名用小寫加連字號（沿襲 Device Tree 傳統），單位直接寫進名字（`-hz`、`-ms`）讓讀的人不用猜。

補充說明：一套 API 通吃 ACPI 與 Device Tree

`device_property_*` 是 **fwnode** 抽象層的門面：同一份 driver 程式碼，在 x86 上從 `_DSD` 讀值、在 ARM Device Tree 平台上從 DT 節點讀值，一行都不用改。寫跨平台 driver 時，優先用它而不是 ACPI 專屬 API——這也是 kernel 社群 review 時的既定要求。

25.7 韌體端：EDK II 的 ACPI 表生產線

視角換回韌體端。你寫好的 `Dsdt.asl` 在 UEFI 韌體專案（**EDK II**）裡是這樣被生產、運送、交貨的。先看它住哪裡：

EDK II 平台 package 中的 ACPI 檔案

```
PlatformPkg/AcpiTables/
```

```
├── AcpiTables.inf      ← 打包元件：收集以下所有表
├── Dsdt/Dsdt.asl       ← 本書主角：手寫的 ASL
├── Facp/Facp.aslc      ← FADT：C 結構逐欄位填
├── Madt/Madt.aslc      ← MADT：CPU 與中斷拓撲
└── Mcfg/Mcfg.aslc      ← MCFG：PCIe ECAM 基底
```

`.asl` 由 iasl 編譯；`.aslc` 是 EDK II 慣例的「用 C 寫的表」——像 FADT 這種純欄位的表，直接定義一個 25.2.2 風格的 `__packed` struct 常數即可。建置時 iasl 以 `-tc` 選項執行，除了 `.aml` 還會多吐一個 `.hex` ——AML 映像的 C 陣列版，讓韌體能以原始碼形式攜帶這張表：

dsdt.hex（節錄）：AML 變成 C 陣列

```
/* dsdt.hex (excerpt), generated by "iasl -tc dsdt.asl".
 * The array symbol name varies with the iasl version.  */
unsigned char AmlCode[] =
{
    0x44, 0x53, 0x44, 0x54, 0xC0, 0x0F, 0x00, 0x00, /* DSDT,len */
    0x02, 0x8D, 0x44, 0x45, 0x4D, 0x4F, 0x20, 0x20, /* rev,chks */
    /* ... 4032 bytes in total ... */
};
```

開頭八個 byte 正是 25.2.2 表頭 struct 的前兩個欄位：`"DSDT"` 與 little-endian 的長度 0xFC0 = 4032。整條生產線：



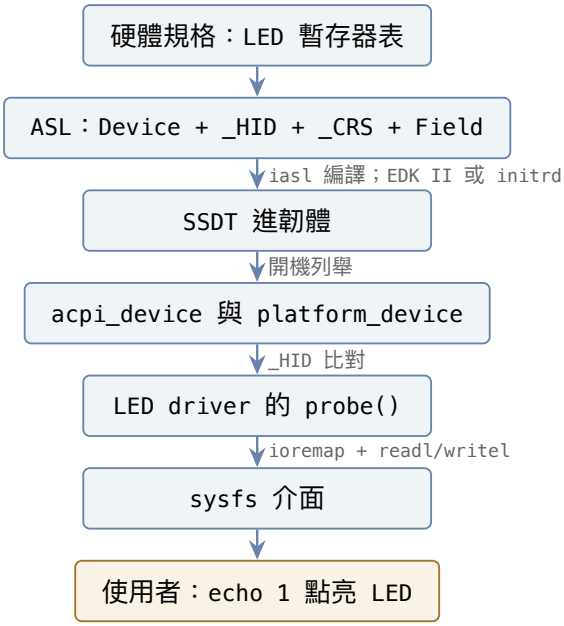
開機時，平台的 `AcpiPlatformDxe` driver 在 FV 裡按 GUID 找到裝表的 FFS 檔，逐張呼叫 `InstallAcpiTable()`：把表複製到 ACPI 保留記憶體、填進 XSDT、重算 checksum——第 1 章那條「RSDP → XSDT → 各表」的鏈就是在這一刻搭好的。

最後，bring-up 時三張「非 AML」的表各自要填什麼（DSDT 描述裝置，這三張描述平台骨架）：

表	bring-up 時的必填重點
FADT	SCI 中斷號、PM1a Event / Control block 位址、PM Timer、SMI_CMD 與 enable / disable 命令值、DSDT / FACS 位址、HW-reduced 旗標；填錯常見電源鍵沒反應、睡眠失敗
MADT	Local APIC 位址、每顆 CPU 一筆 LAPIC / x2APIC entry、IOAPIC entry、Interrupt Source Override（IRQ0 對 GSI2 是經典）；填錯常見只認到一顆 CPU、中斷錯亂
MCFG	PCIe ECAM 基底位址、segment 編號、起訖 bus 號；填錯常見 lspci 讀不到延伸組態空間

25.8 端到端案例：一顆 LED 的旅程

收尾前，把全書串成一個故事。硬體工程師遞來一頁規格：「LED 控制器，MMIO 基底 0xFEDF0000，offset 0 是控制暫存器：bit 0 enable、bits 7..4 亮度，開機預設關。讓使用者能從 Linux 控制它。」你的整合路線：



每一站都是你已經修過的學分：讀懂暫存器規格、規劃 bit 欄位靠 25.2 的 C 功力與第 9–11 章的 Field 語意；寫 Device、_HID、_CRS 是第 8 章與第 18–20 章的 SSDT 實戰；編譯與靜態驗證交給第 21 章的 iasl 與 fwts；整合進韌體走 25.7 的 EDK II 生產線（或第 23 章的 initrd 捷徑）；列舉與 driver 綁定是 25.3 與 25.4；驗證與除錯的整套工具則在 Module 10（第 22–24 章）。

ASL 端你已經會了：25.2.3 的 LEDR region 加 LEDE / LEDB 兩個欄位，包成一個 _HID 為 "DEMO0005" 的 Device，配上對應的 _CRS。driver 端沿用 25.4 的骨架，兩處小改：probe 用 devm_platform_ioremap_resource() 把資源映成 __iomem 指標；sysfs attribute 換成可寫版本，先以 kstrtou8() 把字串轉成 0–15 的亮度值，再照搬 25.2.3 的 led_set() 那組 RMW 序列寫進暫存器，回傳 count 收工。

於是「`echo 8 > /sys/devices/platform/DEMO0005:00/brightness`」這個再平凡不過的指令，背後是：韌體描述（ASL）→ 編譯（iasl）→ 韌體安裝（EDK II）→ 核心列舉（ACPI subsystem）→ driver 綁定（`_HID`）→ 位址翻譯（`_CRS` 加 `ioremap`）→ 位元操作（RMW）→ 光子離開 LED。若哪一站斷了，Module 10 的整套工具（dmesg、sysfs、Debug 物件、ACPI Error 解剖）都在等你。第 26 章起的實戰專案，會讓你把這條鏈親手走上五遍。

核心觀念：本章重點回顧

- ASL 與 C 是同一份暫存器規格的兩種寫法：OperationRegion / Field 對應 pointer、mask、`readl` / `writel`；Field 存取天生 volatile。
- 列舉鏈：DSDT 的 Device → `acpi_device` → `platform_device` → driver probe；I2C / SPI 裝置由 controller 列舉，PCI 節點只當 companion。
- platform driver 三件套：`acpi_device_id` 比對表、`platform_get_resource()` 取 `_CRS`、`acpi_evaluate_integer()` 呼叫 Method。
- `_DSD` 傳 key-value 參數，C 端用 `device_property_*` 讀取，同一套 API 通吃 ACPI 與 Device Tree。
- EDK II 生產線：`.asl` → iasl（`-tc`）→ FFS / FV → AcpiPlatformDxe 安裝 → XSDT；FADT / MADT / MCFG 是 bring-up 的平台骨架。

動手練習

1. `volatile` 解決什麼問題、不解決什麼問題？為什麼 kernel 要求用 `readl` / `writel` 而不是直接解參考 `__iomem` 指標？
2. 手算 `GENMASK(7, 4)` 的值，並寫出把亮度欄位設成 5 而不動其他 bit 的完整 RMW 序列（C 與 ASL 各一版）。
3. 在 `demo.dsl` 的 `_DSD` 加一個 `"blink-hz"` 屬性，修改 driver 讀出並印到 dmesg。
4. 動手：完整跑一遍 25.4 的 Lab；然後把 ASL 的 `TEST` 改成回傳 `Arg0 * 3`，重新載入 SSDT 與模組，用 sysfs 驗證結果變成 63。

Module 12

實戰專案

難度：★★★★★

前面十一個模組給了你零件：語法、Namespace、Device、OperationRegion、Method、電源管理、工具鏈與 Linux 端的除錯手段。本模組用五個由淺入深的專案把零件組成完整的生產線——每個專案都要求你走完「撰寫 ASL、iasl 編譯、acpiexec 模擬、上機驗證」的完整循環。做完 Project 5，你就具備 ACPI Firmware Engineer 的基本實戰能力：拿到一台陌生的機器，能萃取、看懂、修改並驗證它的 ACPI Tables。

第 26 章

Project 1：Virtual ACPI Device

這是你第一次不跟著課文抄範例，而是以工程師的身分交付一個「產品」：一個從零開始的虛擬 ACPI 裝置。它只有一個 _HID、一個 _STA 和一個會回傳 1234 的 Method，小得不能再小，但五臟俱全——你會走完 ASL 生產線的每一站：撰寫、編譯、在 acpiexec 裡執行、最後塞進 QEMU 或真機的 Namespace，在 /sys/bus/acpi/devices 底下親眼看到它。

之後四個專案的所有基礎設施（工作目錄、編譯流程、驗證手法）都在本章建立，請把每一步都確實做過，不要只用讀的。

26.1 專案目標與驗收標準

本章學習目標

專案目標

- 寫出一個完整、可編譯的 SSDT，內含裝置 DEV0：_HID、_UID、_STA 與方法 TEST。
- 說明 Namespace、Device、Method 三者的關係：裝置掛在樹的哪裡、OS 怎麼找到它、方法怎麼被呼叫。
- 熟練 iasl 與 acpiexec 的基本工作循環（第 21 章工具鏈的實戰化）。
- 用 initrd override 把自製 SSDT 載入 QEMU 或真機。

驗收標準——完成後你必須能拿出以下證據：

- `iasl project1.dsl` 輸出 `0 Errors, 0 Warnings, 0 Remarks`。
- acpiexec 中 `execute \_SB.DEV0.TEST` 回傳 `[Integer] = ...04D2`（即十進位 1234）。
- QEMU 或真機的 `/sys/bus/acpi/devices/` 出現 `DEMO0001:00`，其 path 檔內容為 `\_SB_.DEV0`。

26.2 背景與設計

26.2.1 為什麼用 SSDT，而不是改 DSDT？

第 1 章介紹過表鏈結構：DSDT 是主表、開機必載，SSDT 則是可以有很多張的「補充表」。DSDT 屬於 OEM，改它等於動手術；SSDT 則像外掛模組，OS 載入後會把內容「嫁接」到同一棵 Namespace 樹上（Module 8 的核心觀念）。自製裝置一律先做成 SSDT——錯了就拔掉，不會弄壞原廠描述。這也是業界替既有平台補描述（例如加一顆新感測器）的標準做法。

26.2.2 用到前面哪些章的知識

- 第 **2** 章：ASL 經 iasl 編譯成 AML；`DefinitionBlock` 的參數就是表頭欄位的來源。
- 第 **3** 章：Namespace 樹與 `\_SB` 掛載點——所有實體裝置都活在 System Bus 底下。
- 第 **8** 章：`_HID` / `_UID` / `_STA` 這組「身分證三件套」的格式與位元定義。
- 第 **12** 章：Method 的宣告、`Return` 與 `Debug` 物件。
- 第 **21** 章：iasl 與 acpiexec 的操作。
- **Module 10**：Linux 的 initrd table override 與 `/sys/bus/acpi` 觀察點。

26.2.3 開發循環與目標 Namespace

本專案（以及之後所有專案）都遵循同一條開發循環：



先在 acpiexec 裡把邏輯調到對，再上機——模擬器的一輪迭代只要幾秒，上機驗證一輪至少要一次重開機，順序反了就是浪費生命。載入成功後，Namespace 會長出這樣的新枝：

Project 1 完成後的 Namespace（節錄）

```
\ (root)
├── _SB_                    ← 既有的 System Bus Scope
│   ├── ...                 ← 原有裝置（DSDT 描述）
│   └── DEV0                ← 本專案新增（來自 SSDT）
│       ├── _HID  "DEMO0001" ← OS 據此比對 driver
│       ├── _UID  0          ← 同型裝置的流水號
│       ├── _STA  Method     ← 回報存在狀態
│       └── TEST  Method     ← Return (1234)
└── _TZ_ ...
```

補充說明：`_HID` 字串的格式規則

字串型 `_HID` 只有兩種合法長度：7 字元（3 個大寫字母的 vendor ID ＋ 4 位十六進位，如 `PNP0C0D`）或 8 字元（4 個字元的 vendor ID ＋ 4 位十六進位，如本專案的 `DEMO0001`）。iasl 會檢查格式，寫成 `DEMO001` 或 `demo0001` 都過不了關。第 8 章介紹的 `EisaId ("PNP0C0D")` 則是把 7 字元格式壓縮成 32 位元整數的舊式寫法，兩者在 AML 層面等價。

26.3 逐步實作

Lab 實作：Project 1 實作步驟

1. 建立工作目錄 `proj1/`，之後每個專案一個目錄，養成把 `.dsl`、`.aml`、筆記放在一起的習慣。
2. 用編輯器建立 `project1.dsl`，輸入下方完整程式碼。
3. 執行 `iasl project1.dsl`，確認零錯誤零警告。
4. 進入 acpiexec 互動模式，用 `namespace` 與 `execute` 驗證（下一節）。
5. 打包 initrd，丟進 QEMU 或真機，檢查 `/sys`（下一節）。

完整程式碼如下，總共不到三十行：

project1.dsl——完整可編譯

```
DefinitionBlock ("project1.aml", "SSDT", 2, "DEMO", "PROJ1",
                 0x00000001)
{
    Scope (\_SB)
    {
        Device (DEV0)
        {
            Name (_HID, "DEMO0001")   // 4-char vendor + 4 hex
            Name (_UID, Zero)         // instance number

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)         // present+enabled+shown+ok
            }

            Method (TEST, 0, NotSerialized)
            {
                Debug = "DEV0.TEST called"
                Return (1234)         // = 0x4D2
            }
        }
    }
}
```

26.3.1 逐段拆解

DefinitionBlock 標頭——六個參數依序是：AML 輸出檔名（現代 iasl 其實以命令列為準，此欄位僅存於原始碼）、表簽章 `"SSDT"`、ComplianceRevision 2、OEM ID（最多 6 字元）、OEM

Table ID（最多 8 字元）、OEM Revision。兩個細節值得注意：

- ComplianceRevision ≥ 2 代表整數是 64 位元（第 2 章）。之後你會看到 `Ones` 顯示成 `FFFFFFFFFFFFFFFF`，就是這個欄位的效果。
- OEM ID 與 Table ID 是你在 dmesg 裡認出自己表的依據——載入時核心會印出 `DEMO` 與 `PROJ1`，取個有辨識度的名字，除錯時會感謝自己。

`Scope (\_SB)`——SSDT 不是重新種一棵樹，而是宣告「以下內容接到既有的 `\_SB` 底下」。第 3 章說過：Scope 打開既有節點，Device 建立新節點。如果這裡誤用 `Device (_SB)`，會變成企圖重複定義 System Bus。

身分證三件套——`_HID` 決定 OS 拿哪個 driver 來配對；`_UID` 在同型裝置有多顆時區分個體；`_STA` 回報狀態位元（第 8 章）：bit 0 存在、bit 1 已啟用、bit 2 顯示於 UI、bit 3 功能正常，`0x0F` 就是四者全開。省略 `_STA` 時 OS 視同全開，本專案仍刻意寫出，因為 Project 4 會開始操作它。

`TEST` 方法——`Debug = "..."` 是 ASL 2.0 寫法，等價於 legacy 的 `Store ("...", Debug)`（第 12 章）；寫進 `Debug` 物件的東西會被 Interpreter 印到除錯通道，在 acpiexec 是直接印在畫面上，在 Linux 則要開 ACPI debug 才看得到。`Return (1234)` 用十進位字面值；AML 層面沒有進位制的概念，所以 acpiexec 顯示結果時一律用十六進位——`0x4D2`。

26.4 驗證

26.4.1 iasl 編譯

編譯，檢查三個零

```
$ iasl project1.dsl

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20260408
...
ASL Input:     project1.dsl -  593 bytes  9 keywords  0 source lines
AML Output:    project1.aml -  115 bytes  3 opcodes   6 named objects

Compilation successful. 0 Errors, 0 Warnings, 0 Remarks, ...
```

115 bytes 的 AML——之後上機排錯時，看到 dmesg 裡的表長度 `0x73`（= 115），你就知道核心載到的是這一版。

26.4.2 acpiexec 模擬執行

acpiexec 互動驗證（版本字串與位址依環境而異）

```
$ acpiexec project1.aml
```

```
...
ACPI: SSDT 0x7FB5D87069D0 000073 (v02 DEMO PROJ1 00000001 ...)
Table [SSDT: PROJ1] (id 02) - 5 Objects with 1 Devices, 2 Methods
ACPI: 2 ACPI AML tables successfully acquired and loaded

- namespace
ACPI Namespace (from Namespace Root):
 0  _SB_ Device       0x7fbf...1f0 000
 1    DEV0 Device       0x7fbf...390 002
 2      _HID String       0x7fbf...590 002 Len 08 "DEMO0001"
 2      _UID Integer      0x7fbf...810 002 = 0000000000000000
 2      _STA Method       0x7fbf...8f0 002 Args 0 Len 0003
 2      TEST Method       0x7fbf...6c0 002 Args 0 Len 0019

- execute \_SB.DEV0.TEST
Evaluating \_SB.DEV0.TEST
ACPI Debug:  "DEV0.TEST called"
Evaluation of \_SB.DEV0.TEST returned object, buffer length 18
  [Integer] = 00000000000004D2

- execute \_SB.DEV0._STA
Evaluating \_SB.DEV0._STA
  [Integer] = 000000000000000F
```

逐一核對：`namespace` 證明 `DEV0` 確實掛在 `\_SB` 底下、四個子物件齊全；`TEST` 先印出 Debug 字串，再回傳 `0x4D2`——1234 到手。順帶觀察兩件事：acpiexec 連同自己的迷你 DSDT 一共載了 2 張表；而 `namespace` 輸出裡還有 `_OS_`、`_REV` 這些 Interpreter 內建的預定義物件，Project 5 會再遇到它們。

26.4.3 initrd 載入 QEMU / 真機

Module 10 教過核心的 ACPI table upgrade 機制：AML 要放進 cpio 封存檔內的固定路徑（`kernel/firmware/acpi/`），再把封存檔接在 initrd 的最前面，開機時核心就會自動安裝這張 SSDT。實際操作：

打包 override initrd 並用 QEMU 開機

```
$ mkdir -p kernel/firmware/acpi
$ cp project1.aml kernel/firmware/acpi/
$ find kernel | cpio -H newc --create > project1.img
1 block

$ cat project1.img /boot/initrd.img-$(uname -r) > combined.img
$ qemu-system-x86_64 -m 2G -nographic \
```

```
    -kernel /boot/vmlinuz-$(uname -r) \
    -initrd combined.img \
    -append "console=ttyS0"
```

注意：cpio 的路徑必須「相對」

打包時必須站在 kernel 目錄的上一層執行 `find kernel`，讓封存檔內的路徑恰好是 `kernel/firmware/acpi/project1.aml`。若不小心打包成絕對路徑（`/home/you/kernel/...`），核心找不到約定的路徑，會安靜地忽略你的表——沒有錯誤訊息，這是本專案最常見的「無聲失敗」。

開機後檢查三個觀察點：

在 QEMU（或真機）內驗收

```
$ sudo dmesg | grep -i "table upgrade"
ACPI: Table Upgrade: install [SSDT-DEMO  -PROJ1   ]

$ ls /sys/bus/acpi/devices/ | grep DEMO
DEMO0001:00

$ cat /sys/bus/acpi/devices/DEMO0001:00/path
\_SB_.DEV0
$ cat /sys/bus/acpi/devices/DEMO0001:00/status
15
```

`path` 顯示 `_SB_`——四字元補齊的內部名稱，第 3 章的知識在 sysfs 裡直接可見；`status` 的 15 正是 `_STA` 回傳的 `0x0F`。在真機上，同樣的 `project1.img` 可透過 GRUB 掛載（完整流程見第 30 章階段四）；建議先在 QEMU 練熟再碰實體機器。

26.5 常見問題

注意：學生最容易卡住的三個點

- **`_HID` 格式錯誤**：`DEMO001`（7 字元但 vendor 佔 4 字）或小寫字母都會被 iasl 拒絕，錯誤訊息是 `Invalid leading ASCII character` 或 `suffix must be 4 hex digits` 一類。記住規則：3 字母 + 4 hex，或 4 字元 + 4 hex。
- **acpiexec 路徑打錯**：`execute DEV0.TEST` 會回 `AE_NOT_FOUND`——互動模式下請用完整路徑 `\_SB.DEV0.TEST`。若用 `-b` 參數從 shell 一次性執行，記得整串用單引號包起來，否則反斜線會先被 shell 吃掉。
- **QEMU 裡看不到裝置**：先確認 dmesg 有沒有 `Table Upgrade` 那一行。沒有，就是

initrd 打包或串接順序的問題（override 的 cpio 必須在最前面、不可壓縮）；有這行但 `/sys` 沒裝置，再回頭檢查 `_STA` 與 `Scope` 路徑。

26.6 加分挑戰

動手練習：延伸任務

1. **第二個裝置**：在同一張 SSDT 加入 `DEV1`，`_HID` 同為 `DEMO0001`、`_UID` 設為 1。載入後觀察 `/sys/bus/acpi/devices/` 如何用 `DEMO0001:00`、`DEMO0001:01` 區分兩顆同型裝置。
2. **帶參數的方法**：新增 `Method (ADDX, 2) { Return (Arg0 + Arg1) }`，在 acpiexec 用 `execute \_SB.DEV0.ADDX 0x15 0x2A` 驗證回傳 `0x3F`——這是第 12 章 `Arg` 物件的實戰。
3. **回傳複合物件**：把 `TEST` 改成回傳 `Package () { 1234, "hello", Buffer () {1, 2} }`，觀察 acpiexec 如何逐層列印。真實韌體的 `_BST`、`_BIF` 回傳的正是這種 Package（第 30 章會解剖）。

第 27 章

Project 2：GPIO Controller

Project 1 的裝置是「空殼」——有身分，但不碰任何硬體。本專案補上韌體工程師每天在做的那件事：透過暫存器控制硬體。你將打造一顆虛擬的 GPIO 控制器，完整走一遍 Device → OperationRegion → Field → Method 這條 ACPI 描述硬體的標準鏈路：宣告一塊 MMIO、把它切成方向 / 輸出 / 輸入三個暫存器，再用位元運算寫出「設方向、寫 pin、讀 pin」三個控制方法。

真實平台上，GPIO 是 ACPI 出鏡率最高的資源之一：電源開關、LED、感測器中斷腳、觸控板的 reset 線，背後全是同一套操作。把這條鏈路練熟，你就看得懂真實 DSDT 裡九成的暫存器操作程式碼。

27.1 專案目標與驗收標準

本章學習目標

專案目標

- 用 `OperationRegion` 宣告一塊假想 MMIO（SystemMemory `0xFED40000`），並以 `Field` 切出 DIR / OUT / IN 三個 32 位元暫存器。
- 寫出 `SETD`（設方向）、`SETP`（寫 pin）、`GETP`（讀 pin）三個 Method，以位元運算精準操作單一 pin 而不干擾其他 pin。
- 在 `_CRS` 中宣告 MMIO 資源，讓 OS 知道這塊位址被誰占用。
- 理解 acpiexec 如何模擬記憶體區域，並利用它驗證讀寫邏輯。

驗收標準

- `iasl` 編譯零錯誤、零警告、零 remark。
- acpiexec 中依序執行 `SETP (5,1)`、`SETP (3,1)`、`SETP (5,0)` 後，`GOUT` 的值依序為 `0x20`、`0x28`、`0x08`——證明單一 pin 的 set 與 clear 都沒有波及鄰兵。
- 能口頭解釋：為什麼 `SETD` / `SETP` 必須 `Serialized`，而 `GETP` 不必。

27.2 背景與設計

27.2.1 暫存器地圖：先畫圖，再寫碼

寫任何暫存器操作程式之前，先把 datasheet 的暫存器地圖抄下來——這是韌體工程師的肌肉記憶。我們的虛擬控制器管 32 支 pin，三個 32 位元暫存器、每 pin 一個 bit：

Offset	名稱	Field	意義
0x00	DIR	GDIR	方向：bit n = 1 表示 pin n 為輸出
0x04	OUT	GOUT	輸出鎖存：輸出 pin 的驅動準位
0x08	IN	GPIN	輸入準位：讀取腳位目前狀態（唯讀）

基底位址選在 `0xFED40000`——這是本教材的假想值。呼叫鏈自上而下是這樣接起來的：



27.2.2 用到前面哪些章的知識

- 第 **6** 章：`<<`、`|`、`&`、`~` 位元運算子——本專案的靈魂。
- 第 **8** 章：`_CRS` 與 `Memory32Fixed`，向 OS 申報資源占用。
- 第 **9**、**10** 章（Module 5）：`OperationRegion` 的 SystemMemory 空間與 `Field` 的切割規則。
- 第 **12**、**13** 章（Module 6）：帶參數的 Method、`Serialized` 對 read-modify-write 的保護。

注意：絕對不要把這張表載到真機

`0xFED40000` 在許多 Intel 平台上是 TPM 的暫存器空間；對著別人的硬體亂寫 MMIO，輕則裝置異常、重則系統當機。本專案的驗證只在 **acpiexec** 模擬器中進行——它根本不會碰真實記憶體，這正是我們敢放手實驗的原因。想在真機玩 GPIO，請等第 30 章教你找出「自己機器上真正的」GPIO 控制器之後。

27.3 逐步實作

Lab 實作：Project 2 實作步驟

1. 建立 `proj2/`，新增 `project2.dsl`，先只寫 Device 骨架 + `_CRS`，編譯一次確認無誤。
2. 加入 `OperationRegion` 與 `Field`，再編譯一次——增量開發，錯誤才好定位。
3. 依序實作 `SETD`、`SETP`、`GETP`，每加一個方法就編譯一次。

4. 進 acpiexec，照驗證一節的劇本逐步操作暫存器。

project2.dsl——完整可編譯

```
DefinitionBlock ("project2.aml", "SSDT", 2, "DEMO", "PROJ2",
                 0x00000001)
{
    Scope (\_SB)
    {
        Device (GPI0)
        {
            Name (_HID, "DEMO0002")
            Name (_UID, Zero)

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)
            }

            Name (_CRS, ResourceTemplate ()
            {
                Memory32Fixed (ReadWrite, 0xFED40000, 0x00001000)
            })

            OperationRegion (GREG, SystemMemory, 0xFED40000, 0x0C)
            Field (GREG, DWordAcc, NoLock, Preserve)
            {
                GDIR, 32,   // 0x00: direction, 1 = output
                GOUT, 32,   // 0x04: output latch
                GPIN, 32,   // 0x08: input level (RO on real HW)
            }

            // SETD: set direction. Arg0 = pin 0..31, Arg1 = dir
            Method (SETD, 2, Serialized)
            {
                Local0 = (One << Arg0)
                If (Arg1)
                {
                    GDIR |= Local0        // set one bit
                }
                Else
                {
                    GDIR &= ~Local0       // clear one bit
```

```
                }
                Return (GDIR)
            }

            // SETP: drive an output pin. Arg0 = pin, Arg1 = level
            Method (SETP, 2, Serialized)
            {
                Local0 = (One << Arg0)
                If (Arg1)
                {
                    GOUT |= Local0
                }
                Else
                {
                    GOUT &= ~Local0
                }
                Return (GOUT)
            }

            // GETP: read an input pin. Arg0 = pin
            Method (GETP, 1, NotSerialized)
            {
                Return ((GPIN >> Arg0) & One)
            }
        }
    }
}
```

27.3.1 逐段拆解

Region 與 Field 的對應——`OperationRegion` 宣告「從 `0xFED40000` 起、長 `0x0C` 的 SystemMemory」；`Field` 再依宣告順序把位元往上堆：`GDIR` 佔 bit 0–31（offset 0x00）、`GOUT` 佔 bit 32–63（offset 0x04）、`GPIN` 佔 bit 64–95（offset 0x08）——與暫存器地圖一一對應。`DWordAcc` 要求 Interpreter 以 32 位元為單位存取，符合多數 GPIO 控制器 datasheet 的規定；如果硬體只接受 32 位元對齊的存取卻用了 `ByteAcc`，真機上就會讀寫出垃圾。

mask 的產生——`Local0 = (One << Arg0)` 把 1 左移到目標 pin 的位置。這一行是第 6 章位元運算的直接應用：pin 5 得到 `0x20`、pin 3 得到 `0x08`。

read-modify-write 三部曲——`GOUT |= Local0` 只點亮目標 bit；`GOUT &= ~Local0` 只熄滅目標 bit。兩者展開後都是「讀出 32 位元、改一個 bit、整包寫回」。用 legacy 運算子寫，等價形式是：

同一件事的 legacy 寫法（片段，僅供對照）

```
    // GOUT |= Local0
    Or (GOUT, Local0, GOUT)
    // GOUT &= ~Local0
    And (GOUT, Not (Local0), GOUT)
```

真實 DSDT 兩種寫法都常見（取決於 OEM 程式碼的年代），必須都讀得懂。另外注意一個 64 位元陷阱：`Not (Local0)` 是對 64 位元整數取反（第 2 章的 ComplianceRevision 效應），pin 5 的反罩其實是 `0xFFFFFFFFFFFFFFDF`；寫回 32 位元 Field 時高位被截掉，結果仍然正確——但如果有一天你把 Field 改成 64 位元寬，行為就完全不同了。

為什麼 **`Serialized`**——`SETD` / `SETP` 是 read-modify-write：兩條執行緒同時 `SETP (5,1)`、`SETP (3,1)`，若交錯執行，後寫的會蓋掉先寫的 bit（第 13 章的競態問題）。宣告 `Serialized` 讓 Interpreter 對整個 Method 自動上鎖。`GETP` 只做單次讀取、沒有中間狀態，維持 `NotSerialized` 即可。

回傳暫存器現值——`SETD` / `SETP` 最後 `Return (GDIR)` / `Return (GOUT)`，純為驗證方便：acpiexec 每執行一次就直接把暫存器新值印給你看，不用另外 `evaluate`。真實韌體裡這類方法通常不回傳值。

27.4 驗證

27.4.1 iasl 編譯與 acpiexec 劇本

編譯無誤後，照下面的劇本操作。留意每一步的回傳值——那就是暫存器的即時內容：

acpiexec 驗證劇本（輸出節錄）

```
$ iasl project2.dsl
...
Compilation successful. 0 Errors, 0 Warnings, 0 Remarks, ...

$ acpiexec project2.aml
- execute \_SB.GPI0.SETD 0x5 0x1
  [Integer] = 0000000000000020     ; DIR: pin5 -> output

- execute \_SB.GPI0.SETP 0x5 0x1
  [Integer] = 0000000000000020     ; OUT: 0010 0000

- execute \_SB.GPI0.SETP 0x3 0x1
  [Integer] = 0000000000000028     ; OUT: 0010 1000

- execute \_SB.GPI0.SETP 0x5 0x0
  [Integer] = 0000000000000008     ; OUT: 0000 1000

- execute \_SB.GPI0.GETP 0x5
```

```
  [Integer] = 0000000000000000     ; IN reads as 0 (simulated)

- evaluate \_SB.GPI0.GOUT
  [Integer] = 0000000000000008     ; direct field read-back
```

三個關鍵觀察：

- `0x20` → `0x28` → `0x08`：set pin 3 沒動到 pin 5，clear pin 5 沒動到 pin 3 ——位元運算的隔離性驗證完成，正是驗收標準。
- `evaluate` 可以直接讀一個 Field 單元：Interpreter 會真的執行一次 region 讀取。這是檢查暫存器狀態最快的手段。
- `GETP` 回 0 不是 bug——見下面的補充說明。

補充說明：acpiexec 如何模擬記憶體

acpiexec 不會存取真實實體記憶體：它替每塊 OperationRegion 維護一份私有緩衝區。寫入會記進緩衝區，之後讀同一位址就回讀寫過的值（所以 `GOUT` 讀得回 `0x08`）；從未寫過的位址則回傳預設填充值 0（所以模擬輸入的 `GPIN` 永遠是 0）。想改變填充值，可用 `acpiexec -f 0xFF project2.aml` 之類的參數，讓「未初始化讀取」在測試中無所遁形。換句話說：模擬器驗證的是你的邏輯，不是硬體的電氣行為——後者只能上真機。

27.5 常見問題

注意：學生最容易卡住的三個點

- **`GETP` 永遠回 0 就以為程式錯了**：模擬的 `GPIN` 沒有真實訊號源，0 是正常的。想驗證 `GETP` 的位移邏輯，先用 `evaluate` 直接對 `GPIN` 寫值再讀（或暫時把 `GETP` 改讀 `GOUT`），確認 shift-and-mask 正確。
- **忘記 `Serialized`**：單人測試看不出差別，上機後偶發性 pin 狀態錯亂，極難重現。凡是 read-modify-write 的方法，一律 `Serialized`。
- **Field 順序與 offset 對不上**：`Field` 是「依序堆疊」，在 `GDIR` 前多插一個 8 位元欄位，後面所有暫存器全部位移。改地圖時請用 `Offset (0x04)` 明確錨定，不要依賴默數位元。

27.6 加分挑戰

動手練習：延伸任務

1. **GPIO consumer 裝置**：新增裝置 `LED0`，在其 `_CRS` 用 `GpioIo` 描述子引用 `\_SB.GPI0` 的 pin 5（第 8 章資源描述子的進階用法），體會「控制器提供資源、裝置消費資源」的分工。
2. **GPIO 事件**：替 `GPI0` 加上 `_AEI`（ACPI Event Interrupt）宣告 pin 7 為事件腳，並實作對應的事件方法 `_E07`，在其中 `Notify` 一個裝置——這是筆電「上蓋開關接 GPIO」的標準模式。
3. **一行翻轉**：實作 `TGLP`（toggle pin），用 XOR 一行完成：`GOUT ^= (One << Arg0)`，並用 acpiexec 連按兩次驗證值恢復原狀。

第 28 章

Project 3：I2C Sensor

MMIO 之後，第二種你一定會遇到的硬體介面是序列匯流排。筆電裡的觸控板、溫度感測器、電量計，多半掛在 I2C 上——它們不像 MMIO 那樣「位址一寫就到」，每次存取都是一筆要透過控制器代發的**交易**（transaction）。ACPI 用 GenericSerialBus 這種特殊的 OperationRegion 把交易包裝成 Field 存取，讓 ASL 程式碼幾乎感覺不到差異。

本專案模擬一顆經典的 I2C 溫度感測器（以 TI TMP102 為藍本，位址 `0x48`）：宣告 I2C 控制器與感測器兩個裝置、用 `_CRS` 綁定匯流排位址、透過 GenericSerialBus 讀出兩個 raw byte，再用定點數技巧換算成攝氏溫度回傳——完整走一遍「ACPI → I2C 裝置 → 暫存器 → 讀值 → 回傳溫度」的資料流。

28.1 專案目標與驗收標準

本章學習目標

專案目標

- 建立 I2C 控制器 `I2C0` 與感測器 `SEN0` 兩個裝置，用 `I2cSerialBusV2` 把兩者接起來。
- 宣告 GenericSerialBus OperationRegion，以 `AttribByte` 存取感測器的兩個溫度暫存器。
- 實作 `TEMP` 方法：讀兩個 raw byte、位元組合成 12 位元讀值、以整數運算換算成攝氏溫度後 `Return`。
- 掌握 GenericSerialBus 交易緩衝區的固定格式（狀態 + 長度 + 資料）。

驗收標準

- `iasl` 編譯零錯誤、零警告（允許一個 Remark 2173，拆解中會說明它的意義）。
- acpiexec 中 `CONV (0x190)` 回傳 `0x9C4` （= 2500，即攝氏 25.00 度）——換算邏輯正確。
- acpiexec 中 `TEMP` 回傳 `0x3EBE`，且你能解釋這個數字是怎麼來的（提示：模擬器的填充值）。

28.2 背景與設計

28.2.1 兩個裝置、一條匯流排

I2C 的世界有兩種角色：控制器（controller）發起交易，週邊（target）回應交易。ACPI 如實描述這個拓撲——`I2C0` 是控制器裝置，真實平台上它會有自己的 MMIO 與中斷（我們也照樣宣告，讓範例更貼近實物）；`SEN0` 是感測器，它的 `_CRS` 用 `I2cSerialBusV2` 描述子說明：「我掛在 `\_SB.I2C0` 上、位址 `0x48`、速率 400 kHz」。OS 的 I2C 子系統就靠這行把感測器掛到正確的匯流排上（第 8 章資源描述子的序列匯流排版本）。

模擬對象 TMP102 的暫存器地圖（節錄）：

Command	Field	意義
0x00	`TMPH`	溫度高位元組（12 位元讀值的 bit 11–4）
0x01	`TMPL`	溫度低位元組（bit 3–0 靠左對齊於高 nibble）

溫度解析度是每 LSB 0.0625 °C：raw 讀值 400（`0x190`）代表 $400 \times 0.0625 = 25$ °C。

28.2.2 用到前面哪些章的知識

- 第 **6** 章：位移與遮罩組合位元組、整數乘除實作定點數換算。
- 第 **8** 章：`_CRS` 描述子——這次是 `I2cSerialBusV2` 而不是 `Memory32Fixed`。
- **Module 5（第 9–11 章）**：OperationRegion 的位址空間分類。GenericSerialBus 是「需要 Connection」的特殊空間：Field 的 offset 不再是位元組位址，而是傳給裝置的 command byte。
- 第 **12**、**13** 章：方法回傳值、`Serialized` 與方法內建立具名物件的互動。

28.2.3 一筆溫度讀取的完整資料流



注意中段：一次 `BUFF = TMPH` 的 Field 讀取，在真機上會展開成一整筆 I2C 交易（start、位址、command、restart、讀資料、stop），由 OS 的控制器驅動代勞——這就是 GenericSerialBus 與 SystemMemory 最大的差異：**Field 存取不再是同步的記憶體讀寫，而是一筆可能失敗的匯流排交易**，所以協定裡才需要狀態位元組。

28.3 逐步實作

Lab 實作：Project 3 實作步驟

1. 建立 `proj3/project3.dsl`，先寫 `I2C0` 與 `SEN0` 兩個裝置骨架（含 `_CRS`），編譯確認。
2. 加入 GenericSerialBus OperationRegion 與 Field （`Connection` + `AccessAs` 兩行是關鍵），編譯。
3. 先寫純數學的 `CONV`，這是可以獨立單元測試的部分。
4. 最後寫 `TEMP`，把交易、錯誤處理、位元組合、換算串起來。

project3.dsl——完整可編譯

```
DefinitionBlock ("project3.aml", "SSDT", 2, "DEMO", "PROJ3",
                 0x00000001)
{
    Scope (\_SB)
    {
        Device (I2C0)
        {
            Name (_HID, "DEMO0003")
            Name (_UID, Zero)

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)
            }

            Name (_CRS, ResourceTemplate ()
            {
                Memory32Fixed (ReadWrite, 0xFED50000, 0x00001000)
                Interrupt (ResourceConsumer, Level, ActiveHigh,
                           Exclusive) { 20 }
            })
        }

        Device (SEN0)
        {
```

```
            Name (_HID, "DEMO0004")
            Name (_UID, Zero)

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)
            }

            Name (_CRS, ResourceTemplate ()
            {
                I2cSerialBusV2 (0x0048, ControllerInitiated,
                    400000, AddressingMode7Bit, "\\_SB.I2C0",
                    0x00, ResourceConsumer, , Exclusive, )
            })

            OperationRegion (SREG, GenericSerialBus, Zero, 0x0100)
            Field (SREG, BufferAcc, NoLock, Preserve)
            {
                Connection (I2cSerialBusV2 (0x0048,
                    ControllerInitiated, 400000,
                    AddressingMode7Bit, "\\_SB.I2C0", 0x00,
                    ResourceConsumer, , Exclusive, )),
                AccessAs (BufferAcc, AttribByte),
                Offset (0x00),
                TMPH, 8,    // command 0x00: temperature MSB
                TMPL, 8,    // command 0x01: temperature LSB
            }

            // CONV: 12-bit raw value -> temperature in 0.01 degC
            Method (CONV, 1, NotSerialized)
            {
                // 1 LSB = 0.0625 degC; *625/100 -> centi-deg
                Return ((Arg0 * 625) / 100)
            }

            // TEMP: read both raw bytes, return 0.01 degC units
            Method (TEMP, 0, Serialized)
            {
                Name (BUFF, Buffer (0x10) {})
                CreateByteField (BUFF, 0x00, STAT)
                CreateByteField (BUFF, 0x02, DATA)

                BUFF = TMPH             // I2C read, command 0x00
                If (STAT != Zero)
```

```
                {
                    Return (Ones)       // bus error
                }
                Local0 = DATA           // raw MSB

                BUFF = TMPL             // I2C read, command 0x01
                If (STAT != Zero)
                {
                    Return (Ones)
                }
                Local1 = DATA           // raw LSB

                // 12-bit raw = MSB<<4 | LSB>>4 (TMP102 layout)
                Local2 = ((Local0 << 4) | (Local1 >> 4))
                Local0 = CONV (Local2)
                Debug = Local0
                Return (Local0)
            }
        }
    }
}
```

28.3.1 逐段拆解

Connection 與 AccessAs——GenericSerialBus 的 Field 宣告裡有兩行別的空間看不到的東西。`Connection (...)` 指定「接下來的欄位透過哪條連線存取」，內容就是一份 I2C 描述子（與 `_CRS` 中那份相同；一個 Field 裡可以有多個 Connection 段服務多顆裝置）。`AccessAs (BufferAcc, AttribByte)` 指定交易協定——`AttribByte` 對應 SMBus 的 read/write byte：一個 command、一個資料位元組。之後的 `TMPH, 8` 就代表「command `0x00` 那個暫存器」，**Field offset 在這裡的語義是 command byte，不是記憶體位址。**

交易緩衝區的固定格式——BufferAcc 的 Field 讀寫的不是整數而是一包緩衝區，格式由 ACPI 規格固定：

Byte	欄位	說明
0	Status	0 = 成功；非零 = 匯流排錯誤（由控制器驅動填入）
1	Length	實際傳輸的資料長度
2…	Data	資料本體（`AttribByte` 時只有 1 byte）

所以程式碼用 `CreateByteField` 在 `BUFF` 上開兩扇窗：`STAT` 對準 byte 0、`DATA` 對準 byte 2。初學者十個有九個把資料對到 byte 1——那是長度欄位，讀出來的「溫度」會永遠是 1。

錯誤處理——每次交易後先驗 `STAT`，失敗就回傳 `Ones`（64 位元全 1，慣用的錯誤哨兵值）。真實筆電上 I2C 交易可能因裝置忙碌、線路 NAK 而失敗，不檢查狀態的 ASL 會把垃圾當溫度回報給 OS。

位元組合——TMP102 把 12 位元讀值靠左放進兩個 byte：高 8 位在 `TMPH`，低 4 位在 `TMPL` 的高 nibble。`(Local0 << 4) | (Local1 >> 4)` 把兩段拼回 12 位元整數，純粹是第 6 章位移與 OR 的應用。

定點數換算——ASL 只有整數，沒有浮點數。要乘上 0.0625，訣竅是改成「乘 625、除 100」，讓結果以 0.01 度為單位：$400 \times 625/100 = 2500$，即 25.00 度。**先乘後除**是鐵則——若寫成 `Arg0 / 16`（$0.0625 = 1/16$）雖然數學等價，但整數除法會直接把小數截掉，解析度從 0.0625 度掉到 1 度；先乘 625 把精度墊高再除，才留得住小數兩位。

Remark 2173——iasl 會對 `Name (BUFF, ...)` 給一個 remark：「在方法內建立具名物件效率不佳」。這不是錯誤：ACPI 規格書的 GenericSerialBus 範例正是這樣寫的，具名物件會在方法結束時銷毀、下次呼叫重建，效能損失在這個場景可忽略。但方法必須宣告 `Serialized`——若 `NotSerialized` 且兩條路徑同時進入，第二次的 `Name` 會撞上第一次尚未銷毀的 `BUFF`，拋出 `AE_ALREADY_EXISTS`（第 13 章的教訓，iasl 也會直接警告你）。

28.4 驗證

28.4.1 先驗數學，再驗交易

`CONV` 是純函式，最適合先做單元測試：

acpiexec：用已知向量驗證換算

```
$ iasl project3.dsl
...
Compilation successful. 0 Errors, 0 Warnings, 1 Remarks, ...

$ acpiexec project3.aml
- execute \_SB.SEN0.CONV 0x190
  [Integer] = 00000000000009C4    ; 400 raw  -> 25.00 degC

- execute \_SB.SEN0.CONV 0x168
  [Integer] = 00000000000008CA    ; 360 raw  -> 22.50 degC

- execute \_SB.SEN0.CONV 0x001
  [Integer] = 0000000000000006    ; 1 raw -> 0.06 degC (truncated)
```

對照手算：`0x9C4` = 2500、`0x8CA` = 2250，換算正確。最後一筆展示整數除法的截斷——真實值 0.0625 度被截成 0.06 度，這是定點數設計時就該心裡有數的誤差。

28.4.2 執行 TEMP：看懂模擬值

acpiexec：完整交易路徑

```
- execute \_SB.SEN0.TEMP
Evaluating \_SB.SEN0.TEMP
ACPI Debug:  0x0000000000003EBE
Evaluation of \_SB.SEN0.TEMP returned object, buffer length 18
  [Integer] = 0000000000003EBE

- evaluate \_SB.SEN0.TMPH
Evaluation of \_SB.SEN0.TMPH returned object, buffer length 20
  [Buffer] Length 03 =  0000: 00 01 A0
```

`0x3EBE` = 16062，也就是攝氏 160.62 度——顯然不是真溫度。用 `evaluate` 直接讀 `TMPH` 就真相大白：acpiexec 對 GenericSerialBus 交易回傳的模擬緩衝區是 `00 01 A0`——狀態 0（成功）、長度 1、資料 `0xA0`。兩個暫存器都讀到 `0xA0`，於是 raw $= (\texttt{0xA0} \ll 4) \,|\, (\texttt{0xA0} \gg 4) = \texttt{0xA0A} = 2570$，$2570 \times 625/100 = 16062 = \texttt{0x3EBE}$。數字全部對上——這正是驗收標準要你「解釋這個數字」的原因：**模擬器驗證的是交易流程、錯誤處理與換算邏輯；資料本身是假的**。真實溫度要等這張表接上真硬體才會出現。

補充說明：規格出處與真機行為

GenericSerialBus 的交易緩衝區格式定義在 ACPI 規格「GenericSerialBus 位址空間」一節（ACPI 6.5 的 5.5.2.4.5）。真機上，Linux 由 `i2c-core-acpi` 註冊這個空間的 handler：AML 發起交易時，核心會找到 `Connection` 指到的控制器、組出真正的 I2C 訊息送上線，再把結果與狀態填回緩衝區。也就是說，同一份 ASL 在模擬器與真機上走的是同一條程式路徑，只是最底層的 handler 不同——這就是先在 acpiexec 開發、後上真機的理論保證。

28.5 常見問題

注意：學生最容易卡住的三個點

- **資料對錯位元組**：`CreateByteField (BUFF, 0x01, DATA)` 對到的是 Length 欄位，讀回來永遠是 1。資料從 byte 2 開始——狀態、長度、資料，背起來。
- **ResourceSource 字串的反斜線**：字串常值裡的 \ 必須寫成 `"\\_SB.I2C0"`（兩根），否則 iasl 報 escape 錯誤或路徑解析失敗。這與 `Scope (\_SB)` 不同——後者不在字串裡，一根就好。
- **把 Remark 當錯誤、或反過來忽略警告**：Remark 2173 可以安心留著；但若你忘了 `Serialized`，iasl 給的是警告（method creates named object, should be serialized）——那個必須修。分清楚 Error / Warning / Remark 三個層級的嚴重度。

28.6 加分挑戰

動手練習：延伸任務

1. 加上 **`_DSD` 屬性**：用 Device Properties UUID 替 `SEN0` 附上 `"compatible" = "demo,tmp102"` 與取樣率屬性，讓 Linux driver 能用 `device_property_read_*` 讀到（第 25 章的橋樑）。
2. **溫度告警**：新增門檻物件 `Name (THRS, 2800)` 與檢查方法 `CHKT`：呼叫 `TEMP`，超過門檻就 `Notify (SEN0, 0x80)`。在 acpiexec 觀察 Notify handler 印出的訊息（下一章你會大量使用它）。
3. **負溫度**：TMP102 的 raw 是 12 位元二補數。擴充 `CONV`：bit 11 為 1 時先做符號延伸再換算，用 `CONV (0xE70)` 應得攝氏 -25 度的表示。想想看：ASL 整數無號，你要怎麼表達負值？（提示：參考 `_BST` 用大整數表放電率的慣例，或回傳帶符號旗標的 Package。）

第 29 章

Project 4：Power Management

前三個專案的裝置都「永遠有電」。真實世界不是這樣：筆電闔上蓋子，觸控板該斷電；喚醒時，電源軌（power rail）要先穩、裝置才能初始化。Module 7 講過這套 D-state 理論，本專案把它做出來：一顆 `PowerResource` 控制假想的電源軌、一個裝置掛上 `_PR0` / `_PR3` 與 `_PS0` / `_PS3`，再用一個 `PWTG`（power toggle）方法模擬 OSPM 的完整開關時序，最後以 `Notify` 通知狀態改變。

這個專案最重要的產出不是程式碼，而是時序感：**上電時誰先誰後、斷電時誰先誰後**——順序寫反的韌體，輕則裝置喚不醒，重則對沒電的裝置下指令造成匯流排 hang。

29.1 專案目標與驗收標準

本章學習目標

專案目標

- 實作 `PowerResource (PWR0)`：`_ON` / `_OFF` 操作 GPIO 式的 rail 控制位元，`_STA` 回報 rail 狀態。
- 裝置 `DEV0` 以 `_PR0` / `_PR3` 宣告電源依賴、以 `_PS0` / `_PS3` 執行狀態切換，D-state 記錄在 `Name (PSTA, ...)`。
- 實作 `PWTG`：按 OSPM 的正確順序切換電源，並 `Notify (DEV0, 0x80)`。
- 畫出（並背下）D3 → D0 與 D0 → D3 兩個方向的呼叫時序。

驗收標準

- `iasl` 編譯零錯誤、零警告、零 remark。
- acpiexec 初始狀態：`PWR0._STA` 回 0、`DEV0._PSC` 回 3（開機在 D3）。
- 執行一次 `PWTG`：Debug 依序印出 `_ON` 在前、`_PS0` 在後，並看到 Notify 0x80 的訊息；再執行一次，順序變成 `_PS3` 在前、`_OFF` 在後。
- 能解釋：為什麼兩個方向的順序恰好相反。

29.2 背景與設計

29.2.1 宣告式與程序式：_PRx 與 _PSx 的分工

第 16 章介紹過裝置電源的兩套機制，本專案讓它們同台演出：

- `_PR0` / `_PR3` 是宣告式的：「我要進 D0，需要 Package 裡這些 PowerResource 是開的」。OSPM 據此做引用計數——多個裝置共享同一條 rail 時，最後一個離開的人才關燈。
- `_PS0` / `_PS3` 是程序式的：rail 已就緒後，裝置本身要做的初始化 / 收尾動作。

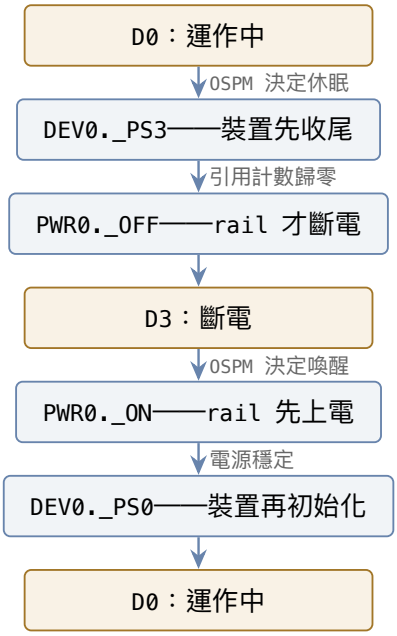
物件之間的關係如下：

Project 4 的物件關係

```
\_SB
├── PMIC (OperationRegion)
│   └── RAIL, 1          ← 假想 PMIC 的 rail 控制位元
├── PWR0 (PowerResource)
│   ├── _STA             ← 回報 0/1 (rail 是否供電)
│   ├── _ON              ← RAIL = 1 (實際動硬體)
│   └── _OFF             ← RAIL = 0
└── DEV0 (Device)
    ├── _PR0 = { PWR0 }    ← 宣告：D0 依賴 PWR0
    ├── _PR3 = { PWR0 }    ← 宣告：D3hot 仍依賴 PWR0
    ├── _PS0 / _PS3        ← 程序：切換時的裝置動作
    ├── PSTA               ← 目前 D-state 的記錄
    └── PWTG               ← 實驗用：模擬 OSPM 完整時序
```

29.2.2 OS 觀點的完整時序

這張圖是本章的核心，OSPM 切換裝置電源時的呼叫順序：



記憶口訣：**下行先裝置後電源，上行先電源後裝置**。理由很物理：收尾動作（存暫存器、通知韌體）需要裝置還有電；初始化動作同樣需要電源已經穩定。兩個方向都是「有電的那一端先動作」。

29.2.3 用到前面哪些章的知識

- 第 16 章：D-state、_PSx / _PRx / PowerResource 的語義（本專案的理論來源）。
- 第 14 章：Notify 事件通知——0x80 是「狀態改變，請 OS 重新查詢」的慣用值。
- 第 9、10 章：rail 控制位元用 OperationRegion + Field 實作，手法同 Project 2。
- 第 12 章：跨 Scope 的方法呼叫（DEV0 的方法呼叫 _SB.PWR0._ON）。

29.3 逐步實作

Lab 實作：Project 4 實作步驟

1. 建立 proj4/project4.dsl：先宣告 PMIC OperationRegion 與 RAIL 位元（1 位元就夠）。
2. 實作 PowerResource（PWR0）三件套：_STA、_ON、_OFF，每個都加 Debug 輸出——時序驗證全靠這些訊息。
3. 實作 DEV0：身分三件套 + PSTA + _PR0 / _PR3 + _PS0 / _PS3 + _PSC。
4. 最後實作 PWTG，嚴格按照時序圖排列呼叫順序。

project4.dsl——完整可編譯

```
DefinitionBlock ("project4.aml", "SSDT", 2, "DEMO", "PROJ4",
                 0x00000001)
{
    Scope (\_SB)
    {
        // Virtual PMIC: one register, bit 0 = rail enable
        OperationRegion (PMIC, SystemMemory, 0xFED41000, 0x04)
        Field (PMIC, DWordAcc, NoLock, Preserve)
        {
            RAIL, 1,    // bit 0: 1 = rail powered
        }

        PowerResource (PWR0, 0x00, 0x0000)
        {
            Name (PWRS, Zero)       // cached rail state

            Method (_STA, 0, NotSerialized)
            {
                Return (PWRS)       // 0 = off, 1 = on
            }

            Method (_ON, 0, Serialized)
```

```
            {
                RAIL = One          // touch the hardware bit
                PWRS = One          // then update the cache
                Debug = "PWR0._ON : rail powered"
            }

            Method (_OFF, 0, Serialized)
            {
                RAIL = Zero
                PWRS = Zero
                Debug = "PWR0._OFF: rail off"
            }
        }

        Device (DEV0)
        {
            Name (_HID, "DEMO0005")
            Name (_UID, Zero)

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)
            }

            Name (PSTA, 0x03)       // D-state bookkeeping: D3

            Name (_PR0, Package () { PWR0 })
            Name (_PR3, Package () { PWR0 })

            Method (_PS0, 0, Serialized)
            {
                PSTA = Zero
                Debug = "DEV0._PS0: enter D0"
            }

            Method (_PS3, 0, Serialized)
            {
                PSTA = 0x03
                Debug = "DEV0._PS3: enter D3"
            }

            Method (_PSC, 0, NotSerialized)
            {
                Return (PSTA)       // current D-state
```

```
            }

            // PWTG: lab-only toggle, emulates OSPM ordering
            Method (PWTG, 0, Serialized)
            {
                If (\_SB.PWR0._STA () == One)
                {
                    // D0 -> D3: _PS3 first, then cut the rail
                    _PS3 ()
                    \_SB.PWR0._OFF ()
                }
                Else
                {
                    // D3 -> D0: rail up first, then _PS0
                    \_SB.PWR0._ON ()
                    _PS0 ()
                }
                Notify (\_SB.DEV0, 0x80)
                Return (PSTA)
            }
        }
    }
}
```

29.3.1 逐段拆解

PowerResource 的參數——`PowerResource (PWR0, 0x00, 0x0000)` 的第二個參數是 SystemLevel（此資源在哪個睡眠深度以上仍需維持，0 表示 S0）、第三個是 ResourceOrder（多資源時的開關排序權重）。實驗用途填 0 即可，真機上 OEM 用它們編排多條 rail 的上電順序。

`PWRS` 快取——`_ON` / `_OFF` 寫完硬體位元後，順手把狀態記在 `PWRS`，`_STA` 只讀快取。這是真實韌體的常見模式：有些 rail 控制暫存器是唯寫的（讀不回狀態），或讀取代價高（要走 I2C 問 PMIC），快取一份是最務實的解法。代價是快取與現實可能脫鉤——所以更新順序必須是「先動硬體、再改快取」，萬一寫硬體失敗，快取才不會謊報。

`_PR0` 與 `_PR3` 都掛 PWR0——D0 需要供電理所當然；D3 也宣告依賴，代表這裡的 D3 是 **D3hot**（主功能斷了、但輔助電源還在，裝置仍可被列舉）。若 OSPM 決定進一步斷光（**D3cold**），才會把 `_PR3` 的資源也放掉。一個 Package 的差異，就把第 16 章 D3hot / D3cold 的抽象定義變成看得見的程式碼。

`PWTG` 只是代打——真實系統中，走這套時序的是 OSPM（Linux 的 ACPI device PM 程式碼），韌體只提供 `_ON` / `_OFF` / `_PSx` 這些「零件」。acpiexec 裡沒有 OSPM 幫我們跑流程，所以用 `PWTG` 把時序自己排一遍——它存在的目的就是讓你把順序寫對一次、印出來看一次。最

後的 `Notify (\_SB.DEV0, 0x80)` 通知 OS「狀態變了，重新查詢我」，真機上會觸發 driver 的 notify handler（第 14 章）。

補充說明：兩種 _STA，兩種語義

裝置的 _STA 回傳位元遮罩（`0x0F`：存在、啟用、顯示、正常）；PowerResource 的 _STA 只回傳 0 或 1（off / on）。名字相同、語義不同，混用是新手常見錯誤：如果 `PWR0._STA` 回傳 `0x0F`，OSPM 會把它當「on」處理沒錯，但你的程式若拿它跟 One 比較就會出錯——本專案的 PWTG 恰好就這樣寫，故意讓你踩不得。

29.4 驗證

按劇本執行，把每一行輸出跟時序圖對照：

acpiexec 完整驗證（輸出節錄）

```
$ iasl project4.dsl
...
Compilation successful. 0 Errors, 0 Warnings, 0 Remarks, ...

$ acpiexec project4.aml
- execute \_SB.PWR0._STA
  [Integer] = 0000000000000000      ; rail off

- execute \_SB.PWR0._ON
ACPI Debug:  "PWR0._ON : rail powered"

- execute \_SB.PWR0._STA
  [Integer] = 0000000000000001      ; rail is now on

- execute \_SB.PWR0._OFF
ACPI Debug:  "PWR0._OFF: rail off"

- execute \_SB.DEV0._PSC
  [Integer] = 0000000000000003      ; device still in D3

- execute \_SB.DEV0.PWTG
ACPI Debug:  "PWR0._ON : rail powered"
ACPI Debug:  "DEV0._PS0: enter D0"
ACPI Exec: Global:  Received a Device Notify on [DEV0]
           Value 0x80 (Status Change)
  [Integer] = 0000000000000000      ; now in D0

- evaluate \_SB.RAIL
```

```
  [Integer] = 0000000000000001      ; hardware bit really set

- execute \_SB.DEV0.PWTG
ACPI Debug:  "DEV0._PS3: enter D3"
ACPI Debug:  "PWR0._OFF: rail off"
ACPI Exec: Global:  Received a Device Notify on [DEV0]
           Value 0x80 (Status Change)
  [Integer] = 0000000000000003      ; back to D3
```

劇本分兩段。前半把 `_ON` / `_STA` / `_OFF` 當獨立零件逐一驗證——它們本來就是設計給 OSPM 單獨呼叫的；注意此時 `_PSC` 仍回 3：光開關 rail 不會改變 D-state 的記錄，兩套狀態是分開的。後半的 `PWTG` 才是整套時序：驗收重點全在 Debug 行的順序——上行 `_ON` 先於 `_PS0`；下行 `_PS3` 先於 `_OFF`，與時序圖一致。acpiexec 內建的 Notify handler 如實回報了 0x80；`evaluate \_SB.RAIL` 直接讀硬體位元，證明 `_ON` 不是只改了快取。

29.5 常見問題

注意：學生最容易卡住的三個點

- **時序寫反**：先 `_OFF` 再 `_PS3`，acpiexec 照樣跑完不報錯——模擬器裡沒有會壞掉的硬體。但上真機就是對斷電裝置下指令。驗收時請盯著 Debug 順序核對，不要只看回傳值。
- **以為 OSPM 會呼叫 `PWTG`**：不會。OSPM 只認識 `_PSx` / `_PRx` / `_ON` / `_OFF` 這些規格定義的名字；`PWTG` 是我們的實驗治具。真機上驅動 D-state 轉換的是 OS 的 runtime PM，不是你手動執行方法。
- **開機初值不一致**：`PSTA` 初值 3、`PWRS` 初值 0，兩者必須描述同一個事實（D3 = rail off）。如果 `PSTA` 寫 0 但 rail 是關的，OSPM 會以為裝置在 D0，直接跳過上電流程——這類「紙面狀態與硬體脫鉤」是電源韌體最典型的 bug。

29.6 加分挑戰

動手練習：延伸任務

1. **共享 rail 與引用計數**：加入 `DEV1`，`_PR0` 同樣掛 `PWR0`，並在 `PWR0` 內維護 `Name (REFC, 0)`：提供 `PON` / `POFF` 方法做計數（進來加一、離開減一，歸零才真正 `_OFF`）。模擬「兩個裝置都睡了，rail 才能關」的 OSPM 行為。
2. **喚醒能力 `_PRW`**：替 `DEV0` 加上 `Name (_PRW, Package () { 0x0D, 3 })`——GPE 0x0D、最深可從 S3 喚醒（第 17 章）。說明它與 `_PR3` 的關係：喚醒源在 D3 仍需要哪些資源維持供電。
3. **接上 Project 2**：把 `RAIL` 位元換成呼叫 `\_SB.GPI0.SETP`（rail 掛在 GPIO pin 2 上），

體會真實平台「PMIC 或 GPIO 控制 rail」的兩種常見接法，以及跨裝置呼叫的路徑寫法。

第 30 章

Project 5：DSDT Reverse Engineering（Final Project）

畢業專案。前四個專案的對象都是你自己寫的表——乾淨、短小、沒有祕密。這次的對象是你手上那台真實筆電的 **DSDT**：數萬行、沒有註解、混雜著十年來各代 OEM 工程師的手筆。你的任務是把它萃取出來、反組譯、畫出這台機器的「地圖」、深挖其中一個裝置，最後安全地做一次修改並驗證。這正是 ACPI Firmware Engineer 的日常：多數時候你不是從零寫表，而是在別人的表裡找答案、修問題。

本章採「任務書 + 指引」的形式：每個階段給你目標、指令與搜尋策略，但不再手把手——地圖要你自己畫，因為每台機器都不同。請準備：一台 Linux 筆電（或桌機）、Module 9 裝好的 acpica-tools，以及至少半天不被打擾的時間。

30.1 專案目標與驗收標準

本章學習目標

專案目標

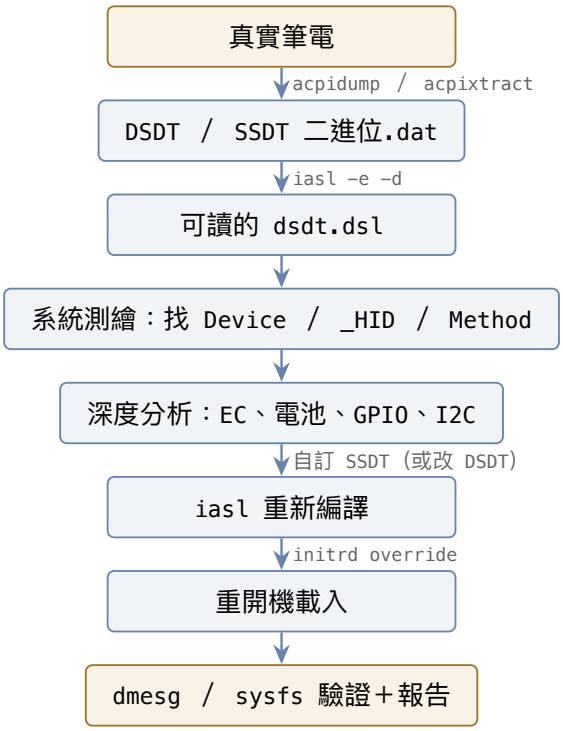
- 完整執行「萃取 → 反組譯 → 測繪 → 深度分析 → 修改 → 驗證」的逆向工程流程。
- 產出你這台機器的**機器地圖**：EC、電池、LID、電源鍵、I2C 觸控板、GPIO 控制器、PCIe root ports 分別在 Namespace 的哪裡、用什麼 `_HID`。
- 深度解剖一個裝置（建議電池或 LID）：畫出 OperationRegion / Field / Method 的呼叫關係，並與 dmesg、`/sys` 的實際行為對照。
- 以自訂 SSDT 的方式對真機做一次安全修改並驗證；了解（但不強制執行）直接改 DSDT 的完整流程。

驗收標準——繳交四樣交付物：

- 工作目錄：含 `acpi.dat`、各 `.dat` / `.dsl`、你的 SSDT 原始碼與筆記。
- 機器地圖（表格或樹狀圖，格式不拘但必須完整）。
- 上機證據：dmesg 的 Table Upgrade 訊息、`/sys/bus/acpi/devices` 中你的裝置、以及還原後系統正常開機的證明。
- 一頁分析報告（骨架見階段五）。

30.2 背景與設計

30.2.1 整體流程



這條流程整合了全書：萃取與反組譯是第 21 章工具鏈；看懂表靠 Module 2–7 的語法與物件知識；override 機制是 Module 8 與 Module 10 的實戰；驗證手段則是 Module 10 的 sysfs 與 dmesg 觀察點。

30.2.2 安全哲學：先加法，後改法

修改真機韌體描述有兩條路。**加法**——寫一張自己的 SSDT 疊加上去：影響範圍只有你新增的物件，拔掉 initrd 就完全還原。**改法**——反組譯 DSDT、修改、重編譯、整張替換：能改到既有行為，但反組譯再重編譯本身就可能引入差異（工具版本、External 解析），風險高一個量級。本章的原則：**能用加法就不用改法；改法只在受控環境（QEMU 或備用機）先演練過才碰主力機**。這不是膽小，是職業素養——你改的是開機路徑上的東西。

30.3 逐步實作

前四個階段照順序完成；最後的驗證與報告（階段五）獨立成節。每完成一個階段就把產出物歸檔進工作目錄，不要憑記憶做下一步。

30.3.1 階段一：萃取與反組譯

Lab 實作：階段一操作

1. 建立有日期與機型的工作目錄——逆向工程的檔案會越長越多，從第一天就分門別類。
2. 用 `acpidump` 一次倒出所有表，`acpixtract` 拆檔。
3. 反組譯 DSDT 時，用 `-e` 把所有 SSDT 掛進來解析跨表引用，否則 External 符號會滿天飛。
4. 立刻把原始 `.dat` 檔做一份唯讀備份。

萃取與反組譯（輸出節錄，數字依機器而異）

```
$ mkdir -p ~/acpi-lab/mylaptop && cd ~/acpi-lab/mylaptop
$ sudo acpidump -o acpi.dat
$ acpixtract -a acpi.dat
...
  DSDT -   65239 bytes written (0x0000FEE7) - dsdt.dat
  SSDT -    1972 bytes written (0x000007B4) - ssdt1.dat
  SSDT -    3186 bytes written (0x00000C72) - ssdt2.dat
...
$ iasl -e ssdt*.dat -d dsdt.dat
...
Disassembly completed
ASL Output:    dsdt.dsl - 1.2 MB

$ cp *.dat /some/readonly/backup/
```

另一條等價的取表路徑是 sysfs——單獨要某一張表時更快：

替代做法：直接從 sysfs 取表

```
$ sudo cat /sys/firmware/acpi/tables/DSDT > dsdt.dat
```

30.3.2 階段二：系統測繪——畫出機器地圖

打開 `dsdt.dsl`，你面對的是幾萬行陌生程式碼。不要從頭讀——**用搜尋策略定位地標**。下表是七個必找目標與對應的搜尋線索（`rg` 是你的好朋友）：

目標	搜尋線索	要記下什麼
EC	`PNP0C09`、`EmbeddedControl`	裝置路徑、Region 大小、`_Qxx` 事件方法清單
電池	`PNP0C0A`、`_BST`、`_BIF`	路徑、讀哪些 EC 欄位、誰對它 `Notify`
LID	`PNP0C0D`、`_LID`	路徑、`_LID` 讀的來源欄位
電源鍵	`PNP0C0C`（固定功能則在 FADT）	存在與否、`_PRW` 喚醒宣告
I2C 觸控板	`PNP0C50`、`I2cSerialBus`	匯流排位址、掛在哪個控制器、中斷來源
GPIO 控制器	`GpioInt`、`GpioIo`、Intel 常見 `INT34` 開頭的 `_HID`	控制器路徑、哪些裝置消費它的 pin
PCIe root ports	`PNP0A08`、`RP0` 命名、`_ADR`	root complex 路徑、各 port 的位址

實際操作長這樣：

測繪範例：定位 EC 與它的事件方法

```
$ rg -n "PNP0C09" dsdt.dsl
10382:  Name (_HID, EisaId ("PNP0C09"))

$ rg -n "EmbeddedControl" dsdt.dsl
10394:  OperationRegion (ECRM, EmbeddedControl, 0x00, 0xFF)

$ rg -n "Method \(_Q" dsdt.dsl | head -4
10771:  Method (_Q09, 0, NotSerialized)
10786:  Method (_Q0B, 0, NotSerialized)
10801:  Method (_Q0C, 0, NotSerialized)
10817:  Method (_Q11, 0, NotSerialized)
```

把找到的東西整理成地圖（樹狀圖或表格都行），每一項都附上行號——之後深度分析與寫報告會反覆回來查。找不到某一項也是有效資訊：例如桌機沒有 LID、新平台的電源鍵可能是 FADT 描述的固定功能而非 Namespace 裝置（把「為什麼沒有」寫進報告，比硬湊一個答案更專業）。

30.3.3 階段三：深度分析一個裝置（以電池為例）

從地圖挑一個裝置往下鑽。建議電池：它的資料流長、橫跨 EC 與事件機制，最能練功。目標是畫出這張呼叫關係圖（以下是常見筆電的典型結構，你的機器細節會不同）：

電池裝置的呼叫關係（典型結構示意）

```
\_SB.PCI0.LPCB.EC0_            ← EC (PNP0C09)
├── OperationRegion ECRM (EmbeddedControl, 0x00, 0xFF)
├── Field：BTEX,1  BST0,8  BRTE,16  BVOL,16 ...
```

```
└── Method _Q09                ← EC 事件：充放電狀態改變
     └── Notify (BAT0, 0x80)   ← 通知 OS：重新讀 _BST

\_SB.PCI0.LPCB.BAT0            ← 電池 (PNP0C0A)
├── _STA ← 讀 EC0.BTEX：電池在位？ (0x1F / 0x0F)
├── _BIF ← 讀設計容量、電壓等欄位，組成 Package
└── _BST ← Acquire (ECMX)、讀 BST0 / BRTE / BVOL、
           組 Package 回傳 (狀態、電流、餘量、電壓)
```

分析時回答四個問題：_BST 讀了哪些 Field？有沒有拿 Mutex 保護 EC 存取（第 13 章）？_Q09 之類的事件方法對誰 Notify、值是多少？_STA 的回傳值在電池拔除時變成什麼？然後把紙上分析跟活系統對照：

對照 dmesg 與 sysfs 的實際行為

```
$ sudo dmesg | grep -i "battery"
ACPI: battery: Slot [BAT0] (battery present)

$ cat /sys/class/power_supply/BAT0/voltage_now
12341000

$ acpi_listen    # unplug AC, watch the events
battery BAT0 00000080 00000001
ac_adapter ACAD 00000080 00000000
```

acpi_listen 印出的 00000080 正是 AML 裡 Notify (BAT0, 0x80) 的值——紙上讀到的程式碼與機器的實際行為在這裡對上，逆向工程的閉環完成。

30.3.4 階段四：安全修改

做法 A（必做）：疊加自訂 SSDT。把下面這張探測表載入你的真機——它加一個虛擬裝置、在 _INI 印出 Debug 訊息，證明你有能力把程式碼送進自己筆電的 ACPI Interpreter：

probe.dsl——完整可編譯

```
DefinitionBlock ("probe.aml", "SSDT", 2, "DEMO", "PROBE",
                 0x00000001)
{
    Scope (\_SB)
    {
        Device (PRB0)
        {
            Name (_HID, "DEMO0099")
            Name (_UID, Zero)
```

```
            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)
            }

            Method (_INI, 0, Serialized)
            {
                Debug = "PROBE: custom SSDT is alive"
                Debug = \_OS        // interpreter's OS name
                Debug = \_REV       // ACPI revision implemented
            }

            Method (PING, 0, NotSerialized)
            {
                Return (Ones)
            }
        }
    }
}
```

`_INI` 會在 OS 建完 Namespace、確認 `_STA` 回報存在之後被自動執行一次——是「開機時跑一段我的 AML」最正當的掛載點。`\_OS` 與 `\_REV` 是 Interpreter 的預定義物件；先用 acpiexec 試跑，你會看到有趣的答案：

先在 acpiexec 預演（節錄）

```
$ iasl probe.dsl && acpiexec probe.aml
...
Initializing Device/Processor/Thermal objects ...
ACPI Debug:  "PROBE: custom SSDT is alive"
ACPI Debug:  "Microsoft Windows NT"
ACPI Debug:  0x0000000000000002
    Executed 1 _INI methods requiring 1 _STA executions
```

`_OS` 回答「Microsoft Windows NT」——包括 Linux 在內的 ACPICA 系 Interpreter 都固定回這個字串，因為太多 OEM 的 AML 會比對 `_OS`，回答別的反而觸發沒人測過的程式路徑。接著上真機（GRUB 的 early initrd 機制）：

上真機：打包並掛進 GRUB

```
$ mkdir -p kernel/firmware/acpi
$ cp probe.aml kernel/firmware/acpi/
$ find kernel | cpio -H newc --create > probe.img
$ sudo cp probe.img /boot/
```

```
$ echo 'GRUB_EARLY_INITRD_LINUX_CUSTOM="probe.img"' \
      | sudo tee -a /etc/default/grub
$ sudo update-grub
$ sudo reboot
```

做法 **B**（了解流程，在 **QEMU** ／備用機演練）：直接改 **DSDT**。完整流程：反組譯（同階段一）→ 修改內容 → 把表頭的 OEM Revision 加一 → 重新編譯 → 與做法 A 相同的打包與掛載。表頭修改示意：

dsdt.dsl 表頭（片段——非完整檔案）

```
// Before:
//   DefinitionBlock ("", "DSDT", 2, "OEMID ", "OEMTBL  ",
//                    0x00000001)
// After: bump the OEM revision by one
DefinitionBlock ("", "DSDT", 2, "OEMID ", "OEMTBL  ",
                 0x00000002)
```

重編譯與打包（在 QEMU 或備用機演練）

```
$ iasl -ve dsdt.dsl
$ mkdir -p kernel/firmware/acpi
$ cp dsdt.aml kernel/firmware/acpi/
$ find kernel | cpio -H newc --create > dsdt-override.img
```

補充說明：OEM Revision 為什麼要加一

核心的 table upgrade 以簽章比對要替換的表——DSDT 一定整張換，不加版號其實也會生效。加一是專業習慣，理由有二：dmesg 與 `acpidump` 會顯示版號，一眼分辨機器上跑的是原廠表還是你的表；而 Windows 對表有快取機制，版號不變可能繼續用舊表。順帶一提，反組譯再重編譯的 DSDT 常帶著一批 iasl 警告——那是 OEM 用舊版編譯器留下的歷史，修掉它們是本章的加分挑戰之一。

30.4 驗證與報告（階段五）

重開機後，依序收集證據：

上機驗證清單

```
$ sudo dmesg | grep -i "table upgrade"
ACPI: Table Upgrade: install [SSDT-DEMO  -PROBE   ]

$ ls /sys/bus/acpi/devices/ | grep DEMO
DEMO0099:00
```

```
$ cat /sys/bus/acpi/devices/DEMO0099:00/path
\_SB_.PRB0
```

想在真機 dmesg 看到 `_INI` 的 Debug 輸出，核心需編譯 `CONFIG_ACPI_DEBUG` 並加開機參數 `acpi.debug_layer=0xffffffff acpi.debug_level=0x2`（Module 10 的 Debug object 章節）；看不到訊息不代表失敗，sysfs 裡有裝置就是載入成功。你也可以用 acpiexec 對萃取出的原廠表做離線對照——例如確認某方法存在、試跑不碰硬體的純邏輯方法：

離線對照：把原廠表餵給 acpiexec

```
$ acpiexec dsdt.dat ssdt1.dat
- find _BST
\_SB_.PCI0.LPCB.BAT0._BST Method ...
```

最後對照驗收清單，一項一項打勾：

項目	通過條件
萃取	`acpi.dat`、全部 `.dat` 與備份齊全，`dsdt.dsl` 反組譯成功
地圖	七個目標都有結論（找到路徑，或說明為何不存在）
深度分析	呼叫關係圖完成，且至少一項與 dmesg / sysfs / `acpi_listen` 的實際行為互相印證
安全修改	dmesg 有 Table Upgrade 訊息、sysfs 看得到 `DEMO0099:00`
還原	移除 override 後重開機，dmesg 無 Table Upgrade、系統一切正常

分析報告寫一頁就好，骨架如下：

1. **機器概況**：機型、BIOS 版本、核心版本、表清單（幾張 SSDT、各自大小）。
2. **機器地圖**：七個目標的路徑與 `_HID` 總表。
3. **深度分析**：你挑的裝置、呼叫關係圖、與實際行為的對照證據。
4. **修改與驗證**：改了什麼、怎麼載入、驗證輸出、如何還原。
5. **問題與心得**：卡住的地方、與教材範例不同之處、下一步想挖的東西。

30.5 常見問題

注意：動真機前必讀的三件事

- **Secure Boot 與 kernel lockdown**：啟用 Secure Boot 時核心進入 lockdown，會拒絕 ACPI table override ——dmesg 出現 lockdown 相關訊息、你的表被無聲忽略。先用 `mokutil --sb-state` 與 `cat /sys/kernel/security/lockdown` 檢查；必要時在 BIOS 關閉 Secure Boot，或改在 QEMU 完成實驗。
- **永遠準備還原路**：override 只存在於 initrd，不會燒進韌體——刪掉 `probe.img`、移除 GRUB 那行、`update-grub` 就完全復原；最壞情況在 GRUB 選單挑舊核心或用救援 USB 開機即可。動手前先確認自己知道這條退路怎麼走。
- **絕不在生產機器上第一次實驗**：所有流程先在 QEMU 或備用機跑通，確定指令與檔案路徑都對了，才在日常工作機上重演。改壞 DSDT 的典型症狀是開機卡死在早期階段——雖然救得回來，但你不會想在趕工作的早上處理這件事。

30.6 加分挑戰

動手練習：延伸任務（畢業加冕）

1. **修掉一個原廠警告**：反組譯你的 DSDT 重新編譯，挑一個 iasl 警告（常見如「Not all control paths return a value」或未使用的 Local 變數），讀懂該方法、修正它、重編譯到警告消失。在 QEMU 驗證後，寫下你對這個警告成因的推測。
2. **把 Project 1 接上驅動**：把 `DEV0`（`DEMO0001`）的 SSDT 載入真機，再用第 25 章的 platform driver 範本寫一個比對 `DEMO0001` 的 `acpi_match_table`，載入模組後在 dmesg 看到 probe 成功——從 ASL 到 C driver 的全鏈路貫通。
3. **比較研究**：找第二台不同品牌的機器重跑階段一與二，比較兩家 OEM 的 EC 事件方法（`_Qxx` 的數量與用途）與電池實作差異，寫成半頁筆記。看過兩家的手筆，你對「什麼是慣例、什麼是怪癖」會有全新的體感。

附錄 A

開發環境安裝指南

本附錄整理各平台的工具安裝步驟。全書實作至少需要 `iasl`；建議完整安裝 acpica-tools 與 QEMU。

A.1 Linux

Ubuntu / Debian

```
$ sudo apt update
$ sudo apt install acpica-tools        # iasl, acpidump, acpixtract, acpiexec
$ sudo apt install qemu-system-x86 ovmf   # QEMU + UEFI firmware (optional)
```

Fedora / RHEL

```
$ sudo dnf install acpica-tools
$ sudo dnf install qemu-kvm edk2-ovmf
```

A.2 macOS

```
$ brew install acpica      # iasl, acpiexec, acpixtract
$ brew install qemu
```

macOS 沒有 `/sys/firmware/acpi`，抓取實機 ACPI tables 的練習（第 22 章、Project 5）請在 Linux 實體機或虛擬機上進行。

A.3 驗證安裝

```
$ iasl -v
Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20230628

$ echo 'DefinitionBlock ("t.aml", "SSDT", 2, "TEST", "TEST", 1) {}' > t.dsl
$ iasl t.dsl && ls t.aml
```

```
ASL Input:     t.dsl -   1 lines, 55 bytes, 0 keywords
AML Output:    t.aml -   36 bytes, 0 named objects
t.aml
```

A.4 建議的專案目錄結構

```
acpi-lab/
 ├── src/        ← 自己寫的 .dsl
 ├── build/      ← iasl 產出的 .aml
 ├── dump/       ← acpidump 取得的機器表
 └── notes/      ← 分析筆記與機器地圖
```

附錄 B

ACPI 預定義物件速查表

以下按用途分類整理本書用到的預定義物件（完整清單見 ACPI Spec 第 5.6.6 節與第 6 章）。

B.1 識別與列舉

物件	用途
`_HID`	Hardware ID：driver 綁定依據（EISA ID 或 8 字元字串）
`_CID`	Compatible ID：相容識別碼（單一或 Package）
`_UID`	Unique ID：同 `_HID` 多實例時區分用
`_ADR`	可列舉匯流排上的位址（如 PCI 的 device/function）
`_CLS`	PCI class code（給無 `_HID` 的裝置）
`_STR`	裝置的人類可讀名稱（Unicode）
`_SUN`	插槽編號（slot user number）

B.2 狀態與資源

物件	用途
`_STA`	裝置狀態位元圖（見下表）
`_INI`	裝置初始化方法（namespace 載入後呼叫）
`_REG`	OperationRegion handler 可用性通知
`_CRS`	目前使用的硬體資源（ResourceTemplate）
`_PRS`	可能的資源組態
`_SRS`	設定資源組態
`_DIS`	停用裝置資源解碼
`_DSD`	Device Specific Data：key-value 裝置屬性
`_DSM`	Device Specific Method：UUID 分派的萬用方法

`_STA` 回傳值位元定義：

位元	意義	說明
bit 0	Present	裝置存在
bit 1	Enabled	已啟用並解碼資源
bit 2	Show in UI	應顯示於使用者介面
bit 3	Functioning	運作正常（0 = 故障）
bit 4	Battery Present	電池裝置專用
常見值：`0x0F` 正常存在、`0x0B` 存在但不顯示、`0x00` 不存在		

B.3 裝置電源管理

物件	用途
`_PS0`–`_PS3`	將裝置切換到 D0–D3 的方法
`_PR0`–`_PR3`	各 D-state 所需的 PowerResource 列表
`_ON` / `_OFF` / `_STA`	PowerResource 的開、關、查詢
`_PRW`	喚醒能力：GPE 編號 + 最低可喚醒 S-state
`_PSW` / `_DSW`	啟用 / 設定裝置喚醒
`_S0W`–`_S4W`	各 S-state 下可喚醒的最深 D-state

B.4 系統睡眠

物件	用途
`_PTS`	Prepare To Sleep：進睡眠前韌體準備
`_WAK`	喚醒後處理，回傳喚醒狀態
`_TTS`	Transition To State：睡眠轉換前後各呼叫一次
`_S0_`–`_S5_`	各 S-state 的 SLP_TYP 值（Package）

B.5 電池、按鍵與熱管理

物件	用途
`_BIF` / `_BIX`	電池靜態資訊（設計容量等）
`_BST`	電池動態狀態（電量、充放電）
`_LID`	上蓋狀態（1 = 開）
`_TMP`	熱區目前溫度（單位 0.1 K）
`_PSV` / `_CRT` / `_HOT`	Passive / Critical / Hibernate 觸發點
`_AC0`–`_AC9`	Active cooling 觸發點（風扇檔位）
`_PSS` / `_PCT` / `_PPC`	CPU P-state 表 / 控制暫存器 / 上限
`_CST` / `_CSD`	CPU C-state 表 / 相依性

B.6 常用 Notify 代碼

代碼	名稱	典型用途
0x00	Bus Check	重新列舉整條匯流排
0x01	Device Check	檢查裝置存在狀態（熱插拔）
0x02	Device Wake	裝置要求喚醒
0x03	Eject Request	退出請求
0x80	裝置自訂	電池狀態改變、熱區溫度變化等
0x81	裝置自訂	熱區 trip point 改變、電池資訊改變等

附錄 C

工具指令速查

C.1 iASL 常用旗標

指令	用途
`iasl test.dsl`	編譯，產生 `test.aml`
`iasl -d dsdt.dat`	反組譯，產生 `dsdt.dsl`
`iasl -e ssdt*.dat -d dsdt.dat`	反組譯時解析外部（SSDT）符號
`iasl -tc test.dsl`	另輸出 C hex 陣列（EDK II 嵌入用）
`iasl -l test.dsl`	產生 ASL/AML 對照 listing（`.lst`）
`iasl -p out test.dsl`	指定輸出檔名
`iasl -vw 3144 ...`	抑制特定編號的警告
`iasl -g`	取得系統 ACPI tables（僅 Windows；Linux 用 acpidump）
`iasl -h` / `iasl -v`	說明 / 版本

C.2 取表與執行

Linux 取得並反組譯 ACPI tables

```
$ sudo acpidump > acpi.out       # dump all tables
$ acpixtract -a acpi.out         # split into dsdt.dat, ssdt1.dat, ...
$ iasl -e ssdt*.dat -d dsdt.dat  # disassemble with externals
```

acpiexec 常用指令

```
$ acpiexec dsdt.dat              # load table(s) into userspace interpreter
- namespace                      # dump whole namespace
- find _HID                      # find objects by name
- execute \_SB.DEV0.TEST         # run a method
- debug \_SB.DEV0.TEST           # single-step a method
- quit
```

C.3 Linux ACPI 觀察點

路徑／指令	用途
`/sys/firmware/acpi/tables/`	原始 ACPI tables（可直接 dd 出來）
`/sys/bus/acpi/devices/`	列舉出的 ACPI 裝置（path、hid、status）
`/sys/firmware/acpi/interrupts/`	GPE 觸發計數（抓 GPE storm）
`/sys/module/acpi/parameters/`	執行期調整 debug_layer ／ debug_level
`dmesg | grep -i acpi`	開機與執行期 ACPI 訊息
`acpi.debug_layer= acpi.debug_level=`	開機參數：開啟 AML Debug 輸出

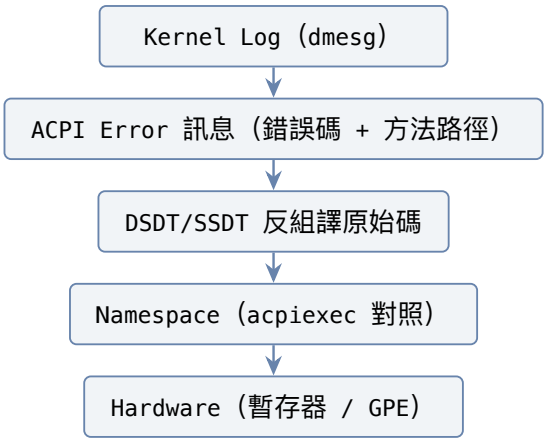
附錄 D

ACPI Error Code 對照表

ACPICA 的錯誤碼以 `AE_` 開頭，出現在 kernel log 與 acpiexec 輸出中。完整定義見 ACPICA 原始碼 `include/acexcep.h`。

錯誤碼	意義	典型成因與修法
`AE_ALREADY_EXISTS`	名稱重複建立	NotSerialized 方法內 `Name()` 被重入；SSDT 與 DSDT 同名衝突。改 Serialized 或改名。
`AE_NOT_FOUND`	找不到參照的物件	引用不存在的名稱；SSDT 缺 `External`；韌體版本差異。用 `CondRefOf` 防禦。
`AE_AML_OPERAND_TYPE`	運算元型別錯誤	對 Package 做算術、把 Buffer 當 Integer 用。檢查隱含轉換規則（第 4 章）。
`AE_AML_UNINITIALIZED_ARG`	Arg 未初始化	呼叫端傳的參數數量不足。核對方法宣告的參數個數。
`AE_AML_UNINITIALIZED_LOCAL`	Local 未初始化	某分支忘了先賦值就讀取 LocalX。
`AE_AML_PACKAGE_LIMIT`	Package 索引越界	`Index()` 超出 Package 長度。先用 `SizeOf` 檢查。
`AE_AML_BUFFER_LIMIT`	Buffer 索引越界	CreateXXXField 或 Index 超出 Buffer 大小。
`AE_AML_INVALID_RESOURCE_TYPE`	資源描述子型別錯誤	`_CRS` 回傳的 ResourceTemplate 內容不合法。
`AE_AML_LOOP_TIMEOUT`	While 迴圈逾時	輪詢硬體旗標永遠等不到。檢查 Field 位址與硬體狀態。
`AE_TIME`	逾時	Acquire mutex / Wait event 逾時。檢查鎖的釋放路徑。

除錯時的定位順序（詳見第 24 章）：


Kernel Log（dmesg）
ACPI Error 訊息（錯誤碼 + 方法路徑）
DSDT/SSDT 反組譯原始碼
Namespace（acpiexec 對照）
Hardware（暫存器 / GPE）

參考資料

官方規格與文件

- **ACPI Specification 6.5**, UEFI Forum.
  `https://uefi.org/specifications`
  一切問題的最終答案。建議搭配本書：第 5 章（tables）、第 6 章（device configuration）、第 19 章（ASL reference）。
- **ACPICA —ACPI Component Architecture**, Intel.
  `https://www.acpica.org/`、`https://github.com/acpica/acpica`
  iASL、acpiexec、acpidump 的官方專案；Linux kernel 內嵌同一套核心。
- **iASL Compiler/Disassembler User Guide**, ACPICA.
  `https://www.acpica.org/documentation`
- **ASL Tutorial（Introduction to ASL）**, Intel ACPICA 文件集。
  本書 Module 2–6 的官方對照讀物。
- **ASL 2.0 Introduction（Symbolic Operators）**, ACPICA 文件集。
- **ACPICA User Guide and Programmer Reference**, ACPICA 文件集。

Linux 相關

- **Linux Kernel Firmware Guide —ACPI**.
  `https://docs.kernel.org/firmware-guide/acpi/`
  含 DSDT override、`_DSD` 屬性、GPIO/I2C 列舉等實務文件。
- **Linux ACPI 子系統原始碼**：`drivers/acpi/`（kernel tree）。
- **FirmWare Test Suite（fwts）**, Canonical.
  `https://wiki.ubuntu.com/FirmwareTestSuite`

韌體與 UEFI

- **EDK II**, TianoCore.
  `https://github.com/tianocore/edk2`
  開源 UEFI 參考實作；ACPI 表的生產端（第 25 章）。
- **UEFI Specification**, UEFI Forum.
  `https://uefi.org/specifications`

- **OSDev Wiki —ACPI**.
  https://wiki.osdev.org/ACPI
  入門視角的補充讀物。

延伸學習路徑

完成本書後，建議依職涯方向擇一深入：

方向	下一步
BIOS/UEFI 工程師	EDK II 開發、平台 bring-up、Intel FSP / AMD AGESA
Linux 核心工程師	ACPI 子系統、platform/I2C/GPIO driver、電源管理框架
Embedded / SoC	Device Tree 與 ACPI 的對照、ARM SBSA/SBBR 平台標準
韌體資安	AML 攻擊面、SMM 安全、fwupd / LVFS 生態