

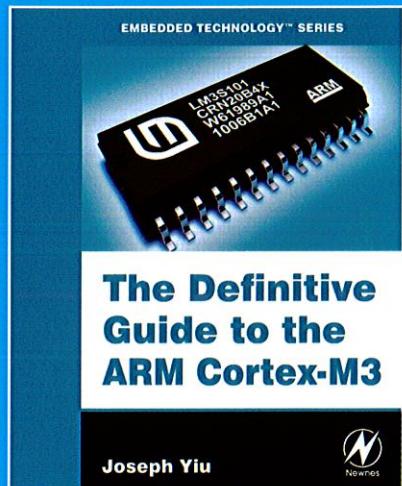
Joseph Yiu 著 Jason 嘴書—EETOP 世界唯一貼  
劉榮春 譯

# ARM Cortex-M3 嵌入式系統 設計入門

官方  
教材

The Definitive Guide to the ARM Cortex-M3

- ARM 官方 Cortex-M3 教材
- 市面上唯一針對 Cortex-M3 微處理器的教科書
- ARM7 與 Cortex-M3 微處理器差異說明
- 同時涵蓋 GNU 工具鏈與 KEIL RealView Microcontroller Development Kit(RVMDK) 開發工具
- 同時涵蓋組合語言與 C 程式語言



Jason 嘴書—EETOP 世界唯一貼

Flag Publishing

<http://www.flag.com.tw>

# ARM Cortex-M3 嵌入式系統 設計入門

官方  
教材

Definitive Guide to the ARM Cortex-M3

Joseph Yiu 原著 劉榮春譯

## 前言

微控制器程式設計師的天性富於活用資源，他們藉著獨特的方式去實作微控制器，在固定的設計上創造出新的產品。他們總是從最儉省的系統設計裡，獲取高效率的運算。完成這樣點石成金工作的要素就是工具鏈環境；因此，ARM 本身的工具鏈部門的工程師聯合 CPU 設計師組成了一個團隊，以合理、簡化、並改良 ARM7TDMI 處理器的設計。

此組合的成果，也就是 ARM Cortex-M3，代表原有的 ARM 架構的嶄新發展。此設計將 32 位元 ARM 架構的最佳特性與相當成功的 Thumb-2 指令集設計融合，並增加一些新的功能。即使作了這些改變，所有現存的 ARM 愛好者應該很容易察覺到，Cortex-M3 依舊保留一個簡化的程式設計模型。

--Wayne Lyons

Director of Embedded Solution, ARM

## 作者序

此書是為了對 ARM 的 Cortex-M3 處理器有興趣的硬體與軟體工程師兩者所寫。雖然 Cortex-M3 Technical Reference Manual (TRM) 與 ARMv7-M Architecture Application Level Reference Manual 已經對此新處理器提供了許多資訊，但是都太過於描述細節而不適合新手閱讀。

本書意圖為程式設計師、嵌入式產品設計者、系統單晶片(SoC)工程師、電子產品玩家、學術研究員、以及其他具有一些微控制器或微處理器經驗的人，提供較易閱讀的材料。內容包括了新架構的介紹、指令集整理、一些指令的範例、硬體特性的資訊、以及處理器的進階除錯系統的綜觀。本書並提供應用範例，包括使用 ARM 工具以及 GNU 工具鏈來進行 Cortex-M3 處理器軟體開發的基本步驟。此書亦以熟悉 ARM7TDMI 處理器而正過渡到 Cortex-M3 處理器的工程師為目標，因此亦涵蓋這些處理器之間的差異，以及從 ARM7TDMI 移植應用軟體至 Cortex-M3。

## 致謝

在此我向校閱本書並提供意見與回饋的下列人員致謝：

Alan Tringham, Dan Brook, David Brash, Haydn Povey, Gary Campbell, Kevin McDermott, Richard Earnshaw, Samin Ishtiaq, Shyam Sadasivan, Simon Axford, Simon Smith, Stephen Theobald, 與 Wayne Lyons。

我也要感謝 CodeSourcery 提供的技術支援，以及 Luminary Micro 為本書英文版封面提供的圖片，並且當然地，感謝 Elsevier 員工對本書出版的專業工作。

最後，向鼓勵我寫此書的 Peter Cole 與 Ivan Yardley 深致謝意。

## 術語與縮寫

縮寫	意義
ADK	AMBA 設計套件
AHB	進階高效能匯流排
AHB-AP	AHB 存取埠
AMBA	進階微控制器匯流排架構
APB	進階週邊匯流排
ARM ARM	ARM 架構參考手冊
ASIC	應用特定積體電路
ATB	進階追蹤匯流排
BE8	Byte Invariant Big Endian 模式
CPI	每一指令的週期
CPU	中央處理單元
DAP	除錯存取埠
DSP	數位信號處理器/數位信號處理
DWT	資料觀察點與追蹤
ETM	嵌入式追蹤巨集格
FPB	快閃補丁與中斷點
FSR	錯誤狀態暫存器
HTM	CoreSight AHB 追蹤巨集格

接下頁

# Jason 嘴書—EETOP 世界唯一貼

ICE	線上模擬器
IDE	整合開發環境
IRQ	中斷要求(通常意指外部中斷)
ISA	指令集架構
ISR	中斷服務程式
ITM	設備追蹤巨集格
JTAG	聯合測試工作組(為測試/除錯介面的標準)
JTAG-DP	JTAG 除錯埠
LR	連結暫存器
LSB	最低有效位元
LSU	載入/儲存單元
MCU	微控制器單元
MMU	記憶體管理單元
MPU	記憶體保護單元
MSB	最高有效位元
MSP	主要堆疊指標
NMI	不可遮罩中斷
NVIC	巢狀向量中斷控制器
OS	作業系統
PC	程式計數器
PSP	程序堆疊指標
PPB	私有週邊匯流排
PSR	程式狀態暫存器
SCS	系統控制空間
SIMD	單一指令多重資料
SP, MSP, PSP	堆疊指標, 主要堆疊指標, 程序堆疊指標
SoC	系統單晶片
SW	序列線
SW-DP	序列線除錯埠
SWJ-DP	序列線 JTAG 除錯埠
SWV	序列線檢視器(TPIU 之一運算模式)
TPA	追蹤埠分析儀
TPIU	追蹤埠介面單元
TRM	技術參考手冊

## 使用慣例

本書採用各種表示慣例, 如下所示:

◆ 一般的組合語言程式碼:

```
MOV R0, R1 ; 將資料從暫存器 R1 移到暫存器 R0
```

◆ 組合語言程式的通用語法; 在 <> 內的項目必須換成暫存器的名稱:

```
MRS <reg>, <special_reg> ;
```

◆ C 程式碼:

```
for (i=0;i<3;i++) { func1(); }
```

◆ 虛擬碼:

```
if (a>b) { ... }
```

◆ 數值:

1. 4'hC, 0x123 兩者皆為十六進位值
2. #3 意指第 3 個項目(例如, IRQ #3 意指 IRQ 編號 3)
3. #immed\_12 表示 12-bit 的立即值資料
4. 暫存器位元

一般用來表示根據位元位置定義的部分數值。例如, bit[15:12]就表示第 15 至 12 位元的數值。

## ◆ 暫存器存取種類：

1. R 表唯讀
2. W 表唯寫
3. R/W 表可讀取或寫入
4. R/Wc 表可讀取並被寫入存取所清除

**參考資料**

參考號碼	文件
1	Cortex-M3 Technical Reference Manual (TRM) 可從 ARM 文件網站下載： <a href="http://www.arm.com/documentation/ARMProcessor_Cores/index.html">www.arm.com/documentation/ARMProcessor_Cores/index.html</a>
2	ARMv7-M Architecture Application Level Reference Manual 可從 ARM 文件網站下載： <a href="http://www.arm.com/products/CPUs/ARM_Cortex-M3_v7.html">www.arm.com/products/CPUs/ARM_Cortex-M3_v7.html</a>
3	CoreSight Technology System Design Guide 可從 ARM 文件網站下載： <a href="http://www.arm.com/documentation/Trace_Debug/index.html">www.arm.com/documentation/Trace_Debug/index.html</a>
4	AMBA Specification 可從 ARM 文件網站下載： <a href="http://www.arm.com/products/solutions/AMBA_Spec.html">www.arm.com/products/solutions/AMBA_Spec.html</a>
5	AAPCS Procedure Call Standard for the ARM Architecture 可從 ARM 文件網站下載： <a href="http://www.arm.com/pdfs/aapcs.pdf">www.arm.com/pdfs/aapcs.pdf</a>
6	RVCT 3.0 Compiler and Library Guide 可從 ARM 文件網站下載： <a href="http://www.arm.com/pdfs/DUI0205G_rvct_compiler_and_libraries_guide.pdf">www.arm.com/pdfs/DUI0205G_rvct_compiler_and_libraries_guide.pdf</a>
7	ARM Application Note 179: Cortex-M3 Embedded Software Development 可從 ARM 文件網站下載： <a href="http://www.arm.com/documentation/Application_Notes/index.html">www.arm.com/documentation/Application_Notes/index.html</a>

**第 1 章 簡介**

什麼是 ARM Cortex-M3 處理器？ .....	1-1
ARM 與 ARM 架構的背景 .....	1-3
○ 簡史 .....	1-3
○ 各種架構版本 .....	1-4
○ 處理器命名 .....	1-6
指令集的發展 .....	1-8
Thumb-2 指令集架構(ISA) .....	1-9
Cortex-M3 處理器應用 .....	1-10
本書架構 .....	1-11
進一步的閱讀資料 .....	1-11

**第 2 章 Cortex-M3 概觀**

基礎 .....	2-1
暫存器 .....	2-2
○ R0 至 R12：一般用途暫存器 .....	2-2
○ R13：堆疊指標 .....	2-3
○ R14：連結暫存器 .....	2-3
○ R15：程式計數器 .....	2-4
○ 特殊暫存器 .....	2-4
操作模式 .....	2-5
內建的巢狀向量中斷控制器 .....	2-6
○ 巢狀中斷支援 .....	2-6
○ 向量中斷支援 .....	2-7
○ 動態優先權變動支援 .....	2-7
○ 減少中斷延遲 .....	2-7
○ 中斷遮罩 .....	2-7
記憶體映射 .....	2-7
匯流排介面 .....	2-8
記憶體保護單元 .....	2-9
指令集 .....	2-9
中斷與例外 .....	2-11
低功率與高電能效率 .....	2-12
除錯支援 .....	2-13
特性總結 .....	2-14
○ 高性能 .....	2-14
○ 進階中斷處理功能 .....	2-15
○ 低電力耗損 .....	2-15

○ 系統特性 .....	2-16
○ 除錯的支援 .....	2-16
<b>第 3 章 Cortex-M3 基礎</b>	
<b>暫存器 .....</b>	<b>3-1</b>
○ 一般用途暫存器 R0-R7 .....	3-1
○ 一般用途暫存器 R8-R12 .....	3-1
○ 堆疊指標暫存器 R13 .....	3-2
○ 連結暫存器 R14 .....	3-4
○ 程式計數器 R15 .....	3-5
<b>特殊暫存器 .....</b>	<b>3-5</b>
○ 程式狀態暫存器(Program Status Register, PSRs) .....	3-6
○ PRIMASK, FAULTMASK, 和 BASEPRI 暫存器 .....	3-8
○ 控制暫存器 .....	3-9
○ CONTROL[1] .....	3-9
○ CONTROL[0] .....	3-9
<b>操作模式 .....</b>	<b>3-10</b>
<b>例外與中斷 .....</b>	<b>3-12</b>
<b>向量表 .....</b>	<b>3-13</b>
<b>堆疊記憶體運算 .....</b>	<b>3-14</b>
○ 堆疊基本運算 .....	3-14
○ Cortex-M3 堆疊實現 .....	3-16
○ Cortex-M3 雙堆疊模式 .....	3-17
<b>重置順序 .....</b>	<b>3-18</b>
<b>第 4 章 指令集</b>	
<b>組合語言基礎 .....</b>	<b>4-1</b>
○ 組譯器語言：基本語法 .....	4-1
○ 組譯器語言：使用後綴字 .....	4-3
○ 組譯器語言：統一組譯器語言 .....	4-3
<b>指令列表 .....</b>	<b>4-5</b>
○ 未支援的指令 .....	4-9
<b>指令描述 .....</b>	<b>4-11</b>
○ 組譯器語言：資料移動 .....	4-11
○ 虛擬指令 LDR 和 ADR .....	4-15
○ 組譯器語言：資料處理 .....	4-16
○ 組譯器語言：Call 和 Unconditional(無條件的)跳躍 .....	4-21
○ 組譯器語言：抉擇與條件式跳躍 .....	4-22
○ 組譯器語言：比較與條件式跳躍的結合 .....	4-26
○ 組譯器語言：使用 IT 指令的條件式跳躍 .....	4-26

○ 組譯器語言：指令障礙與記憶體障礙的指令 .....	4-28
○ 組譯器語言：飽和運算 .....	4-29
<b>Cortex-M3 裡一些有用的指令 .....</b>	<b>4-31</b>
○ MSR 和 MRS .....	4-31
○ IF-THEN .....	4-32
○ CBZ 與 CBNZ .....	4-33
○ SDIV 和 UDIV .....	4-34
○ REV、REVH、和 REVSH .....	4-34
○ RBIT .....	4-35
○ SXTB、SXTH、UXTB、和 UXTH .....	4-35
○ BFC 和 BFI .....	4-36
○ UBFX 和 SBFX .....	4-37
○ LDRD 和 STRD .....	4-37
○ TBB 和 TBH .....	4-38

## 第 5 章 記憶體系統

<b>記憶體系統特性概觀 .....</b>	<b>5-1</b>
<b>記憶體映射 .....</b>	<b>5-2</b>
<b>記憶體存取的屬性 .....</b>	<b>5-5</b>
<b>預設的記憶體存取權限 .....</b>	<b>5-6</b>
<b>Bit-Band 運算 .....</b>	<b>5-7</b>
○ Bit-Band 運算的優點 .....	5-11
○ 不同資料大小的 Bit-Band 運算 .....	5-15
○ C 程式中的 Bit-Band 運算 .....	5-15
<b>非對齊傳輸 .....</b>	<b>5-16</b>
<b>獨占存取 .....</b>	<b>5-18</b>
<b>Endian 模式 .....</b>	<b>5-20</b>

## 第 6 章 Cortex-M3 架構實作概觀

<b>管線技術 .....</b>	<b>6-1</b>
<b>Cortex-M3 詳細的方塊圖 .....</b>	<b>6-3</b>
<b>Cortex-M3 上的匯流排介面 .....</b>	<b>6-6</b>
○ I-Code 匯流排 .....	6-7
○ D-Code 匯流排 .....	6-7
○ 系統匯流排 .....	6-7
○ 外部私有周邊匯流排 .....	6-7
○ 除錯存取埠匯流排 .....	6-7
<b>Cortex-M3 上的其他介面 .....</b>	<b>6-8</b>
<b>外部的私有周邊匯流排 .....</b>	<b>6-8</b>

典型的連接方式 .....	6-10
重置信號 .....	6-11

## 第 7 章 例外

例外類型 .....	7-1
優先權定義 .....	7-3
向量表 .....	7-9
中斷輸入與等待行為 .....	7-10
錯誤例外 .....	7-14
◎ 匯流排錯誤 .....	7-14
◎ 記憶體管理錯誤 .....	7-16
◎ 用法錯誤 .....	7-17
◎ 硬錯誤 .....	7-19
◎ 應付錯誤的方式 .....	7-20
SVC 與 PendSV .....	7-20

## 第 8 章 NVIC 與中斷控制

NVIC 總觀 .....	8-1
基本中斷組態 .....	8-2
中斷致能與清除致能 .....	8-2
中斷等待與清除等待 .....	8-4
◎ 優先權等級 .....	8-5
◎ 活動狀態 .....	8-5
◎ PRIMASK 與 FAULTMASK 特殊暫存器 .....	8-6
◎ BASEPRI 特殊暫存器 .....	8-6
◎ 其它例外的組態暫存器 .....	8-7
設定中斷的範例程序 .....	8-9
軟體中斷 .....	8-11
SYSTICK 計時器 .....	8-12

## 第 9 章 中斷行為

中斷/例外序列 .....	9-1
◎ 堆疊 .....	9-1
◎ 向量擷取 .....	9-4
◎ 暫存器更新 .....	9-4
例外出口 .....	9-4
巢狀中斷 .....	9-5
末尾連鎖中斷 .....	9-6

最後到達例外 .....	9-6
再論例外返回值 .....	9-7
中斷延遲 .....	9-9
與中斷相關的錯誤 .....	9-10
◎ 進堆疊 .....	9-10
◎ 去堆疊 .....	9-10
◎ 向量擷取 .....	9-11
◎ 不合法返回 .....	9-11

## 第 10 章 Cortex-M3 程式設計

綜觀 .....	10-1
◎ 使用組合語言 .....	10-1
◎ 使用 C .....	10-2
組合語言與 C 之間的介面 .....	10-3
典型開發流程 .....	10-4
第一步 .....	10-5
產生輸出 .....	10-6
◎ “Hello World” 範例 .....	10-7
使用資料記憶體 .....	10-11
使用獨占存取作號誌 .....	10-12
使用 Bit-Band 作號誌 .....	10-15
使用位元欄位取出與表格跳躍 .....	10-16

## 第 11 章 例外程式設計

中斷的使用 .....	11-1
◎ 堆疊設定 .....	11-1
◎ 向量表設定 .....	11-2
◎ 中斷優先權設定 .....	11-3
◎ 致能中斷 .....	11-4
例外/中斷處理程式 .....	11-5
軟體中斷 .....	11-7
例外處理程式範例 .....	11-8
使用 SVC .....	11-11
SVC 範例：作為輸出函數 .....	11-12
以 C 來使用 SVC .....	11-15

## 第 12 章 進階程式設計特性與系統行為

使用兩個不同堆疊來執行系統 .....	12-1
---------------------	------

Double-Word 堆疊對齊 .....	12-4
Nonbase 執行緒致能 .....	12-5
效能考慮 .....	12-8
鎖住情形 .....	12-10
○ 鎖住時會發生哪些事情？ .....	12-10
○ 避免鎖住 .....	12-11
錯誤遮罩 .....	12

## 第 13 章 記憶體保護單元

綜觀 .....	13-1
MPU 暫存器 .....	13-2
設定 MPU .....	13-7
典型設定 .....	13-13
○ 使用次區域除能的範例 .....	13-13

## 第 14 章 其它 Cortex-M3 的特性

SYSTICK 計時器 .....	14-1
電源管理 .....	14-5
多個處理器之間的通信 .....	14-8
自我重置的控制 .....	14-11

## 第 15 章 除錯架構

除錯特性概觀 .....	15-1
CoreSight 概觀 .....	15-2
○ 處理器除錯介面 .....	15-2
○ 除錯 Host 介面 .....	15-3
○ DP 模組、AP 模組、與 DAP .....	15-3
○ 追蹤介面 .....	15-5
○ CoreSight 特性 .....	15-5
除錯模式 .....	15-7
除錯事件 .....	15-10
Cortex-M3 內的中斷點 .....	15-12
在除錯時存取暫存器內容 .....	15-13
其它核心除錯的特性 .....	15-15

## 第 16 章 除錯的元件

簡介 .....	16-1
----------	------

○ Cortex-M3 內的追蹤系統 .....	16-2
追蹤元件：資料觀察點與追蹤(Data Watchpoint and Trace) .....	16-3
追蹤元件：儀器追蹤巨集格(Instrumentation Trace Macrocell, ITM) .....	16-5
○ 使用 ITM 作軟體追蹤 .....	16-5
○ 以 ITM 與 DWT 作硬體追蹤 .....	16-6
○ ITM 時戳 .....	16-6
追蹤元件：嵌入式追蹤巨集格(Embedded Trace Macrocell, ETM) .....	16-7
追蹤元件：追蹤埠介面單元(Trace Port Interface Unit, TPIU) .....	16-8
快閃補丁與中斷點單元(Flash Patch and Breakpoint Unit, FPB) .....	16-8
AHB 存取埠 .....	16-11
ROM 表 .....	16-12

## 第 17 章 Cortex-M3 開發入門

選擇 Cortex-M3 產品 .....	17-1
Cortex-M3 Revision 0 與 Revision 1 之間的差別 .....	17-2
○ 修正版 1 的改變：由 JTAG-DP 傳輸到 SWJ-DP .....	17-4
Cortex-M3 Revision 1 與 Revision 2 之間的差別 .....	17-5
○ 預設組態為 Double word 堆疊對齊 .....	17-5
○ 新的輔助控制暫存器 .....	17-5
○ ID 暫存器值的更新 .....	17-6
○ 除錯特性 .....	17-6
○ 睡眠特性 .....	17-6
修正版 2 的好處與影響 .....	17-7
開發工具 .....	17-8
○ C 編譯器 .....	17-9
○ 嵌入式作業系統的支援 .....	17-9

## 第 18 章 從 ARM7 移植應用程式到Cortex-M3

概觀 .....	18-1
系統特色 .....	18-2
○ 記憶體映射 .....	18-2
○ 中斷 .....	18-2
○ MPU .....	18-3
○ 系統控制 .....	18-3
○ 運算模式 .....	18-4
組合語言檔案 .....	18-5
○ Thumb 狀態 .....	18-5
○ ARM 狀態 .....	18-5
C 程式檔案 .....	18-8

預先編譯的目的檔.....	18-8
最佳化.....	18-8

## 第 19 章 使用 GNU 工具鏈開始 Cortex-M3 的開發

背景.....	19-1
取得 GNU 工具鏈.....	19-2
開發流程.....	19-2
範例.....	19-3
○ 範例 1：第一個程式.....	19-3
○ 範例 2：連結多個檔案.....	19-5
○ 範例 3：一個簡單的"Hello World"程式.....	19-7
○ 範例 4：RAM 裡面的資料.....	19-9
○ 範例 5：僅有 C 而無組合語言的檔案.....	19-10
○ 範例 6：僅用 C, 以及標準的 C 啟動程式.....	19-13
存取特殊暫存器.....	19-16
使用未支援的指令.....	19-16
GNU C 編譯器裡的行內組譯器.....	19-16

## 第 20 章 KEIL RealView 微控制器開發套件入門

概觀.....	20-1
$\mu$ Vision 入門.....	20-2
藉由 UART 輸出"Hello World"訊息.....	20-8
測試軟體.....	20-11
使用除錯器.....	20-13
指令集模擬器.....	20-17
修改向量表.....	20-19
使用中斷的碼表範例.....	20-19

## 附錄 A Cortex-M3 指令摘要

## 附錄 B 16-Bit Thumb 指令與各種架構版本

## 附錄 C Cortex-M3 例外快速參考

## 附錄 D NVIC 暫存器快速參考

## 附錄 E Cortex-M3 問題解決指南

## 簡介

本章內容包括：

- ✓ 什麼是 ARM Cortex-M3 處理器？
- ✓ ARM 與 ARM 架構的背景
- ✓ 指令集的發展
- ✓ Thumb-2 指令集架構(ISA)
- ✓ Cortex-M3 處理器應用
- ✓ 本書架構
- ✓ 進一步的閱讀資料

## 什麼是 ARM Cortex-M3 處理器？

微控制器的市場廣大，估計至 2010 年每一年會出廠兩百億個元件。成陣成列令人目為之眩的廠商、元件、架構在此市場裡競爭。隨著工業需求的變動，全面性地驅使了更高效能微控制器的要求；例如，在不增加產品的頻率或功率下，微控制器需要去處理更多的工作。除此以外，微控制器藉著通用串聯匯流排(Universal Serial BUS, USB)、乙太網(Ethernet)、或無線電波，彼此互連的狀況更為頻繁，因而支援通信通道，與進階周邊的處理能力的需求日益增長。同樣地，更複雜的使用介面、多媒體需求、系統速度、功能合併等等，驅使一般的應用更為複雜。

ARM 於 2006 發表的第一個 Cortex 世代處理器：ARM Cortex-M3 處理器，原先主要目標為 32-bit 的微控制器市場。Cortex-M3 處理器以低開數提供了優良表現，且具備先前只有高階處理器才有的新的功能。Cortex-M3 以下列的方式，因應了 32-bit 嵌入式處理器市場的需求：

- ◆ **高效能表現**: 在不增加頻率與功率要求, 可完成更多的工作。
- ◆ **低功率消耗**: 可增長電池壽命, 對包括無線網路應用的可攜式產品, 特別重要。
- ◆ **強化決定**: 保證在已知週期數內儘速執行關鍵的工作與中斷。
- ◆ **更高的程式碼密度**: 保證程式適用於最小的執行參考記憶體。
- ◆ **使用便利**: 為了日益增多地從 8-bit 和 16-bit 轉到撰寫 32-bit 的人, 提供更容易撰寫程式與除錯的方式。
- ◆ **更低成本的解決方案**: 降低了 32-bit 基礎系統的成本, 接近舊的 8-bit 與 16-bit 元件的成本, 並使得低階 32-bit 微控制器, 第一次定價低於一塊美金。
- ◆ **廣泛的開發工具選擇**: 許多開發工具廠商, 提供了各種低成本或免費的編譯器, 及全功能的開發套件。

以 Cortex-M3 處理器為基礎的微控制器, 和以其它多種架構為基礎的元件正進行著正面競爭。更多的設計工程師以傳統元件成本作比較, 尋求降低系統成本。因此, 各種機構實現了元件集成, 使單一卻更有威力的元件具備取代三個到四個傳統 8-bit 元件的潛力。

另外, 也可藉由提高程式碼在整個系統中的重複使用量來節省成本。因為以 Cortex-M3 為基礎的微控制器, 可輕易地使用 C 語言來撰寫程式, 且根據的架構為大家所熟知, 故應用程式碼能容易地被移植和重複利用, 進而降低開發時間與測試成本。

值得強調的是 Cortex-M3 處理器, 並非 ARM 處理器中第一個用來製作不受商標管制的微控制器。舊有 ARM7 處理器在這個市場非常成功, 合作對象包括 NXP(Philips)、Texas Instruments、Atmel、OKI 和其它發表強韌的 32-bit 微控制器單元(Microcontroller Units, MCUs)的廠商。ARM7 為歷來使用最廣的 32-bit 嵌入式處理器, 每年生產超過十億個, 用途非常多樣, 包括行動電話與汽車的電子產品等等。

Cortex-M3 立足於 ARM7 處理器的成功之上, 交出了極易撰寫程式與除錯、且具備高度處理能力的元件。此外, Cortex-M3 處理器, 引入了一些符合微控制應用特殊需求的功能與特性, 例如: 對關鍵工作不可遮罩、高度決定性的巢狀向量中斷、不可切割的位元運作 atomic bit manipulation)、可選用的記憶體保護單元。這些要素使得

Cortex-M3 吸引了現在 ARM 處理器的使用者、及許多正考慮在其產品上使用 32-bit MCUs 的新使用者。

### Cortex-M3 處理器與以 Cortex-M3 為基礎的 MCUs 的區別

Cortex-M3 處理器是微控制器晶片的中央處理單元。此外, Cortex-M3 為基礎的微控制器則需加上其它元件。晶片製造商獲得 Cortex-M3 授權以後, 可將 Cortex-M3 處理器放進其矽晶片設計, 並加上記憶體、周邊、輸入/輸出、和其它功能等。不同廠商基於 Cortex-M3 製作的晶片, 具有不同的記憶體大小、種類、周邊、和功能等。本書聚焦於處理器核心的架構。對晶片其它的細節, 請參考個別的晶片製造商文件。

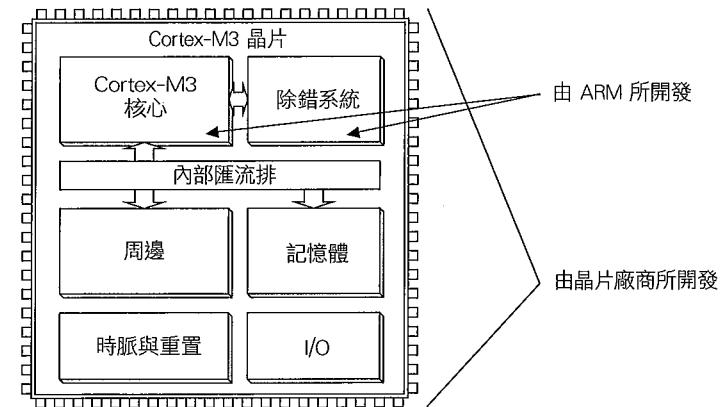


圖 1-1 Cortex-M3 處理器與以 Cortex-M3 為基礎的 MCUs 的區別

## ARM 與 ARM 架構的背景

### 簡史

為了幫助你了解各種 ARM 處理器與架構的版本, 讓我們稍微看一下 ARM 的歷史。

ARM 成立於 1990 年, 全名為 Advanced RISC Machines Ltd. ; 為蘋果電腦、Acorn Computer Group、及 VLSI Technology 聯合創投。ARM 於 1991 推出了 ARM6 處理器系列, VLSI 是最早獲得授權的公司。接下來, 更多的公司授權了 ARM 處理器設計, 包括德州儀器、NEC、夏普、ST Microelectronics 等, 將 ARM 處理器的應用推廣到行動電話、電腦硬碟、個人數位助理(PDAs)、家庭娛樂系統、及其它消費性產品。

現今 ARM 夥伴每年出廠超過二十億個 ARM 處理器。不同於其它半導體公司，ARM 並不直接製造處理器或販賣晶片。反之，ARM 將處理器設計授權給生意夥伴，包括了全世界大部分居領先地位的半導體公司。根據 ARM 低成本和高功率效能的處理器設計，這些合作夥伴再進行製作處理器、微控制器、以及單晶片系統解決方案。這樣的生意模式經常被稱為智慧財(intellectual property, IP)授權。

除了處理器設計外，ARM 也授權了系統層級的智慧財和各種軟體智慧財。為了支援這些產品，ARM 也發展了堅強陣容的開發工具、硬體、軟體產品，以幫助合作夥伴開發產品。

## 各種架構版本

多年來 ARM 持續開發新的處理器與系統區塊，包括廣受採用的 ARM7TDMI 處理器，以及更近期使用於高階應用產品(例如智慧型手機)的 ARM1176TZ(F)-S 處理器。長期以來功能的演化與處理器的加強，帶來了 ARM 架構後續的版本。注意到架構版本的編號與處理器名稱並沒有關係，例如：ARM7TDMI 處理器是以 ARMv4T 架構為基礎(T 意指支援 Thumb 指令模式)。

ARMv5E 的架構是在 ARM9E 處理器系列裡導入。此系列包括 ARM926E-S 和 ARM946E-S 處理器。此處理器增加了應用於多媒體應用的增強版(Enhanced)數位信號處理(DSP)指令。

隨著 ARM11 處理器系列的到來，架構擴充為 ARMv6。此架構的新功能包括記憶體系統功能，和單指令多重資料(SIMD)的指令。基於 ARMv6 架構的處理器包括 ARM1136J(F)-S、ARM1156T2(F)-S、以及 ARM1176JZ(F)-S 等。

在導入了 ARM11 系列之後，決定了諸如最佳化的 Thumb-2 指令集等許多新的技術也可應用於低成本市場的微控制器與自動元件；也決定了雖然最低階 MCU 至最高效能處理器需有一致性，但仍需要導入一個最適合應用的處理器架構，使其適用於具有一定的數目、低閾數的處理器，以因應對成本敏感的市場，以及適用於高階應用的多功能高效率處理器。

在過去幾年中，ARM 藉著多樣化開發 CPU，擴充了它的產品目錄，產生版本 7 架構(v7)。於此版本，架構設計區分為下列三型：

- ◆ **A 型**：為高效能開放平台設計
- ◆ **R 型**：為需要即時(real-time)表現的高階嵌入式系統設計
- ◆ **M 型**：為深層嵌入式微控制器型態的系統設計

讓我們更進一步地看一下此三型的細節：

- ◆ **A 型(ARMv7-A)**：需要執行複雜應用的應用處理器，例如高階的嵌入式作業系統(OSs)：Symbian、Linux、Windows Embedded 等。其需要最高的處理能力、虛擬記憶體系統支援(藉著記憶體管理單元，Memory Management Units, MMUs)，以及可選用的增強的 Java 支援和保全的程式執行環境。產品的例子包括高階行動電話和作財務交易的電子錢包。
- ◆ **R 型(ARMv7-R)**：即時、高效能處理器。其目標主要為高階即時<sup>1</sup> 市場；在這些應用中，例如高階煞車系統以及硬碟控制器等，高處理能力、高依賴度、和低延遲等極為重要。
- ◆ **M 型(ARMv7-M)**：以低成本應用及工業控制應用(包括即時控制系統)為其目標的處理器；重點為具備處理效率，並且成本、功率消耗、低中斷延遲、易於使用等極為重要。

Cortex 處理器系列為開發於架構 v7 的第一個產品。而 Cortex-M3 處理器是基於 v7 架構中，被稱作 ARMv7-M 的一型，為微控制器產品的一個架構規格。

本書聚焦於 Cortex-M3 處理器，但它僅是使用了 ARMv7 架構的 Cortex 產品系列成員之一。其它的 Cortex 系列處理器，包括基於 ARMv7-A 型的 Cortex-A8(應用處理器)，以及基於 ARMv7-R 型的 Cortex-R4(即時處理器)。

1 對於是是否我們能夠藉著一般用途的處理器以得到「即時的」系統，一直受到廣泛的爭議。「即時」的定義意指系統能在保證的時間內得到回應。在基於 ARM 處理器的系統裡，因為不同的選擇，例如作業系統、中斷延遲、或記憶體延遲，以及 CPU 是否正進行更高優先權的中斷，你可能會(或可能不會)得到這樣的回應。

# Jason 嘴書—EETOP 世界唯一貼

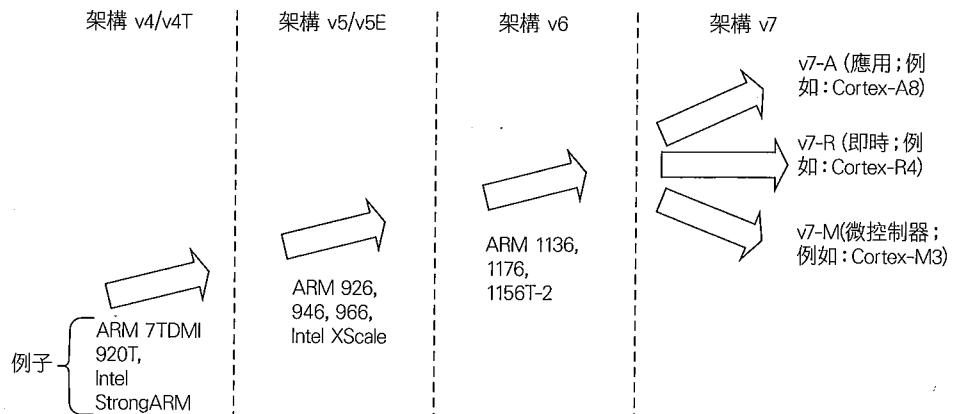


圖 1-2 ARM 處理器架構的演變

ARMv7-M 架構的細節記載於 The ARMv7 Architecture Application Level Reference Manual (Ref 2)。此文件可在 ARM 網站，經過一個簡單的註冊手續後得到。ARMv7-M 架構包括下列主要的部分：

- ◆ / 程式設計師模型 ◆ / 指令集
- ◆ / 記憶體模型 ◆ / 除錯架構

處理器特定資訊，例如介面的細節與計時等，記載於 Cortex-M3 Technical Reference Manual (TRM) (Ref 1)。此手冊可於 ARM 網站自由下載。Cortex-M3 TRM 也包含了架構規格未曾包含到的一些實作細節，例如所支援指令的列表，這是因為有些包含於 ARMv7-M 架構規格的指令，在 ARM7 元件上為選擇性的。

## 處理器命名

傳統上，ARM 採取了一個編號的方法，來為處理器命名。在早期(1990 年代)，亦使用後綴字來顯示處理器的功能。例如，ARM7TDMI 處理器，其 T 意指 Thumb 指令的支援，D 意指 JTAG 除錯，M 意指快速乘法器(Multiplier)，I 意指嵌入式 ICE 模組。後來，因為決定了上述功能應該成為未來的 ARM 處理器的標準功能；所以這些後綴字便不再加在新的處理器系列名字裡。反之，不同的記憶體介面、快取、緊密耦合記憶體(Tightly Coupled Memory)等，創造了處理器命名的新方法。

例如，具快取與 MMUs 的 ARM 處理器，得到的後綴字為： 26, 36。而具記憶體保護單元(Memory Protection Units, MPUs)的處理器，得到 46 的後綴字(例如 ARM946E-S)。此外，加上其它後綴字以顯示 synthesizable<sup>2</sup> (S) 和 Jazelle(J) 技術。表 1.1 呈現了處理器命名的總結。

表 1.1 ARM 處理器名稱

處理器名稱	架構版本	記憶體管理特性	其他的特性
ARM7TDMI	ARMv4T		
ARM7TDMI-S	ARMv4T		
ARM7EJ-S	ARMv5E		DSP, Jazelle
ARM920T	ARMv4T	MMU	
ARM922T	ARMv4T	MMU	
ARM926EJ-S	ARMv5E	MMU	
ARM946E-S	ARMv5E	MPU	DSP
ARM966E-S	ARMv5E		DSP
ARM968E-S	ARMv5E		DMA, DSP
ARM966HS	ARMv5E	MPU(選擇性的)	DSP
ARM1020E	ARMv5E	MMU	DSP
ARM1022E	ARMv5E	MMU	DSP
ARM1026EJ-S	ARMv5E	MMU 或 MPU	DSP, Jazelle
ARM1036J(F)-S	ARMv6	MMU	DSP, Jazelle
ARM1076JZ(F)-S	ARMv6	MMU + TrustZone	DSP, Jazelle
ARM11 MPCore	ARMv6	MMU + 多重處理器快取支援	DSP, Jazelle
ARM1156T2(F)-S	ARMv6	MPU	DSP
Cortex-M3	ARMv7-M	MPU(選擇性的)	NVIC
Cortex-R4	ARMv7-R	MPU	DSP
Cortex-R4F	ARMv7-R	MPU	DSP+浮點運算
Cortex-A8	ARMv7-A	MPU + TrustZone	DSP, Jazelle

自 version 7 架構開始，ARM 捨棄了難以理解的複雜命名方式，改採一致的處理器系列命名方式，並以 Cortex 為其名牌開端。這樣的方式除了能夠顯示處理器之間的相容性，也去除了架構版本與處理器系列號碼之間的混淆；例如 ARM7TDMI 並非 v7 的處理器，而是基於 v4T 的架構。

<sup>2</sup> 一個可合成的(synthesizable)核心設計，可以得到使用硬體描述語言(HDL，例如 Verilog HDL 或者 VHDL)的形式，並且可以藉由合成的軟體將其轉換為設計的網列(netlist)。

## 指令集的發展

ARM 處理器指令集的加強與擴展，一直是驅使架構演變的一個主要力量。

歷來(自從 ARM7TDMI)ARM 處理器支援了兩個不同的指令集：32-bit 的 ARM 指令與 16-bit 的 Thumb 指令。在程式執行時，處理器可以動態地切換於 ARM 狀態與 Thumb 狀態間，以使用任一種指令。Thumb 指令集僅提供了部分 ARM 指令，但它提供了更高的程式碼密度，可用在對記憶體要求較嚴格的產品上。

隨著架構版本的更新，更多指令加入了 ARM 指令與 Thumb 指令裡。附錄二提供了架構改進過程中，Thumb 指令變動的一些資訊。在 2003 年，ARM 宣布了 Thumb-2 指令集，其為包含 16-bit 與 32-bit Thumb 指令的新超集合。

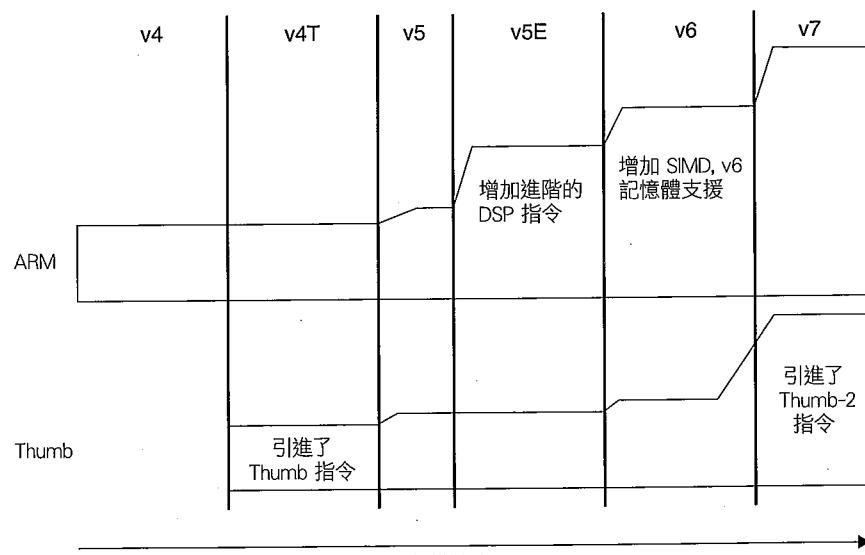


圖 1-3 指令集的改進

有關指令集的細節，可在此文件中找到：The ARM Architecture Reference Manual(亦稱作 ARM ARM)。此手冊已針對下列架構作了更新：ARMv5 架構、ARMv6 架構、以及 ARMv7 架構。針對 ARMv7 架構，因為已經成長為多種類型，其規格亦分開於不同的文件裏。對 Cortex-M3 開發者，ARM v7-M Architecture Application Level Reference Manual (Ref 2)包含了所有需要的指令集細節。

## Thumb-2 指令集架構(ISA)

Thumb-2<sup>3</sup> 指令集架構(Instruction Set Architecture, ISA)為一個高效率且具威力的指令集，可在易於使用、程式大小、以及效能各方面達到非常好的效果。Thumb-2 指令集為先前 16-bit Thumb 指令集的超集合，除了 32-bit 指令外，另增加了一些 16-bit 指令。它允許在 Thumb 狀態裡執行更多運算，故可藉著減少在 ARM 狀態與 Thumb 狀態間切換的次數，提高運算效率。

為了專注在像是微控制器等具有小型記憶體系統的設備，以及減少處理器的大小，Cortex-M3 僅提供 Thumb-2(和傳統的 Thumb)指令集。它捨棄了傳統 ARM 處理器裡某些運算會用到的 ARM 指令，而使用 Thumb-2 指令集來作所有的運算。此舉導致 Cortex-M3 處理器無法與傳統的 ARM 處理器回溯相容。也就是說，你不能在 Cortex-M3 裡執行 ARM7 處理器的影像。然而，另一方面 Cortex-M3 處理器幾乎可執行所有的 16-bit Thumb 指令(包含所有 ARM7 系列處理器支援的 16-bit Thumb 指令)，使得應用程式更易移植。

因為 Thumb-2 指令同時支援 16-bit 與 32-bit，因此並不需要在 Thumb 狀態(16-bit 指令)與 ARM 狀態(32-bit 指令)間切換。例如在 ARM7 或 ARM9 系列處理器中，如果你想要執行複雜計算或大量條件運算，且要求高的效能，則你可能得切換到 ARM 狀態。然而，就 Cortex-M3 處理器而言，你可混用 32-bit 與 16-bit 的指令，而不需切換狀態，在不增加複雜度下，得到高程式碼密度與高效能表現。

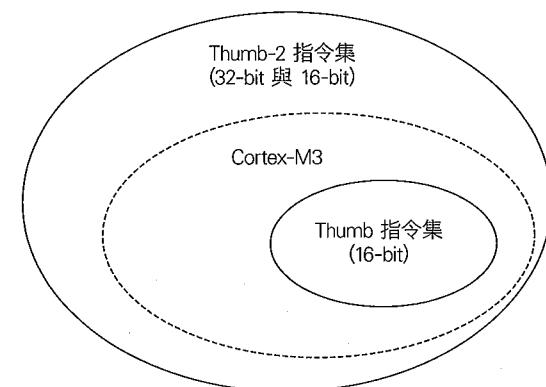


圖 1-4 Thumb-2 指令集與 Thumb 指令集之間的關係

<sup>3</sup> Thumb 與 Thumb-2 為屬於 ARM 的註冊商標。

Thumb-2 指令集是 ARMv7 架構中非常重要的特點。與 ARM 系列處理器(ARMv4T 架構)支援的指令作比較, Cortex-M3 處理器指令集有大量的新功能。ARM 處理器第一次可使用硬體除法指令, 並且 Cortex-M3 亦提供了一些乘法指令以改善消化資料(data crunching)的表現。Cortex-M3 亦支援非對齊(unaligned)資料存取(此功能先前只有高階處理器提供)。

## Cortex-M3 處理器應用

因其高效能表現、高程式碼密度、低的矽佔用域(silicon footprint), 故 Cortex-M3 處理器為廣泛應用的理想選擇：

- ◆ **低成本的微控制器**: Cortex-M3 非常適用於低成本的微控制器, 經常使用在消費性產品, 從玩具到電器等都有。因為市場上已有許多為人熟悉的 8-bit 與 16-bit 微控制器產品, 故此為一個高度競爭的市場。藉由 Cortex-M3 處理器低功率、高效能、且易於使用的優勢, 使嵌入式開發者得以轉換到 32-bit 系統, 並開發使用 ARM 架構的產品。
- ◆ **自動化**: 另一個 Cortex-M3 處理器理想的應用為自動化工業。Cortex-M3 處理器具有非常高的表現效率, 和低的中斷延遲, 故其可使用於即時系統。Cortex-M3 支援最多 240 個外部向量中斷, 藉著內建的中斷控制器(具有巢狀的中斷支援與可選用的記憶體保護單元), 使得它成為需要高整合性且對成本敏感的自動化應用理想的選擇。
- ◆ **資料通信**: 處理器的低功率與高效能, 結合了 Thumb-2 指令的 bit-field 運作, 使得 Cortex-M3 為許多通信應用(例如藍芽與 ZigBee)的理想選擇。
- ◆ **工業控制**: 在工業控制應用中, 簡單、快速反應、可靠等為主要關鍵。再一次地, Cortex-M3 處理器的中斷特性、低中斷延遲、強化的錯誤處置功能等等, 使它成為此領域強有利的候選者。
- ◆ **消費性產品**: 在許多消費性產品中, 都會使用一個(或多個)具備高效能表現的微處理器。Cortex-M3 這個小小的處理器, 由於具有高效能與低功率, 加上它支援 MPU, 因而能執行複雜的軟體, 並提供強韌的記憶體保護。

目前市場中已經有非常多基於 Cortex-M3 處理器的產品, 包括定價低至 1 美元的低階產品, 這使得 ARM 微控制器的成本, 與大部分的 8-bit 微控制器相近或更便宜。

## 本書架構

- ◆ 第一章與第二章:Cortex-M3 簡介與概觀
- ◆ 第三章至第六章:Cortex-M3 基礎
- ◆ 第七章至第九章:例外與中斷
- ◆ 第十章與第十一章:Cortex-M3 程式撰寫
- ◆ 第十二章至第十四章:Cortex-M3 硬體特性
- ◆ 第十五章與第十六章:Cortex-M3 裡的除錯支援
- ◆ 第十七章至第二十章:Cortex-M3 的應用開發
- ◆ 附錄

## 進一步的閱讀資料

本書並未包含 Cortex-M3 處理器的所有技術細節, 而是為了新接觸 Cortex-M3 處理器的人們所寫的新手指導手冊, 對於使用基於 Cortex-M3 處理器的微控制器的人來說, 也可作為補充參考。若想得到 Cortex-M3 處理器更多的細節, 則下列文件(可於 ARM 網站: [www.arm.com](http://www.arm.com) 和 ARM 合作夥伴的網站)應該包含了大部分所需的細節：

- ◆ The Cortex-M3 Technical Reference Manual (TRM) (Ref 1)提供了處理器詳細的資訊, 包括程式設計師的模型、記憶體映射、以及指令計時等等。
- ◆ The ARMv7-M Architecture Application Level Reference Manual (Ref 2)包含指令集與記憶體模型的詳細資訊。
- ◆ 參考為基於 Cortex-M3 處理器的微控制器產品而寫的資料表; 或者造訪你正計畫使用的(基於 Cortex-M3 處理器的)微控制器產品的製造廠商, 以取得資料表。

- ◆ 參考 AMBA Specification 2.0 (Ref 4), 以取得更多有關內部 AMBA 介面匯流排協定的細節。
- ◆ 關於 Cortex-M3 裡, C 程式撰寫要領可在這裡找到: ARM Application Note 179: Cortex-M3 Embedded Software Development (Ref 7)。

本書假設你對嵌入式程式已有一些知識與經驗(最好是使用 ARM 處理器)。如果你是一位經理或學生, 想要學習基礎知識, 而不要花太多時間去閱讀整本書或 TRM, 則第二章值得一讀, 因為它提供了 Cortex-M3 處理器的總整理。

## Chapter

## 2

# Cortex-M3 概觀

本章內容包括：

- |                |             |
|----------------|-------------|
| ✓ 基礎           | ✓ 記憶體保護單元   |
| ✓ 暫存器          | ✓ 指令集       |
| ✓ 操作模式         | ✓ 中斷與例外     |
| ✓ 內建的巢狀向量中斷控制器 | ✓ 除錯支援      |
| ✓ 記憶體映射        | ✓ 低功率與高電能效率 |
| ✓ 匯流排介面        | ✓ 特性總結      |

## 基礎

Cortex-M3 為 32-bit 的微處理器, 其資料路徑、暫存器庫、記憶體介面皆為 32-bit。處理器採用哈佛架構, 這意味著它擁有分開的指令匯流排與資料匯流排, 允許同時去存取指令與資料, 得以提升處理器的表現, 原因就在於資料的存取不會影響指令的管線作用 (Pipeline)。此特徵產生了 Cortex-M3 裡的多重匯流排介面, 每一個介面皆可達到最佳化的運用, 也能被同時使用。然而, 指令與資料匯流排共用了相同的記憶體空間(一個統一的記憶體系統)。換句話說, 雖然擁有獨立的匯流排介面, 但卻不會因此得到 8GB 的記憶體空間。

對需要更多記憶體系統性能的複雜應用, Cortex-M3 處理器亦有一個可選用的 MPU, 而且, 需要時也可使用一個外接快取。Cortex-M3 處理器支援 little endian 與 big endian 兩個記憶體系統。

Cortex-M3 處理器包括了一些固定的內部除錯元件。這些元件提供了除錯操作的支援及功能，例如一些中斷點(breakpoints)和觀察點(watchpoints)。除此之外，亦提供有可選用的元件，支援例如指令追蹤以及不同型態的除錯介面等除錯功能。

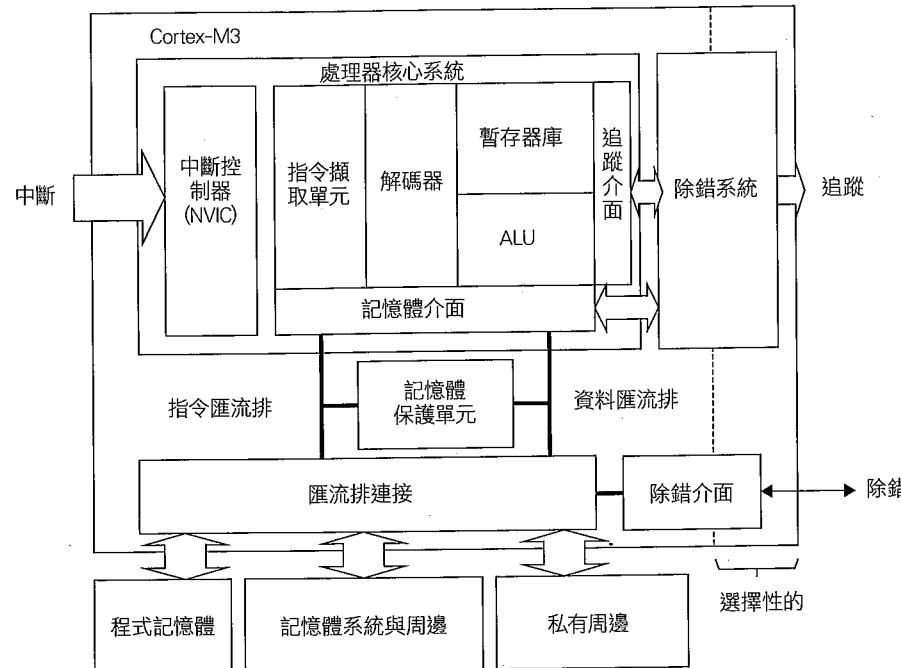


圖 2-1 Cortex-M3 簡示圖

## 暫存器

Cortex-M3 處理器擁有 R0 至 R15 的暫存器。R13 (堆疊指標) 暫存器一組兩個，在同一時間點只能見到其中一個 R13 暫存器。

### R0 至 R12：一般用途暫存器

R0 至 R12 是資料運算用的 32-bit 一般用途暫存器，一些 16-bit 的 Thumb 指令僅只能存取這些暫存器的一部分(低暫存器 R0 至 R7)。

### R13：堆疊指標

Cortex-M3 包含兩個堆疊指標，即 R13。它們兩個為一組，同時間只能看到其中一個：

- **主要堆疊指標(Main Stack Pointer, MSP)**: 預設的堆疊指標；供 OS 核心與例外處理程式(exception handlers)使用。

- **程序堆疊指標(Process Stack Pointer, PSP)**: 供應用程式碼使用。

堆疊指標的最低兩個 bits 永遠為 0，意謂其永遠以 word 為單位對齊。

名稱	功能 (與庫存的暫存器)
R0	一般用途暫存器
R1	一般用途暫存器
R2	一般用途暫存器
R3	一般用途暫存器
R4	一般用途暫存器
R5	一般用途暫存器
R6	一般用途暫存器
R7	一般用途暫存器
R8	一般用途暫存器
R9	一般用途暫存器
R10	一般用途暫存器
R11	一般用途暫存器
R12	一般用途暫存器
R13(MSP)	主要堆疊指標(MSP)
R14	連結暫存器(LR)
R15	程式計數器(PC)

圖 2-2 Cortex-M3 裡的暫存器

### R14：連結暫存器

當呼叫副程式時，返回位址存放在連結(link)暫存器裡。

## R15：程式計數器

程式計數器(program counter, PC)存放目前執行程式的位址。可寫值入此暫存器以控制程式的流程。

## 特殊暫存器

Cortex-M3 處理器也有一些特殊暫存器：

- ◊ 程式狀態暫存器(Program Status Register, 即 PSRs)
- ◊ 中斷遮罩暫存器(Interrupt Mask Register, 即 PRIMASK, FAULTMASK, 和 BASEPRI)
- ◊ 控制暫存器(Control Register, 即 CONTROL)

這些暫存器有特殊的功能，而且僅能以特殊指令來存取，不可以使用於正常的資料處理。

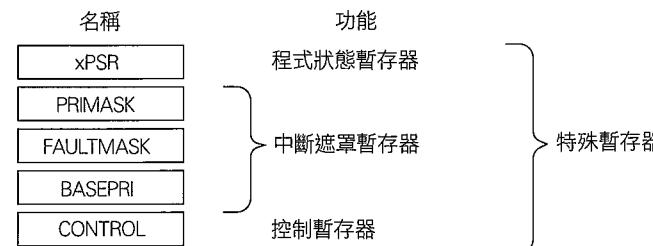


圖 2-3 Cortex-M3 裡的特殊暫存器

表 2-1 暫存器及其功能

暫存器	功能
xPSR	提供 ALU 旗標(零旗標、進位旗標)、執行狀態、與現在執行中的中斷號碼
PRIMASK	除能所有的中斷，除了不可遮罩的中斷(NMI)與硬錯誤
FAULTMASK	除能所有的中斷，除了 NMI
BASEPRI	除能所有的特定優先權等級或更低優先權等級的中斷
CONTROL	定義特權的狀態與堆疊指標之選擇

你會在第三章找到更多這些暫存器的資訊。

## 操作模式

Cortex-M3 處理器有兩種模式(modes)和兩種權限等級(privilege levels)。操作模式(執行緒模式與處理程式模式)決定處理器正執行的是正常的程式，還是例外處理程式，例如中斷處理程式或系統例外處理程式。特權的等級和用戶等級則提供了一個機制，此機制一方面保護了重要區域的記憶體存取，另一方面提供了基本的安全模型。

	特權的	用戶
當執行例外時	處理程式模式	處理程式模式
當執行主程式時	執行緒模式	執行緒模式

圖 2-4 Cortex-M3 裡的操作模式和權限等級

當處理器執行一個主程式時(在執行緒模式下)，它可能處於特權的狀態或用戶狀態，但執行例外處理程式時，一定是特權的狀態。處理器離開重置時，它將處於執行緒模式並擁有特權的存取權。處於特權的狀態時，程式可存取整個記憶體範圍(被 MPU 設定禁止的區域除外)，並可使用所有支援的指令。

在特權的存取等級裡的軟體，可藉著控制暫存器切換程式至用戶存取等級。當例外發生時，處理器通常會切換回特權的狀態，並在離開例外處理程式時，返回例外發生前的狀態。用戶程式不能藉著寫入控制暫存器，切換回特權的狀態。它需要經過一個例外處理程式去安排控制暫存器，在程式返回執行緒模式時，切換處理器回到特權的存取等級。

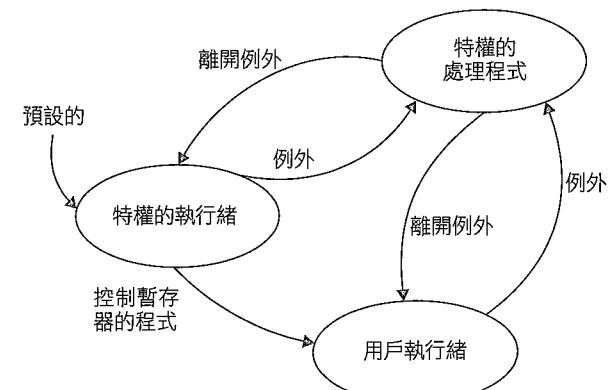


圖 2-5 允許的操作模式的轉換

藉由特權的與用戶等級的區隔，可防止系統組態暫存器受到不信任程式的存取或修改，增進系統的可靠性。如果有 MPU，也可利用它和特權的等級來保護重要的記憶體位置，例如作業系統的程式與資料。

舉例來說，在通常為 OS 核心使用的特權的存取狀態下，可以存取所有的記憶體位置(除了 MPU 設定禁止的位置)。當 OS 啟動應用程式時，通常是在用戶存取等級下執行，以保護系統不致因為不可信任的程式而崩潰(crash)，造成系統失敗。

## 內建的巢狀向量中斷控制器

Cortex-M3 處理器包含了一個中斷控制器，或叫做巢狀向量中斷控制器(Nested Vectored Interrupt Controller, NVIC)。它和處理器核心密切連結，並提供了一些功能：

- ◆ 巢狀中斷支援
- ◆ 向量中斷支援
- ◆ 動態優先權變動支援
- ◆ 減少中斷延遲
- ◆ 中斷遮罩

### 巢狀中斷支援

NVIC 提供了巢狀中斷支援。所有的外部中斷和大部分的系統例外，可以程式區分為不同的優先等級。當中斷發生時，NVIC 把此中斷優先權與正執行程式的優先等級作比較。如果新的中斷其優先權高於正執行程式的優先等級，則新的中斷的中斷處理程式將取代正執行中的工作而優先執行。

### 向量中斷支援

Cortex-M3 處理器有向量中斷的支援。當接到中斷後，會從記憶體的向量表，尋找中斷服務程式(interrupt service routine, ISR)的起始位址。由於並不需要使用軟體來做決定並跳躍至 ISR 的起始位址，因而可花用較少的時間去處理中斷要求。

### 動態優先權變動支援

中斷的優先等級可在執行時，藉由軟體改變。正使用的中斷在其中斷服務程式未完成前，不會再次被啟動。故改變其優先權，並不會導致意外地重新進入。

### 減少中斷延遲

Cortex-M3 處理器亦包含了一些進階的功能來減低中斷延遲。這些包括了自動儲存與復原一些暫存器的內容、減少 ISR 之間切換的延遲(參考第九章末尾連鎖(tail chaining)中斷)、和對晚到中斷的處理。

### 中斷遮罩

中斷與系統例外可藉著中斷遮罩暫存器 BASEPRI、PRIMASK、FAULTMASK 等全部遮罩，或根據優先等級加以遮罩，以保證時間緊要的工作，能不受中斷地如期完成。

### 記憶體映射

Cortex-M3 有著預先定義的記憶體映射。因此，可使用簡單的記憶體存取指令，去存取中斷控制器與除錯元件等內建的周邊設備。如此，大部分的系統功能可在 C 程式碼裡取用。預先定義的記憶體映射也允許高度最佳化 Cortex-M3 處理器，增進其速度，更易整合於系統單晶片(system-on-a-chip, SOC)設計中。

整體來看，4GB 的記憶體空間可劃分的範圍如圖 2-6 所示。

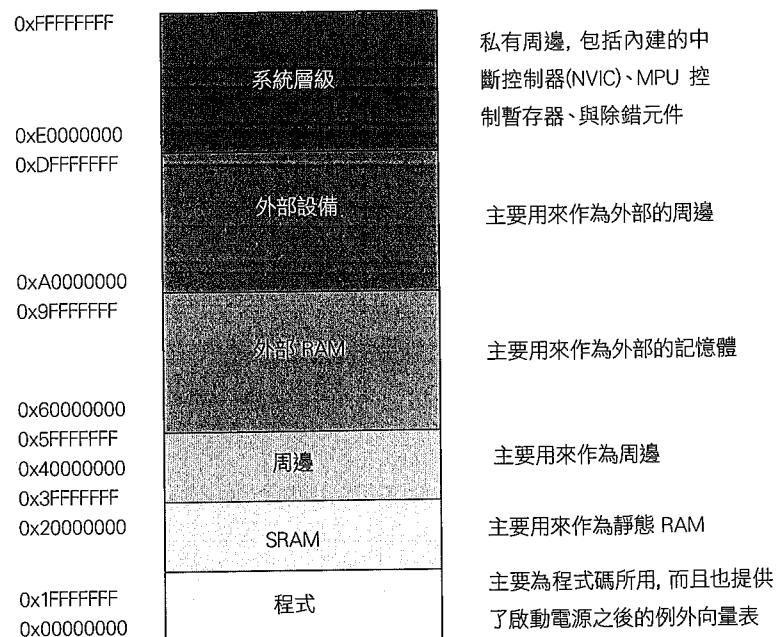


圖 2-6 Cortex-M3 記憶體映射

Cortex-M3 在設計上有一個針對此記憶體用法最佳化過的匯流排基礎架構。此外，其設計允許這些區域作不同方式使用。例如，資料記憶體可置於 CODE 區，程式碼也可在外部的 RAM 區執行。

系統層級的記憶體區域，包括了中斷控制器與除錯元件。這些元件有固定的位置，將於本書第五章記憶體系統詳加討論。固定了這些周邊設備的位址，使你更容易於不同的 Cortex-M3 間移植應用程式。

## 匯流排介面

Cortex-M3 處理器擁有幾個匯流排介面。它們允許 Cortex-M3 在同一時間擷取指令與存取資料。主要的匯流排介面為：

- ◆ 程式記憶體匯流排
- ◆ 系統匯流排
- ◆ 私有周邊匯流排

程式記憶體區的存取，是在程式記憶體匯流排上進行。程式記憶體匯流排實際上有兩個：一個稱作 I-Code，另一個稱作 D-Code。此兩者對指令的存取作最佳化，以得到最快的指令執行速度。

系統匯流排是用來存取記憶體與周邊。此匯流排提供對 SRAM、周邊設備、外部 RAM、外部設備、以及部分的系統層級記憶體區域等的存取。

私有周邊匯流排提供了存取部分的系統層級記憶體區域，此記憶體區域專供私有周邊(例如除錯元件)使用。

## 記憶體保護單元

Cortex-M3 有一個可選用的記憶體保護單元(Memory Protection Unit, MPU)。此單元允許設定特權的存取與用戶應用程式存取的規則。當違反存取規則時，將發出錯誤例外，且錯誤例外處理程式會分析問題並儘可能地改正。

MPU 可以有多種用途。在通常的情境下，MPU 為作業系統所設定，保護特權的程式(即作業系統核心)使用的資料，不致遭受不被信任的應用程式影響。MPU 也可用來限制記憶體區域為唯讀，避免意外地刪除資料，並可隔離多工系統不同工作的記憶體區域。總括來說，它使得嵌入式系統更強健可靠。

MPU 功能是選用的，可在處理器的實現階段或 SoC 設計時，決定是否選用。第十三章有更多關於 MPU 的資訊。

## 指令集

Cortex-M3 支援了 Thumb-2 指令集。此為 Cortex-M3 處理器重要的特性之一，因此允許同時使用 32-bit 指令與 16-bit 指令，以得到更高的程式碼密度與效率。它具有彈性、威力、且易於使用。

在先前的 ARM 處理器，其 CPU 有兩種運算狀態：32-bit ARM 狀態與 16-bit Thumb 狀態。在 ARM 狀態，指令為 32-bit 並能高效率地執行所有支援的指令。在 Thumb 狀態，指令為 16-bit，故其具較高的程式密度，但 Thumb 狀態並不具備所有的 ARM 指令功能，且需要以較多的指令去完成某些運算。

很多應用混和了 ARM 和 Thumb 程式碼，以活用此兩者最佳特性。然而，混和程式碼的安排並不一定會得到最好的效果。在兩個狀態之間切換必須付出額外代價(就執行時間與指令佔用空間而言)；且 ARM 程式與 Thumb 程式，可能需要於不同檔案裡分開編譯，這樣會增加軟體開發的複雜度，並降低了 CPU 核心的最高效率。

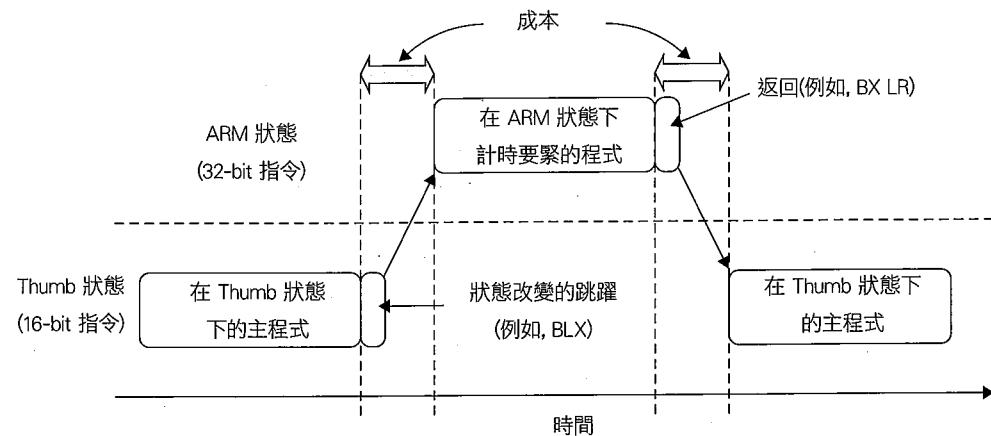


圖 2-7 傳統 ARM 處理器例如 ARM7 中 ARM 程式與 Thumb 程式的切換

藉由引入 Thumb-2 指令集，現在可於單一狀態下應付所有處理要求，而不必於兩個狀態間作切換。事實上，Cortex-M3 並不支援 ARM 程式碼，甚至，中斷亦以Thumb 狀態處理。(先前 ARM 核心是在 ARM 狀態下進入中斷處理器)。因其不必於狀態間切換，Cortex-M3 處理器比起傳統的 ARM 處理器具有以下優點：

- ◆ 沒有狀態間切換的額外代價，節省了執行時間與指令佔用空間
- ◆ 不需有分開的 ARM 程式與 Thumb 程式原始檔案，使得軟體開發與維護更為容易
- ◆ 更容易得到最佳效能與表現，相對地軟體的開發更容易，因為不再需要於 ARM 與 Thumb 之間切換，可以求得最佳的密度/表現

Cortex-M3 擁有一些有趣且具威力的指令，舉例如下：

- ◆ **UFBX、BFI、BFC**: bit field 萃取、插入、清除的指令
- ◆ **UDIV、SDIV**: 無號數與有號數除法指令

◆ **SEV、WFE、WFI**: Send-Event、Wait-For-Event、Wait-For-Interrupt；處理器可藉著這些指令，去處理多處理器系統的同步工作，或進入休眠狀態

◆ **MSR、MRS**: 用來存取特殊暫存器

因為 Cortex-M3 僅支援 Thumb-2 指令集，現存的 ARM 程式碼就需要移植到新的架構上。大部分的 C 應用程式僅需於支援 Cortex-M3 的新編譯器上重新編譯即可。一些組譯程式碼則需要加以修改與移植，才能利用新的架構與新的統一組譯器框架。

注意：並非 Thumb-2 指令集裡的所有指令都可以在 Cortex-M3 上使用。ARMv7-M Architecture Application Level Reference Manual (Ref 2) 僅要求實作Thumb-2 指令的子集合。例如，Cortex-M3 並不支援輔助處理器指令(可加裝外部資料處理器)，而 SIMD (Single Instruction, Multiple Data) 亦未在 Cortex-M3 中實作。此外，一些 Thumb 指令並未被支援，例如：以立即值作 BLX(將處理器從 Thumb 切換至 ARM 狀態)、幾個 CPS(Current Processor Status, 進行中處理器狀態)指令、architecture v6 引入的 SETEND(用以 set 和 clear 狀態 E-bit)指令。請參考本書附錄 A 或 Cortex-M3 技術參考手冊(Ref 1)，以取得完整的支援指令列表。

## 中斷與例外

Cortex-M3 處理器建置了在 ARMv7-M 架構裡所引進的新例外模式。此例外模式有別於傳統 ARM 的例外模式，可以高效率地處理例外。它包括了一些系統例外與外部 IRQs(外部中斷輸入)。Cortex-M3 中並無 FIQ(ARM7/9/10/11 的快速中斷)；然而，中斷優先權處理與巢狀中斷支援現在均已納入中斷架構中，因此，可容易地設立一個支援巢狀的中斷系統(高優先權中斷可以複寫或強占低優先權中斷)，其行為同於傳統 ARM 處理器的 FIQ。

在 Cortex-M3 裡中斷功能實現於 NVIC 中。除了支援外部中斷，Cortex-M3 也支援一些內部中斷源，例如系統錯誤處理。因此，Cortex-M3 有一些如表 2-2 所示，預先定義的例外種類。

表 2-2 Cortex-M3 的例外種類

例外號碼	例外類型	優先權(若可程式則預設為 0)	描述
0	NA	NA	沒有例外正在執行
1	重置	-3 (最高的)	重置
2	NMI	-2	不可遮罩中斷 (外部的 NMI 輸入)
3	硬錯誤	-1	所有錯誤的狀況，若相關的錯誤處理程式沒有被啟能
4	MemManage 錯誤	可程式的	記憶體管理錯誤；違反 MPU 或存取非法的位置
5	匯流排錯誤	可程式的	匯流排錯誤(預先擷取廢除或資料廢除)
6	用法錯誤	可程式的	因為程式錯誤造成的例外
7-10	保留的	NA	保留的
11	SVCALL	可程式的	系統服務呼叫
12	除錯監視器	可程式的	除錯監視器(中斷點、觀察點、或外部除錯要求)
13	保留的	NA	保留的
14	PendSV	可程式的	為了系統設備可置於等待的請求
15	SYSTICK	可程式的	系統滴答計時器
16	IRQ #0	可程式的	外部中斷 #0
17	IRQ #1	可程式的	外部中斷 #1
...	...	...	...
255	IRQ #239	可程式的	外部中斷 #239

外部中斷輸入的數目，由晶片製造廠定義。最多可支援 240 個外部中斷輸入。此外，Cortex-M3 也有一個 NMI(不可遮罩的中斷，Non-maskable Interrupt)中斷輸入。一旦被宣告，將無條件執行 NMI 中斷服務程式。

## 低功率與高電能效率

Cortex-M3 處理器的設計具有各種的特性，以允許設計工程師開發低功率與高電能效率的產品。首先，它具備睡眠模式與深度睡眠模式的支援，這些可以配合各樣的系統設計方法以減少在閒置期間的功率消耗。再者，它的低閾數與設計技術減低處理器內的

電路活動，因而允許了活動功率的減少。除此之外，因為 Cortex-M3 具有高的程式密度，故它減低了程式大小的要求；同時，它能讓處理的工作能在短期間完成，所以處理器可以儘速返回睡眠模式以減低能量的使用。其結果是：Cortex-M3 的電能效率比許多 8-bit 與 16-bit 微處理器還好。

此外，從 Cortex-M3 revision 2 開始有一個稱作喚醒中斷控制器(Wakeup Interrupt Controller, WIC)的新的特性。此特性允許調降整個處理器核心的功率，而在保留整個處理器狀態下，處理器可以在中斷發生時，幾乎立即地返回活動狀態。這使得 Cortex-M3 適合許多先前僅可使用 8-bit 或 16-bit 微控制器實作的超低功率之應用。

## 除錯支援

Cortex-M3 處理器包括了一些除錯功能，例如程式執行控制：暫停與步進、指令中斷點、資料觀察點、暫存器與記憶體存取、profiling、追蹤等等。

Cortex-M3 處理器的除錯硬體，採取 CoreSight 架構。異於傳統的 ARM 處理器，其 CPU 核心本身並無 JTAG(聯合工作測試組 Joint Task Action Group)介面。反之，有一個與核心分開的除錯介面模組，加上一個稱為即除錯存取埠(Debug Access Port, DAP)、在核心層級提供的匯流排介面。藉著這個匯流排介面，即使在處理器運行下，外部除錯程式也可存取控制暫存器以作硬體除錯，或者存取系統記憶體。此匯流排介面的控制，是由除錯埠(Debug Port, DP)裝置進行。目前可用的除錯埠裝置為 SW-DP(僅支援序列線協定，Serial Wire protocol)，也可使用 ARM CoreSight 產品系列的 JTAG-DP 模組。晶片製造商可選擇其中一個 DP 模組，以提供除錯介面。

晶片製造商也可以包含一個內嵌追蹤巨集格(Embedded Trace Macrocell, ETM)來作指令追蹤。追蹤的資訊可經由追蹤埠介面單元(Trace Port Interface Unit, TPIU)來作輸出。而除錯主機(通常為 PC)，則藉著外部追蹤-擷取硬體，來收集指令執行後的資訊。

在 Cortex-M3 處理器裡，有一些事件可觸發除錯的動作。包括：中斷點、觀察點、錯誤狀況、外部除錯要求輸入信號等等。當除錯事件發生時，Cortex-M3 可以進入暫停模式，或者執行除錯監視例外處理程式。

Cortex-M3 處理器裡的資料觀察與追蹤(Data Watchpoint and Trace, DWT)單元可提供資料觀察點的功能。此功能可停止處理器(或觸發除錯監視例外程式), 或者產生資料追蹤資訊。資料追蹤時, 則藉著 TPIU 輸出追蹤的資料。

除了這些基本的偵錯功能外, Cortex-M3 處理器也提供了快閃補丁與中斷點(Flash Patch and Breakpoint, FPB)單元。此單元提供簡單的中斷點功能, 或是將指令存取由快閃重新映射至 SRAM 裡不同的位置。

設備追蹤巨集格(Instrumentation Trace Macrocell, ITM)為程式開發者提供了輸出資料到除錯器的新途徑。藉著將資料寫入 ITM 暫存器的記憶體, 除錯器可透過追蹤介面蒐集資料, 加以顯示或處理。此方法比 JTAG 輸出更易於使用, 且更快。

所有這些除錯元件均可由 Cortex-M3 上的 DAP 介面匯流排、或處理器核心執行的程式控制, 並且所有的追蹤資訊, 可於 TPIU 存取。

## 特性總結

為何 Cortex-M3 處理器為一個革命性的產品？採用 Cortex-M3 的優點何在？其好處與優點總結如下：

### 高性能

- ◆ 許多指令, 包括乘法等, 為單一週期指令。Cortex-M3 比大部分微控制器產品, 表現更好。
- ◆ 分開的資料與指令匯流排, 允許同時進行資料與指令的存取。
- ◆ Thumb-2 指令集使得切換狀態的額外代價成為歷史, 不再需要花時間切換於 ARM 狀態(32-bit)與 Thumb 狀態(16-bit), 降低了指令週期與程式大小。此特性簡化了軟體開發、加速進入市場時間、且程式更易維護。
- ◆ Thumb-2 指令集在撰寫程式上, 提供了更多彈性。許多資料運算可使用更精簡的程式碼, 故 Cortex-M3 有更高的程式密度, 並降低了記憶體需求。

◆ 指令的擷取為 32-bit。於一個週期內最多可擷取兩個指令, 因此, 有更多頻寬可用在資料的傳遞。

◆ Cortex-M3 的設計允許微控制器產品運作在更高的時脈頻率(現代半導體製程可超過 100MHz)。即使以其它大多數微控制器產品相同的頻率運作, Cortex-M3 有著更好的時脈/指令(Clock per instruction, CPI)比率, 因此可得更好的工作量/MHz 比值, 或者其設計可在較低的時脈頻率執行, 以降低其電力消耗。

### 進階中斷處理功能

- ◆ 內建的巢狀向量中斷控制器(NVIC)最多可支援 240 個外部中斷輸入。因為不再需要使用軟體以決定欲服務的 IRQ 處理程式, 故向量中斷功能大為降低了中斷延遲。此外, 也不需要以軟體程式來設定巢狀的中斷支援。
- ◆ Cortex-M3 處理器於中斷進入點, 自動地將暫存器 R0-R3、R12、LR、PSR、PC 等等 push 進堆疊, 並在離開中斷時, 將其 pop 回來, 因此降低了 IRQ 處理延遲, 並允許以普通的 C 函數作為中斷處理程式(第八章將加以解釋)。
- ◆ 因為對任一中斷, NVIC 具可程式的中斷優先權控制, 故中斷的安排有很大的彈性。最低支援了八個等級、且可動態更改的優先權。
- ◆ 特殊的最佳化用來降低中斷延遲, 例如: 接受遲到中斷, 和末尾連鎖中斷進入點。
- ◆ 有些多週期的運算: 包括多重載入(Load-Multiple, LDM)、多重儲存(Store-Multiple, STM)、PUSHI-POP 等等, 現在可接受中斷。
- ◆ 在接收到不可遮罩(Nonmaskable Interrupt, NMI)要求時, 除非整個系統被完全鎖住, 否則一定會立即地執行 NMI 處理程式。NMI 對許多安全性緊要的應用程式而言, 非常重要。

### 低電力耗損

- ◆ 因為閘數少, 故 Cortex-M3 處理器適合用在低電力設計。

- ◆ 支援了省電模式(SLEEPING 和 SLEEPDEEP)。處理器可藉著等待中斷(Wait for Interrupt, WFI)指令, 或等待事件(Wait for Event, WFE)指令進入休眠模式。此設計對主要的區塊有獨立的時脈, 故處理器大部分的時脈電路在休眠時可被停止。
- ◆ 全穩定、同步、可合成的設計, 故易使用低功率或標準的半導體製程, 來製造處理器。

## 系統特性

- ◆ 系統提供了 bit-band 運算、byte-invariant big endian 模式、無對齊(unaligned)資料存取支援。
- ◆ 進階的錯誤處理功能, 包括許多的例外型態與錯誤狀態暫存器, 更易找到問題所在。
- ◆ 藉著隱藏的堆疊指標, 核心與應用程序的堆疊記憶體可以分開。藉著可選用的 MPU, 處理器有相當充分的能力, 以開發強韌的軟體與可靠的產品。

## 除錯的支援

- ◆ 支援 JTAG 與序列線除錯介面。
- ◆ 以 CoreSight 除錯方案為基礎; 可一邊執行核心, 一邊存取處理器狀態與記憶體內容。
- ◆ 內建支援了六個中斷點與四個觀察點。
- ◆ 可選用 ETM 作指令追蹤, DWT 作資料追蹤。
- ◆ 新的除錯功能, 包括錯誤狀態暫存器、新的錯誤例外、快閃補丁運算等, 使得除錯更為簡單。
- ◆ ITM 提供了易於使用的方式, 以從測試碼輸出除錯資訊。
- ◆ DWT 裡的 PC 取樣器與計數器提供了程式剖繪(profiling)的資訊。

# 3

Chapter

## Cortex-M3 基礎

本章內容包括：

- ✓ 暫存器
- ✓ 特殊暫存器
- ✓ 操作模式
- ✓ 例外與中斷
- ✓ 向量表
- ✓ 堆疊記憶體運算
- ✓ 重置順序

## 暫存器

如我們之前所見, Cortex-M3 處理器擁有暫存器 R0-R15, 以及一些特殊暫存器。其中 R0-R12 作為一般用途, 但是某些 16 位元的 Thumb 指令只能取用 R0-R7(低暫存器), 而 32 位元的 Thumb-2 指令可取用所有的暫存器(R0-R12)。特殊暫存器有預定的功能, 而且只能經由特殊的暫存器存取指令取用。

### 一般用途暫存器 R0-R7

一般用途暫存器 R0-R7 亦稱為低暫存器, 可為所有的 16 位元的 Thumb 指令和 32 位元的 Thumb-2 指令取用。這些暫存器皆是 32 位元, 其重置的值不可預測。

### 一般用途暫存器 R8-R12

一般用途暫存器 R8-R12 亦稱為高暫存器, 可為所有的 Thumb-2 指令取用, 但是

並非所有的 16 位元的 Thumb 指令都可以取用它們。這些暫存器皆是 32 位元，其重置的值不可預測。

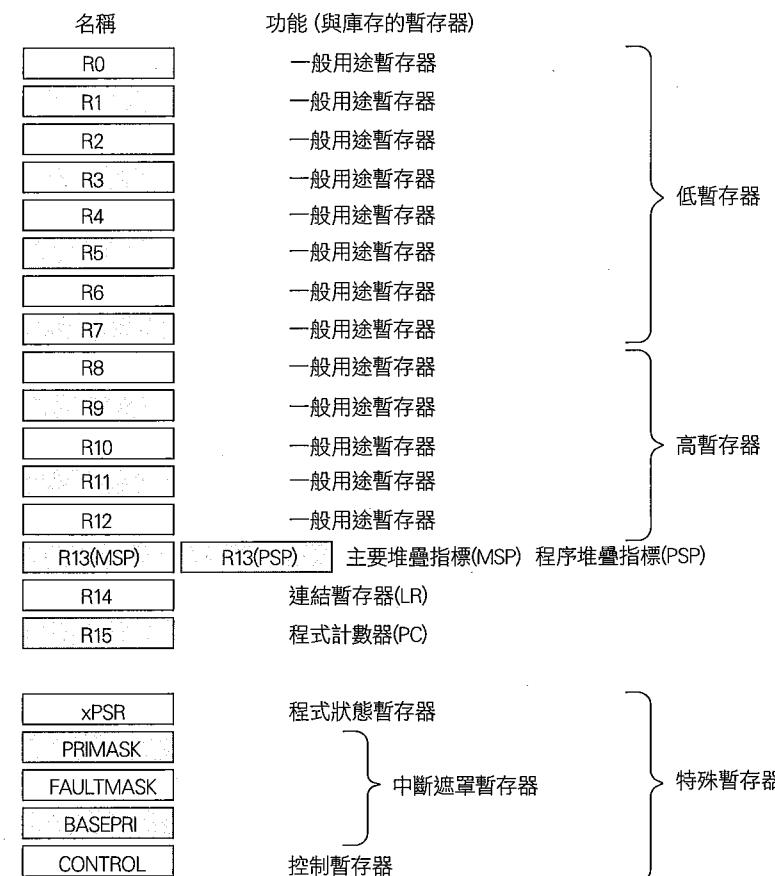


圖 3-1 在 Cortex-M3 裡的暫存器

## 堆疊指標暫存器 R13

R13 為堆疊指標。Cortex-M3 處理器擁有兩個堆疊指標，此雙重性(duality)可用來設定兩個分開的堆疊記憶體。使用暫存器 R13 的名稱時，只能存取當時作用的堆疊指標，除非使用特殊指令 MSR 或 MRS，否則無法存取另一個堆疊指標。兩個堆疊指標為：

● **主要堆疊指標(Main Stack Pointer, MSP)**，在 ARM 文件亦稱為 SP\_main：此為預設的堆疊指標；其使用者包括 OS 核心、例外處理程式(exception handlers)、和需要特權才能存取(privileged access)的應用程式碼。

● **程序堆疊指標(Process Stack Pointer, PSP)**，在 ARM 文件亦稱為 SP\_process：其為基層(base-level)應用程式碼使用(當不在進行例外處理時)。

使用到兩個堆疊指標並非必要，簡單的應用僅僅依靠 MSP 即可。堆疊指標是用在存取堆疊記憶體的處理上，例如 PUSH 和 POP。

3

### 區別堆疊 PUSH 和 POP

堆疊是一種記憶體使用模型。其僅是系統記憶體的一部份，利用(位於處理器內的)指標暫存器，將其作為先進後出(first-in/last-out)的緩衝器。在某些資料處理之前，堆疊通常用來存放暫存器的內容，並在處理的工作完成後，從堆疊將存放的內容取出，以恢復暫存器的內容。

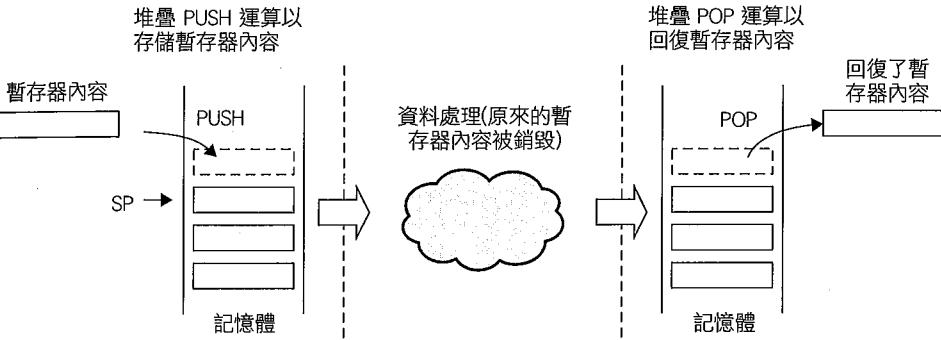


圖 3-2 堆疊記憶體的基本觀念

當執行 PUSH 和 POP 動作時，指標暫存器，通常稱作堆疊指標(Stack Pointer, SP)，會自動地調整內容，以避免先前堆入的資料被之後的堆入資料覆蓋。本章後面將提供更多堆疊運作的細節。

Cortex-M3 藉由 PUSH 和 POP 指令，存取堆疊記憶體。組合語言的語法如下(在分號以後的文字為註解)：

PUSH {R0}	; R13 = R13 - 4, then Memory[R13] = R0
POP {R0}	; R0 = Memory[R13], then R13 = R13+4

# Jason 嘴書—EETOP 世界唯一貼

Cortex-M3 使用全遞減(full-descending)的堆疊安排。(更多這方面的細節, 可在本章堆疊記憶體操作的段落找到。)所以, 當新的資料存進堆疊時, 堆疊指標將減少。PUSH 和 POP 指令通常是用在當副程式起始時將暫存器內容儲存於堆疊記憶體, 並在副程式結束時, 將其從堆疊取出並回復至暫存器。你可以用單一個指令, 將多個暫存器 PUSH 或 POP :

```
Subroutine_1
    PUSH {R0-R7, R12, R14}      ; 儲存暫存器值
    ...
    POP   {R0-R7, R12, R14}      ; 復原暫存器值
    BX    R14                   ; 回到呼叫程式
```

在程式碼中, SP(即堆疊指標 stack pointer)與 R13 可互換使用, 它們代表同一件事。在程式碼裡, MSP 和 PSP 兩者皆可叫做 R13/SP。然而, 你可藉由特殊暫存器存取指令(MRS/MSR), 去存取它們之中特定的一個。

MSP, 在 ARM 文件裡亦稱作 SP\_main, 為電源啟動後預設的堆疊指標, 使用於核心程式碼以及例外處理。PSP, 在 ARM 文件裡亦稱作 SP\_process, 主要使用於執行緒程序。

因為暫存器 PUSH 和 POP 運算以 word 為單位對齊(其位址應該是 0x0, 0x4, 0x8, ...), 堆疊指標 R13 的第 0 bit 和第 1 bit 以硬體設定為 0, 而且其讀值永遠為 0 (RAZ)。

## 連結暫存器 R14

R14 為連結暫存器(Link Register, LR)。在組合語言程式中, 你可將其寫作 R14 或 LR。LR 用以存放叫用副程式或函數時, 返回的程式計數器值(return program counter), 例如, 當你使用 BL(branch and link, 跳躍並連結)指令時:

```
main      :     主程式
...
BL      function1      ; 以 branch with link 指令
          ; 呼叫 function1
          ; PC = function1 並且
```

接下頁

```
...
function1
...
BX      LR
```

; LR = 主程式的下一個指令  
; function1 的程式碼  
; return

雖然實際上程式計數器 bit 0 通常為 0(因為指令以 word 或者 half word 單位對齊), LR 的 bit 0 是可讀可寫的。原因在於 Thumb 指令集, bit 0 常常用來顯示 ARM/Thumb 的狀態。為了讓 Cortex-M3 上的 Thumb-2 程式, 能運作在其他支援 Thumb-2 指令集的 ARM 處理器上, 此 LSB(即 bit 0)是可讀可寫的。

3

## 程式計數器 R15

R15 為程式計數器。在組合語言程式中, 你可以透過 R15 或 PC 來取用它。因為 Cortex-M3 處理器管線(pipelined)的性質, 當你讀此暫存器, 將發現其值與正執行指令位置相差為 4。例如:

```
0x1000 : MOV R0, PC ; R0 = 0x1004
```

把值寫入程式記數器將造成跳躍(branch)(但連結暫存器值並未因此更新)。因為指令的位址應以 half word 排列, 程式計數器的 LSB(即 bit 0)的讀值, 通常為 0。然而, 不論是藉由寫入 PC 或使用分歧指令跳躍, 目標(target)位址的 LSB 應設定為 1, 以表示是在 Thumb 狀態中運作。如果其值為 0, 則意味著正欲切換至 ARM 狀態, 這會在 Cortex-M3 導致錯誤例外。

## 特殊暫存器

Cortex-M3 的特殊暫存器包括如下:

- ◆ 程式狀態暫存器(Program Status Register, 即 PSRs)
- ◆ 中斷遮罩暫存器(Interrupt Mask Register, 即 PRIMASK, FAULTMASK, 和 BASEPRI)

### ◆ 控制暫存器(Control Register, 即 Control)

特殊暫存器僅能藉由 MSR 和 MRS 指令來存取；它們沒有記憶體位址：

MRS <reg>, <special_reg>	; 讀特殊暫存器值
MSR <special_reg>, <reg>	; 寫入特殊暫存器

你可以使用 MRS 指令去讀取程式狀態暫存器，亦可以使用 MSR 指令去改變 APSR 值，但 EPSR 和 IPSR 則是唯讀的。例如：

MRS r0, APSR	; 讀取旗標狀態給 R0
MRS r0, IPSR	; 讀取例外/中斷狀態
MRS r0, EPSR	; 讀取執行狀態
MSR APSR, r0	; 寫入旗標狀態

## 程式狀態暫存器(Program Status Register, PSRs)

程式狀態暫存器細分為 3 個狀態暫存器：

### ◆ 應用 PSR(Application PSR, 即 APSR)<sup>1</sup>

### ◆ 中斷 PSR(Interrupt PSR, 即 IPSR)

### ◆ 執行 PSR(Execution PSR, 即 EPSR)

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR																例外號碼
EPSR						IC/I/T	T			IC/I/T						

圖 3-3 在 Cortex-M3 裡的程式狀態暫存器(PSR)

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	IC/I/T	T			IC/I/T						例外號碼

圖 3-4 在 Cortex-M3 裡結合的程式狀態暫存器(xPSR)

在 ARM 組譯器，可用 PSR 符號去存取 xPSR(3 個 PSR 集合成一項使用)：

MRS r0, PSR	; 讀取複合的程式狀態 word
MSR PSR, r0	; 寫入複合的程式狀態 word

PSR 各個位元的描述，參照表 3-1。

表 3-1 在 Cortex-M3 程式狀態暫存器裡的位元欄位

位元	描述
N	負
Z	零
C	進位/借位
V	溢位
Q	黏性的飽和旗標
IC/I/T	中斷-可連續指令(Interrupt-Continuable Instruction, IC)位元, IF-THEN 指令狀態位元
T	Thumb 狀態，通常為 1；試圖清除此位元將會造成錯誤的例外
例外號碼	顯示處理器正處理的例外

如果你將此 PSR 與 ARM7 的進行中程式狀態暫存器 CPSR(Current Program Status Register)互相比較，會發覺某些 ARM7 有的位元不見了。模式(Mode)M 位元被刪除了，因為 Cortex-M3 並無此定義在 ARM7 的操作模式。Thumb-bit(T)移至 bit 24。中斷狀態(I 和 F)位元則由與 PSR 分開的新的中斷遮罩暫存器(PRIMASKs)所取代。為了比較起見，圖 3.5 展示了傳統 ARM 處理器中的 CPSR。

1 在 Cortex-M3 開發的初期，APSR 被稱作旗標程式狀態 Word (Flags Program Status Word, FPSR)，故一些開發工具的早期版本可能使用 FPSR 的用詞以取代 APSR。可藉由特殊暫存器存取指令 MSR 和 MRS，一起或分別地存取此 3 個 PSR。當他們被集合成一項使用時，則總稱之為 xPSR。

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
ARM	N	Z	C	V	Q	IT	J	Resrv	GE[3:0]	IT	E	A	I	F	T	M[4:0]

圖 3-5 在傳統的 ARM 處理器裡的現行程式狀態暫存器(CPSR)

## PRIMASK, FAULTMASK, 和 BASEPRI 暫存器

PRIMASK, FAULTMASK, 和 BASEPRI 等暫存器, 是用來抑止各種例外(參考表 3-2)。

表 3-2 Cortex-M3 中斷遮罩暫存器

暫存器名稱	描述
PRIMASK	一個 1-bit 暫存器。設定此則允許 NMI 與硬錯誤例外；其他的中斷與例外皆被遮罩。預設值為 0, 其表示並無設定遮罩。
FAULTMASK	一個 1-bit 暫存器。設定此則僅允許 NMI；並且其他的中斷與錯誤處理例外皆被除能。預設值為 0, 其表示並無設定遮罩。
BASEPRI	一個最多至 9 bits 的暫存器(視為優先權等級實作的位元欄位而定)。它定義了遮罩的優先權等級。設定此則除能了所有相同的或更低等級的(較大的優先權數值)中斷。但依然可以允許更高優先權的中斷。若此被設定為 0, 其表示遮罩功能被除能(此為預設值)。

在時間相關緊要的工作中, PRIMASK 和 BASEPRI 暫存器可用來暫時抑止中斷。當工作當掉(crash)的時候, OS(作業系統)可使用 FAULTMASK 來暫時抑止錯誤處理。在這種情況下, 可能會在工作當掉時產生一堆錯誤, 而當核心開始清理時, 可能不希望被當掉程序造成的其他錯誤影響而中斷清理工作。此時, FAULTMASK 就能夠給予 OS 核心足夠的時間去處理錯誤。

MRS 和 MSR 指令可用來存取 PRIMASK、FAULTMASK、以及 BASEPRI 暫存器。例如：

```

MRS r0, BASEPRI           ; 將 BASEPRI 暫存器內容讀進 r0
MRS r0, PRIMASK            ; 將 PRIMASK 暫存器內容讀進 r0
MRS r0, FAULTMASK          ; 將 FAULTMASK 暫存器內容讀進 r0
MSR BASEPRI, r0             ; 將 r0 內容寫入 BASEPRI 暫存器
MSR PRIMASK, r0             ; 將 r0 內容寫入 PRIMASK 暫存器
MSR FAULTMASK, r0           ; 將 r0 內容寫入 FAULTMASK 暫存器

```

PRIMASK、FAULTMASK、以及 BASEPRI 不能在用戶存取等級下設定。

## 控制暫存器

控制暫存器用來定義權限的級別和堆疊指標的選項。如表 3-3 所示, 此暫存器有兩位元。

表 3-3 Cortex-M3 CONTROL 暫存器

位元	功能
CONTROL[1]	<p>堆疊狀態： 1 = 使用了替代的堆疊 0 = 使用了預設的堆疊(MSP)</p> <p>如果它在執行緒或基底等級, 其替代的堆疊為 PSP。處理程式模式並無替代的堆疊, 故在處理器處於處理程式模式時此位元必須為 0。</p>
CONTROL[0]	<p>0 = 在執行緒模式之特權的 1 = 在執行緒模式之用戶狀態</p> <p>如果在處理程式模式(而非執行緒模式), 處理器運算於特權的模式。</p>

## CONTROL[1]

Cortex-M3 在處理程式模式時, 其 CONTROL[1] bit 通常為 0。而在執行緒或 base 等級時, 則可為 0 或 1。

要寫入此位元, 核心需作用於執行緒模式且為特權的。在用戶狀態以及處理程式模式下, 不允許寫入此位元。除了寫入此暫存器, 還有另一個改變此位元的方式：就是從例外返回時, 更改 LR 的 bit 2。此方式將於第八章敘述例外細節時加以討論。

## CONTROL[0]

CONTROL[0]位元僅可在特權的狀態寫入。一旦進入用戶狀態, 唯一返回特權的狀態的方法, 則需要觸發中斷並於例外處理程式下作改變。

MRS 和 MSR 指令可用來存取控制暫存器：

```

MRS r0, CONTROL           ; 將控制暫存器值讀入 r0
MSR CONTROL, r0             ; 將 r0 值寫入控制暫存器

```

## 操作模式

Cortex-M3 處理器支援兩種模式(modes)和兩種權限等級(privilege levels)。

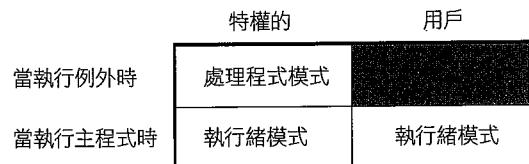


圖 3-6 在 Cortex-M3 裡的運算模式與特權的等級

處理器運作於執行緒模式時，它可以是在特權的(privileged)或用戶(user)的等級；但運作於處理程式模式時，僅能是特權的等級。處理器離開重置時，它將處於執行緒模式並擁有特權的存取權限。

在用戶存取級別(執行緒模式)下，不能存取系統控制空間(System Control Space, 或 SCS)，SCS 為記憶體區的一部分，用以組態暫存器(configuration registers)和除錯元件。此時，也不可使用存取特殊暫存器的指令(例如 MSR，但存取 APSR 除外)。如果運作在用戶存取級別的程式試著存取 SCS 或特殊暫存器，就會造成錯誤例外。

特權存取等級下的軟體，可藉由控制暫存器，將程式轉換至用戶存取等級。當例外產生時，處理器通常會轉換至特權的狀態，並在離開例外處理程式時，回到先前的狀態。用戶程式不可藉由控制暫存器直接切換到特權的狀態，必須經由例外處理程式去安排控制暫存器，並在返回執行緒 模式時，將處理器切換至特權的存取等級。

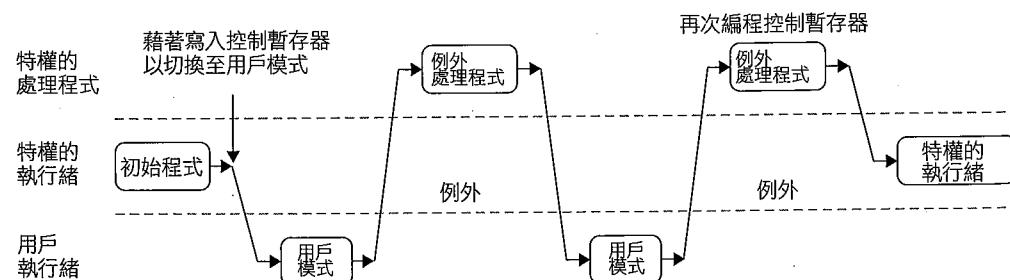


圖 3-7 藉由設定控制暫存器或例外以切換運算模式

特權的與用戶存取等級的支援，提供了更安全與強健的結構。例如，即使用戶程式出錯，也不會破壞 NVIC 中的控制暫存器。除此之外，如果 MPU 存在，則可阻擋用戶程式去存取特權的程序使用的記憶體區。

為了避免因用戶程式裡堆疊使用錯誤，造成系統當機的可能，你可將用戶應用堆疊與核心堆疊記憶體分開。在此安排下，用戶程式(運作於執行緒模式)使用 PSP，而例外處理程式使用 MSP。在進入或離開例外處理程式時，會主動地切換堆疊指標。本書第 8 章將更詳細地討論此主題。

控制暫存器定義了處理器的模式與存取等級。當控制暫存器 bit 0 為 0 時，處理器會在例外發生時改變模式。

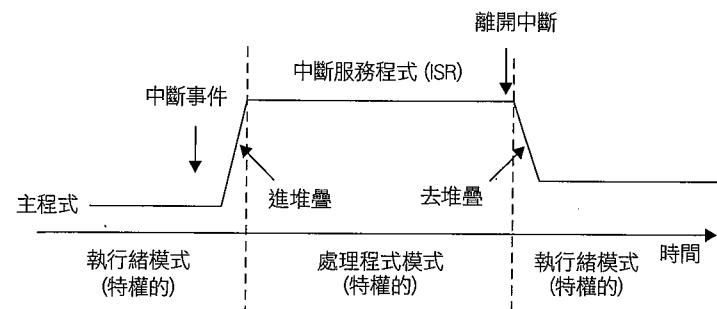


圖 3-8 在中斷時切換處理器模式

當控制暫存器 bit 0 為 1(執行緒執行用戶應用程式)時，處理器的模式和存取等級皆會於例外發生時改變。

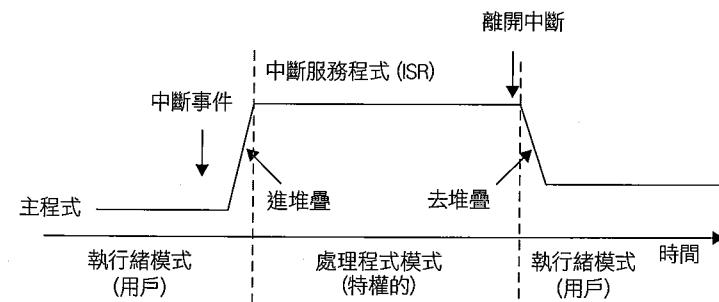


圖 3-9 在中斷時切換處理器模式與特權等級

唯有處於特權的等級時，才能藉由程式去更動控制暫存器的 bit 0。如果用戶等級程式要切換至特權的狀態，就必須引發中斷(例如，SVC，亦即系統服務呼叫 System Service Call)，並在處理程式裡寫入 CONTROL[0]。

## 例外與中斷

Cortex-M3 支援例外處理，包括固定數量的系統例外和一些通稱為 IRQ 的中斷。Cortex-M3 微控制器的中斷輸入之數量，隨著各自的設計而不同。除了系統計時器 (System Tick Timer)之外，周邊設備發出的中斷，亦連結至中斷輸入信號。典型中斷輸入的數量為 16 或 32。然而，你也可能找到擁有更多或更少中斷輸入的微控制器。

在中斷輸入外，還有稱為 NMI 的輸入信號。NMI 的實際用途，隨著你使用的微控制器或 SoC 產品而定。多數情形下，NMI 可連結到 watchdog 監時器，或電壓監視區塊，其功能為當電壓下降至某準位以下時，對處理器發出警告。可隨時、甚至緊接核心離開重置的瞬間，啟動 NMI 信號。

表 3-4 為 Cortex-M3 的例外處理列表。某些系統例外是隨各類錯誤情形觸發的錯誤處理例外。NVIC 亦提供一些錯誤狀態暫存器，供錯誤處理程式判斷例外的原因。

第 7-9 章會討論更多 Cortex-M3 的例外操作的細節。

表 3-4 在 Cortex-M3 裡的例外類型

例外號碼	例外類型	優先權	功能
1	重置	-3(最高的)	重置
2	NMI	-2	不可遮罩的中斷
3	硬錯誤	-1	所有種類的錯誤，當相關的錯誤處理程式，因為正在被除能或者受到例外遮罩而被遮罩，而不能被啟動時
4	MemManage	可設定的	記憶體管理錯誤；因為 MPU 違規或不合法存取而造成(例如由不可執行的區域作指令擷取)
5	匯流排錯誤	可設定的	從匯流排系統接收到的錯誤反應；因為指令預先擷取的失敗或資料存取錯誤而造成
6	用法錯誤	可設定的	用法錯誤；典型的造成原因是不合法的指令或是不合法的狀態轉移嘗試(例如在 Cortex-M3 裡試圖切換到 ARM 狀態)
7-10			保留的

例外號碼	例外類型	優先權	功能
11	SVC	可設定的	經由 SVC 指令的系統服務呼叫
12	除錯監視器	可設定的	除錯監視器
13			保留的
14	PendSV	可設定的	為了系統服務可置於等待的要求
15	SYSTICK	可設定的	系統滴答計時器
16-255	IRQ	可設定的	IRQ 輸入 #0-239

## 向量表

當例外事件在 Cortex-M3 中產生，並被處理器核心接受後，相對應的例外處理程式將被執行。例外處理程式的起始位址是藉由向量表的機制來決定。向量表為 word 資料陣列，每一 word 代表每一例外類型的起始位址。向量表是可重新定位的 (relocatable)，其移動位置由 NVIC 裡的移動位置暫存器來控制。

重置之後，此移動位置暫存器被重置為 0；所以，在重置之後，向量表位於位址 0x0。

表 3-5 重置之後的向量表定義

例外類型	位址位移值	例外向量
18-255	0x48-0x3FF	IRQ #2-239
17	0x44	IRQ #1
16	0x40	IRQ #0
15	0x3c	SYSTICK
14	0x38	PendSV
13	0x34	保留的
12	0x30	除錯監視器
11	0x2c	SVC
7-10	0x1c-0x28	保留的
6	0x18	用法錯誤
5	0x14	匯流排錯誤
4	0x10	MemManage 錯誤
3	0x0c	硬錯誤
2	0x08	NMI
1	0x04	重置
0	0x00	MSP 的起始值

例如，如果重置(reset)是 type 1 例外，則重置向量的位置是 1 乘以 4(每一 word 有 4 bytes)，亦即等於 0x00000004。而 NMI 向量(type 2)則位於  $2 \times 4 = 0x00000008$ 。位址 0x00000000 則是儲存 MSP 的初始值。

每一個例外向量的最低位元 LSB 顯示此例外是否在 Thumb 狀態下執行。因為 Cortex-M3 僅支援 Thumb 指令，故所有的例外向量的 LSB 應該設定為 1。

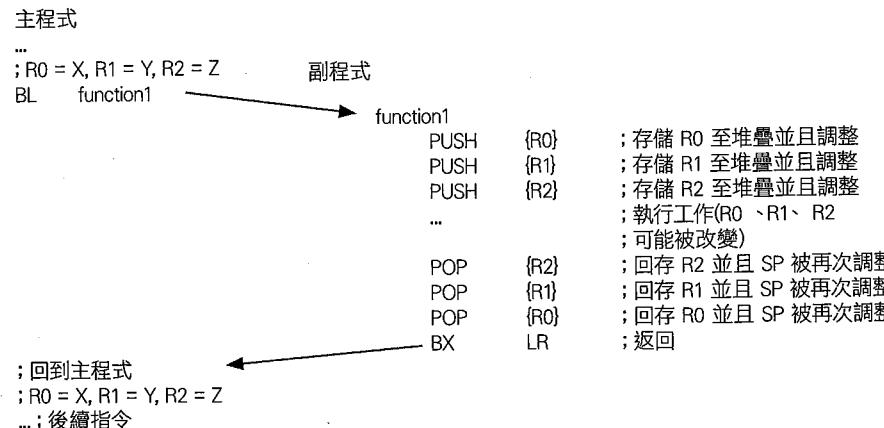
## 堆疊記憶體運算

在 Cortex-M3 裡，除了正常的軟體控制的堆疊 PUSH 和 POP，堆疊 PUSH 和 POP 運算也會在進入或離開例外/中斷處理程式時被自動執行。此節，我們將探討軟體堆疊的運算。

### 堆疊基本運算

大致上，堆疊運算為記憶體的讀或寫的運算，其位址由堆疊指標(SP)所決定。在暫存器裡的資料，藉著 PUSH 運算儲存至堆疊記憶體，之後，藉著 POP 運算以回復至暫存器。堆疊指標隨著 PUSH 和 POP 自動調整，故執行多次的資料 PUSH，也不致於造成舊的堆疊資料被清除。

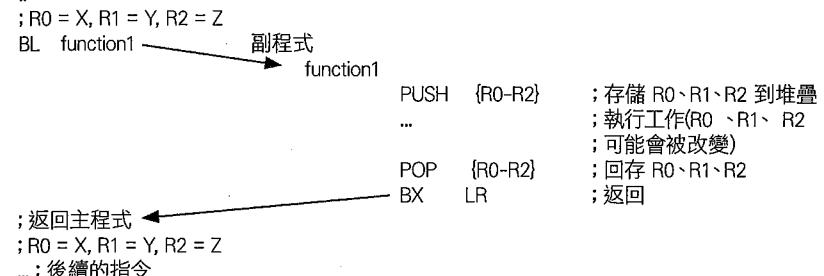
堆疊的功能為儲存暫存器的內容，以使得這些內容能於處理工作完成後，順利回復。正常使用下，每一 store(即 PUSH)必伴隨著一 read(即 POP)，而且 POP 運算的位址，必與 PUSH 運算吻合(參照圖 3.10)。隨著 PUSH/POP 指令，堆疊指標將主動遞增/遞減。



當程式控制回到主程式時，r0-r2 內容與之前相同。注意 PUSH 和 POP 的次序：POP 的次序必與 PUSH 次序相反。

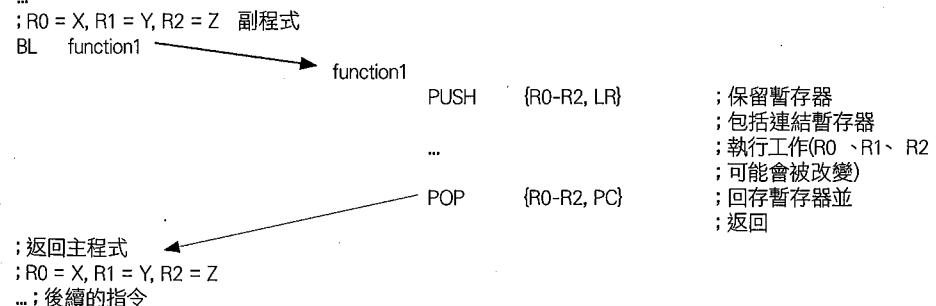
因為 PUSH 和 POP 允許多重載入與儲存，故上面的運算可更為簡化。在這個例子中，處理器主動地將暫存器 POP 的次序加以相反(參照圖 3.11)。

主程式



你亦可將 RETURN 與 POP 運算結合。其做法是把 LR 值 PUSH 到堆疊，並在副程式結束時，POP 回 PC(參照圖 3.12)。

主程式



## Cortex-M3 堆疊實現

Cortex-M3 採取全遞減的堆疊運算模型，堆疊指標 SP 指向最後被 PUSH 進堆疊記憶體的資料，並在新的 PUSH 運算前，SP 先減少。圖 3-13 為執行 PUSH {R0} 指令的例子。

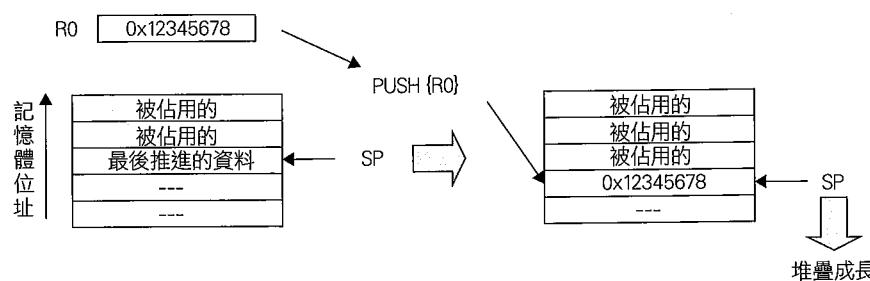


圖 3-13 Cortex-M3 堆疊 PUSH 的實現

POP 如下運算：先把 SP 指向的記憶體位置的資料讀出，再增加 SP。記憶體的內容維持不變，但會被下一個 PUSH 運算所覆蓋(參照圖 3-14)。

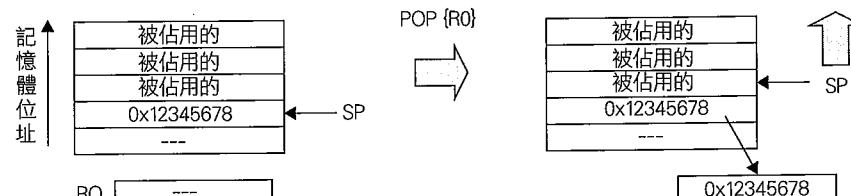


圖 3-14 Cortex-M3 堆疊 POP 的實現

因為每一次 PUSH/POP 運算，移動了 4 bytes 資料(每一暫存器包含 1 word = 4 bytes)，SP 每一次增加/減少 4 或者 4 的倍數(如果多個暫存器被 push 或 pop)。

在 Cortex-M3 裡，R13 被定義為 SP。當中斷發生時，會自動 push 一些暫存器，R13 將用作此次堆疊程序的 SP。同樣地，離開中斷處理程式時，被 push 的暫存器將主動地被 restored/popped，而 SP 亦會被調整。

## Cortex-M3 雙堆疊模式

如前所述，Cortex-M3 擁有兩個堆疊指標：主要堆疊指標(Main Stack Pointer, MSP)和程序堆疊指標(Process Stack Pointer, PSP)。控制暫存器的 bit 1, CONTROL[1]操控了 SP 暫存器的選用。

當 CONTROL[1]為 0 時，執行緒模式與處理程式模式都是使用 MSP。在這樣安排下，主程式和例外處理程式合用相同的堆疊記憶體區。此為電源開啟後的預設值。

當 CONTROL[1]為 1 時，執行緒模式下會改用 PSP。如此安排下，主程式和例外處理程式可以有分開的堆疊記憶體區。這樣做可以避免用戶應用程式的堆疊錯誤，破壞了 OS 使用的堆疊(假設用戶應用程式僅操作於執行緒模式，而 OS 核心執行於處理程式模式)。

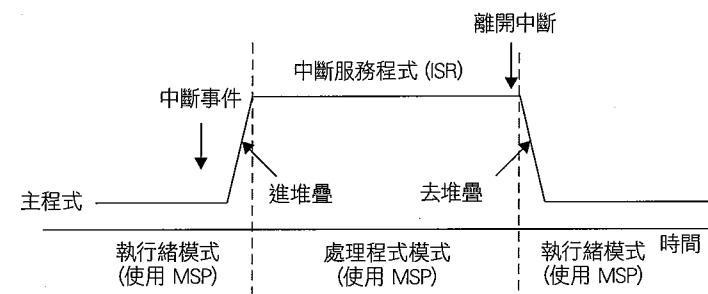


圖 3-15 Control[1]=0: 執行緒等級與處理程式兩者皆使用主要堆疊

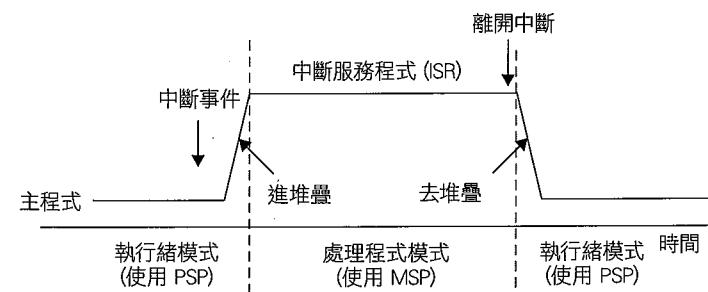


圖 3-16 Control[1]=1: 執行緒等級使用程序堆疊而處理程式使用主要堆疊

注意在此情形下，自動堆疊與還原堆疊的機制將使用 PSP，而在處理程式內的堆疊運算將使用 MSP。

你也可以直接對 MSP 和 PSP 執行讀/寫運算，避免混淆 R13 到底指哪一個的問題。假設你正處於特權的等級，則可以 MRS 和 MSR 指令去存取 MSP 和 PSP：

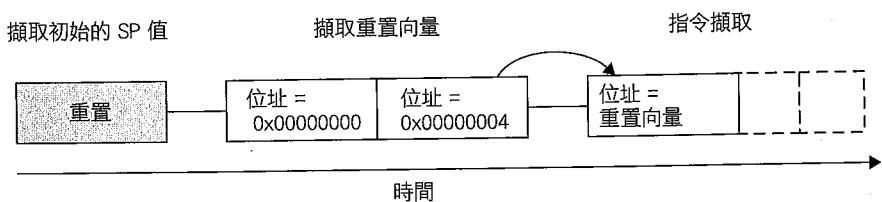
MRS R0, MSP	; 將主要堆疊指標的值，讀進 R0
MSR MSP, R0	; 將 R0 的值，寫入主要堆疊指標
MRS R0, PSP	; 將程序堆疊指標的值，讀進 R0
MSR PSP, R0	; 將 R0 的值，寫入程序堆疊指標

藉著使用 MRS 指令讀取 PSP 值，OS 即可讀進用戶應用程式堆疊的資料(例如在系統服務呼叫 SVC 之前暫存器的內容)。此外，OS 可以在像是多工系統內容切換時，更改 PSP 指標的值。

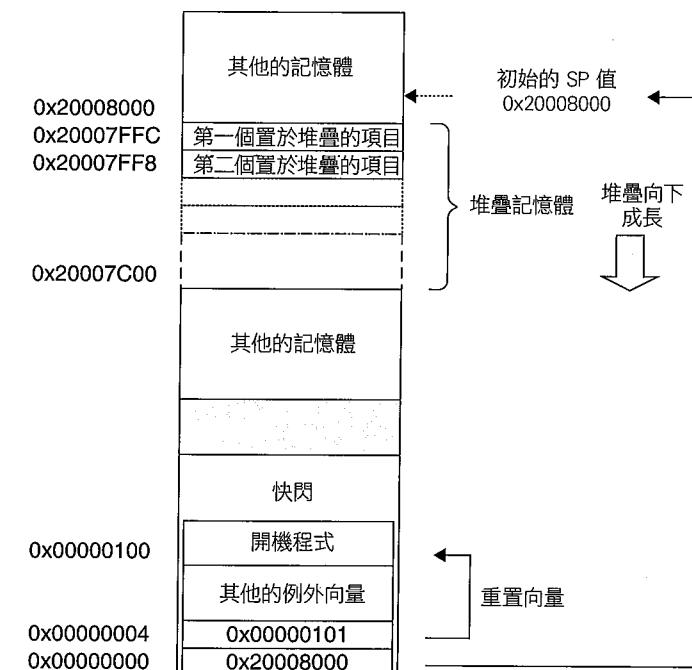
## 重置順序

處理器離開重置後，會從記憶體讀入兩個 word 資料：

- ◆ 位址 0x00000000：堆疊指標 R13 的起始值
- ◆ 位址 0x00000004：重置向量(程式執行的起始位址；其 LSB 應設為 0 以顯示為 Thumb 狀態)



這不同於傳統 ARM 處理器的行為。先前的 ARM 處理器是從位址 0x0 開始執行程式碼。更且，先前 ARM 元件的向量表為指令(你需要在那裡放置跳躍指令，才能將例外處理程式放在其它位置)。在 Cortex-M3 裡，MSP 的初始值放在記憶體映射(memory map)的起始位置，向量表緊接其後，其內含為向量位址值。(之後當程式執行時，可以將向量表重新移動至新的位置)。此外，向量表的內容為位址值，而非跳躍指令。向量表的第一個項目(例外 type 1)為重置向量(reset vector)，其為處理器重置之後，抓取的第二塊資料。



既然 Cortex-M3 裡的堆疊運算為全遞減堆疊(先減少堆疊指標，再存放資料)，初始堆疊指標值，應設定在堆疊區頂端之後的第一個記憶體處。例如，你的堆疊記憶體位於 0x20007C00 至 0x20007FFF(1 K bytes)，則其初始的堆疊指標值，應設定為 0x20008000。

向量表緊接在初始的 SP 值後面，第一個向量為重置向量。注意在 Cortex-M3 裡，向量表中的向量位址，需將其 LSB 設定為 1，以顯示其為 Thumb 程式碼。因此，在圖 3-18 的例子裡，重置向量值雖然為 0x101，但 boot 程式碼卻起始於位址 0x100。抓取了重置向量之後，接著 Cortex-M3 從重置向量位址開始執行程式，並進行正常的運算。堆疊指標的初值化是必要的動作，因為某些例外(例如 NMI)可能會在重置後立刻出現，而這些例外的處理程式可能需要用到堆疊記憶體。

不同的軟體開發工具，可能會以不同方式，去指定堆疊指標的起始值和重置向量。如果你需要更多這方面的資訊，最好在開發工具提供的專案例子裡尋找。本書提供了一些簡單的例子，在第 10 章和第 20 章是 ARM 工具，第 19 章則是 GNU 工具鏈。

# 4

## Chapter

# 指令集

本章內容包括：

- ✓ 組合語言基礎
- ✓ 指令列表
- ✓ 指令描述
- ✓ Cortex-M3 裡一些有用的指令

本章對 Cortex-M3 的指令集，和一些指令的例子，提供了一些洞察。你亦可在本書附錄 A 找到支援指令的快速參考。欲了解每一指令的完整細節，可參考 ARM v7-M Architecture Application Level Reference Manual(ARM v7-M 架構應用層參考手冊) (Ref 2)。

## 組合語言基礎

在此我們介紹一些 ARM 組合語言的基本語法，以期能更容易地了解本書接下來的程式碼範例。本書大部分組合語言程式的範例，以 ARM 組譯器為根據，但在第 19 章的例子則例外，因為該章的焦點為 GNU 工具鏈。

## 組譯器語言：基本語法

組合語言程式裡，通常使用如下的指令格式：

```
label
opcode operand1, operand2, ... ; 註解
```

# Jason 嘴書—EETOP 世界唯一貼

標籤(label)可有可無。在指令之前加上了標籤，則可藉著標籤決定指令的位址。接下來你會看到 opcode(指令)，並有幾個運算元(operand)跟隨其後。通常第一個運算元為此運算的目的地。運算元的個數視指令種類而定，運算元亦擁有不同的語法格式。例如，立即值(immediate)資料有著如下#number 的型態：

```
MOV R0, #0x12 ; 設定 R0 = 0x12 (16 進位)
MOV R1, #'A' ; 設定 R1= A 字母的 ASCII 值
```

每一個分號(;)後的文字為註解。註解會使得程式更易了解，但不會影響程式運算的結果。

你可以使用 EQU 定義一些常數，再於程式中使用它們。例如：

```
NVIC_IRQ_SETEN0 EQU 0xE000E100
NVIC IRQ0_ENABLE EQU 0x1
...
LDR R0, = NVIC_IRQ_SETEN0 ; 在此 LDR 為虛指令,
; 組譯器會將之轉成
; 以 PC 為相對位址的載入
MOV R1, # NVIC IRQ0_ENABLE ; 將立即值資料移進暫存器
STR R1, [R0] ; 將 R1 值寫入 R0 內的位址處
; 以致能 IRQ
```

當組譯器無法產生你需要的正確指令，而你也知道此指令的二進位碼時，則可使用 DCI 來寫此指令：

```
DCI 0xBE00 ; Breakpoint (BKPT 0), 一個 16 進位的指令
```

我們可以使用 DCB(大小為 1 byte 的常數，例如字元 character)和 DCD(大小為 1 word 的常數)來定義程式裡的 2 進位資料：

```
LDR R3, =MY_NUMBER ; 取得 MY_NUMBER 的記憶體位址值
LDR R4, [R3] ; 將數值 0x12345678 載入 R4
...
LDR R0, =HELLO_TXT ; 取得 HELLO_TXT 記憶體位址
; 之起始值
BL PrintText ; 呼叫 PrintText 函式
; 以顯示字串
```

接下頁

MY_NUMBER	
DCD	0x12345678
HELLOW_TXT	
DCB	"Hello\n", 0 ; null 結尾的字串

注意：組譯器語法隨你所使用的組譯器工具而定。在此介紹者為 ARM 組譯器工具的語法。要了解其它組譯器的語法，最好從那些工具提供的程式例子下手。

## 組譯器語言：使用後綴字

ARM 處理器的組譯器，如表 4-1 所示，可在指令之後加上後綴字。

表 4-1 指令裡的後綴字

後綴字	描述
S	更新 APSR(旗標)；例如： ADDS R0, R1 ; 此將更新 APSR
EQ, NE, LT, GT 等等	條件式執行；EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, 等等。例如： BEQ <Label> ; 若相等則跳躍

在 Cortex-M3 裡，條件執行後綴字，常用於跳躍的指令。然而，在 IF-THEN 指令區塊內，其他非跳躍指令亦可與條件執行後綴字一起使用。(此觀念將於本章後面部分介紹)。在這些情形下，後綴字 S 和條件執行後綴字可以同時使用。本章後面部分將提到，共有 15 種條件選擇可供使用。

## 組譯器語言：統一組譯器語言

為了支援並發揮 Thumb-2 指令集的最佳效果，遂開發了統一組譯器語言(Unified assembler Language, UAL)，以允許選擇 16-bit 和 32-bit 指令，並且藉著兩者相同的語法，使 ARM 程式和 Thumb 程式間，更加容易移植應用程式。(使用 UAL，則Thumb 指令與 ARM 指令有相同的語法)

ADD R0, R1 ; R0 = R0 + R1，使用傳統的 Thumb 語法
ADD R0, R0, R1 ; 如上相同的指令，使用 UAL 語法

# Jason 嘴書—EETOP 世界唯一貼

傳統 Thumb 的語法仍然適用。但你需要留意一件事，意即當使用傳統 Thumb 指令的語法時，即使未使用後綴字 S，某些指令也會改變 APSR 裡的旗標值。然而，當使用 UAL 語法時，指令會不會影響旗標，則隨後綴字 S 而定。例如：

AND R0, R1 ; 傳統的 Thumb 語法	ANDS R0, R0, R1 ; 等效的 UAL 指令(加上後綴字 S)
---------------------------	---------------------------------------

在新的 Thumb-2 指令支援之下，有些運算可使用 Thumb 指令或 Thumb-2 指令來處理。例如， $R0 = R0 + 1$  可以用 16-bit 的 Thumb 指令，或 32-bit 的 Thumb-2 指令來實現。使用 UAL，你可以加上後綴字，來指定要用哪一個指令：

ADDS R0, #1 ; 使用 16-bit 的 Thumb 指令 ；預定值，以減少其大小
ADDS.N R0, #1 ; 使用 16-bit 的 Thumb 指令(N = Narrow)
ADDS.W R0, #1 ; 使用 32-bit 的 Thumb-2 指令(W = Wide)

後綴字.W(即 wide)指定了 32-bit 的指令。未加上後綴字時，組譯器工具可任意選擇指令，但通常預定為 16-bit 的 Thumb 程式碼，以得到較小的程式。如果組譯器工具支援的話，你也可以藉著後綴字.N(即 narrow)來指定選用 16-bit 的 Thumb 指令。

再一次強調，上述語法適用於 ARM 組譯器工具。其它組譯器可能會有稍微不一樣的語法。如果未加上後綴字，則組譯器可能依最小程式碼，來替你選擇指令。

大部分情形下，應用程式會以 C 撰寫，而 C 編譯器會儘可能使用 16-bit 的指令，以得到較小的程式碼。然而，如果立即值的資料大小超出某特定範圍，或運算較易以 32-bit 的 Thumb-2 指令來處理，則會選用 32-bit 的指令。

32-bit 的 Thumb-2 指令，可能以 half word 為單位來對齊。例如，你可以有一個置於 half word 位置的 32-bit 指令：

0x1000 : LDR r0, [r1] ; 16-bit 指令(佔用 0x1000-0x1001)
0x1002 : RBIT.W r0 ; 32-bit 的 Thumb-2 指令 ；(佔用 0x1002-0x1005)

大部分 16-bit 指令僅可以存取暫存器 R0 至 R7；32-bit 的 Thumb-2 指令則無此限制。然而，PC(即 R15)並不允許使用於某些指令裡。如果你需要找到這方面的細

節，請參考 ARM v7-M Architecture Application Level Reference Manual( Ref 2: section A4.6)。

## 指令列表

表 4-2 至表 4-9 列出所有支援的指令。每一個指令完整的細節，請參考 ARM v7-M Architecture Application Level Reference Manual( Ref 2)。本書附錄 A 為支援指令集的摘要。

表 4-2 16-bit 資料處理指令

指令	功能
ADC	Add with carry, 考慮進位的加法
ADD	Add, 加
AND	Logical AND, 邏輯 AND
ASR	Arithmetic shift right, 算術右移
BIC	Bit clear, 位元清除 (將某值與其邏輯上相反的值作邏輯 AND)
CMN	Compare negative, 負值比較 (將一個資料與其它資料的 2 的補數做比較並且更新旗標值)
CMP	Compare, 比較 (比較 2 筆資料並且更新旗標值)
CPY	Copy, 複製 (architecture v6 開始可用；從一個高或低的暫存器移動數值到另一個高或低的暫存器)
EOR	Exclusive OR, 互斥 OR 運算
LSL	Logical shift left, 邏輯左移
LSR	Logical shift right, 邏輯右移
MOV	Move, 移動(可作暫存器到暫存器的傳輸或載入立即值資料)
MUL	Multiply, 乘
MVN	Move NOT (獲取邏輯上相反的值)
NEG	Negate (取得 2 的補數)
ORR	Logical OR, 邏輯 OR
RROR	Rotate right, 向右旋轉
SBC	Subtract with carry, 考慮進位的減法
SUB	Subtract, 減
TST	Test , 測試(使用如 logical AND；更新 Z 旗標，但並不保存 AND 結果)
REV	Reverse the byte order 在 32-bit 暫存器將 byte 次序倒置(architecture v6 開始可用)
REVH	Reverse the byte order 在 32-bit 暫存器的每一個 16-bit half word 將 byte 次序倒置(architecture v6 開始可用)
REVSH	Reverse the byte order 在 32-bit 暫存器的低的 16-bit half word 將 byte 次序倒置，並做正負號的位元擴充至 32 bits(architecture v6 開始可用)
SXTB	Signed extend byte , 有號數擴充位元組(architecture v6 開始可用)
SXTH	Signed extend half word, 有號數擴充 half word(architecture v6 開始可用)
UXTB	Unsigned extend byte, 無號數擴充 byte (architecture v6 開始可用)
UXTH	Unsigned extend half word, 無號數擴充 half word (architecture v6 開始可用)

表 4-3 16-bit 跳躍指令

指令	功能
B	Branch, 跳躍
B<cond>	Conditional branch, 條件式跳躍
BL	Branch with link: 呼叫副程式並且將返回位址儲存於 LR
BLX	Branch with link and change state (BLX <reg> only) <sup>1</sup> , 跳躍連結並改變狀態
CBZ	Compare and branch if zero (architecture v7), 比較若為零則跳躍
CBNZ	Compare and branch if nonzero (architecture v7), 比較若為非零則跳躍
IT	IF-THEN (architecture v7)

表 4-4 16-bit Load and Store 指令

指令	功能
LDR	Load word from memory to register, 從記憶體載入 word 至暫存器
LDRH	Load half word from memory to register, 從記憶體載入 half word 至暫存器
LDRB	Load byte from memory to register, 從記憶體載入 byte 至暫存器
LDRSH	Load half word from memory, sign extend it, and put it in register, 從記憶體載入 half word, 做正負號擴充並置入暫存器
LDRSB	Load byte from memory, sign extend it, and put it in register, 從記憶體載入 byte, 做正負號擴充並置入暫存器
STR	Store word from register to memory, 從暫存器儲存 word 至記憶體
STRH	Store half word from register to memory, 從暫存器儲存 half word 至記憶體
STRB	Store byte from register to memory, 從暫存器儲存 byte 至記憶體
LDMIA	Load multiple increment after, 多重載入之後遞增
STMIA	Store multiple increment after, 多重儲存之後遞增
PUSH	Push multiple register, 推進多個暫存器
POP	Pop multiple register, 推出多個暫存器

表 4-5 其它 16-bit 指令

指令	功能
SVC	System service call, 系統服務呼叫
BKPT	Breakpoint, 中斷點；若除錯已啟能則進入除錯模式(暫停), 或除錯監視例外被啟能則喚起除錯例外, 否則將喚起錯誤例外
NOP	No operation, 不做運算
CPSIE	啟能 PRIMASK (CPSIE i)/FAULTMASK (CPSIE f) 暫存器(設定暫存器為 0)
CPSID	除能 PRIMASK (CPSID i)/FAULTMASK (CPSID f) 暫存器(設定暫存器為 1)

<sup>1</sup> 並未支援使用立即值的 BLX 指令, 因為此指令經常會試圖切換至 Cortex-M3 未支援的 ARM 狀態。嘗試使用 BLX<reg>以切換至 ARM 狀態也將會造成錯誤的例外。

表 4-6 32-bit 資料處理指令

指令	功能
ADC	Add with carry, 考慮進位的加法
ADD	Add, 加
ADDW	Add wide (#immed_12), 加上立即值
AND	Logical AND, 邏輯 AND
ASR	Arithmetic shift right, 算術右移
BIC	Bit clear, 位元清除(將某值與其邏輯上相反的值作 Logical AND)
BFC	Bit field clear, bit field 清除
BFI	Bit field insert, bit field 插入
CMN	Compare negative (將一個資料與其它資料的 2 的補數做比較並且更新旗標值)
CMP	Compare (比較 2 筆資料並且更新旗標值)
CLZ	Count lead zero, 計數開頭 0 的個數
EOR	Exclusive OR, 互斥 OR
LSL	Logical shift left, 邏輯左移
LSR	Logical shift right, 邏輯右移
MLA	Multiply accumulate, 乘後累加
MLS	Multiply and subtract, 乘後相減
MOV	Move, 移動
MOVW	Move wide (將 16-bit 立即值寫入暫存器)
MOVT	Move top (寫入一個立即值至目的暫存器的頂端 half word )
MVN	Move negative, 移入負值
MUL	Multiply, 乘
ORR	Logical OR, 邏輯 OR
ORN	Logical OR NOT 邏輯 NOR
RBIT	Reverse bit, 位元值反向
REV	Byte reverse word, word 內做 byte 反轉
REVH / REV16	Byte reverse packed half word, half word 內做 byte 反轉
REVS	Byte reverse signed half word, 低 half word 內做 byte 反轉並且高 half word 做正負號擴充
ROR	Rotate right register, 暫存器向右旋轉
RSB	Reverse subtract, 反轉並相減
RRX	Rotate right extended, 向右旋轉並擴充
SBFX	Signed bit field extract, 擷取正負號欄位
SDIV	Signed divide, 有號數除法
SMLAL	Signed multiply accumulate long, 有號數乘法累加 long
SMLL	Signed multiply long, 有號數乘法 long
SSAT	Signed saturate, 有號數飽和
SBC	Subtract with carry, 考慮進位的減法

接下頁

指令	功能
SUB	Subtract, 減
SUBW	Subtract wide (#immed_12), 與立即值相減
SXTB	Sign extend byte 正負號擴充至 byte
TEQ	Text equivalent (用法有如邏輯 exclusive OR; 更新旗標但不儲存結果)
TST	Test, 測試(使用如 logical AND; 更新 Z 旗標, 但並不保存 AND 結果)
UBFX	Unsigned bit field extract, 無號數 bit field 檢取
UDIV	Unsigned divide, 無號數除法
UMLAL	Unsigned multiply accumulate long, 無號數乘法並累加 long
UMULL	Unsigned multiply long, 無號數乘法 long
USAT	Unsigned saturate, 無號數飽和
UXTB	Unsigned extend byte, 無號數擴充 byte
UXTH	Unsigned extend half word, 無號數擴充 half word

表 4-7 32-bit Load and Store 指令

指令	功能
LDR	Load word data from memory to register, 從記憶體載入 word 資料至暫存器
LDRB	Load byte data from memory to register, 從記憶體載入 byte 資料至暫存器
LDRH	Load half word data from memory to register, 從記憶體載入 half word 資料至暫存器
LDRSB	Load byte data from memory, sign extend it, and put it in register, 從記憶體載入 byte 資料, 做正負號擴充並置入暫存器
LDRSH	Load half word data from memory, sign extend it, and put it in register, 從記憶體載入 half word 資料, 做正負號擴充並置入暫存器
LDM	Load multiple data from memory to register, 從記憶體載入多重資料至暫存器
LDRD	Load double word data from memory to register, 從記憶體載入 double word 資料至暫存器
STR	Store word to memory, 儲存 word 至記憶體
STRB	Store byte data to memory, 儲存 byte 資料至記憶體
STRH	Store half word data to memory, 儲存 half word 資料至記憶體
STM	Store multiple words from register to memory, 從暫存器儲存多重 word 資料至記憶體
STRD	Store double word data from register to memory, 從暫存器儲存 double word 資料至記憶體
PUSH	Push multiple register, 推進多個暫存器
POP	Pop multiple register, 推出多個暫存器

表 4-8 32-bit 跳躍指令

指令	功能
B	Branch, 跳躍
BL	Branch and link, 跳躍並且連結
TBB	Table branch byte, 使用 byte 位移表做往前的跳躍
TBH	Table branch half word, 使用 half word 位移表做往前的跳躍

表 4-9 其它 32-bit 指令

指令	功能
LDREX	Exclusive load word, 獨佔載入 word
LDREXH	Exclusive load half word, 獨佔載入 half word
LDREXB	Exclusive load byte, 獨佔載入 byte
STREX	Exclusive store word, 獨佔儲存 word
STREXH	Exclusive store half word, 獨佔儲存 half word
STREXB	Exclusive store byte, 獨佔儲存 byte
CLREX	Clear the local exclusive access record of local processor, 清除區域處理器的區域獨佔存取紀錄
MRS	Move special register to general-purpose register, 從特殊暫存器到一般用途暫存器作移動
MSR	Move to special register from general-purpose register, 從一般用途暫存器到特殊暫存器作移動
NOP	No operation, 不做運算
SEV	Send event, 送出事件
WFE	Sleep and wake for event, 睡眠並以事件喚起
WFI	Sleep and wake for interrupt, 睡眠並以中斷喚起
ISB	Instruction synchronization barrier, 指令同步障礙
DSB	Data synchronization, 資料同步障礙
DMB	Data memory barrier, 資料記憶體障礙

## 未支援的指令

表 4-10 列出了一些 Cortex-M3 不支援的 Thumb 指令。

表 4-10 不被支援的傳統 ARM 處理器的 thumb 指令

未支援的指令	功能
BLX label	此為連結的跳躍並改變狀態。在立即值資料格式下, BLX 通常切換至 ARM 狀態。因為 Cortex-M3 並不支援 ARM 狀態, 會試圖切換至 ARM 狀態的這種指令將造成稱作用法錯誤的錯誤例外。
SETEND	此於 architecture v6 導入的 Thumb 指令, 會於執行期間切換 endian 組態, 但因為 Cortex-M3 並不支援動態 endian, 使用 SETEND 指令會造成錯誤例外。

Cortex-M3 亦不支援列於 ARM v7-M Architecture Application Level Reference Manual 的一些指令。ARM v7-M 架構允許 Thumb-2 幫助處理器指令, 但是 Cortex-M3 並不支援 coprocessor。所以, 執行列於表 4-11 的輔助處理器指令時, 會出現錯誤的例外(用法錯誤, NVIC 裡的 NOCP 旗標被設定為 1)。

表 4-11 不被支援的輔助處理器的指令

未支援的指令	功能
MCR	Move to coprocessor from ARM processor, 從 ARM 處理器移至輔助處理器
MCR2	Move to coprocessor from ARM processor, 從 ARM 處理器移至輔助處理器
MCRR	Move to coprocessor from two ARM registers, 從兩個 ARM 暫存器移至輔助處理器
MRC	Move to ARM register from coprocessor, 從輔助處理器移至 ARM 暫存器
MRC2	Move to ARM register from coprocessor, 從輔助處理器移至 ARM 暫存器
MRRC	Move to two ARM registers from coprocessor, 從輔助處理器移至兩個 ARM 暫存器
LDC	Load coprocessor; 載入輔助處理器。從連續的記憶體位址循序載入記憶體資料到輔助暫存器
STC	Store coprocessor; 儲存輔助處理器。從輔助暫存器儲存資料到一連續記憶體位址中。

Cortex-M3 亦不支援一些改變程序狀態(Change Process State, CPS)的指令(參考表 4-12)。這是因為 PSR 的定義已經改變了, 故一些 ARM 架構 v6 定義的 bits, 在 Cortex-M3 裡不再使用。

表 4-12 不被支援的改變處理狀態 CPS 指令

未支援的指令	功能
CPS<E D>.WA	Cortex-M3 裡並無 A bit
CPS.W #mode	Cortex-M3 PSR 裡並無 mode bit

此外, 如表 4-13 所示的暗示(hint)指令, 在 Cortex-M3 當作 NOP 使用。

表 4-13 不被支援的暗示指令

未支援的指令	功能
DBG	除錯與追蹤系統的暗示指令
PLD	Preload data. 預先載入資料, 此為對快取記憶體的暗示指令, 但是因為 Cortex-M3 處理器裡並無快取, 此指令作用有如 NOP。
PLI	Preload Instruction. 預先載入指令, 此為對快取記憶體的暗示指令, 但是因為 Cortex-M3 處理器裡並無快取, 此指令作用有如 NOP。
YIELD	此暗示指令允許多執行緒軟體去告訴硬體, 其執行工作可移出以改善整體系統的效率。

執行任何其它未定義的指令, 將導致產生用法錯誤的例外。

## 指令描述

在此, 我們介紹一些 ARM 組合語言程式裡慣用的語法。對於有多種用法選擇的指令, 例如 barrel shifter 移位器, 本章並不會全部涵蓋。

### 組譯器語言：資料移動

在處理器裡其中一個最基本的功能是資料傳輸。在 Cortex-M3 裡, 資料傳輸可以為如下型態之一：

◆ 暫存器與暫存器之間資料移動

◆ 特殊暫存器與暫存器之間資料移動

◆ 記憶體與暫存器之間資料移動

◆ 將立即值資料移進暫存器

暫存器之間資料移動的指令是 MOV (即 move)。例如, 下面是把資料從 R3 移至 R8 :

```
MOV R8, R3
```

另一個可產生原資料的負值的指令, 稱為 MVN(即 move negative)。

存取記憶體的基本指令是 Load 和 Store。Load( LDR )將資料從記憶體載入暫存器, Store 將資料從暫存器存進記憶體。可以傳輸不同大小的資料( byte, half word, word, double word), 如表 4-14 的摘要所示。

表 4-14 慣用的記憶體存取指令

例子	說明
LDRB Rd, [Rn, #offset]	從記憶體位址 Rn + offset 讀取 byte
LDRH Rd, [Rn, #offset]	從記憶體位址 Rn + offset 讀取 half-word
LDR Rd, [Rn, #offset]	從記憶體位址 Rn + offset 讀取 word
LDRD Rd1, Rd2, [Rn, #offset]	從記憶體位址 Rn + offset 讀取 double word
STRB Rd, [Rn, #offset]	儲存 byte 至記憶體位址 Rn + offset
STRH Rd, [Rn, #offset]	儲存 half-word 至記憶體位址 Rn + offset
STR Rd, [Rn, #offset]	儲存 word 至記憶體位址 Rn + offset
STRD Rd1, Rd2, [Rn, #offset]	儲存 double word 至記憶體位址 Rn + offset

多個 Load 和 Store 指令可合成單獨的指令，稱作 LDM(即 Load Multiple)和 STM(即 Store Multiple)，如表 4-15 的摘要所示。

表 4-15 多重記憶體存取指令

例子	說明
LDMIA Rd!, <reg list>	從記憶體裡 Rd 指定的位置讀取多重 words。在每一傳輸之後遞增(Increment After, IA)位址(16-bit Thumb 指令)。
STMIA Rd!, <reg list>	儲存多重 words 到記憶體裡 Rd 指定的位置。在每一傳輸之後遞增位址(16-bit Thumb 指令)。
LDMIA.W Rd(), <reg list>	從記憶體裡 Rd 指定的位置讀取多重 words。在每一讀取之後遞增位址(W 表示其為 32-bit Thumb-2 指令)。
LDMDB.W Rd(), <reg list>	從記憶體裡 Rd 指定的位置讀取多重 words。在每一讀取之前遞減(Decrement Before, DB)位址(W 表示其為 32-bit Thumb-2 指令)。
STMIA.W Rd(), <reg list>	寫入多重 words 到記憶體裡 Rd 指定的位置。在每一讀取之後遞增位址(W 表示其為 32-bit Thumb-2 指令)。
STMDB.W Rd(), <reg list>	寫入多重 words 到記憶體裡 Rd 指定的位置。在每一讀取之前遞減位址(W 表示其為 32-bit Thumb-2 指令)。

指令中的驚嘆號(!)，用來指定指令執行後，是否更新暫存器 Rd 的值。以 R8 等於 0x8000 為例：

STMIA.W	R8!, {R0-R3}	; 執行 store 後, R8 值改為 0x8010 ; (增加 4 words = 16 bytes)
STMIA	R8 , {R0-R3}	; 執行 store 後, R8 值不變

ARM 處理器亦支援前索引(pre-index)與後索引(post-index)來存取記憶體。執行前索引時，先調整內含記憶體位址的暫存器值，再以更新的位址，作記憶體資料移轉。例如：

LDR.W	R0, [R1, #offset]!	; 讀 memory[R1+offset] ; 事先把 R1 值更新為 R1+offset
-------	--------------------	--

驚嘆號！表示是否更新基底暫存器 R1。在此，驚嘆號！可以不用；在不使用驚嘆號的情形下，此指令可看成平常的基底位址加上位移(offset)的記憶體資料傳輸。前索引記憶體存取指令，包括不同資料大小的 load 和 store 指令(參考表 4-16)。

表 4-16 前索引記憶體存取指令的例子

例子	說明
LDR.W Rd, [Rn, #offset]! LDRB.W Rd, [Rn, #offset]! LDRH.W Rd, [Rn, #offset]! LDRD.W Rd1, Rd2, [Rn, #offset]!	不同大小的前索引載入指令(word, byte, half word, double word 等)。
LDRSB.W Rd, [Rn, #offset]! LDRSH.W Rd, [Rn, #offset]!	做正負號擴充的不同大小的前索引載入指令(byte, half word 等)。
STR.W Rd, [Rn, #offset]! STRB.W Rd, [Rn, #offset]! STRH.W Rd, [Rn, #offset]! STRD.W Rd1, Rd2, [Rn, #offset]!	不同大小的前索引儲存指令(word, byte, half word, double word 等)。

後索引記憶體存取指令會先根據暫存器指定的基底位址，執行記憶體資料傳輸，之後再更新暫存器的位址內容。例如：

LDR.W	R0, [R1], #offset	; 讀取 memory[R1] ; 並更新 R1 至 R1 + offset
-------	-------------------	---

當使用後索引指令時，並不需要使用！驚嘆號，因為所有的後索引指令，皆會更新基底位址暫存器，而作前索引時，要不要更新基底位址暫存器，則由你選擇。

跟前索引的情形類似，後索引記憶體存取指令也包括不同資料大小的 load 和 store 指令(參考表 4-17)。

表 4-17 後索引記憶體存取指令的例子

例子	說明
LDR.W Rd, [Rn], #offset LDRB.W Rd, [Rn], #offset LDRH.W Rd, [Rn], #offset LDRD.W Rd1, Rd2, [Rn], #offset	不同大小的後索引載入指令(word, byte, half word, double word 等)。
LDRSB.W Rd, [Rn], #offset LDRSH.W Rd, [Rn], #offset	做正負號擴充的不同大小的後索引載入指令(byte, half word 等)。
STR.W Rd, [Rn], #offset STRB.W Rd, [Rn], #offset STRH.W Rd, [Rn], #offset STRD.W Rd1, Rd2, [Rn], #offset	不同大小的後索引儲存指令(word, byte, half word, double word 等)。

另外有兩個記憶體運算指令，即堆疊 PUSH 與堆疊 POP。例如：

PUSH {R0, R4-R7, R9}	; Push R0, R4, R5, R6, R7, R9 ; 到堆疊記憶體
POP {R2, R3}	; 從堆疊 Pop 到 R2, R3

通常每一個 PUSH 指令，伴隨著相同暫存器名單的相關 POP 指令，但不一定非如此不行。例如：一個常有的例外情形為把 POP 用來從函數返回：

PUSH {R0-R3, LR}	; 在副程式開始時 ; 保留暫存器的內容
....	; 執行副程式
POP {R0-R3, PC}	; 回復暫存器並返回

在這個例子中，我們直接將位址值 POP 進程式記數器，而不是 POP 回 LR 暫存器值，再跳躍至 LR 內的位址。

如第 3 章題及，Cortex-M3 擁有一些特殊暫存器。我們使用 MSR 或 MRS 去存取這些暫存器。例如：

MRS R0, PSP	; 將 processor status word 讀進 R0
MSR CONTROL, R1	; 將 R1 值寫至 control 暫存器

除了存取 APSR，你僅可在特權的模式裡，使用 MSR 或 MRS 去存取其餘的特殊暫存器。

把立即值資料移至暫存器是常見的動作。例如，當你需要存取周邊暫存器時，你需要先將其位址值放進暫存器。對於小數值(8 bits 以下)，可使用 MOV(即 move)指令。例如：

MOV R0, #0x12	; 令 R0 為 0x12
---------------	---------------

對於大一點的數值(大於 8 bits)，則需使用 Thumb-2 的 move 指令。例如：

MOV.W R0, #0x789A	; 令 R0 為 0x789A
-------------------	-----------------

或者，當數值為 32-bit 時，你可使用兩個指令來分別設定上半(高位元)與下半(低位元)的值：

MOVW.W R0, #0x789A	; 設定 R0 低位元的值為 0x789A
MOVT.W R0, 0x3456	; 設定 R0 高位元的值為 0x3456
	; 故 R0 = 0x3456789A

亦可採行另一個替代設定方式，即使用 LDR (AMR 組譯器提供的虛擬指令)。例如：

LDR R0, =0x3456789A
---------------------

此非真正的組合語言命令，但是 ARM 組譯器會將它轉換為以 PC 為相對位址的 load 指令，來產生要求的資料。欲產生 32-bit 立即值資料時，建議使用 LDR 而非 MOVM.W 及 MOVT.W 的組合，這樣做，則程式的可讀性比較高，並且如果同一程式多次參考相同立即值的話，組譯器可據以減少佔用的記憶體。

## 虛擬指令 LDR 和 ADR

LDR 和 ADR 兩者虛擬指令，皆可以將暫存器設定為程式碼位址值。但兩者的語法和行為相異。使用 LDR，如其位址是程式碼位址值，組譯器將主動地把 LSB 設為 0。例如：

# Jason 嘴書—EETOP 世界唯一貼

```

LDR      R0, =address1          ; R0 被設定為 0x4001
...
address1
0x4000:   MOV      R0,      R1          ; address1 包含了程式碼
...

```

你將發現 LDR 指令把 0x4001 放進 R0, LSB 被設定為 1, 以示其為 Thumb 程式碼。但如果 address1 是資料位址, LSB 將不被更改。例如：

```

LDR      R0, =address1          ; R0 被設定為 0x4000
...
address1
0x4000:   DCD      0x0          ; address1 包含了資料
...

```

你也可以使用 ADR 指令, 將程式碼的位址值, 載入暫存器, 此時 LSB 不被主動設定。例如：

```

ADR      R0, address1
...
address1
0x4000:   MOV      R0,      R1          ; address1 包含了程式碼
...

```

使用 ADR 指令, 你將得到 0x4000。注意, 等號( = )並不在 ADR 的敘述中出現。

LDR 獲得立即值資料的方式是先將資料置於程式碼裡, 再藉由 PC 相對的載入動作, 將資料放進暫存器。ADR 則藉著加減的指令來產生立即值(例如, 根據現在 PC 的值)。因此, 利用 ADR 指令並不能夠產生任意的立即值, 且目的位址的標籤須在近距離內。然而, 使用 ADR 相較於使用 LDR, 可產生比較小的程式。

## 組譯器語言：資料處理

Cortex-M3 提供了許多不同的資料處理的指令, 在此介紹一些基本資料處理指令。許多的資料處理指令有多重指令型態。例如, ADD 指令可運作於兩個暫存器之間或一個暫存器與一個立即值之間：

```

ADD      R0,      R1          ; R0 = R0 + R1
ADD      R0, #0x12          ; R0 = R0 + 0x12
ADD.W   R0, R1, R2          ; R0 = R1 + R2

```

上述皆為 ADD 指令, 但其語法與二進位碼皆不相同。

在使用 16-bit 的 Thumb code 時, ADD 指令會改變 PSR 的旗標值。然而, 在 32-bit 的 Thumb-2 碼時, 則可以指定保留或更動旗標值。如果後面的運算會受到旗標值影響, 則可以使用後綴字 S 來區別兩種不同的 ADD 運算：

```

ADDS.W  R1, R2          ; 保留旗標值
ADDS.W  R0, R1, R2        ; 更動旗標值

```

在 ADD 指令之外, Cortex-M3 支援的算術運算功能包括 SUB( subtract, 減), MUL( Multiply, 乘 ), 和 UDIV/SDIV (unsigned and signed divide 無號數及有號數的除法)。表 4-18 為最常用的算術運算指令。

表 4-18 算術運算指令的例子

指令	運算
ADD Rd, Rn, Rm ; Rd = Rn + Rm	加法運算
ADD Rd, Rm ; Rd = Rd + Rm	
ADD Rd, #immed ; Rd = Rn + #immed	
ADC Rd, Rn, Rm ; Rd = Rn + Rm + carry	包含進位的加法運算
ADC Rd, Rm ; Rd = Rd + Rm + carry	
ADC Rd, #immed ; Rd = Rn + #immed + carry	
ADDW Rd, Rn, #immed ; Rd = Rn + #immed	暫存器與 12-bit 立即值相加
SUB Rd, Rn, Rm ; Rd = Rn - Rm	減法運算
SUB Rd, #immed ; Rd = Rd - #immed	
SUB Rd, Rn, #immed ; Rd = Rn - #immed	
SBC Rd, Rm ; Rd = Rn - Rm - carry flag	包含借位(進位)的加法運算
SBC.W Rd, #immed ; Rd = Rd - #immed - carry flag	
SBC.W Rd, Rn, Rm ; Rd = Rn - Rm - carry flag	
RSB.W Rd, Rn, #immed ; Rd = #immed - Rn	正負反向的減法
RSB.W Rd, Rn, Rm ; Rd = Rm - Rn	
MUL Rd, Rm ; Rd = Rd * Rm	乘法
MUL.W Rd, Rn, Rm ; Rd = Rn * Rm	
UDIV Rd, Rn, Rm ; Rd = Rd / Rm	無號數與有號數除法
SDIV Rd, Rn, Rm ; Rd = Rn / Rm	

# Jason 嘴書—EETOP 世界唯一貼

Cortex-M3 亦支援 32-bit 的乘法指令，與乘後累加(multiply accumulate)指令，其結果為 64-bit。這些指令支援有號數與無號數(參考表 4-19)。

表 4-19 32-bit 乘法指令

指令	運算
SMULL RdLo, RdHi, Rn, Rm ; {RdHi, RdLo} = Rn * Rm SMLAL RdLo, RdHi, Rn, Rm ; {RdHi, RdLo} += Rn * Rm	有號數值的 32-bit 乘法指令
UMULL RdLo, RdHi, Rn, Rm ; {RdHi, RdLo} = Rn * Rm UMLAL RdLo, RdHi, Rn, Rm ; {RdHi, RdLo} += Rn * Rm	無號數值的 32-bit 乘法指令

另一種資料處理的指令為邏輯運算指令，包括 AND、ORR、移位(shift)、旋轉(rotate)等功能。表 4-20 為最常用的邏輯運算指令。

表 4-20 邏輯運算指令

指令	運算
AND Rd, Rn ; Rd = Rd & Rn	
AND.W Rd, Rn, #immed ; Rd = Rd & #immed	位元逐次做 AND
AND.W Rd, Rn, Rm ; Rd = Rn & Rm	
ORR Rd, Rn ; Rd = Rd   Rn	
ORR.W Rd, Rn, #immed ; Rd = Rd   #immed	位元逐次做 OR
ORR.W Rd, Rn, Rm ; Rd = Rn   Rm	
BIC Rd, Rn ; Rd = Rd & (~Rn)	
BIC.W Rd, Rn, #immed ; Rd = Rd & (~#immed)	清除位元
BIC.W Rd, Rn, Rm ; Rd = Rn & (~ Rm)	
ORN.W Rd, Rn, #immed ; Rd = Rd   (~#immed)	
ORN.W Rd, Rn, Rm ; Rd = Rn   (~ Rm)	位元逐次做 OR NOT
EOR Rd, Rn ; Rd = Rd ^ Rn	
EOR.W Rd, Rn, #immed ; Rd = Rd ^ #immed	位元逐次做 Exclusive OR
EOR.W Rd, Rn, Rm ; Rd = Rn ^ Rm	

Cortex-M3 提供旋轉和位移指令。在某些情形下，旋轉運算可結合其它運算(例如，load/store 裡，記憶體位址位移的計算)。表 4-21 為單獨的 rotate/shift 指令。

表 4-21 位移與旋轉指令

指令	運算
ASR Rd, Rn, #immed ; Rd = Rn >> immed	算數右移
ASR Rd, Rn ; Rd = Rd >> Rn	
ASR.W Rd, Rn, Rm ; Rd = Rn >> Rm	
LSL Rd, Rn, #immed ; Rd = Rn << immed	邏輯左移
LSL Rd, Rn ; Rd = Rd << Rn	
LSL.W Rd, Rn, Rm ; Rd = Rn << Rm	
LSR Rd, Rn, #immed ; Rd = Rn >> immed	邏輯右移
LSR Rd, Rn ; Rd = Rd >> Rn	
LSR.W Rd, Rn, Rm ; Rd = Rn >> Rm	
ROR Rd, Rn ; Rd rot by Rn	向右旋轉
ROR.W Rd, Rn, Rm ; Rd = Rn rot by Rm	
RRX.W Rd, Rn ; {C, Rd} = {Rn, C}	向右旋轉並擴充

藉著後綴字 S，旋轉和位移運算亦可改變進位旗標(在 16-bit Thumb 碼中，進位旗標恆受影響)。參照圖 4-1。

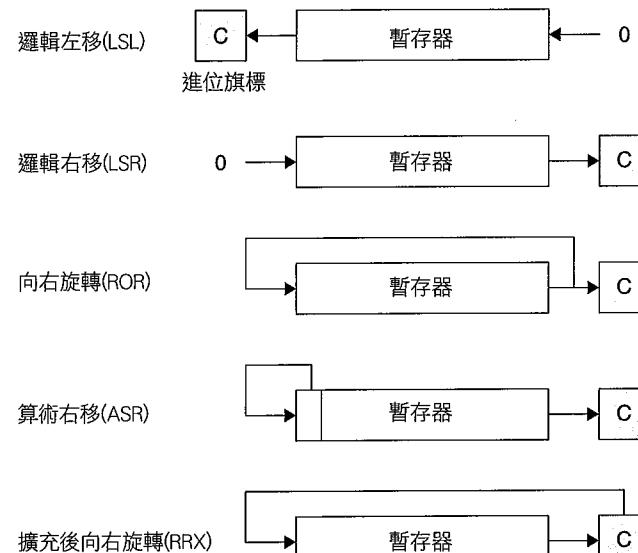


圖 4-1 位移和旋轉指令

如果位移或旋轉運算移動了多個位元，進位旗標 C 的值為移出暫存器的最後一個位元。

### 為何有 Rotate Right 但無 Rotate Left 指令？

Rotate left(左旋轉)的指令，可被不同旋轉位移值的 Rotate right(右旋轉)取代。例如：指令 rotate left by 4-bit = 指令 rotate right by 28-bit，兩者運算結果與執行所需時間相同。

Cortex-M3 提供表 4-22 所示的兩個指令，可將有號數資料，從 byte 或者 half word 轉為 word。

表 4-22 有號的擴充(Sign Extend)指令

指令	運算
SXTB.W Rd, Rn, ; Rd = signext (Rn[7:0])	Byte 資料擴充正負號至 word
SXTH.W Rd, Rn, ; Rd = signext (Rn[15:0])	Half word 資料擴充正負號至 word

另一組資料處理指令，可將暫存器的資料作 byte 反轉(參考表 4-23 及圖 4.2)。這些指令可用來做 little endian 和 big endian 資料之間的互轉。

表 4-23 資料反轉次序(Reverse Ordering)指令

指令	運算
REV.W Rd, Rn, ; Rd = rev (Rn)	反轉 word 中的 Byte
REV16.W <Rd>, <Rn>; Rd = rev16 (Rn)	反轉每一個 half word 中的 Byte
REVSH.W <Rd>, <Rn>; Rd = revsh (Rn)	反轉底部 half word 中的 Byte, 之後做正負號擴充

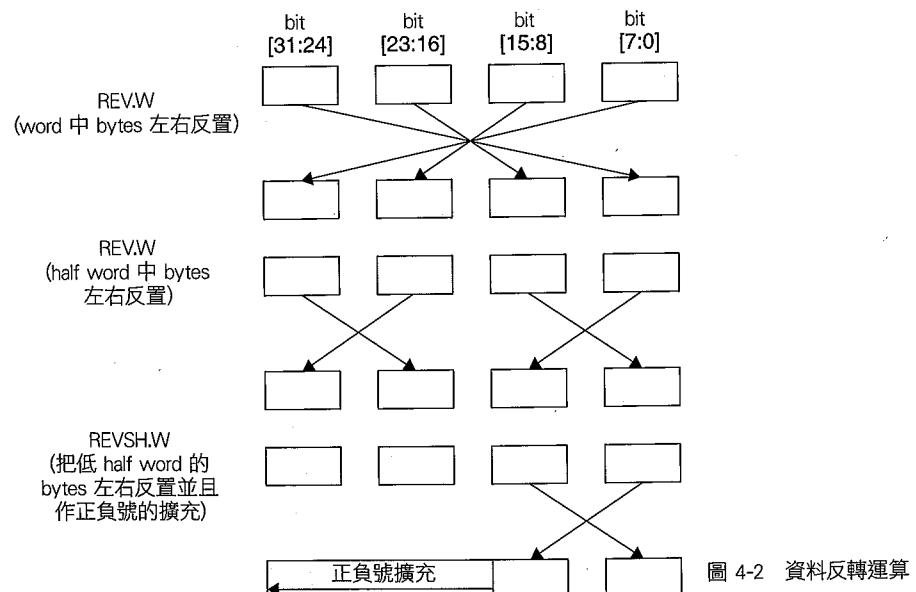


圖 4-2 資料反轉運算

最後一組資料處理指令可做 bit field 處理。如表 4-24 所列出的指令。本章後面將提供這些指令的例子。

表 4-24 Bit Field 處理與操縱指令

指令	運算
BFC.W Rd, Rn, #<width>	清除暫存器內的 bit field
BFI.W Rd, Rn, #<lsb>, #<width>	插入 bit field 至暫存器
CLZ.W Rd, Rn	計數開頭 0 的個數
RBIT.W Rd, Rn	反轉暫存器裡的位元的次序
SBFX.W Rd, Rn, #<lsb>, #<width>	複製來源的 bit field 並做正負號擴充
UBFX.W Rd, Rn, #<lsb>, #<width>	複製來源暫存器的 bit field

### 組譯器語言：Call 和 Unconditional(無條件的)跳躍

最基本的跳躍指令為：

B	label	; 跳躍至標籤所在位址
BX	reg	; 跳躍至暫存器指定之位址

使用 BX 指令，則暫存器存放資料的 LSB 指定了處理器的下一個狀態(Thumb/ARM)。因為 Cortex-M3 總是作用在 Thumb 狀態，其 LSB 應設為 1，當被設為 0，程式將造成一個用法錯誤的例外，因為它會嘗試切換處理器至 ARM 狀態。

呼叫函數應使用跳躍和連結指令：

BL	label	; 跳躍至標籤所在位址
		; 並將 return 位址存放在 LR
BLX	reg	; 跳躍至暫存器指定之位址
		; 並將 return 位址存放在 LR

上面的指令會將返回位址存放在連結(link)暫存器，故可使用 BX LR 來結束函式呼叫，以將程式控制還給呼叫程序。然而，當使用 BLX 時，應確定其暫存器之 LSB 為 1。否則，處理器將產生一個錯誤例外，因為它會嘗試切換至 ARM 狀態。

你也可以藉由 MOV 和 LDR 指令來做跳躍運算，例如：

# Jason 嘴書—EETOP 世界唯一貼

MOV	R15, R0	; 跳躍至 R0 內位址值
LDR	R15, [R0]	; 跳躍至 R0 指定位址 ; 的記憶體位置
POP	{R15}	; 執行堆疊 POP 運算 ; 將程式記數器之值 ; 改為運算結果值

當使用上述方法去執行跳躍，你也需要確定更新的程式記數器值的 LSB 為 0x1。否則，將產生一個用法錯誤的例外，因為它會嘗試切換處理器至 Cortex-M3 不允許的 ARM 狀態。

## 當你需要呼叫副程式時，記得保存 LR 值

BL 指令會破壞正存放在 LR 暫存器的內容，所以，如果之後的程式碼需要 LR 暫存器值，應先將 LR 保存，再使用 BL。常用方法是在副程式開始 push 暫存器 LR 至堆疊。例如：

```

main
...
    BL    functionA
...
functionA
    PUSH {LR}      ; 保存 LR 內容於堆疊
...
    BL    functionB
...
    POP  {PC}      ; 使用堆疊 LR 值，以返回 main
functionB
    PUSH {LR}
...
    POP  {PC}      ; 使用堆疊 LR 值，以返回 functionA

```

此外，如果你呼叫了 C 語言的副程式，你可能也需保存 R0-R3，和 R12 的內容，以便在後面的程式裡使用。

根據 AAPCS(Ref 5)，這些暫存器裡的內容，可能會被 C 程式改變。

## 組譯器語言：抉擇與條件式跳躍

大多數 ARM 處理器裡條件式跳躍利用 APSR(Application Program Status Register 應用程式狀態暫存器)內的旗標，來決定跳躍是否執行。APSR 有 5 個旗標位元，其中 4 個用來做跳躍抉擇(參考表 4-25)。

表 4-25 APSR 中用來做條件式跳躍的旗標位元

旗標	PSR 位元	描述
N	31	負值旗標(先前運算結果為負值)
Z	30	零(先前運算結果回傳零的值)
C	29	進位(先前運算回傳進位或借位)
V	28	溢位(先前運算結果溢位)

另外一個旗標位元 bit[27]，稱為 Q 旗標，它作用在飽和算數運算，而不在條件式跳躍中使用。

## ARM 處理器中的旗標

資料處理指令經常會改變 PSR 內的旗標值。旗標可用以作跳躍抉擇，或者作為下一個指令輸入的一部分，ARM 處理器通常至少包括 Z, N, C, 和 V 等旗標，執行資料處理指令會更改這些旗標：

- Z(Zero) 旗標：當指令執行結果為 0，或比較資料兩者相等時，則設定此旗標。
- N(Negative) 旗標：當指令執行結果為負值(bit 31 為 1)，則設定此旗標。
- C(Carry) 旗標：此旗標用在無號數資料處理，例如：作加法(ADD)運算，若溢位則設定此旗標；作減法(SUB)運算，若無借位(borrow)則設定此旗標(借位為進位之相反)。
- V(Overflow) 旗標：此旗標用在有號數資料處理，例如：作加法(ADD)運算，若兩正數相加結果為負，或兩負數相加結果為正則設定此旗標。

當使用位移和旋轉指令時，這些旗標也會有特別的結果，細節請參考 ARM v7-M Architecture Application Level Reference Manual (Ref 2)。

表 4-26 跳躍或其它條件式運算的條件表示法

符號	條件	旗標
EQ	Equal, 相等	設定 Z
NE	Not equal, 不相等	清除 Z
CS/HS	Carry set/ unsigned higher or same, 進位設定/無號數更高或相同	設定 C
CC/LO	Carry clear/unsigned lower, 進位 f 清除/無號數比較低	清除 C
MI	Minus/negative or zero, 負/負數或零	設定 N
PL	Plus/positive or zero, 正/正數或零	清除 N
VS	Overflow, 溢位	設定 V

接下頁

# Jason 嘴書—EETOP 世界唯一貼

符號	條件	旗標
VC	No overflow, 無溢位	清除 V
HI	Unsigned higher, 無號數更高	設定 C 並清除 Z
LS	Unsigned lower or same, 無號數更低或相同	清除 C 或設定 Z
GE	Signed greater than or equal, 有號數較大或相同	設定 N 並且設定 V, 或者清除 N 並清除 V( $N=V$ )
LT	Signed less than, 有號數較小	設定 N 並清除 V, 或者清除 N 並設定 V( $N \neq V$ )
GT	Signed greater than, 有號數較大	清除 Z, 並且設定 N 或 V, 或者清除 N 並設定 V ( $Z=1$ or $N \neq V$ )
AL	Always (unconditional), 永遠	-

四種旗標(N, Z, C, 和 V)組合出 15 種跳躍條件(參照表 4-26)。加上這些條件, 跳躍指令可如下：

```
BEQ label ; 當 Z 旗標設定時, 跳躍至 'label' 的位址
```

如果跳躍的目的在更遠端, 你亦可使用 Thumb-2 的版本來寫。例如：

```
BEQ.W label ; 當 Z 旗標設定時, 跳躍至 'label' 的位址
```

亦可在 IF-THEN-ELSE 結構裡, 使用跳躍的條件, 例如：

```
CMP R0, R1 ; 比較 R0 和 R1
ITTEE GT ; 如果 R0 > R1 敘述
            ; 成立, 則執行前面兩個指令
            ; 不成立, 則執行後面兩個指令
MOVGT R2, R0 ; R2 = R0
MOVGT R3, R1 ; R3 = R1
MOVLE R2, R1 ; ELSE R2 = R1
MOVLE R3, R0 ; R3 = R0
```

PSR 旗標會受到下面影響：

◆ 16-bit ALU 指令

◆ 32-bit(Thumb-2) ALU 指令加上了後綴字；例如, ADDS.W

◆ 比較(例如: CMP)和測試(例如: TST, TEQ)

◆ 直接寫入 APSR/xPSR

大多數的 16-bit Thumb 的算術指令會影響 N, Z, C 和 V 旗標。使用 32-bit 的 Thumb-2 指令, ALU 運算則可能影響或不影響旗標, 例如：

ADDS.W	R0, R1, R2	; 此 32-bit Thumb-2 指令會更新旗標
ADD.W	R0, R1, R2	; 此 32-bit Thumb-2 指令不會更新旗標
<hr/>		
ADDS	R0, R1	; 此 16-bit Thumb 指令會更新旗標
ADD	R0, #0x1	; 此 16-bit Thumb 指令會更新旗標

把 ALU 指令作 Thumb 和 Thumb-2 之間代換時需留意。即使無後綴字 S, Thumb 指令可能會更新旗標, 而 Thumb-2 指令則不會, 故你可能會得到不同的結果。為了確保程式可使用於不同的工具, 你應該使用後綴字 S 使旗標得以更新, 以進行條件式運算用。

比較(CMP)指令把兩個值相減, 並更新旗標(有如 SUBS), 但相減結果並不存放在任何暫存器裡。CMP 有如下兩種形式：

CMP	R0, R1	; 計算 R0-R1 並更新旗標
CMP	R0, #0x12	; 計算 R0-0x12 並更新旗標

CMN(compare negative)指令的作用類似。它把前者與後者的負值(即 2 的補數)作比較, 旗標被更新了, 但結果並不存放在任何暫存器裡：

CMN	R0, R1	; 計算 R0 - (-R1) 並更新旗標
CMN	R0, #0x12	; 計算 R0 - (-0x12) 並更新旗標

TST(test) 指令的作用較像是 AND 指令。它 AND 兩個值並更新旗標, 但其結果並不存放在任何暫存器。類似 CMP 指令, 它有兩種形式：

TST	R0, R1	; 計算 R0 與 R1 並更新旗標
TST	R0, #0x12	; 計算 R0 與 0x12 並更新旗標

## 組譯器語言：比較與條件式跳躍的結合

根據 ARM 架構 v7-M, Cortex-M3 提供了兩個新的指令，以作為一個簡單地與 0 比較再條件式跳躍的運算。包括 CBZ (compare and branch if zero, 比較若為 0 則跳躍) 和 CBNZ (compare and branch if nonzero, 比較若為非 0 則跳躍) 兩者。

比較並作條件式跳躍的指令僅支援往前(forward) 跳躍。例如：

```
i = 5      ;
while ( i !=0 ) {
    func1( );      ; 呼叫函式
    i-- ;
}
```

此可編譯為：

MOV R0, #5	; 設定 loop 計數
loop1 CBZ R0, loopexit	; 如果 loop 計數 = 0, 則離開 loop
BL func1	; 呼叫函式
SUB R0, #1	; 將 loop 計數減一
B loop1	; 下一個 loop
Loopexit	

## 組譯器語言：使用 IT 指令的條件式跳躍

IT( IF-THEN )區塊適合處理小型的條件式程式。因為它不會更動程式流程，故避免了跳躍的代價。它最多可提供四個條件式執行指令。

在 IT 指令區塊，第一行是描述了執行選項的 IT 指令，接下來是要檢查的各個選項。接在 IT 命令(常寫作 ITxxx, 其中 T 表示 THEN, E 表示 ELSE)後的第一個敘述應該是 TRUE-THEN-EXECUTE(真則執行)，第二到第四個敘述可為真(THEN)或者為假(ELSE)：

IT<x><y><z> <cond>	; IT 指令
	; ( <x>, <y>, <z> 指令可為 T 或 E )
instr1<cond> <operands>	; 第一個指令
	; (<cond>需與 IT 相同)
instr2<cond or not cond> <operands>	; 第二個指令
	; (可以是<cond>或<!cond>)
Instr3<cond or not cond> <operands>	; 第三個指令
	; (可以是<cond>或<!cond>)
Instr4<cond or not cond> <operands>	; 第四個指令
	; (可以是<cond>或<!cond>)

如<cond>為假則執行敘述時，指令的後綴字需與<cond>相反。例如，EQ 條件的相反為 NE, GT 條件的相反為 LE, 等等。下面的程式碼為一個簡單的條件式執行的例子：

```
if ( R1 < R2 ) then
    R2 = R2-R1
    R2 = R2/2
else
    R1 = R1-R2
    R1 = R1/2
```

組合語言則寫為：

CMP R1, R2	; 若 R1 < R2
ITTEE LT	; 執行第一個與第二個指令
	; (以 T 標示)
	; 否則執行第三個與第四個指令
	; (以 E 標示)
SUBLT.W R2, R1	; 第一個指令
LSRLT.W R2, #1	; 第二個指令
SUBGE.W R1, R2	; 第三個指令 (注意：GE 為 LT 的相反)
LSRGE.W R1, #1	; 第四個指令

你可以有少於四個的條件式執行指令，但至少為一個。你需要確定 IT 指令裡 T 和 E 出現的次數，與其後出現的條件式執行指令的次數相同。

如果 IT 指令區塊中出現例外，區塊執行狀態將被儲存在堆疊 PSR(位於 IT/ICI bit field)。所以，當例外處理程式完成工作，IT 區塊恢復工作後，區塊內剩餘的指令才能正確地繼續執行。若在 IT 區塊執行多週期的指令(例如，多重 load 和 store)，當例外於動作執行中出現，會先完成整個指令再接受例外。

## 組譯器語言：指令障礙與記憶體障礙的指令

Cortex-M3 支援數個障礙(barrier)指令。因為記憶體系統益形複雜，故需要這些指令。在某些情形下，如果不使用記憶體障礙指令，則會有競爭(race)產生。

例如，如果藉著硬體暫存器來切換記憶體映射(memory map)，在寫入記憶體切換暫存器後，你需使用 DSB 指令。否則，如果寫入記憶體的動作受到緩衝，需經過數個週期才得以完成，但下一個指令會立即存取被切換的記憶體區，則存取可能會誤用舊的記憶體映射。在某些情形下，如果記憶體切換與記憶體存取同時發生，則會導致不合法的存取。在此情況，使用了 DSB 可確定寫入記憶體映射切換暫存器的動作完成後，才會執行新的指令。

Cortex-M3 裡有三個障礙指令：

◆ DMB      ◆ DSB      ◆ ISB

表 4-27 描述了這些指令。

表 4-27 障礙指令

指令	描述
DMB	Data Memory Barrier, 資料記憶體障礙；確保作新的記憶體存取前，所有的記憶體存取皆已完成
DSB	Data Synchronization Barrier, 資料同步障礙；確保下一個指令執行之前，所有的記憶體存取皆已完成
ISB	Instruction Synchronization Barrier, 指令同步障礙；清除管線並且確保執行新的指令之前，所有先前的指令皆已完成

當你於雙埠記憶體中，執行資料寫入後，緊接著作資料讀取，如果記憶體寫入受到緩衝，則需使用 DMB 以確保能讀到更新的值。

DSB 與 ISB 指令對會作自我修正的程式，有其重要性。例如，如果程式會改變其本身的程式，則下一個要執行的指令應根據更新後的程式。然而，因為處理器的管線(pipelined)動作，可能預取了被更動的指令位置，則使用 DSB 再接著 ISB，以確定能讀到更新的程式碼。

記憶體障礙的細節可參照 ARM v7-M Architecture Application Level Reference Manual (Ref 2)。

## 組譯器語言：飽和運算

Cortex-M3 支援兩個可作有號數及無號數飽和(saturation)運算的指令：SSAT(有號數資料型態)和 USAT(無號數資料型態)。飽和通常應用在信號處理上，例如，在信號放大時，放大輸入信號而得的結果可能會超出允許的輸出範圍，如果僅僅去除未使用的MSB(最高位元)，則此溢位的結果會使得整個信號波形變形(參看圖 4.3)。飽和運算並不會完全防止資料變形，但至少大為降低信號波形變形的大小。SSAT 和 USAT 指令的語法概敘於下(參看表 4-28)：

- ◆ **Rn**: 輸入值
- ◆ **Shift**: 執行飽和前對輸入值的位移運算；可有可無，可以是#LSL N 或#ASR N
- ◆ **Immed**: 要執行飽和的 bit 位置
- ◆ **Rd**: 目的暫存器

4

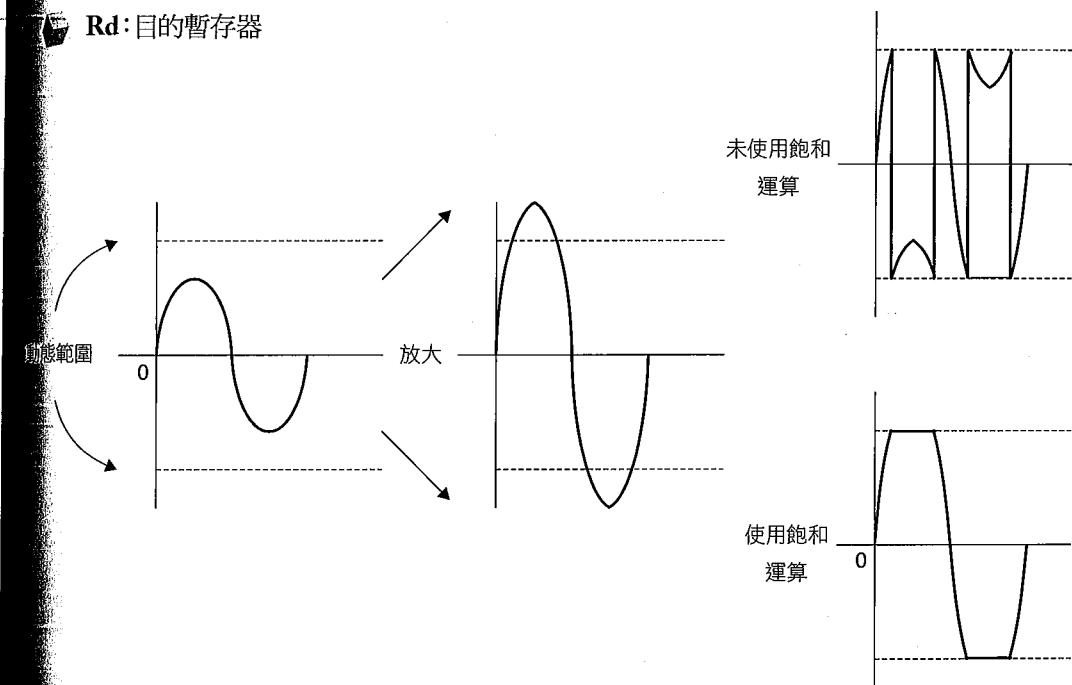


圖 4-3 有號數的飽和運算

表 4-28 飽和指令

指令	描述
SSAT.W <Rd>, #<immed>, <Rn>, {, <shift>}	飽和有號數值
USAT.W <Rd>, #<immed>, <Rn>, {, <shift>}	飽和有號數值以成為無號數值

# Jason 嘴書—ETOP 世界唯一貼

除了目的暫存器, APSR 裡 Q-bit 亦可能會受此結果影響。當飽和發生時 Q 旗標會被設定, 可藉著寫入 APSR 來清除設定(參考表 4-29)。例如, 如果把一個 32-bit 有號數作 16-bit 有號數的飽和運算, 可使用下面的指令：

```
SSAT.W R1, #16, R0
```

表 4-29 有號數飽和運算結果舉例

輸入(R0)	輸出(R1)	Q 位元
0x00020000	0x00007FFF	設定
0x00008000	0x00007FFF	設定
0x00007FFF	0x00007FFF	無改變
0x00000000	0x00000000	無改變
0xFFFF8000	0xFFFF8000	無改變
0xFFFF8001	0xFFFF8000	設定
0xFFE0000	0xFFFF8000	設定

同樣地, 如果把一個 32-bit 有號數作 16-bit 無號數的飽和運算, 可使用下面的指令：

```
USAT.W R1, #16, R0
```

此將出現如圖 4-4 中特性的飽和特徵。

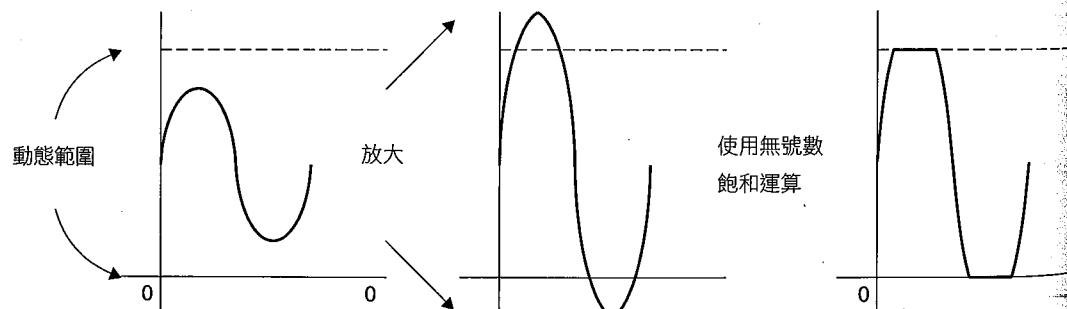


圖 4-4 無號數飽和運算

對上述的 16-bit 饱和指令例子, 將可觀察而得到如同表 4-30 的輸出值。

表 4-30 無號數飽和運算結果舉例

輸入(R0)	輸出(R1)	Q 位元
0x00020000	0x0000FFFF	設定
0x00008000	0x00008000	設定
0x00007FFF	0x00007FFF	無改變
0x00000000	0x00000000	無改變
0xFFFF8000	0x00000000	設定
0xFFFF8001	0x00000000	設定
0xFFFFFFF	0x00000000	設定

飽和指令也可以用來做資料型態的轉換。例如, 它們可用來轉換 32-bit 整數值至 16-bit 整數值。然而, C 編譯器可能無法直接使用這些指令, 所以可能需要加上進行資料轉換的組譯器函式(或內嵌/行內的組譯器程式碼)。

4

## Cortex-M3 裡一些有用的指令

在此介紹從 architecture v7 和 v6 來的幾個有用的 Thumb-2 指令。

### MSR 和 MRS

這兩個指令提供了對 Cortex-M3 裡特殊暫存器的存取。在此為這些指令的語法：

```
MRS <Rn>, <SReg> ; 從特殊暫存器移開  
MSR <SReg>, <Rn> ; 寫入特殊暫存器
```

其中<SReg>可以是表 4-31 的一個選擇。

表 4-31 用在 MRS 與 MSR 指令的特殊暫存器名稱

符號	描述
IPSR	Interrupt status register, 中斷狀態暫存器
EPSR	Execution status register, 執行狀態暫存器
APSR <sup>2</sup>	Flags from previous operation, 先前運算相關旗標
IEPSR	IPSR 與 EPSR 複合體

接下頁

在舊的 ARM Cortex-M3 文件裏, APSR 被稱作 FPSR。如果你使用 Cortex-M3 開發初期階段中發展的較舊的軟體開發工具, 則可能需要在你的組合語言程式裏使用 FPSR 的暫存器名稱。

# Jason 唸書—EETOP 世界唯一貼

符號	描述
IAPSR	IPSR 與 APSR 複合體
EAPSR	EPSR 與 APSR 複合體
PSR	APSR、EPSR 與 IPSR 複合體
MSP	Main stack pointer, 主要堆疊指標
PSP	Process stack pointer, 程序堆疊指標
PRIMASK	正常的例外遮罩暫存器
BASEPRI	正常的例外優先權遮罩暫存器
BASEPRI_MAX	同於正常的例外優先權遮罩暫存器, 但為條件式寫入(新的優先權等級需高於舊的等級)
FAULTMASK	Fault exception mask register, 錯誤例外遮罩暫存器
CONTROL	控制暫存器

例如，下面的程式可用來設定程序堆疊指位器：

```
LDR R0, =0x20008000 ; 把新數值
MSR PSP, R0          ; 寫入程序堆疊指位器(PSP)
```

除非存取 APSR, 否則 MRS 或 MSR 指令僅可使用在特權的模式。如果使用其它的模式，則其運算將被忽略，回傳的讀入(若指令是 MRS)資料為 0。

## IF-THEN

IF-THEN 指令，允許有條件地執行最多至四個連接的指令(稱作 IT 區塊)。IT 使用如表 4-32 其中之一種格式：

表 4-32 可能的 IT 指令語法

IT 區塊裡條件指令的數目	IT 語法	描述
1	IT <cond>	如果 <cond> 為真，則執行下一個指令。各種條件可參考表 4-26。
2	IT<x> <cond>	<x> 可以是 T (true) 或 E (else)。例如： ITT <cond> 如果 <cond> 為真則執行下 2 個指令 ITE <cond> 如果 <cond> 為真則執行下一個指令，如果 <cond> 為假則執行接在下一個指令之後的指令。
3	IT<x><y> <cond>	條件式執行下面三個指令。 <x>, <y> 可以為 T (true) 或 E (else)。
4	IT<x><y><z> <cond>	條件式執行下面四個指令。 <x>, <y> 與 <z> 可以為 T (true) 或 E (else)。

跟隨 IT 指令的下面一個到四個條件指令，需要有與 IT 區塊吻合或相反的條件後綴字(隨著使用的 "T" 與 "E" 序列而定)。使用 IT 指令的範例已經在「組譯器語言：使用 IT 指令的條件式跳躍」一節裡示範過了。

除了直接地使用 IT 指令，從 ARM7TDMI 移植組合語言應用程式到 Cortex-M3 時 IT 指令也有所幫助。當 ARM 組譯器(包括 KEIL RealView Microcontroller Development Kit) 被使用時，並且如果在組合語言程式裡使用了條件式執行指令但並沒有利用 IT 指令，則組譯器會自動地插入所需的 IT 指令。例如表 4-33 所示：

表 4-33 在 ARM 組譯器裡自動插入的 IT 指令

原來的組合語言程式	由產生的工作檔反組譯的組合語言程式
...	...
CMP R1, #2	CMP R1, #2
ADDEQ R0, R1, #1	IT EQ
"	ADDEQ R0, R1, #1
	...

此特性允許在 Cortex-M3 上重新使用許多現存的組合語言程式而不必加以修改。

## CBZ 與 CBNZ

CBZ 指令與 CBNZ 指令將暫存器與 0 作比較，並且對 CBZ 而言，如果暫存器為 0 則條件式跳躍；對 CBNZ 而言，如果結果不為 0 則條件式跳躍。跳躍只能是往前的，且旗標(APSR)並不受這些指令影響。一個使用 CBZ 的簡單迴圈在「組譯器語言：比較與條件式跳躍的結合」一節裡討論過了。

CBNZ 的用法類似於 CBZ，除了是當暫存器為非 0 時作跳越。例如：

```
/* = strchr(email_address_string, '@');
if (x == 0) { // x 為 0 如果@並不在 email_address_string 裡
    show_error_message();
    exit();
}
```

此可編譯為：

# Jason 嘴書—EETOP 世界唯一貼

```

...
BL      strchr          ; strchr 結果傳回 r0 裡
CBNZ   r0, email_looks_valid    ; branch 如果結果不為 0 則跳躍
BL      show_error_message
BL      exit
email_looks_valid
...

```

## SDIV 和 UDIV

有號數和無號數除法指令的語法為：

SDIV.W	<Rd>, <Rn>, <Rm>
UDIV.W	<Rd>, <Rn>, <Rm>

其結果為  $Rd = Rn / Rm$ 。例如：

LDR	R0, = 300	; 十進位 300
MOV	R1, #5	
UDIV.W	R2, R0, R1	

運算結果得到 R2 等於 60(0x3C)。

你可設定 NVIC 組態控制暫存器裡的 DIVBYZERO 位元，當除以 0 出現時，則會顯示一個錯誤例外(用法錯誤)，否則除以 0 出現時，<Rd>將為 0。

## REV、REVH、和 REVSH

REV 把 word 裡的資料以 byte 為單元，作相反排列；REVH 把 half word 裡的資料以 byte 為單元，作相反排列。例如，若 R0 為 0x12345678，執行下列指令後：

REV	R1, R0
REVHR2,	R0

R1 將成為 0x78563412，R2 將成為 0x34127856。REV 和 REVH 對 big endian 和 little endian 之間資料的轉換特別有用。

REVSH 跟 REVH 作用類似。但它僅對低的 half word 作相反排列，而把高的 half word 作正負號的填充。例如，若 R0 為 0x33448899，執行下列指令：

REVSH	R1, R0
-------	--------

R1 將成為 0xFFFF9988。

## RBIT

RBIT 指令把 word 資料，以 bit 為單元，作相反排列。語法如下：

RBIT.W	<Rd>, <Rn>
--------	------------

此指令對處理資料通訊裡連續的位元串流非常有用。例如，若 R0 為 0xB4E10C23 (二進位表示為 1011 0100 1110 0001 0000 1100 0010 0011)，執行下列指令：

RBIT.W	R0, R1
--------	--------

R0 將成為 0xC430872D(二進位表示為 1100 0100 0011 0000 1000 0111 0010 1101)。

## SXTB、SXTH、UXTB、和 UXTH

SXTB、SXTH、UXTB、和 UXTH 四個指令用來將 byte 或 half word 的資料擴充為 word 資料。其語法如下：

SXTB	<Rd>, <Rn>
SXTH	<Rd>, <Rn>
UXTB	<Rd>, <Rn>
UXTH	<Rd>, <Rn>

對 SXTB/SXTH 而言，利用 Rn 的 bit[7]/bit[15]，把資料作正負號的擴充。對 UXTB/UXTH 而言，利用 0 來把資料擴充至 32-bit。

# Jason 嘴書—EETOP 世界唯一貼

例如, 若 R0 為 0x55AA8765 :

```
SXTB    R1, R0 ; R1 = 0x00000065
SXTB    R1, R0 ; R1 = 0xFFFF8765
UXTB    R1, R0 ; R1 = 0x00000065
UXTH    R1, R0 ; R1 = 0x00008765
```

## BFC 和 BFI

BFC(Bit Field Clear)清除暫存器裡任意指定位置相鄰的 1~31 個 bits。其語法如下：

```
BFC.W <Rd>, <#1sb>, <#width>
```

例如：

```
LDR    R0, =0x1234FFFF
BFC.W R0, #4, #8
```

運算結果 R0 = 0x1234F00F。

BFI(Bit Field Insert)會從一個暫存器複製 1~31 個 bits (#width)到另一個暫存器中任意指定的位置(#1sb)。其語法如下：

```
BFI.W <Rd>, <Rn>, <#1sb>, <#width>
```

例如：

```
LDR    R0, =0x12345678
LDR    R1, =0x3355AACC
BFI.W R1, R0, #8, #16 ; 將 R0[15:0] 插入 R1[23:8]
```

運算結果 R1 = 0x335678CC。

## UBFX 和 SBFX

UBFX 和 SBFX 為無號數與有號數的 bit field 摷取指令。其語法如下：

```
UBFX.W <Rd>, <Rn>, <#1sb>, <#width>
SBFX.W <Rd>, <Rn>, <#1sb>, <#width>
```

UBFX 從一個暫存器任意指定的位置(#1sb)開始, 摷取任意指定數目的 bits (#width), 並作 0 的擴充後, 再將它放入目的暫存器。例如：

```
LDR    R0, =0x5678ABCD
UBFX.R1, R0, #4, #8
```

運算結果 R1 = 0x000000BC。

同樣地, SBFX 摷取 bit field, 但作正負號的擴充後, 再將它放入目的暫存器。例如：

```
LDR    R0, =0x5678ABCD
SBFX.R1, R0, #4, #8
```

運算結果 R1 = 0xFFFFFFFBC。

## LDRD 和 STRD

LDRD 和 STRD 將兩個 word 的資料取出或存放到兩個暫存器。其語法如下：

LDRD.W <Rxf>, <Rxf2>, [Rn, #+/-offset]{!}	; 前索引(Pre-indexed)
LDRD.W <Rxf>, <Rxf2>, [Rn], #+/-offset	; 後索引(Post-indexed)
STRD.W <Rxf>, <Rxf2>, [Rn, #+/-offset]{!}	; 前索引(Pre-indexed)
STRD.W <Rxf>, <Rxf2>, [Rn], #+/-offset	; 後索引(Post-indexed)

其中<Rxf>為第一個目的/來源暫存器, <Rxf2>為第二個目的/來源暫存器。

# Jason 嘴書—EE TOP 世界唯一貼

例如，下面的程式從記憶體位址 0x1000 處，把 64-bit 的值讀入 R0 和 R1：

```
LDR      R2, = 0x1000
LDRD.W   R0, R1, [R2]           ; R0 = memory[0x1000]
                                ; R1 = memory[0x1004]
```

同樣的，我們可以使用STRD以存放64-bit值至記憶體。在下面的例子裡，使用了前索引位址模式：

```
LDR      R2, = 0x1000          ; 基底位址
STRD.W   R0, R1, [R2, #0x20]    ; 如此將使得記憶體[0x1000]=R0,
                                ; 記憶體[0x1004]=R1
```

## TBB 和 TBH

TBB(Table Branch Byte)和 TBH(Table Branch Halfword)用以製作跳躍表。TBB 使用以 byte 作位移的跳躍表，TBH 使用以 half word 作位移的跳躍表。因為程式記數器的 bit 0 通常被設定為 0，故先將跳躍表的內容值乘以 2，再加進 PC(即程式記數器)。況且，又因為 PC 的值為正在執行指令的位址加上 4，故 TBB 的跳躍範圍是 $(2 \times 255) + 4 = 514$ ，TBH 的跳躍範圍是 $(2 \times 65535) + 4 = 131074$ 。TBB 和 TBH 僅支援往前的跳躍動作。

TBB 通常使用的語法為：

```
TBB.W   [Rn, Rm]
```

其中 Rn 作為基底記憶體位移，Rm 為跳躍表的索引，故 TBB 指令在跳躍表上的相關項目位於 Rn + Rm 處。如果我們以 PC 當作 Rn，則可看到如圖 4.5 所示的 TBB 運算。

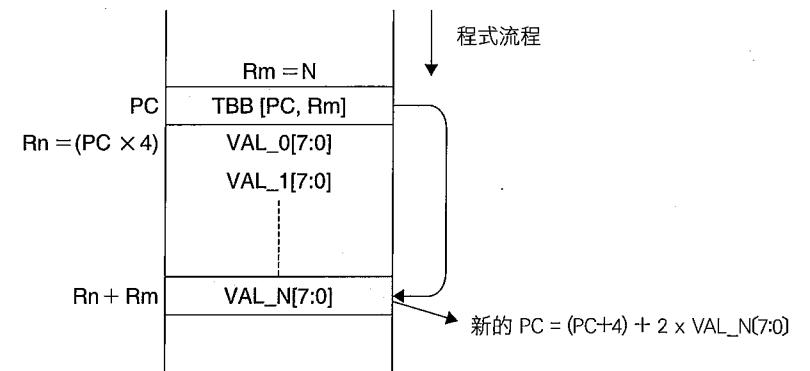


圖 4-5 TBB 運算

TBH 指令，其運算過程類似上述，只是它的跳躍表上相關項目位於  $Rn + 2 \times Rm$  處，並且有更高的跳躍最大位移值。如果我們以 PC 當作 Rn，則可看到如圖 4.6 所示的 TBH 運算。

如果把 table branch 指令中的 Rn 設定為 R15，此時因處理器的管線作用，故 Rn = PC + 4。此兩個指令極可能會被 C 編譯器拿來產生 switch (case) 敘述的程式碼。放在跳躍表的值，為進行中程式記數器的相對值，所以並不容易在組譯器裡，手動編寫跳躍表的內容，因為其相對位址值可能無法在組譯/編譯階段得知，特別是在跳躍目的程式為分開的程式檔案的情形下。計算 TBB/TBH 的跳躍表的內容值所用的語法，可能隨開發工具而定。在 ARM 組譯器 (armasm) 裡，TBB 的跳躍表可用下面的方式產生：

```
TBB.W   [pc, r0]           ; 執行此指令時 PC 等於 branchtable
;
branchtable
DCB ((dest0 - branchtable) / 2) ; DCB : define one or more byte
DCB ((dest1 - branchtable) / 2) ; 在此使用 DCB
DCB ((dest2 - branchtable) / 2) ; 來定義 8-bit 的值
DCB ((dest3 - branchtable) / 2) ;
dest0
...
dest1
...
dest2
...
dest3
... ; 若 r0 = 0 則執行
      ; 若 r0 = 1 則執行
      ; 若 r0 = 2 則執行
      ; 若 r0 = 3 則執行
```

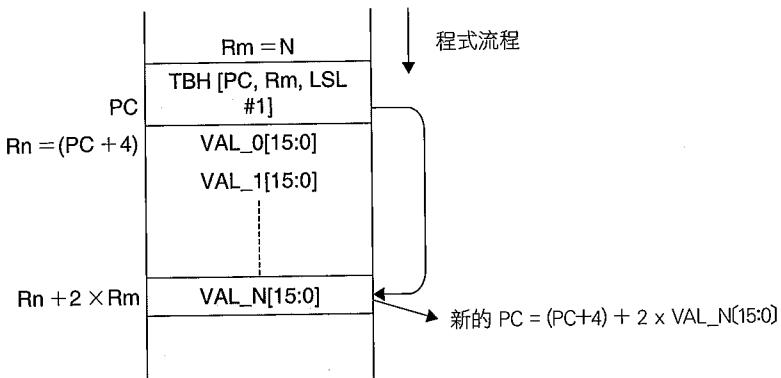


圖 4-6 TBH 運算

當執行 TBB 指令時，其進行中的 PC 值是 branchtable 的位址(因為處理器有管線作用)。同樣的，TBH 指令可以如下使用：

```
TBH.W [pc, r0, LSL #1]
branchtable
DCI ((dest0 - branchtable) / 2) ; DCI : define one or more half word
DCI ((dest1 - branchtable) / 2) ; 在此使用 DCI
DCI ((dest2 - branchtable) / 2) ; 來定義 16-bit 的值
DCI ((dest3 - branchtable) / 2) ;
dest0
...
dest1
...
dest2
...
dest3
...
```

## Chapter

## 5

## 記憶體系統

本章內容包括：

- ✓ 記憶體系統特性概觀
- ✓ 記憶體映射
- ✓ 記憶體存取的屬性
- ✓ 預設的記憶體存取權限
- ✓ Bit-Band 運算
- ✓ 非對齊傳輸
- ✓ 獨占存取
- ✓ Endian 模式

## 記憶體系統特性概觀

Cortex-M3 處理器的記憶體結構不同於傳統 ARM 處理器。首先，它具有預先定義的記憶體映射，指定了存取一個記憶體位置時，所要使用的匯流排介面。此特性也使得處理器的設計可以在存取不同的元件時，最佳化存取的行為。

Cortex-M3 記憶體系統另一個特性為支援了 bit-band，此特性提供了記憶體或周邊裡 bit 資料不可切割單元的(atomic)運算。此 bit-band 運算僅於特殊記憶體區域支援。本章後面將更詳細地探討此主題。

Cortex-M3 記憶體系統，也支援了非對齊(unaligned)傳輸和獨占的(exclusive)存取等方式。這些特性為 v7-M 結構的一部分。最後，Cortex-M3 支援 little endian 和 big endian 兩種記憶體組態。

## 記憶體映射

Cortex-M3 處理器有著固定的記憶體映射，這使得 Cortex-M3 產品之間程式的移植變得簡單。例如，先前章節提及的元件如 NVIC 和 MPU，在所有的 Cortex-M3 產品有著相同的記憶體位置。然而，記憶體映射的定義具有相當大的彈性，故個別廠商可以使用不同於其他廠商的方式，定義其基於 Cortex-M3 的產品。

有些記憶體位置是配置給私有的周邊，例如除錯元件，它們位於私有周邊記憶體區域。這些除錯元件包括：

- ◆ 摷取補丁和中斷點單元(Fetch Patch and BreakPoint Unit, FPB)
- ◆ 資料觀察點和追蹤單元(Data WatchPoint and Trace Unit, DWT)
- ◆ 設備追蹤巨集格(�器化追蹤宏集格, ITM)
- ◆ 嵌入追蹤巨集格(�器化追蹤宏集格, ETM)
- ◆ 追蹤埠介面單元(Trace Port Interface Unit, TPIU)
- ◆ ROM 表

這些元件的細節，將於後面除錯功能的各章討論。

Cortex-M3 處理器擁有總共 4GB 大小的位址空間。程式碼可置於程式碼區、SRAM 區、或外部 RAM 區等。然而，最好將程式放在程式碼區，因為在這樣的安排下，指令擷取與資料存取，可於不同的匯流排介面同時進行。

SRAM 記憶體範圍是作為聯繫內部 SRAM 使用。經由系統介面匯流排，可執行對此區域的存取。在此區域中，有個 32MB 的範圍是定義為 bit-band alias。在這個 32MB bit-band alias 記憶體範圍內，每一個 word 位址代表了在 1Mb bit-band 區域中的一個 bit。對此記憶體區域作資料寫入的存取，將被轉換為對 bit-band 區域一個不可切割的 READ-MODIFY-WRITE 運算，使得程式可以去設定或清除記憶體裡的個別 bit 資料。Bit-band 運算僅適用於資料存取，而不能用在指令擷取。藉著把布林資訊(單獨 bits)放進 bit-band 區域裡，我們可以把多個布林資訊包裹在一個 word 裡，並藉著 bit-band alias 存取個別布林資訊，在不需以軟體處理 READ-MODIFY-WRITE 的同時，節省記憶體空間。本章後面會介紹更多 Bit-band alias 細節。

另外有 0.5GB 的位址範圍區塊是配置給晶片上的 (on-chip)周邊。與 SRAM 區域相似，此區域支援 bit-band alias 且可藉著系統匯流排介面存取。然而，在此區域並不允許執行指令。周邊區域對 bit-band 的支援，使得更易存取周邊，或改變周邊的控制與狀態 bits，故更易規劃周邊控制。



圖 5-1 Cortex-M3 預先定義的記憶體映射

有兩個各 1GB 的記憶體空間，是配置給外部 RAM 與外部設備使用；此兩者的不同在於程式不可於外部設備區域中執行，以及其快取方式有一些相異。

最後的 0.5GB 的記憶體配置給系統層級的元件、內部周邊匯流排、外部周邊匯流排、以及廠商特定的系統周邊等等。私有周邊匯流排包括了兩個區段：

- ◆ **AHB 私有周邊匯流排(僅作 Cortex-M3 內部 AHB 周邊使用)：**此包括了 NVIC、FPB、DWT、ITM 等。
- ◆ **APB 私有周邊匯流排(給 Cortex-M3 內部 APB 設備，以及位於 Cortex-M3 處理器外部的周邊來使用)：**Cortex-M3 允許晶片廠商藉由 APB 介面，於此 APB 私有周邊匯流排上，增加額外的 on-chip APB 周邊。

NVIC 位於一個稱作系統控制空間(System Control Space, SCS)的記憶體區域。除了提供中斷控制功能外，此區域也為 SYSTICK、MPU、和程式除錯控制等，提供了控制暫存器。

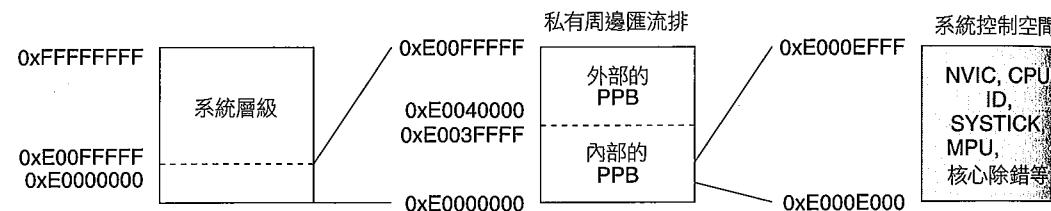


圖 5-2 系統控制空間

剩下未使用的廠商特定記憶體範圍，則可經由系統匯流排介面存取。然而，指令不可於此區域執行。

Cortex-M3 處理器也有一個可供選用的 MPU。晶片製造廠可決定是否將 MPU 包含進其產品中。

如上所顯示的，僅是記憶體映射的樣板。個別的半導體廠商將提供記憶體映射的細節，包括 ROM 和 RAM 實際的位置與大小，以及周邊記憶體的位置。

## 記憶體存取的屬性

記憶體映射顯示了每一個記憶體區域所包含的東西。除了用來解譯所存取的記憶體區塊與設備外，記憶體映射也定義了存取時的記憶體屬性。Cortex-M3 處理器的記憶體屬性包括如下：

- ◆ **可緩衝性：**當處理器繼續下一個指令的執行時，從記憶體寫入到記憶體的動作可被一個寫入緩衝器所延遲。
- ◆ **可快取性：**從記憶體讀取而獲得的資料可以複製到快取記憶體，因此下次可從快取存取此資料以加速程式的執行。
- ◆ **可執行性：**處理器可由此記憶體區域擷取與執行程式碼。
- ◆ **可分享性：**此記憶體區域的資料可以被多重的匯流排 master 所共享。記憶體系統需要確保可分享記憶體區域在不同的匯流排 master 之間的資料一致性。

若有 MPU，且其區域又與預設值規劃不同，則預設的記憶體屬性設定可被覆蓋。雖然事實上，Cortex-M3 並不具有快取記憶體，或快取控制器，但可加上一個外部的快取。此快取屬性可能也會影響晶片上(on-chip) 和晶片外(off-chip) 記憶體控制器的運作，這會隨晶片製造所用的記憶體控制器而定：

- ◆ **程式碼記憶體區域(0x00000000-0x1FFFFFFF)：**此為可執行區域，並且快取屬性為 WT(即 Write Through)。你也可將資料記憶體置於此區域。當資料運算對象是在此區域時，則會經由資料匯流排進行。在此區域，寫入(Write)動作是可緩衝的。
- ◆ **SRAM 記憶體區域(0x20000000-0x3FFFFFFF)：**此區域為 on-chip RAM 使用。寫入動作被緩衝，並且快取屬性為 WB-WA(Write Back, Write Allocated)。此為可執行區域，故可將程式碼複製至此執行。
- ◆ **周邊區域(0x40000000-0x5FFFFFFF)：**此區域為周邊使用。存取動作為不可快取的。你不可在此區域執行指令程式碼(ARM 文件，例如 Cortex-M3 TRM 顯示為 Execute Never，即 XN，表示永不執行)。
- ◆ **外部 RAM 區域(0x60000000-0x7FFFFFFF)：**此區域為 on-chip 或 off-chip 記憶體使用。存取動作為可快取的(WT-Write Through)。你可於此區域執行指令。

- ◆ 外部 RAM 區域(0x80000000-0x9FFFFFFF)：此區域為 on-chip 或 off-chip 記憶體使用。存取動作為可快取的(WT-即 Write through)。你可於此區域執行指令。
- ◆ 外部設備(0xA0000000-0xBFFFFFFF)：此區域指定給外部設備 and/or 需要以有次序/不可緩衝的方式存取的共享記憶體。此亦為不可執行區域。
- ◆ 外部設備(0xC0000000-0xDFFFFFFF)：此區域指定給外部設備 and/or 需要以有次序/不可緩衝的方式存取的共享記憶體。此亦為不可執行區域。
- ◆ 系統區域(0xE0000000-0xFFFFFFFF)：此區域指定給私有周邊與廠商特定設備來使用。此為不可執行區域。在私有周邊匯流排記憶體區域，存取為高度有次序的(不可快取且不可緩衝)；在廠商特定記憶體區域，存取為可緩衝但不可快取。

注意，自從 Cortex-M3 Revision 1 以後，程式碼區域對外輸出到外接的記憶體系統的記憶體屬性，已經硬體設定為可快取但是不可緩衝。此設定不能使用 MPU 組態作覆蓋。此改變僅僅影響在處理器外的記憶體系統之行為；在處理器裡仍然可以藉著寫入傳輸到程式區以使用寫入緩衝器。

## 預設的記憶體存取權限

Cortex-M3 的記憶體映射有一個記憶體存取權限的預設組態，可預防應用程式存取系統控制記憶體空間(例如 NVIC)。預設的記憶體存取權限，使用於下面兩種條件之一：

- ◆ 沒有 MPU
- ◆ 有 MPU，但未致能

如果有 MPU 且致能的情形下，則 MPU 設定裡的存取權限，將決定是否允許使用者存取。

表 5-1 顯示了預設的記憶體存取權限。

表 5-1 預設的記憶體存取權限

記憶體區域	位址	用戶程式的存取
供應廠商特定	0xE0100000-0xFFFFFFFF	全存取
ROM 表	0xE00FF000-0xE00FFFFF	阻擋；用戶存取在匯流排造成錯誤
外部 PPB	0xE0042000-0xE00FEFFF	阻擋；用戶存取在匯流排造成錯誤
ETM	0xE0041000-0xE0041FFF	阻擋；用戶存取在匯流排造成錯誤
TPIU	0xE0040000-0xE0040FFF	阻擋；用戶存取在匯流排造成錯誤
內部 PPB	0xE000F000-0xE003FFFF	阻擋；用戶存取在匯流排造成錯誤
NVIC	0xE000E000-0xE000EFFF	阻擋；用戶存取在匯流排造成錯誤，除非編程軟體觸發中斷暫存器以允許用戶存取
FPB	0xE0022000-0xE0003FFF	阻擋；用戶存取在匯流排造成錯誤
DWT	0xE0001000-0xE0001FFF	阻擋；用戶存取在匯流排造成錯誤
ITM	0xE0000000-0xE0000FFF	允許讀取；除非致能了用戶存取的激勵埠，否則忽略寫入
外部設備	0xA0000000-0xDFFFFFFF	全存取
外部 RAM	0x60000000-0x9FFFFFFF	全存取
周邊	0x40000000-0x5FFFFFFF	全存取
SRAM	0x20000000-0x3FFFFFFF	全存取
程式	0x00000000-0x1FFFFFFF	全存取

當使用者存取被阻擋時，將立即出現錯誤例外。

## Bit-Band 運算

Bit-Band 運算的支援允許透過單一載入/儲存運算，對單一 bit 資料作存取(讀/寫)。在 Cortex-M3 裡，此以預先定義，稱作 bit-band 區域的記憶體區域來支援。其中之一，位於 SRAM 區域第一個 1MB 處，另一個則位於周邊區域第一個 1MB 處。此兩個記憶體區域，可當作正常的記憶體般地存取，但也可經由一個分開的，稱作 bit-band alias 的區域來存取。當使用 bit-band alias 位址時，可以藉由以 word 為單元對齊(word-aligned)的位址中的最低位元 LSB 來存取個別 bit。

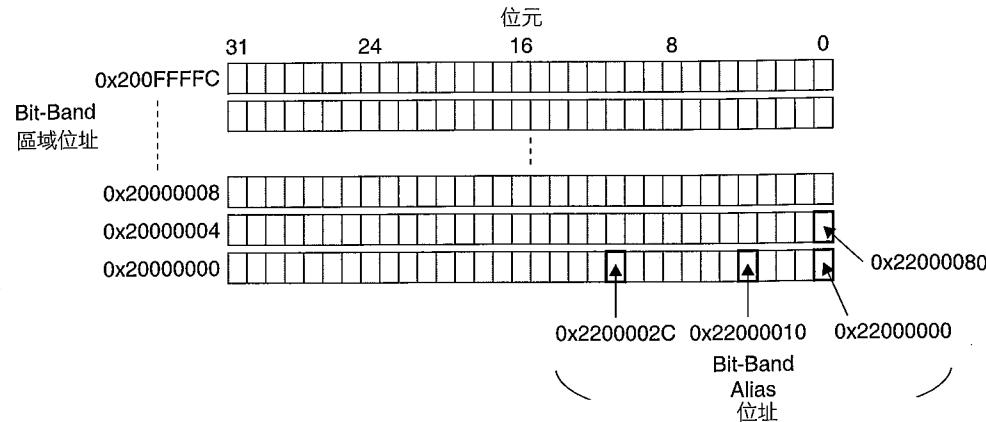


圖 5-3 經由 Bit-Band Alias 對 Bit-Band 區域作 Bit 存取

例如，欲設定位址 0x20000000 處 word 資料裡 bit 2。一種做法是：分別使用三個指令去讀資料、設定 bit、再將結果寫回。反之此工作可以使用一個單獨的指令來執行(參考圖 5-4)。

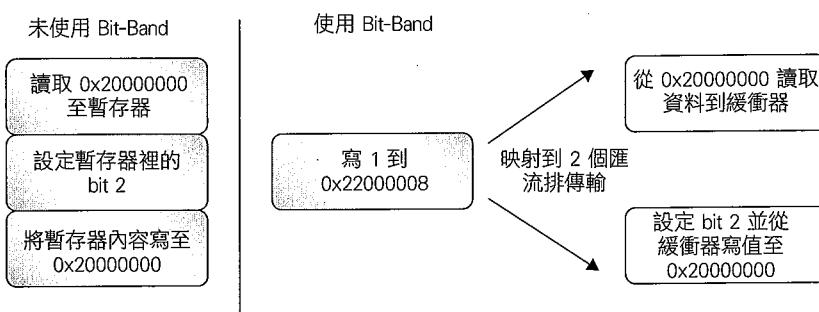


圖 5-4 寫入 Bit-Band Alias

上述兩種做法的組譯器序列可如圖 5-5 所示。

未使用 Bit-Band		使用 Bit-Band	
LDR	R0, =0x20000000 ; 設定位址	LDR	R0, =0x22000008 ; 設定位址
LDR	R1, [R0] ; 讀值	MOV	R1, #1 ; 設定資料
ORR.W	R1, #0x4 ; 修改位元	STR	R1, [R0] ; 寫回結果
STR	R1, [R0] ; 寫回結果		

圖 5-5 使用或不使用 Bit-Band 寫入一個 Bit 的組譯器序列範例

同樣地，當我們需要從一個記憶體位置讀一個 bit 時，bit-band 的支援也可簡化應用程式。例如，如果我們需要決定位址 0x20000000 處 bit 2 的值，我們可採用圖 5-6 所示的步驟。

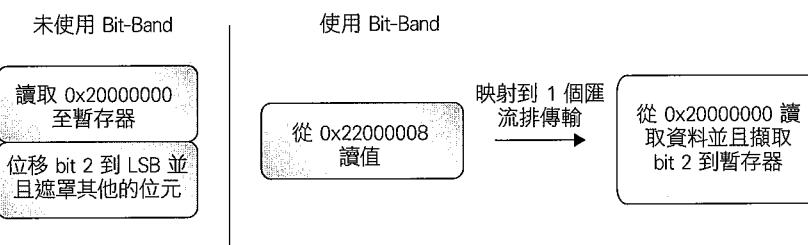


圖 5-6 從 Bit-Band Alias 讀資料

上述兩種做法的組譯器序列可如圖 5-7 所示。

未使用 Bit-Band	使用 Bit-Band
DR R0, =0x20000000 ; 設定位址	LDR R0, =0x22000008 ; 設定位址
LDR R1, [R0] ; 讀值	LDR R1, [R0] ; 讀值
UBFX.W R1, R1, #2, #1 ; 擷取 bit[2]	

圖 5-7 從 Bit-Band Alias 讀資料

Bit-band 運算不是一個新的想法，事實上，在 8-bit 微控制器(例如 8051)裡，有一個存在了超過 30 年的類似功能。雖然 Cortex-M3 並沒有作 bit 運算的特殊指令，但經由定義特殊的記憶體區域，使得對這些區域作資料存取時，將自動轉換為 bit-band 的運算。

注意，Cortex-M3 使用下列名詞稱呼 bit-band 記憶體位址的用法：

- ◆ **Bit-band 區域**:此為支援 bit-band 運算的記憶體位址的區域。
- ◆ **Bit-band alias**:存取 bit-band alias 將造成對 bit-band 區域的存取(一個 bit-band 運算)。(注意：記憶體需執行再映射的動作)

# Jason 嘴書—EE TOP 世界唯一貼

在 bit-band 區域裡, 一個 word 以 bit-band alias 範圍裡的 32 個 words 的 LSB 來表示。實際動作如下：當存取一個 bit-band alias 位址時, 將根據此位址再映射到相對應的 bit-band 位址。如果是讀取的運算, bit-band 位址存放的 word 資料將被讀取, 並且把選定的 bit 位置的值, 移到讀取返回資料(read return data)的 LSB 裡。如果是寫入的運算, 寫入的 bit 資料, 會被移到對應要求的 bit 位置上, 然後一口氣執行了 READ-MODIFY-WRITE 動作。

有兩個記憶體區域可作 bit-band 運算：

- ◆ 0x20000000-0x200FFFFF (SRAM, 1Mb)
- ◆ 0x40000000-0x400FFFFF (周邊, 1Mb)

表 5-2 顯示 SRAM 記憶體區域裡, bit-band alias 再映射的對照表。

表 5-2 SRAM 區域 bit-band 位址的再映射

Bit-Band 區域	Aliased Equivalent
0x20000000 bit[0]	0x22000000 bit[0]
0x20000000 bit[1]	0x22000004 bit[0]
0x20000000 bit[2]	0x22000008 bit[0]
...	...
0x20000000 bit[31]	0x2200007C bit[0]
0x20000004 bit[0]	0x22000080 bit[0]
...	...
0x20000004 bit[31]	0x220000FC bit[0]
...	...
0x200FFFFC bit[31]	0x23FFFFFFC bit[0]

同樣地, 如表 5-3 所示, 周邊記憶體區域的 bit-band 區域, 也可藉由 bit-band alias 位址來存取。

表 5-3 周邊記憶體區域 bit-band 位址的再映射

Bit-Band 區域	Aliased Equivalent
0x40000000 bit[0]	0x42000000 bit[0]
0x40000000 bit[1]	0x42000004 bit[0]
0x40000000 bit[2]	0x42000008 bit[0]
...	...
0x40000000 bit[31]	0x4200007C bit[0]
0x40000004 bit[0]	0x42000080 bit[0]
...	...
0x40000004 bit[31]	0x420000FC bit[0]
...	...
0x400FFFFC bit[31]	0x43FFFFFFC bit[0]

在此舉一個簡單的例子：

1. 位址 0x20000000 內容值設定為 0x3355AAC
2. 讀取位址 0x22000008 的值。此讀取的動作被再映射而去讀取 0x20000000。其回傳值為 1 (bit[2] of 0x3355AAC)。
3. 把 0x0 寫入 0x22000008。此寫入存取動作被再映射而針對 0x20000000 一口氣做了 READ-MODIFY-WRITE 動作。先是從記憶體讀出 0x3355AAC 的值, 再把 bit[2]的值清除為 0, 最後把結果 0x3355AAC8 寫回位址 0x20000000 處。
4. 接著, 再讀取位址 0x20000000 的值。你將得到的回傳值為 0x3355AAC8(bit[2]被清除了)。

當你存取 bit-band alias 位址時, 僅用到資料的 LSB(即 bit[0])。此外, 不可用非對齊(unaligned)的方式去存取 bit-band alias 區域。如果對 bit-band alias 位址範圍做了非對齊存取, 會得到不可預測的結果。

## Bit-Band 運算的優點

那麼, 使用 bit-band 運算有何好處？舉例來說, 我們可以利用它們, 在一般用途輸入/輸出(general-purpose input/output, GPIO)埠上, 實現將串列資料傳輸到串列設備上。因為可以分開存取串列資料與時脈信號, 故可簡單地實現上述的應用程式。

### Bit-Band 對照 Bit-Bang

在 Cortex-M3 裡, 我們使用術語 bit-band, 意指其特性為一個特別的記憶體帶(band, 或 region)提供了 bit 的存取。Bit-bang 則通常指在軟體控制下, 驅動 I/O 接腳以提供串列通信功能。Cortex-M3 裡的 bit-band 特性可用以實現 bit-banging, 但此兩個術語的定義不同。

Bit-band 運算也可用來簡化跳躍決定。例如, 如果根據周邊狀態暫存器的一個 bit 的值, 作執行跳躍的判斷, 可如下動作：

- ◆ 讀取整個暫存器
- ◆ 遮罩不需要的 bits
- ◆ 比較後作跳躍

相反地, 你可簡化整個動作為：

- ◆ 經由 bit-band alias 去讀取狀態 bit (得到 0 或 1)
- ◆ 比較後作跳躍

除了以更少的指令提供了更快的 bit 運算, 在資源被兩個以上的程序分享的情形下, Cortex-M3 裡 bit-band 特性也非常地重要。不可分割(atomic)是 Bit-band 運算中, 一個非常重要的優點。換句話來說, READ-MODIFY-WRITE 執行序列, 不會被其它匯流排的活動打斷。如果沒有這樣的行為模式, 例如假設使用的是軟體 READ-MODIFY-WRITE 序列, 下面的問題將可能發生：考慮一個簡單的輸出埠, 其 bit 0 為主程式所用, 其 bit 1 為一個中斷處理程式所用, 一個基於軟體的 READ-MODIFY-WRITE 運算可能會造成如圖 5-8 所示的資料碰撞。

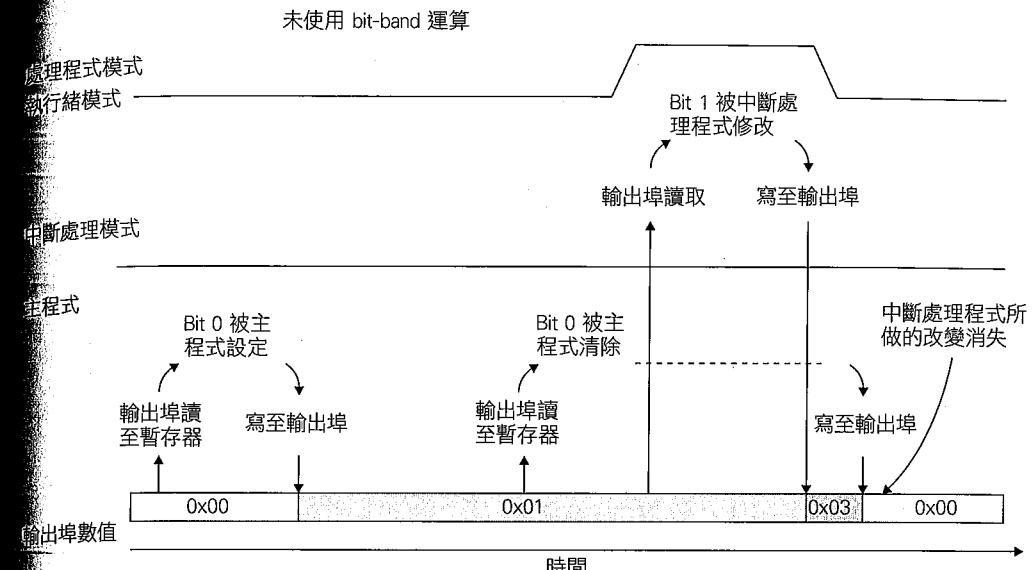


圖 5-8 當例外處理程式修改了分享的記憶體位置, 可能會造成資料遺失

使用 Cortex-M3 bit-band 特性, 可避免這類的競賽(race)情形, 其原因是 READ-MODIFY-WRITE 在硬體層級實現且為不可分割的(兩次傳輸動作不可被分割執行), 因而不會在讀與寫中間發生中斷(參考圖 5-9)。

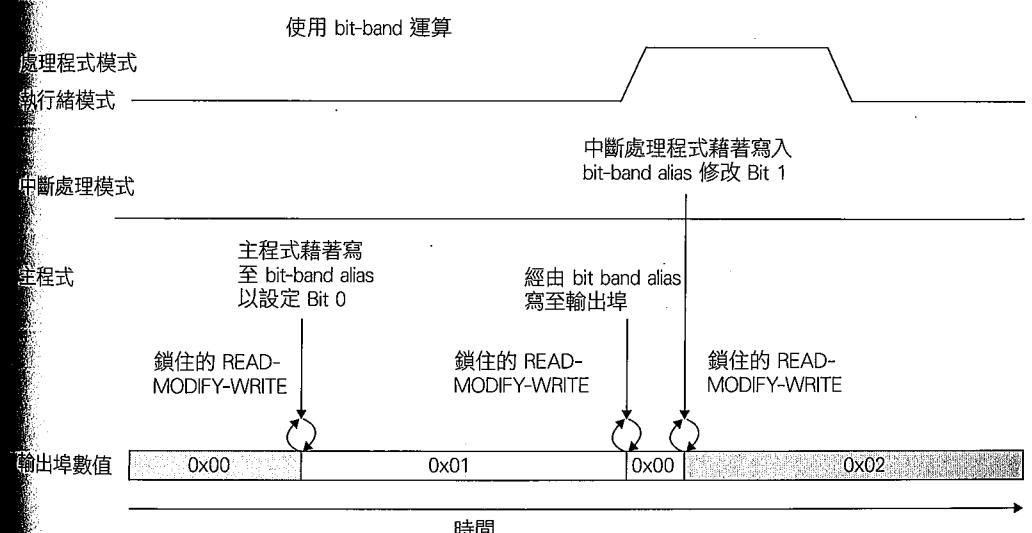


圖 5-9 使用 Bit-Band 的特性做鎖住的傳輸, 以防止資料遺失

多工系統也會看到相同的事情。例如，如果輸出埠 bit 0 為 A 程序所用，bit 1 為 B 程序所用，一個基於軟體的 READ-MODIFY-WRITE 運算可能會造成資料碰撞(參考圖 5-10)。

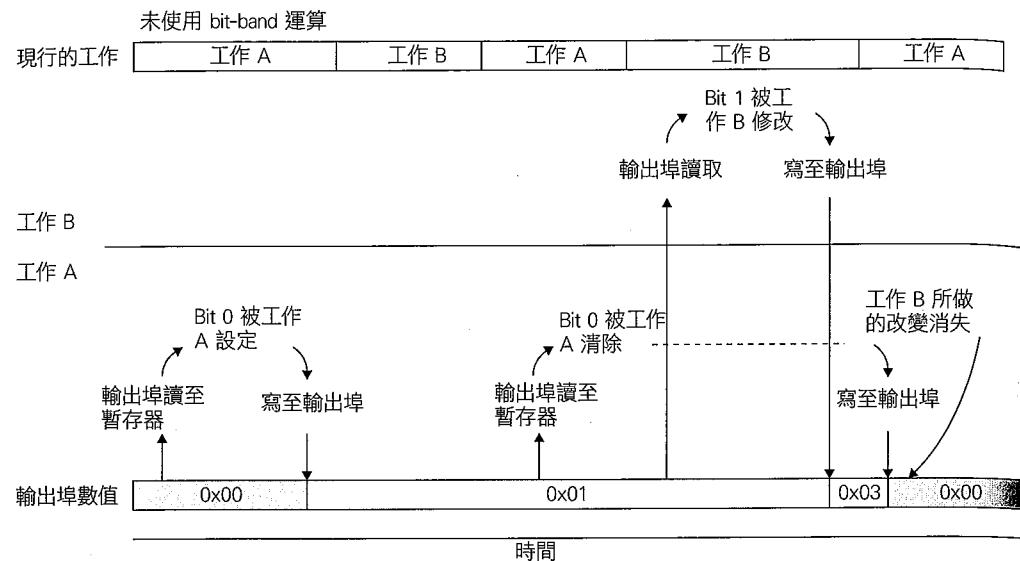


圖 5-10 當不同的工作修改了共用的記憶體位置時，資料將會遺失

再一次地，bit-band 特性可保證任一個工作的存取是分開的，故避免了資料碰撞(參考圖 5-11)。

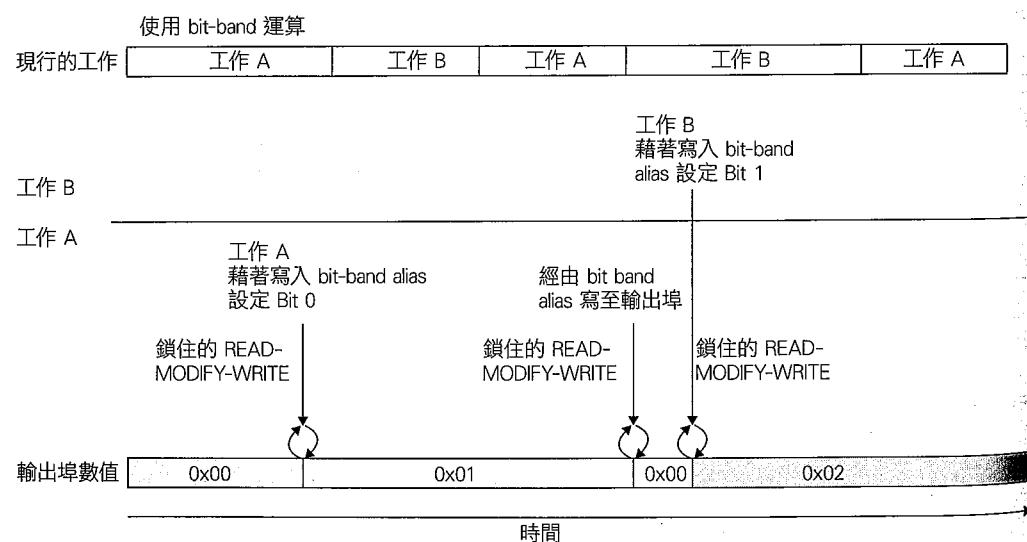


圖 5-11 使用 Bit-Band 的特性做鎖住的傳輸，以防止資料遺失

除了 I/O 功能外，可使用 bit-band 特性以作 SRAM 區域布林資料的儲存與處理。例如，可以把多個布林變數包裹在一個記憶體位置以節省記憶體空間；然而，藉由 bit-band alias 位址區域範圍作存取，依然可以完全分開地存取任一個位元。

如果 SoC 設計工程師要設計一個可作 bit-band 的元件，其元件記憶體位址應該要位於 bit-band 記憶體內，並且要檢查從 AHB 介面出來的鎖住(lock)信號(HMASTLOCK)，以確保當執行一個鎖住的傳輸時，除非經由匯流排，否則不會更改可寫入的暫存器內容。

## 不同資料大小的 Bit-Band 運算

Bit-band 運算並不僅限於 word 傳輸，它也可以 byte 傳輸，或者 half word 傳輸。例如，當一個 byte 存取指令(LDRB/STRB)被用來存取一個 bit-band alias 位址範圍時，則產生至 bit-band 區域的存取將會為 byte 大小，同樣的情形也可以應用到 half word 傳輸(LDRH/STRH)。當你使用非 word 的傳輸至 bit-band alias 位址時，其位址值應該依然以 word 對齊。

5

## C 程式中的 Bit-Band 運算

C 編譯器對於 bit-band 運算並沒有原生的支援。例如，C 編譯器並不知道相同的記憶體可以用不同的位址來存取，也不會知道存取 bit-band alias 僅會存取記憶體位置內資料的 LSB。要在 C 裡面使用 bit-band 特性，最簡單的解決之道，是去分開地宣告記憶體位置的位址，與它的 bit-band alias。例如：

```
#define DEVICE_REG0 ((volatile unsigned long *) (0x40000000))
#define DEVICE_REG0_BIT0 ((volatile unsigned long *) (0x42000000))
#define DEVICE_REG0_BIT1 ((volatile unsigned long *) (0x42000004))

DEVICE_REG0 = 0xAB; //以正常的位址存取硬體暫存器
// ...

DEVICE_REG0 = *DEVICE_REG0 | 0x2 //設定 bit 1,
//但沒用到 bit band 特性

DEVICE_REG0_BIT1 = 0x1 //設定 bit 1，藉由 bit-band alias 位址
//使用 bit band 特性
```

也可以開發 C 巨集，以方便 bit-band alias 的存取。例如，我們可以設定一個巨集，把 bit-band 位址和 bit 數轉換為 bit-band alias 位址，並設定另一個巨集，將位址值當作指標，以存取記憶體位置：

```
// 把 bit band 位址及 bit 數轉換為 bit band alias 位址
#define BITBAND (addr, bitnum) (((addr & 0xF0000000) + 0x20000000 + ((addr &
0xFFFF) <<5) + (bitnum <<2))

// 轉換位址為指標
#define MEM_ADDR (addr) * ((volatile unsigned long *) (addr))
```

根據上面的例子，其程式可改寫如下：

```
#define DEVICE_REG0 0x40000000
#define BITBAND (addr, bitnum) (((addr & 0xF0000000) + 0x20000000 + ((addr &
0xFFFF) <<5) + (bitnum <<2))
#define MEM_ADDR (addr) * ((volatile unsigned long *) (addr))

...
MEM_ADDR (DEVICE_REG0) = 0xAB; //以正常的位址
//存取硬體
...
// 設定 bit 1, 但不使用 bit-band 特性
MEM_ADDR (DEVICE_REG0) = MEM_ADDR (DEVICE_REG0) | 0x2;
...
// 設定 bit 1, 使用 bit-band 特性
MEM_ADDR (BITBAND (DEVICE_REG0, 1)) = 0x1;
```

注意當使用 bit-band 特性時，被存取的變數應該宣告為 volatile 的性質。C 編譯器並不知道同一筆資料會以不同的位址存取，而 volatile 可用來確保每次存取一個變數時，都會到記憶體位置存取，而非存取處理器裡局部備分的資料。

使用 ARM Application Note 179 (Ref 7)中的 ARM RealView Compiler Tools 3.0，你可找到更多以 C 巨集作 bit-band 存取的範例。

## 非對齊傳輸

Cortex-M3 支援以非對齊的(unaligned)傳輸作個別資料存取。資料記憶體存取可定義為對齊的(aligned)與非對齊的。傳統上，ARM 處理器(例如 ARM7/ARM9)

ARM10)僅允許對齊的傳輸。這意味著在存取記憶體時，傳輸一個 word 時需要位址的 bit[1]和 bit[0]為 0，傳輸一個 half word 需要位址的 bit[0]為 0。舉個例子，word 資料可以位於 0x1000 與 0x1004，但不可位於 0x1001、0x1002, 0x1003 等處。若是 half word 資料，位址可為 0x1000 或 0x1002，但不能為 0x1001。

那麼，非對齊傳輸看起來像什麼？圖 5-12 至圖 5-16 顯示了幾個例子。

假設記憶體內部結構為 32-bit (4 bytes)寬，一個非對齊傳輸，可能如圖 5-12 至圖 5-14 所示：以 word 大小的讀取/寫入，且位址並非 4 的倍數。或者如圖 5-15 至圖 5-16 所示：傳輸是以 half word 大小進行，且位址並非 2 的倍數。

	Byte 3	Byte 2	Byte 1	Byte 0
位址 N+4				(31:24)
位址 N	(23:16)	(15:8)	(7:0)	

圖 5-12 非對齊的傳輸，例 1

	Byte 3	Byte 2	Byte 1	Byte 0
位址 N+4			(31:24)	(23:16)
位址 N	(15:8)	(7:0)		

圖 5-13 非對齊的傳輸，例 2

	Byte 3	Byte 2	Byte 1	Byte 0
位址 N+4		(31:24)	(23:16)	(15:8)
位址 N	(7:0)			

圖 5-14 非對齊的傳輸，例 3

	Byte 3	Byte 2	Byte 1	Byte 0
位址 N+4				
位址 N		(15:8)	(7:0)	

圖 5-15 非對齊的傳輸，例 4

	Byte 3	Byte 2	Byte 1	Byte 0
位址 N+4				(15:8)
位址 N	(7:0)			

圖 5-16 非對齊的傳輸，例 5

Cortex-M3 裡, 所有 byte 大小的傳輸皆為對齊的, 因其最小的位址間距為 1 byte。

在 Cortex-M3 裡, 正常的記憶體存取支援非對齊傳輸(例如 LDR、LDRH、STR、STRH 等指令)。但有一些限制如下：

- ◆ Load/Store 多重指令, 並不支援非對齊傳輸。
- ◆ 堆疊運算(PUSH/POP)需為對齊的。
- ◆ 獨占的(exclusive)存取(例如 LDREX 或 STREX) 需為對齊的; 否則將觸發一個錯誤例外(用法錯誤)。
- ◆ Bit-band 運算並不支援非對齊傳輸, 如果你嘗試去做, 將得到不可預測的結果。

當使用非對齊傳輸時, 實際上會被處理器的匯流排介面單元轉換為多個對齊傳輸。此轉換為通透性的, 故並不需要程式師去傷腦筋。然而, 當發生非對齊傳輸時, 將被打散為數個傳輸, 需要花費更多的時脈週期, 對要求高效能表現並非好事。欲得到最佳表現, 就值得去確認一下資料是否適當地對齊。

您也可以設定 NVIC, 在非對齊傳輸發生時觸發例外。此可以藉著設定 NVIC(0xE000ED14)裡的組態控制暫存器中的 UNALIGN\_TRP (Unaligned Trap) bit。如此, 當非對齊傳輸發生時, Cortex-M3 將產生用法錯誤例外。這在軟體開發階段可用來測試應用程式是否會產生非對齊傳輸。

## 獨占存取

你可能注意到了 Cortex-M3 並沒有 SWP 指令(swap), 此指令在傳統的 ARM 處理器(例如 ARM7TDMI)用作號誌(semaphore)的運算。此運算現已被獨占存取所替換。v6 架構(例如 ARM1136 中)第一次支援了獨占存取。

號誌通常用來把分享的資源配置到應用程式。當資源被程序使用時, 它將被此程序鎖住, 直到鎖打開為止, 都不能對其它程序服務。為了設定號誌, 一個記憶體位址被定義為鎖旗標(lock flag), 以顯示分享的資源是否被程序鎖住。當程序或應用程式需要使用分享資源時, 首先需要檢查此資源是否已被鎖住。如果資源尚未被使用, 則可設定鎖旗

標用以顯示資源此時已被鎖住。在傳統的 ARM 處理器裡, 是以 SWP 指令執行對鎖旗標的存取。SWP 指令使得鎖旗標的讀取與寫入, 成為不可分割的動作, 因而防止了資源同時被兩個程序鎖住。

在新的 ARM 處理器中, 讀取/寫入的存取可於分開的匯流排上執行。在這樣的狀況下, SWP 指令不再能夠用來讓記憶體存取成為不可分割的動作, 這是因為被鎖住的傳輸序列裡, 讀取與寫入必須使用同一個匯流排。因此, 鎖住傳輸為獨占存取所替代。獨占存取運算的觀念非常簡單, 但是不同於 SWP; 它允許號誌的記憶體位置能被其它匯流排 master 或執行於相同處理器的程序存取(參考圖 5-17 )。

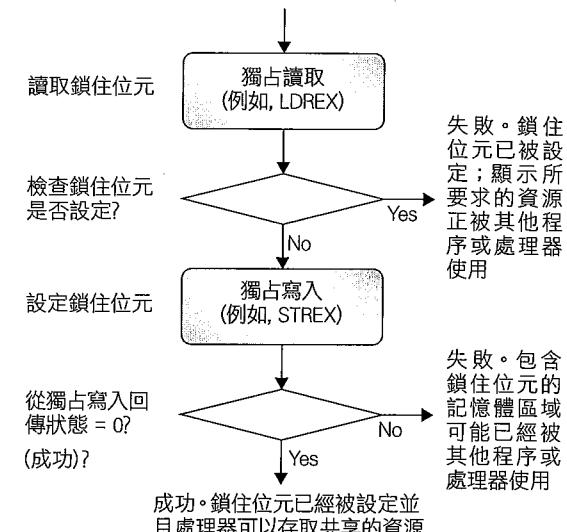


圖 5-17 於號誌中, 使用獨占存取

如果在獨占讀取與獨占寫入之間, 記憶體元件被另一個匯流排 master 存取, 則獨占讀取監控器, 會在處理器嘗試作獨占寫入時, 經由匯流排系統, 標示一個獨占失敗。這將使得獨占寫入的回傳狀態為 1。為了監視多個匯流排 master 的系統(例如多重處理器的設計)的獨占存取, 則需要額外的監控器硬體, 並且需要將此連結到處理器匯流排介面上的獨占存取信號。在 Cortex-M3 處理器裡, D-Code 匯流排(稱作 EXREQD 和 EXRESPD)與系統匯流排(EXREQS 和 EXRESPS)有可使用的獨占存取信號。而作為指令擷取的 I-Code 匯流排並沒有獨占存取信號。

在 Cortex-M3 裡, 獨占存取指令包括 LDREX (word)、LDREXB (byte)、LDREXH (half word)、STREX (word)、STREXB (byte)、STREXH (half word) 等。一個簡單的語法例子如下：

LDREX	<Rxf>, [Rn, #offset]
STREX	<Rd>, <Rxf>, [Rn, #offset]

# Jason 嘴書—EETOP 世界唯一貼

其中<Rd>為獨占寫入的回傳狀態 (0 = 成功, 1 = 失敗)。在第十章中可找到獨占存取的範例程式。

當使用獨占存取時, Cortex-M3 汇流排介面內部的寫入緩衝器將被忽略, 即使 MPU 已定義此區域為可緩衝的。這樣可保證實體記憶體號誌的資訊永遠是最新的, 並且在匯流排 masters 之間是一致的。在多重處理器系統使用 Cortex-M3 的 SoC 設計工程師, 應該要確定當獨占傳輸出現時, 記憶體系統會確保資料一致性。

## Endian 模式

Cortex-M3 支援 little endian (小端)與 big endian (大端)兩個模式。然而, 所支援的記憶體型態也隨微控制器其餘的設計(匯流排連接、記憶體控制器、周邊等等)而定。在開發軟體之前, 必須確實詳細地檢查微控制器的資料表。在大部分的情形下, 基於 Cortex-M3 的微控制器採用的是 little endian 模式。

表 5-4 Cortex-M3 Little endian - 記憶體佈局

位址	Bits 31 - 24	Bits 23 - 16	Bits 15 - 8	Bits 7 - 0
0x1003 - 0x1000	Byte - 0x1003	Byte - 0x1002	Byte - 0x1001	Byte - 0x1000
0x1007 - 0x1004	Byte - 0x1007	Byte - 0x1006	Byte - 0x1005	Byte - 0x1004
...				

表 5-5 The Cortex-M3 Little endian - 不同大小資料的資料佈局

位址, 大小	Bits 31 - 24	Bits 23 - 16	Bits 15 - 8	Bits 7 - 0
0x1000, word	Data[31:24]	Data[23:16]	Data[15:8]	Data[7:0]
0x1000, half word	-	-	Data[15:8]	Data[7:0]
0x1002, half word	Data[15:8]	Data[7:0]	-	-
0x1000, byte	-	-	-	Data[7:0]
0x1001, byte	-	-	Data[7:0]	-
0x1002, byte	-	Data[7:0]	-	-
0x1003, byte	Data[7:0]	-	-	-

於 little endian 模式中, 記憶體視圖裡的資料 byte 通道之位置與 AHB 介面的資料 byte 通道相同。

於 big endian 模式中, 其記憶體視圖中的 byte 通道會交換。

表 5-6 Big endian 記憶體佈局

位址	Bits 31 - 24	Bits 23 - 16	Bits 15 - 8	Bits 7 - 0
0x1003 - 0x1000	Byte - 0x1000	Byte - 0x1001	Byte - 0x1002	Byte - 0x1003
0x1007 - 0x1004	Byte - 0x1004	Byte - 0x1005	Byte - 0x1006	Byte - 0x1007
...				

在 Cortex-M3 裡, big endian 的定義與在 ARM7TDMI 的定義不同。在 ARM7TDMI 裡 big endian 方法被稱作 word-invariant(不變的) big endian, 並且在 ARM 文件寫作"BE-32"方式；而在 Cortex-M3 裡, big endian 方法被稱作 byte-invariant big endian, 並且在 ARM 文件寫作"BE-8"方式。雖然此兩個方式的記憶體佈局相同, 但他們在匯流排介面的 byte 通道使用法不同。ARM 架構 v6 與 v7 中支援 byte-invariant big endian。表 5-7 顯示在 Byte-invariant big endian 方式中不同資料大小的資料佈局。

表 5-7 Cortex-M3 (Byte-Invariant Big Endian) - 不同大小資料的資料佈局

位址, 大小	Bits 31 - 24	Bits 23 - 16	Bits 15 - 8	Bits 7 - 0
0x1000, word	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]
0x1000, half word	Data[7:0]	Data[15:8]	-	-
0x1002, half word	-	-	Data[7:0]	Data[15:8]
0x1000, byte	Data[7:0]	-	-	-
0x1001, byte	-	Data[7:0]	-	-
0x1002, byte	-	-	Data[7:0]	-
0x1003, byte	-	-	-	Data[7:0]

注意在 Byte-Invariant Big Endian 模式中 AHB 汇流排上的資料傳輸, 使用與 little endian 相同的資料 byte 通道(lanes)。然而, 在 half word 與 word 資料之中 的 byte 資料, 其先後次序則是相反(參考表 5-8)。

表 5-8 Cortex-M3 (Byte-Invariant Big Endian) - 在 AHB 汇流排上的資料

位址, 大小	Bits 31 - 24	Bits 23 - 16	Bits 15 - 8	Bits 7 - 0
0x1000, word	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]
0x1000, half word	-	-	Data[7:0]	Data[15:8]
0x1002, half word	Data[7:0]	Data[15:8]	-	-
0x1000, byte	-	-	-	Data[7:0]
0x1001, byte	-	-	Data[7:0]	-
0x1002, byte	-	Data[7:0]	-	-
0x1003, byte	Data[7:0]	-	-	-

此行為相異於 ARM7TDMI。ARM7TDMI 運作於 big endian 模式時，有不同的匯流排通道的安排。使用在 ARM7TDMI 裡的 Word-Invariant Big Endian 的資料 byte 通道使用法如表 5-9 所示。

表 5-9 ARM7TDMI (Word-Invariant Big Endian) - 在 AHB 汇流排上的資料

位址, 大小	Bits 31 - 24	Bits 23 - 16	Bits 15 - 8	Bits 7 - 0
0x1000, word	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]
0x1000, half word	Data[7:0]	Data[15:8]	-	-
0x1002, half word	-	-	Data[7:0]	Data[15:8]
0x1000, byte	Data[7:0]	-	-	-
0x1001, byte	-	Data[7:0]	-	-
0x1002, byte	-	-	Data[7:0]	-
0x1003, byte	-	-	-	Data[7:0]

在 Cortex-M3 中，處理器上的一個硬體輸入信號決定 endian 的模式，此信號會在處理器離開重置時被取樣。此後不可再更改 endian 模式(並無動態的 endian 切換，且不支援 SETEND 指令)。指令擷取永遠採用 little endian 模式；而在系統控制空間(例如 NVIC)以及私有周邊匯流排(例如除錯元件以及外部私有周邊匯流排記憶體範圍)的資料存取也永遠為 little endian 模式。(記憶體範圍 0xE0000000 至 0xE00FFFFF 通常為 little endian)。

如果你的 SoC 或微控制器並不支援 big endian，但是你正使用的一個或一些周邊包含了 big endian 的資料，你可藉著 Cortex-M3 裡幾個指令，輕易地在 little endian 與 big endian 之間作資料轉換。例如：REV 與 REVH 對此種資料轉換非常有用。

## Chapter

## 6

# Cortex-M3 架構實作概觀

本章內容包括：

- ✓ 管線技術
- ✓ Cortex-M3 詳細的方塊圖
- ✓ Cortex-M3 上的匯流排介面
- ✓ Cortex-M3 上的其他介面
- ✓ 外部的私有周邊匯流排
- ✓ 典型的連接方式
- ✓ 重置信號

## 管線技術

Cortex-M3 處理器為三階段的管線模式：包含了指令擷取、指令解碼、指令執行等管線階段(參考圖 6-1)。

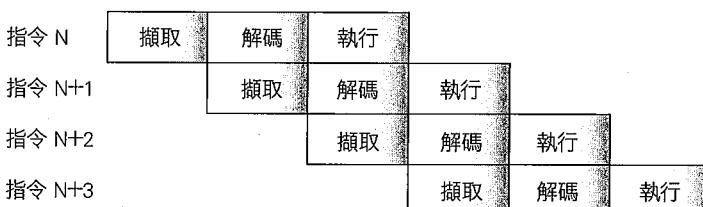


圖 6-1 Cortex-M3 裡的三階段管線

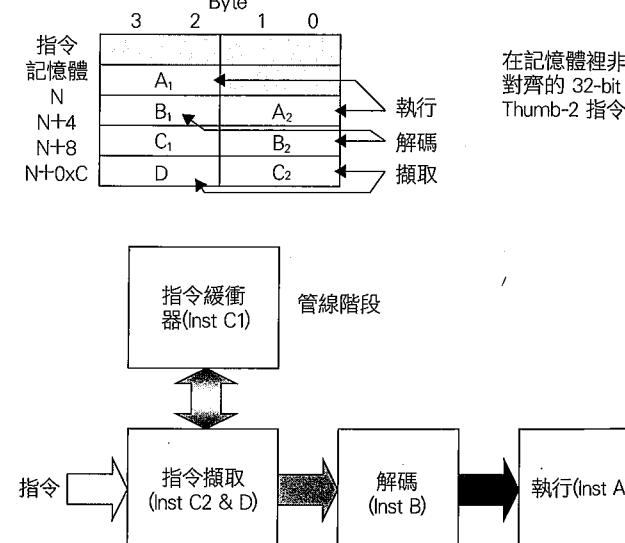
有些人主張其為四階段管線，因為加上了存取記憶體時匯流排介面的管線行為；但此階段在處理器外發生，故處理器本身依然只有三個階段。

當執行絕大部分為 16-bit 的程式時，你會發現處理器可能不會每週期都做指令擷取。其原因是處理器一次至多會擷取兩個指令(32-bit)，故當擷取了一個指令後，其下一個指令已存在於處理器裡了。在此情形下，處理器匯流排介面可能會試著去擷取再下一個指令，但如果緩衝器已滿，則匯流排介面會被閒置。有一些指令需要多個週期來執行，在此情形下，管線就會停頓。

在執行跳躍指令時，管線將被清除。處理器必須從跳躍的目的地擷取指令，再填充管線。然而 Cortex-M3 處理器在 v7-M 結構裡，支援了一些指令，因此可以避免某些短距離的跳躍，而使用條件執行程式加以取代<sup>1</sup>。

因為處理器的管線性質，並且為了保證程式與 Thumb 程式碼的相容性，在指令執行中，程式計數器的讀值，為指令位址加上 4。此位移為常數，不受 16-bit Thumb 指令與 32-bit Thumb-2 指令組合的影響。如此，保證了 Thumb 與 Thumb-2 的一致性。

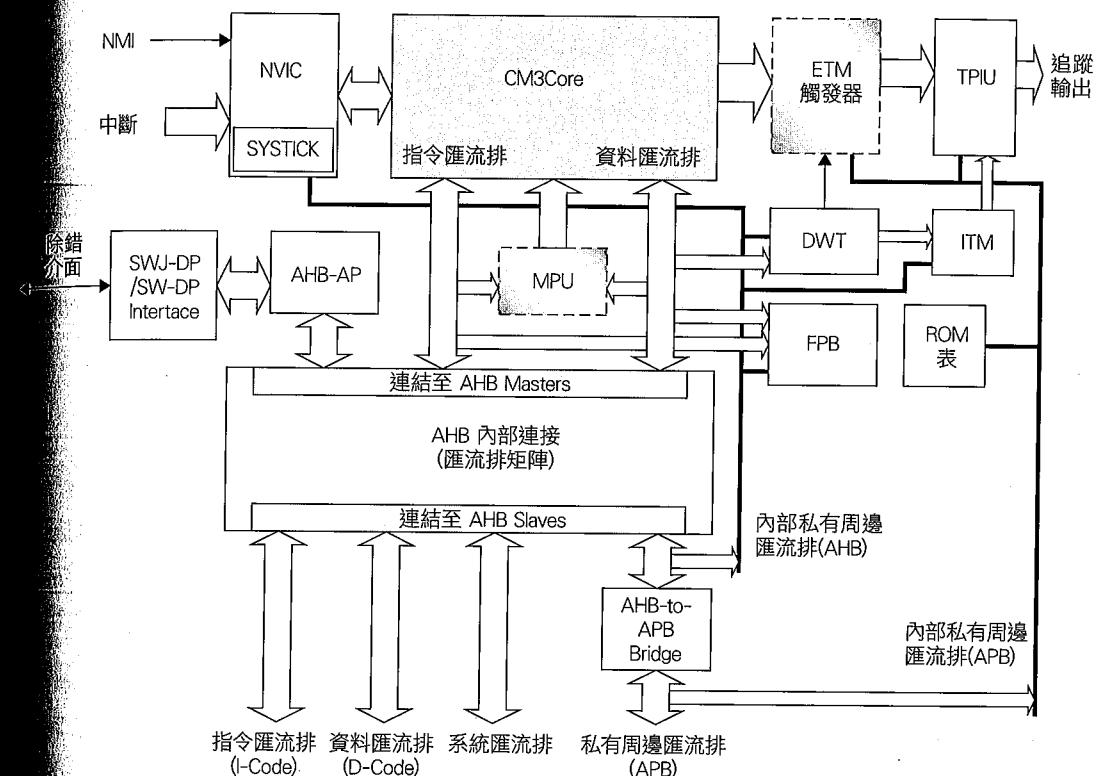
在處理器核心的指令預取單元內，也有一個指令緩衝器。此緩衝器允許將額外的指令，於使用前預先放入佇列。此緩衝器可防止因指令序列包含了非以 word 對齊的 32-bit Thumb-2 指令，造成管線暫停的現象。然而，因為緩衝器並不會增加管線的階段，故亦不會增加跳躍的成本。



<sup>1</sup> 更多的資訊，可參考第四章“IF-THEN 指令”一節。

## Cortex-M3 詳細的方塊圖

Cortex-M3 處理器包含了處理器核心、一些系統管理元件、以及除錯支援元件等。這些元件使用 AHB(Advanced High-Performance Bus 進階高效能匯流排)與 APB(Advanced Peripheral Bus 進階周邊匯流排)連結在一塊。AHB 與 APB 為 AMBA(Advanced Microcontroller Bus Architecture 進階微控制器匯流排架構)(Ref 4)標準的一部分。



注意：MPU 和 ETM 為選擇性的方塊，可於實作微控制器系統時加入。

在圖 6-3 中出現了一些新的元件(參考表 6-1)。

# Jason 嘴書—EEETOP 世界唯一貼

表 6-1 方塊圖裡的簡稱與其定義

名稱	描述
CM3Core	Cortex-M3 處理器的中央處理核心
NVIC	巢狀向量中斷控制器
SYSTICK 計時器	作業系統使用的簡單計時器
MPU	記憶體保護單元(選用的)
CM3BusMatrix	內部的 AHB 內部連結
AHB to APB	轉換 AHB 至 APB 的匯流排橋
SW-DP/SWJ-DP 介面	序列線/序列線 JTAG 除錯埠(DP)介面；使用序列線協定或者傳統的 JTAG 協定(for SWJ-DP)實作的除錯介面連結
AHB-AP	AHB 存取埠；將命令從序列線/SWJ 介面至 AHB 傳輸做轉換
ETM	內嵌的追蹤巨集格；處理為了除錯作指令追蹤的模組(選擇性的)
DWT	資料觀察點與追蹤單元；處理作為除錯的資料觀察點功能的模組
ITM	儀器追蹤巨集格
TPIU	追蹤埠介面單元；傳送除錯資料到外部追蹤抓取硬體
FPB	快閃補釘與中斷點單元
ROM 表	存放組態資訊的一個小型查看表

Cortex-M3 處理器以處理器子系統(subsystem)的方式發佈，其 CPU 核心本身與中斷控制器(NVIC)及各種除錯邏輯方塊密切結合：

- ◆ **CM3Core**: Cortex-M3 核心包括暫存器、ALU、資料通道、匯流排介面等。
- ◆ **巢狀向量中斷控制器(Nested Vectored Interrupt Controller, NVIC)**: NVIC 為一內建中斷控制器。晶片製造廠商可以客製化中斷的數目。NVIC 與 CPU 核心密切結合，並包含了一些系統控制暫存器。它支援巢狀中斷處理，代表在 Cortex-M3 中非常容易處理巢狀中斷。它也具有向量式中斷的特性，故當中斷發生時，它可以直接進入相關的中斷處置程序，而不需使用共用的處置器以決定欲選用的中斷。
- ◆ **SYSTICK 定時器**: 系統滴答(System Tick, SYSTICK)計時器為一基本的倒數計時器，可用來產生固定時間間隔的中斷，即使系統進入休眠模式時亦可。因為不需要改變 OS 的系統定時器程式，所以它使得 Cortex-M3 設備之間 OS 的移植更為簡便。SYSTICK 定時器為 NVIC 建構的一部分。
- ◆ **記憶體保護單元(Memory Protection Unit, MPU)**: MPU 區塊為選擇性的，此意謂某些版本的 Cortex-M3 擁有 MPU，但某些版本沒有。如果包含 MPU，則可使用它來保護記憶體內容，例如，使記憶體區域為唯讀，或防止使用者應用程式去存取特權的應用資料。

◆ **BusMatrix**: BusMatrix 是 Cortex-M3 內部匯流排系統的心臟，為 AHB(Advanced High-performance Bus, 進階高效率匯流排)的連接網路。除了兩個匯流排 master 同時嘗試存取相同的記憶體區域之外，它允許不同的匯流排在同一時間進行傳輸。BusMatrix 也提供其他的資料傳輸管理，包括寫入緩衝器，以及位元取向的(bit-oriented)運算(例如 bit-band)。

◆ **AHB 至 APB**: AHB-to-APB 匯流排橋，連接一些 APB(Advanced Peripheral Bus, 進階周邊匯流排)設備(例如除錯元件)到 Cortex-M3 處理器的私有周邊匯流排。此外，Cortex-M3 允許晶片製造廠商，藉此 APB 匯流排，把額外的 APB 設備附加到外部私有周邊匯流排。

方塊圖的其餘元件是作為除錯的支援，正常情形下，應用程式不能使用這些元件：

◆ **SW-DP/SWJ-DP**: Serial Wire Debug Port (SW-DP, 序列線除錯埠)/Serial Wire JTAG Debug Port (SWJ-DP)與 AHB Access Port (AHB-AP)一起運作，使得外部除錯器產生 AHB 傳輸，以控制除錯活動。Cortex-M3 處理器核心內並無 JTAG 掃描鏈(scan chain)，大部分的除錯功能都是藉由 AHB 存取 NVIC 暫存器來控制。SWJ-DP 支援 Serial Wire Protocol(協定)與 JTAG Protocol，而 SW-DP 僅支援 Serial Wire Protocol。

◆ **AHB-AP**: AHB Access Port(存取埠)提供經由少數暫存器存取整個 Cortex-M3 記憶體的能力。SW-DP/SWJ-DP 經由一個稱作除錯存取埠(Debug Access Port, DAP)的通用(generic)除錯介面，來控制此區塊。要執行除錯功能時，外部除錯硬體需要經由 SW-DP/SWJ-DP 來存取 AHB-AP，以產生所需的 AHB 傳輸。

◆ **嵌入式追蹤巨集格(Embedded Trace Macrocell, ETM)**: ETM 為作指令追蹤的選擇性元件，因而，某些 Cortex-M3 產品可能並沒有即時指令追蹤能力。追蹤的資訊經由 TPIU 輸出至追蹤埠。ETM 控制暫存器會被記憶體映射，除錯器可經由 DAP 來加以控制。

◆ **資料觀察點與追蹤(Data Watchpoint and Trace, DWT)**: DWT 可用以設定資料觀察點。當找到一個吻合的資料位址或者資料值時，此吻合事件可用以產生觀察點事件，以啟動除錯器、產生資料追蹤資訊、或者啟動 ETM。

# Jason 嘴書—EETOP 世界唯一貼

- ◆ **設備追蹤巨集格(Instrumentation Trace Macrocell, ITM)**: ITM 可藉著幾種方式來使用：軟體可直接寫入此模組以將資訊輸出至 TPIU，或者與 DWT 吻合的事件，可藉由 ITM 產生資料追蹤包裹，輸出至追蹤資料串流。
- ◆ **追蹤埠介面單元(Trace Port Interface Unit, TPIU)**: TPIU 用作外部追蹤硬體(例如追蹤埠分析儀)的介面。在 Cortex-M3 內部，追蹤資料以進階追蹤匯流排(Advanced Trace Bus, ATB)包裹形式作格式化，TPIU 則重新格式化資料，以允許資料被外部裝置抓取。
- ◆ **FPB**: FPB 用來提供快閃補丁(Flash Patch)與中斷點的功能。快閃補丁意指如果 CPU 存取的指令，吻合了某一個位址，將由此位址再映射至不同的位置以擷取另一不同的值。或者，吻合的位址可用來觸發一個中斷點事件。快閃補丁特性對測試非常有用，例如：在一個裝置上增加一段除了藉著 FPB 來改變程式控制外，在正常情形下不會被使用到的診斷程式碼。
- ◆ **ROM 表**: Cortex-M3 裡提供了一個小的 ROM 表。此僅是一個小查看表，對不同的系統設備與除錯元件，提供了記憶體映射資訊。除錯系統利用此表格，找出除錯元件的記憶體位址。在大部分的情形下，記憶體映射需要固定在如 Cortex-M3 TRM 文件所規定的標準記憶體位置，但是因為某些除錯元件為選擇性的，並且可能加入額外的元件，個別的晶片製造廠商可能會想要客制化其晶片的除錯特性。在此情形下，ROM 表需要客制化，以便讓除錯軟體決定正確的記憶體映射，並以此偵測可使用的除錯元件。

## Cortex-M3 上的匯流排介面

除非你正使用 Cortex-M3 設計 SoC 產品，否則不大可能直接存取此處描述的匯流排信號。正常情況下，晶片製造廠商把所有的匯流排信號，連到記憶體區塊與周邊，在少數情況下，你可能發現晶片製造廠商，連結匯流排至匯流排橋，並允許外部匯流排系統在晶片外部作連結。Cortex-M3 處理器上的匯流排介面，基於 AHB-Lite 與 APB 協定，其文件在 AMBA Specification (Ref 4) 中。

### I-Code 匯流排

I-Code 匯流排為基於 AHB-Lite 匯流排協定的 32-bit 匯流排，作為記憶體區域 0x00000000 至 0x1FFFFFFF 的指令擷取。即使是 Thumb 指令，指令擷取皆以 word 大小進行，因此，在執行時，CPU 核心最多可在同時間裡，擷取兩個 Thumb 指令。

### D-Code 匯流排

D-Code 匯流排為基於 AHB-Lite 匯流排協定的 32-bit 匯流排，作為記憶體區域 0x00000000 至 0x1FFFFFFF 的資料存取。雖然 Cortex-M3 處理器支援了非對齊單元的傳輸，在此匯流排你不會有非對齊的傳輸，因為處理器核心的匯流排介面，替你將非對齊傳輸換作對齊傳輸。因此，連到此匯流排的設備(例如記憶體)僅需支援 AHB-Lite (AMBA 2.0)的對齊傳輸。

### 系統匯流排

系統匯流排為基於 AHB-Lite 匯流排協定的 32-bit 匯流排，作為記憶體區域 0x20000000 至 0xDFFFFFFF 以及 0xE0100000 至 0xFFFFFFFF 的指令擷取與資料存取。與 D-Code 一樣，皆為對齊傳輸。

### 外部私有周邊匯流排

外部私有周邊匯流排(External Private Peripheral Bus, External PPB) 為基於 APB 匯流排協定的 32-bit 匯流排。然而，因為 APB 記憶體某些部分已用作 TPIU、ETM、和 ROM 表，可在此匯流排連結額外周邊的記憶體區域僅從 0xE0042000 至 0xE00FF000。此匯流排的傳輸皆以 word 作單元對齊。

### 除錯存取埠匯流排

除錯存取埠(Debug Access Port, DAP)匯流排，為基於 APB 規格增強版本的 32-bit 匯流排。它用來連結如 SWJ-DP 或 SW-DP 等等除錯介面區塊。不要把此匯流排作其他用途使用。更多此介面的資訊可在第十五章除錯結構，或 ARM 文件：CoreSight Technology System Design Guide (Ref 3)找到。

## Cortex-M3 上的其他介面

除了匯流排介面, Cortex-M3 有一些作各種用途的其他介面。這些信號不大可能會出現在矽晶片的接腳上, 因為它們大多用來連結 SoC 不同的部分, 或不被使用。這些信號的細節包含於 Cortex-M3 Technical Reference Manual (TRM) (Ref 1)。表 6-2 包含簡單的摘要。

表 6-2 各種介面信號

信號群組	功能
多處理器的通信(TXEV, RXEV)	多個處理器之間的簡單工作同步信號
睡眠信號(SLEEPING, SLEEPDEEP)	電源管理用的睡眠狀態
中斷狀態信號(ETMINTNUM, ETMINTSTATE, C URRPRI)	ETM 運算與除錯用途的中斷運算狀態
重置要求(SYSRESETREQ)	從 NVIC 輸出的重置要求
鎖住 <sup>2</sup> 與暫停狀態 (LOCKUP, HALTED)	顯示處理器核心已經進入鎖住狀態(由在硬錯誤處理程式或 NMI 處理程式之內出現錯誤情形而造成)或暫停的狀態(為了除錯運算)
Endian 輸入(ENDIAN)	當核心重置時設定 Cortex-M3 的 endian
ETM 介面	連結至內嵌追蹤巨集格(ETM)以作指令追蹤
ITM 的 ATB 介面	進階追蹤匯流排(ATB)為在 ARM 的 CoreSight 除錯架構內作為追蹤資料傳輸的匯流排協定；在此這個介面提供從 Cortex-M3 的儀器追蹤巨集格(ITM)的追蹤資料輸出，其中 ITM 被連結至追蹤埠介面單元(TPIU)

## 外部的私有周邊匯流排

Cortex-M3 處理器擁有外部的私有周邊匯流排(Private Peripheral bus, PPB)介面。外部 PPB 介面是根據 AMBA 2.0 規格中的進階週邊匯流排(APB)協定。它用在不可分享的系統設備上(例如除錯元件)。為了支援 CoreSight 設備，此介面包括一個稱做 PADDR31 的額外信號。此信號顯示傳輸的來源：若信號為 0，意指傳輸由 Cortex-M3 上執行的軟體產生；若信號為 1，意指傳輸由除錯硬體產生。根據此信號，可設計一個僅能被除錯器使用，或者當它被軟體使用時，僅允許某些功能的周邊。

<sup>2</sup> 第 12 章包含更多有關鎖住的資訊。

此匯流排異於周邊，不作一般用途使用。雖然並沒有禁止晶片設計師設計並連接通用周邊到此匯流排，但因為特權存取等級管理的原因，致使使用者後來寫程式時遭遇問題，例如：用程式把設備設定為應用狀態；或者在使用 MPU 時，把設備與其他記憶體區域分開。

外部 PPB 並不支援非對齊存取。因為此匯流排資料寬度為 32-bit 且是基於 APB，當你為此記憶體區域設計周邊時，需要確定周邊所有的暫存器位址以 word 為單位對齊。此外在此區域寫軟體去存取設備時，建議先確定所有的存取皆為 word 大小。PPB 的存取，通常是 little endian。

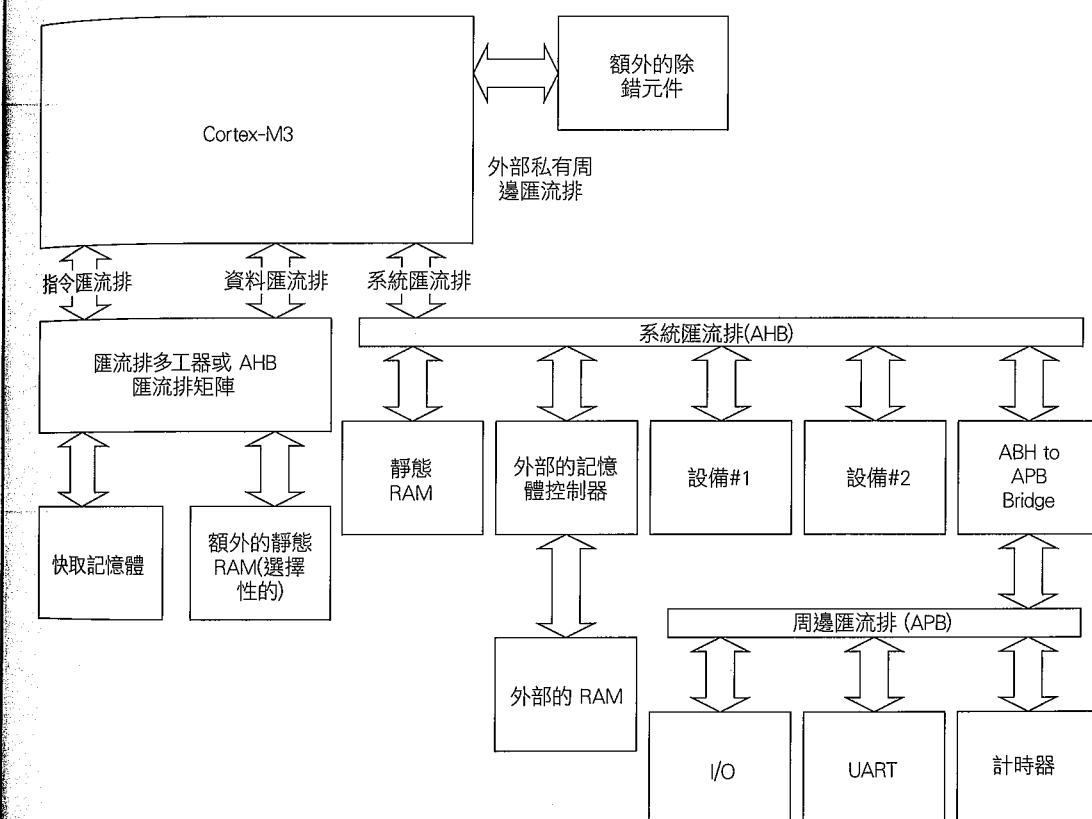


圖 6-4 Cortex-M3 彙流排連接的例子

## 典型的連接方式

因為 Cortex-M3 處理器上有許多匯流排介面，你可能會不曉得如何將它與記憶體或周邊等其他設備連接。圖 6-4 為一個簡化的例子。

因為程式記憶體區域可被指令匯流排(如果作指令擷取)或者資料匯流排(如果作資料存取)來存取，故需要有被稱作 Bus-Matrix<sup>3</sup> 的 AHB 汇流排開關或 AHB 汇流排多工器。使用 Bus-Matrix，則快閃記憶體與額外的 SRAM 記憶體(如果有置入的話)可用兩者其中任一個匯流排作存取。Bus-Matrix 可從 ARM 的 AMBA Development Kit (ADK)<sup>4</sup> 取得。當資料匯流排與指令匯流排兩者同時嘗試存取相同的記憶體設備時，可授與資料匯流排較高的優先權，以得到最佳效果。

使用 AHB Bus-Matrix 時，如果指令匯流排與資料匯流排同時存取不同的記憶體設備(例如，從 Flash 記憶體作指令擷取，同時資料匯流排從額外的 SRAM 讀資料)，則可以同時執行傳輸動作。但如果使用了匯流排多工器，則不能同時傳輸，不過會得到比較小的電路。但是通常 Cortex-M3 微控制器的設計，會使用系統匯流排作 SRAM 連接。

主要的 SRAM 區塊，需要由系統匯流排介面，藉著 SRAM 記憶體位址區域來連接。這樣可以讓資料存取與指令存取在同時間進行。它亦允許藉由 bit-band 特性，以設定布林資料型態。

某些微控制器可能擁有外部記憶體介面。因為你不能把晶片外的記憶體設備，直接連結到 AHB，故需要外部記憶體控制器。外部記憶體控制器可直接連接到 Cortex-M3 的系統匯流排。額外的 AHB 設備，並不需要 Bus-Matrix，也可以容易地連接到系統匯流排。

簡單的周邊可藉由 AHB-to-APB 橋連接到 Cortex-M3，這樣可以讓周邊使用更簡易的 APB 汇流排協定。

<sup>3</sup> 此處所需的 Bus-Matrix 有別於圖 6-3 所示的 Cortex-M3 裡面的內部 Bus-Matrix。Cortex-M3 的內部 Bus-Matrix 為特殊的設計，並且不同於標準的 ADK 版本。

<sup>4</sup> ADK 為 AMBA 元件的集合並且是 VHDL/Verilog 的範例系統。

圖 6-4 的示意圖僅是一個非常簡單的例子，晶片設計師可以選擇不同的匯流排連接的設計。作軟體/韌體開發時，你僅需要明瞭記憶體映射。

示意圖裡顯示的設計區塊，例如：Bus-Matrix、AHB-to-APB 汇流排橋、記憶體控制器、I/O 介面、定時器、和 UART 等，全部可從 ARM 和一些 IP 供給商獲得。因為微控制器有不同的周邊供給商，當你為 Cortex-M3 系統開發軟體時，需要取得微控制器的資料表，以得到正確的程式碼模型。

## 重置信號

在 Cortex-M3 微控制器或者 SoC 上設計的重置電路，是由實作而定。在 Cortex-M3 Technical Reference Manual (Ref 1)，記載了幾個重置信號，然而，實作的 Cortex-M3 晶片可能僅有一個或兩個重置信號，其餘則由晶片商設計的重置產生器內部產生。(請參考製造廠商的資料表，以取得正確地把廠商基於 Cortex-M3 的微控制器重置的指令)。在 Cortex-M3 處理器層級，你可以找到表 6-3 所列的重置信號。

表 6-3 Cortex-M3 上各種重置型態

重置信號	描述
電源啟動重置(PORRESETn)	當設備啟動時必須宣稱的重置；重置了處理器核心與除錯系統兩者
系統重置(SYSRESETn)	系統重置；影響處理器核心，NVIC(除了除錯控制暫存器以外)，以及 MPU 但不影響除錯系統
測試重置(nTRST)	為了除錯系統作重置

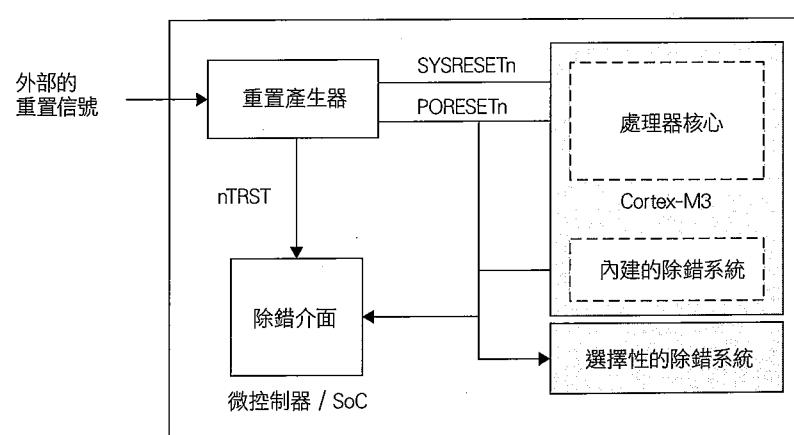


圖 6-5 在典型的 Cortex-M3 微控制器裡，產生內部的重置信號

Chapter

7

# 例外

本章內容包括：

- ✓ 例外類型
- ✓ 優先權定義
- ✓ 向量表
- ✓ 中斷輸入與等待行為
- ✓ 錯誤例外
- ✓ SVC 與 PendSV

## 例外類型

Cortex-M3 提供多功能的例外架構，來支援一些系統例外與外部中斷。例外號碼 1 到 15 作為系統例外使用，號碼 16 以上作為外部中斷輸入使用，大部分的例外可經由程式設定優先權，少部分則有固定的優先權。

Cortex-M3 晶片有不同數量的外部中斷輸入(從 1 到 240)，和不同數量的優先權等級。這是因為晶片設計師可針對不同需求，來對 Cortex-M3 設計的原始程式碼作組態。

例外類型 1 到 15 為系統例外(並無例外類型 0)，於表 7-1 概略介紹；例外類型 16 以上為外部中斷輸入，可參見表 7-2。

正執行中的例外之值會顯示於特殊暫存器 IPSR，或可從 NVIC 的中斷控制狀態暫存器(VECTACTIVE 欄位)得知。

MEMO.

表 7-1 系統例外列表

例外號碼	例外類型	優先權	描述
1	重置	-3(最高的)	重置
2	NMI	-2	不可遮罩的中斷(外部 NMI 輸入)
3	硬錯誤	-1	所有的錯誤情況, 如果相對應的錯誤處理程式未被啟動
4	MemManage 錯誤	可程式的	記憶體管理錯誤, 違反 MPU 或是存取了不合法的位置
5	匯流排錯誤	可程式的	匯流排錯誤; 當 AHB 介面從匯流排 slave 接收到錯誤的反應時發生(若為指令擷取亦稱作預擷取中止, 或者若為資料存取亦稱作資料中止)
6	用法錯誤	可程式的	程式錯誤或試圖存取輔助處理器而造成的例外(Cortex-M3 並無支援輔助處理器)
7-10	保留的	NA(無)	-
11	SVCALL	可程式的	系統服務呼叫(或監督器呼叫)
12	除錯監視器	可程式的	除錯監視器(中斷點, 觀察點, 或者外部除錯要求)
13	保留的	NA	-
14	PendSV	可程式的	可置於等待的系統服務要求
15	SYSTICK	可程式的	系統滴答計時器

表 7-2 外部中斷列表

例外號碼	例外類型	優先權
16	外部中斷#0	可程式的
17	外部中斷#1	可程式的
...	...	...
255	外部中斷#239	可程式的

注意, 此處中斷號碼(例如中斷#0)意指對 Cortex-M3 NVIC 的中斷輸入。在實際的微控制器產品或 SoC, 外部的中斷輸入接腳號碼, 可能並不會與 NVIC 上的中斷輸入號碼吻合。例如, 最前面幾個中斷輸入可能會指定給內部週邊, 接下來幾個中斷輸入則可能指定給外部中斷的接腳。因此, 你需要檢查晶片製造廠商的資料表, 以決定中斷的編號。

當一個致能的例外出現, 但不能立刻執行時(例如, 正執行一個具更高優先權的中斷服務程式, 或者中斷遮罩被設定時), 它將會等待(某些錯誤例外則不會<sup>1</sup>), 這意味著有個

<sup>1</sup> 此將例外置於等待的行為有一些特例：如果錯誤發生, 但因為正執行更高優先權的處理程式而不能執行相關的錯誤處理程式, 則會代而執行硬錯誤處理程式(最高優先權的錯誤處理程式)。本章後面在研究錯誤例外時, 會討論更多此議題的細節；此議題詳盡的解說可以在 Cortex-M3 Technical Reference Manual 與 ARM v7-M Architecture Application Level Reference Manual 找到。

暫存器(等待狀態)將保留例外請求, 直至例外被執行為止。這與傳統的 ARM 處理器不同。先前, 發出中斷(例如 IRQ/FIQ)的設備, 在他們接受服務之前, 需要保留請求。現在, 藉著 NVIC 裡的等待暫存器, 即使請求中斷來源去除其請求信號, 發生的中斷依然可以接受處理。

## 優先權定義

在 Cortex-M3 裡, 一個例外是否被執行, 可能會受到此例外優先權的影響。較高優先權(優先權等級號碼較小)例外強佔較低優先權(優先權等級號碼較大)例外, 此為巢狀的例外/中斷會有的情節。某些例外(重置、NMI、硬錯誤等)有固定的優先權等級, 其號碼為負, 以表示他們的優先權比其他例外為高。其他例外則擁有可程式的優先權等級。

Cortex-M3 支援三個固定的最高優先權等級, 以及多達 256 等級的可程式優先權(最高 128 等級的強佔權)。然而, 大部分 Cortex-M3 晶片支援的等級較少, 可能支援了 8、16、或 32 等級。當設計 Cortex-M3 晶片或 SoC 時, 設計師可藉著客製化以得到所需的等級數。等級的簡化是藉著去掉優先權組態暫存器 LSB 的部分而達成。

例如, 如果在設計中僅實作了 3 位元的優先權等級, 其優先權等級組態暫存器將如圖 7-1 所示。

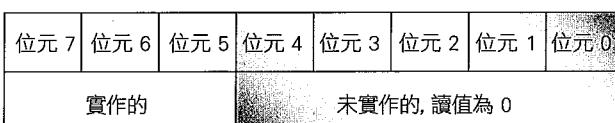


圖 7-1 實作了 3-Bit 的優先權等級暫存器

既然 bit 4 到 bit 0 沒有使用, 其讀值永遠為 0, 且寫入這些位元的動作會被忽略。在這樣的設定下, 我們可能得到下列優先權等級：0x00(最高)、0x20、0x40、0x60、0x80、0xA0、0xC0 和 0xE0(最低)。

同樣地, 如果在設計中實作了 4 位元的優先權等級, 其優先權等級組態暫存器將如圖 7-2 所示。

# Jason 嘴書—EETOP 世界唯一貼

位元 7	位元 6	位元 5	位元 4	位元 3	位元 2	位元 1	位元 0
實作的				未實作的, 讀值為 0			

圖 7-2 實作了 4-Bit 的優先權等級暫存器

如果使用更多位元，則將有更多的優先權等級可用。然而，更多的優先權位元，也會增加閘數和功率耗損。對 Cortex-M3 來說，優先權暫存器欄寬最少使用的數量為 3 位元(8 個等級)。

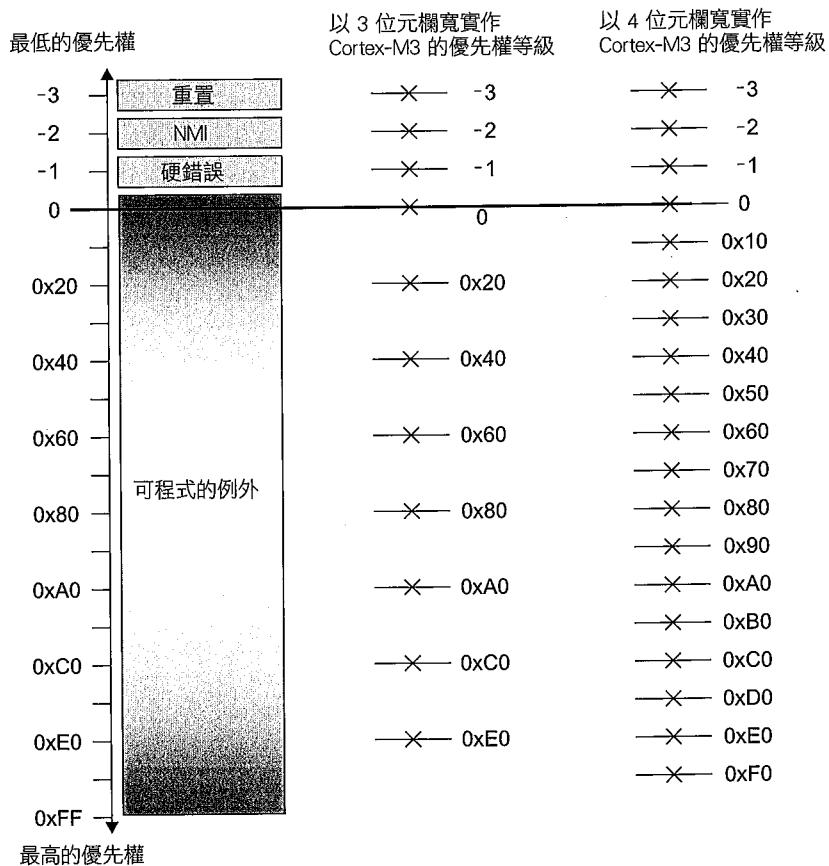


圖 7-3 具 3-Bit 或 4-Bit 優先權欄寬可用的優先權等級

去除暫存器 LSB(最低有效位元)而保留其 MSB(最高有效位元)的理由，是為了利於 Cortex-M3 設備之間軟體的移植。藉著這個做法，具 4-bit 優先權組態暫存器的設備上的程式，就有可能在具 3-bit 優先權組態暫存器的設備上執行。但如果去除 MSB 而保留 LSB，當你在 Cortex-M3 晶片間移植應用程式時，優先權的安排可能會相反。例如，一個應用程式設定 IRQ#0 的優先權等級為 0x05, IRQ#1 的優先權等級為 0x03，顯然 IRQ#1 有較高的優先權；但當 MSB bit 2 被去除時，IRQ#0 的優先權等級變為 0x01，其優先權反而比 IRQ#1 還高。

表 7-3 列出了 3-bit、5-bit、8-bit 優先權暫存器設備上，可使用的例外優先權的等級。

表 7-3 具有 3-bit、5-bit、以及 8-bit 優先權等級暫存器設備可使用的例外優先權的等級

優先權等級	例外類型	具有 3-bit 優先權組態暫存器的設備	具有 5-bit 優先權組態暫存器的設備	具有 8-bit 優先權組態暫存器的設備
-3(最高的)	重置	-3	-3	-3
-2	NMI	-2	-2	-2
-1	硬錯誤	-1	-1	-1
0,	具有可程式優先權等級之例外	0x00	0x00	0x00, 0x01
1,		0x20	0x08	0x02, 0x03
...		...	...	...
0xFF		0xE0	0xF8	0xFE, 0xFF

有些讀者會懷疑，如果優先權等級組態暫存器為 8 位元，為何僅有 128 個強佔等級？這是因為 8-bit 暫存器被進一步區分為兩個部分：強佔優先權(preempt priority)和次優先權(subpriority)。

藉著 NVIC 裡被稱作優先權群的一個組態暫存器(NVIC 中應用程式中斷和重置控制暫存器的一部分，參考表 7-5)，具可程式優先權等級之例外使用的優先權等級組態暫存器，被分作兩部分：上面部分(左端位元)作強佔優先權使用，下面部分(右端位元)作次優先權使用(參考表 7-4)。

表 7-4 不同優先權群組設定裡優先權等級暫存器中，強佔優先權欄位與次優先權欄位的定義

優先權群組	強佔優先權欄位	次優先權欄位
0	位元[7:1]	位元[0]
1	位元[7:2]	位元[1:0]
2	位元[7:3]	位元[2:0]
3	位元[7:4]	位元[3:0]
4	位元[7:5]	位元[4:0]
5	位元[7:6]	位元[5:0]
6	位元[7]	位元[6:0]
7	無	位元[7:0]

表 7-5 應用中斷與重置控制暫存器(位址為 0xE000ED0C)

位元	名稱	類型	重置值	描述
31:16	VECTKEY	R/W	-	存取鑰；欲寫進此暫存器則需將 0x05FA 寫入此欄位，否則寫入將被忽略。讀回時 upper half word 的讀回值為 0xFA05
15	ENDIANNES	R	-	顯示了資料的 endianness：1 表 big endian(BE8)，0 表 little endian；此僅可於重置後改變
10:8	PRIGROUP	R/W	0	優先權群組
2	SYSRESETREQ	W	-	要求晶片控制邏輯以產生重置
1	VECTCLRACTIVE	W	-	清除所有例外的活動狀態資訊；通常用於除錯或 OS 中，以允許系統從系統錯誤恢復（使用重置較為安全）
0	VECTRESET	W	-	重置 Cortex-M3 處理器（除了除錯邏輯），但此不會重置處理器之外的電路

當處理器已經在執行其它的中斷處理時，強佔優先權等級決定中斷是否可以發生。當具有相同的強佔優先等級的兩個例外同時出現時，將考慮次優先權等級的值。在此情形下，會先處理具有較高次優先權（其值較低）的例外。

因為優先權群組，使得強佔優先權的最大欄寬為 7，故得到 128 個等級。當設定優先權群組為第 7 群組時，具可程式優先權等級的所有例外皆在同一等級，在這些例外之間並不會有強佔發生，且僅有硬錯誤（優先權為 -1）、NMI（優先權為 -2）、重置（優先權為 -3）等，可強佔這些例外。

欲決定有效的強佔優先權等級，需考慮下列因素：

建構的優先權等級組態暫存器

優先權群組的設定

例如，如果組態暫存器欄寬為 3（bit 7 至 bit 5 為可用），且其優先權群組被設定為 5，則你可得到四個等級的強佔優先權等級（bit 7 至 bit 6），每一個強佔等級又區分為兩個等級的次優先權（bit 5）。

位元 7	位元 6	位元 5	位元 4	位元 3	位元 2	位元 1	位元 0
強佔優先權	次優先權						

圖 7-4 優先權群組設定為 5 的 3-Bit 優先權等級暫存器裡，優先權欄位的定義

當設定如圖 7-4 所示時，可用的優先權等級將如圖 7-5 所顯示。在相同設計的情形下（亦即組態暫存器欄寬為 3），當其優先權群組被設定為 0x1，則可得到八個等級的強佔優先權等級（bit 7 至 bit 5），但於強佔等級內不再區分次優先權。（因為強佔優先權 Bit[1:0]永遠為 0）。圖 7-6 顯示了優先權等級組態暫存器的定義，圖 7-7 則顯示了其可用的優先權等級。

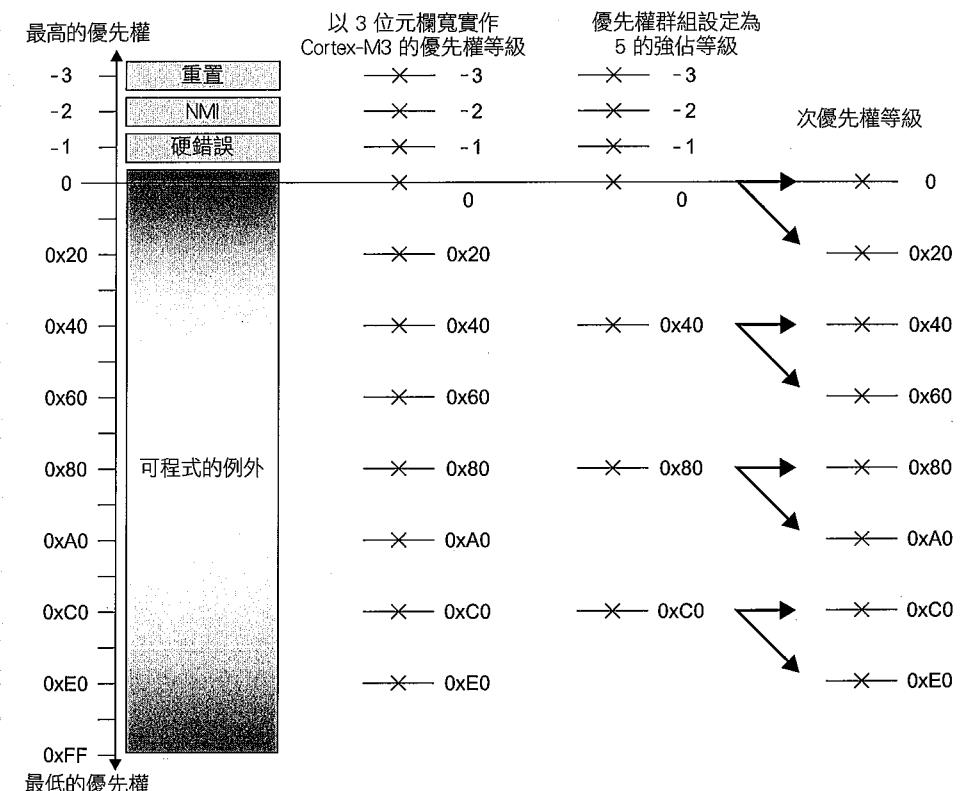


圖 7-5 優先權群組設定為 5，優先權欄位為 3-Bit 時，可用的優先權等級

# Jason 嘴書—EETOP 世界唯一貼

位元 7	位元 6	位元 5	位元 4	位元 3	位元 2	位元 1	位元 0
強占優先權(5:3)		強占優先權(2:0) (通常為 0)		次優先權 (通常為 0)			

圖 7-6 優先權群組設定為 1 的 8-Bit 優先權等級暫存器裡, 優先權欄位的定義

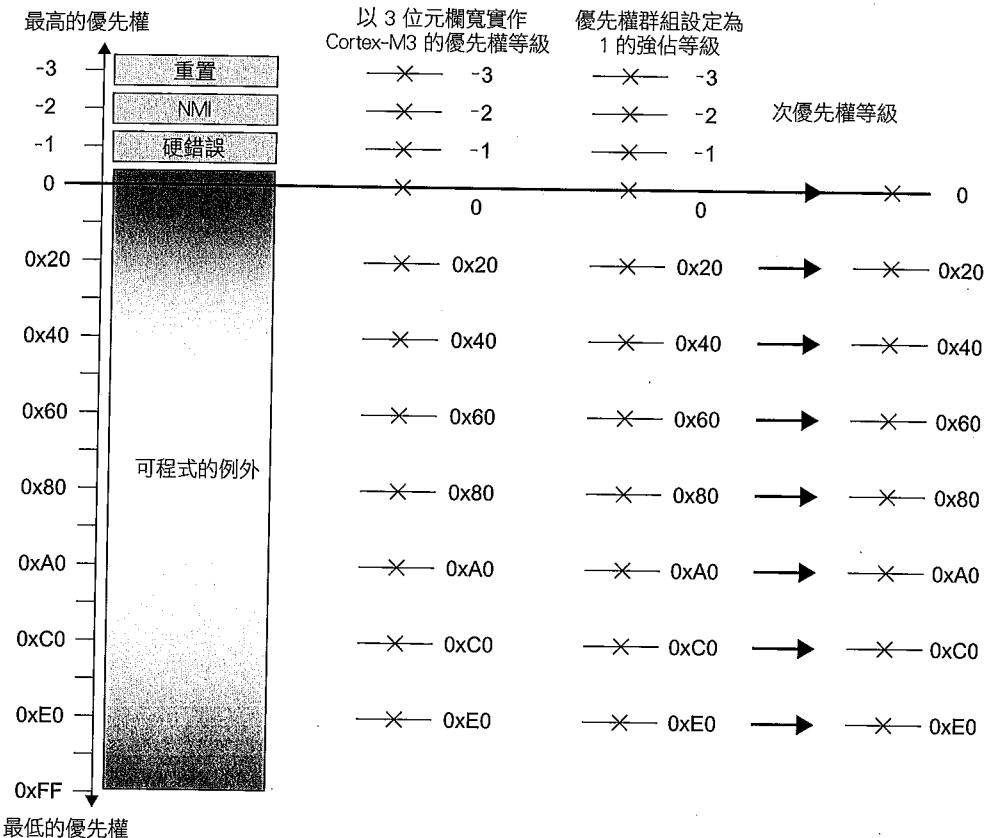


圖 7-7 優先權群組設定為 1, 優先權欄位為 3-Bit 時, 可用的優先權等級

如果 Cortex-M3 設備在優先權等級組態暫存器裡使用了 8 個位元, 則擁有最大數量的強佔等級為 128 個(優先權群組被設定為 0 時), 其優先權欄位的定義顯示於圖 7-8。

位元 7	位元 6	位元 5	位元 4	位元 3	位元 2	位元 1	位元 0
強佔優先權							次優先權

圖 7-8 優先權群組設定為 0 的 8-Bit 優先權等級暫存器裡, 優先權欄位的定義

當同時宣稱的兩個中斷, 具有相同的強佔優先權等級以及次優先權等級, 則有著較小的例外號碼的中斷擁有較高的優先權。(IRQ #0 比起 IRQ #1 具較高的優先權。)

為了防止優先權等級被不經意地改變, 當寫入應用中斷與重置控制暫存器時(位址為 0xE000ED0C), 要相當謹慎。在大多數的情形下, 優先權群組組態完成後, 除非要產生重置, 否則沒有必要使用到此暫存器(參見表 7-5)。

## 向量表

當例外產生且被 Cortex-M3 處理時, 處理器需要找到例外處理程式的起始位址, 此資訊儲存於向量表裡, 預設向量表從位址 0 開始, 且向量位址依照例外號碼乘以 4 來安排(參考表 7-6)。

表 7-6 電源啟動後, 建立的例外向量表

位址	例外號碼	值(Word 大小)
0x00000000	-	MSP 初始值
0x00000004	1	重置向量(程式計數器初始值)
0x00000008	2	NMI 處理程式起始位址
0x0000000C	3	硬錯誤處理程式起始位址
...	...	其他處理程式起始位址

因為位址 0x0 應該是啟動程式, 通常它為快閃記憶體或唯讀記憶體裝置, 且此值不可在執行期間受到更改。然而, 向量表可重新定位到 RAM 中的程式或 RAM 區域裡其它的記憶體位置, 故我們可在執行期間改變處理程式。這可以藉著設定 NVIC 裡所謂的向量表位移暫存器(位址為 0xE000ED08)來達成。位址的位移值需要與向量表大小作對齊排列, 且需要擴展到次一個較大的 2 的指數值。例如, 當 IRQ 輸入為 32 個, 則例外的總數量為  $32 + 16$  (系統例外) = 48 個, 再擴展為次一個較大的 2 的指數值而成為 64 個, 把它乘以 4 得到 256(即 0x100)。所以, 向量表位移值可規劃為 0x0、0x100、0x200 等等。向量表位移暫存器包括的項目列於表 7-7。

表 7-7 向量表位移暫存器(位址 0xE000ED08)

位元	名稱	類型	重置值	描述
29	TBLBASE	R/W	0	表的基底在 Code (0)或 RAM (1)裡
28:7	TBLOFF	R/W	0	從 Code 區或 RAM 區的表的位移值

在應用程式中,如果你希望允許動態地改變例外處理程式,在啟動映射的開始處就需要有下列這些(最小的要求):

- ◆ 起始的主要堆疊指標值
- ◆ 重置向量
- ◆ NMI 向量
- ◆ 硬錯誤向量

需要這些的原因,是因為 NMI 和硬錯誤在你的啟動過程中,有潛在發生的可能,而其它的例外則在被致能前不會發生。

啟動程序完成時,你可定義 SRAM 的一部份為新的向量表,並把向量表重新定位到此新的、可寫入的向量表。

## 中斷輸入與等待行為

本節描述 IRQ 輸入與等待的行為,此亦可應用到 NMI 輸入,不過 NMI 在大部分的情形下會被立即執行,除非遇到以下情形,包括:核心已經正在執行一個 NMI 處理程式、被除錯程式暫停、或因為一些嚴重的系統錯誤而被鎖住(locked up)等。

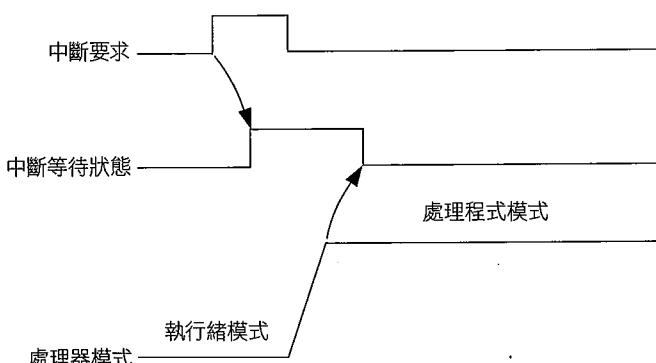


圖 7-9 中斷的等待行為

當中斷輸入被宣稱時,它會處於等待狀態,此意味著它將處於等待處理器去處理要求的狀態中。即使中斷來源停止宣稱此中斷,處於等待狀態的中斷,仍然會導致在優先權獲得允許時,去執行其中斷處理程式。一旦中斷處理程式被啟動,等待狀態會被自動清除(參看圖 7-9 )。

然而,如果等待狀態在處理器開始對等待中斷反應前被清除(例如,當 PRIMASK/FAULTMASK 被設定為 1, 則等待狀態暫存器會被清除),則可以取消中斷(參看圖 7-10 )。中斷的等待狀態可於 NVIC 裡存取,且是可寫入的,所以你可以清除一個等待的中斷,或使用軟體去設定等待暫存器以產生新的中斷等待。

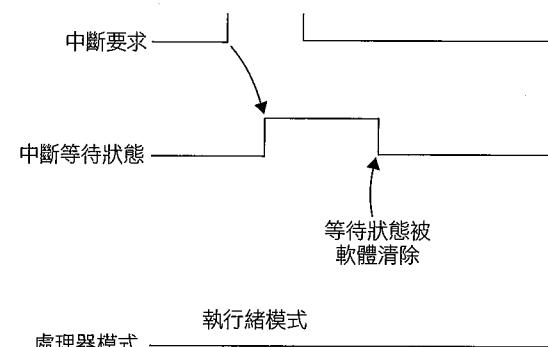


圖 7-10 在處理器採取行動之前,中斷等待被清除

當處理器開始執行一個中斷,此中斷將處於活動狀態,其等待位元會被主動清除(參見圖 7-11 )。當中斷處於活動時,除非中斷服務程式藉著中斷返回(如第九章所論,亦稱作離開中斷)而停止,否則就不能夠再次重新處理同一個中斷。當其活動狀態被清除時,若等待狀態為 1 則此中斷可再被處理。在中斷服務程式結束之前,可以使中斷再次處於等待的狀態。

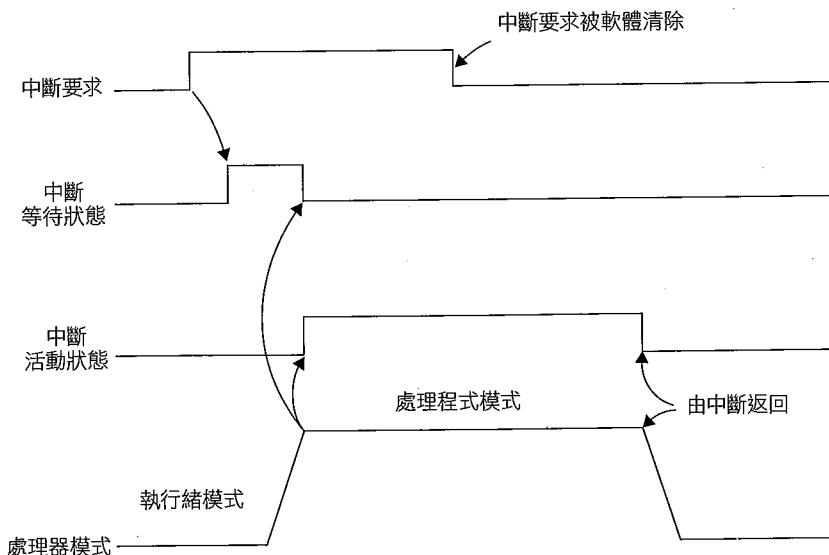


圖 7-11 當處理器進入處理程式時，設定了中斷活動的狀態

如果中斷來源持續發出中斷要求信號，則如圖 7-12 所示，在中斷服務程式結束後，中斷會再一次處於等待狀態。此行為與傳統的 ARM7TDMI 相同。

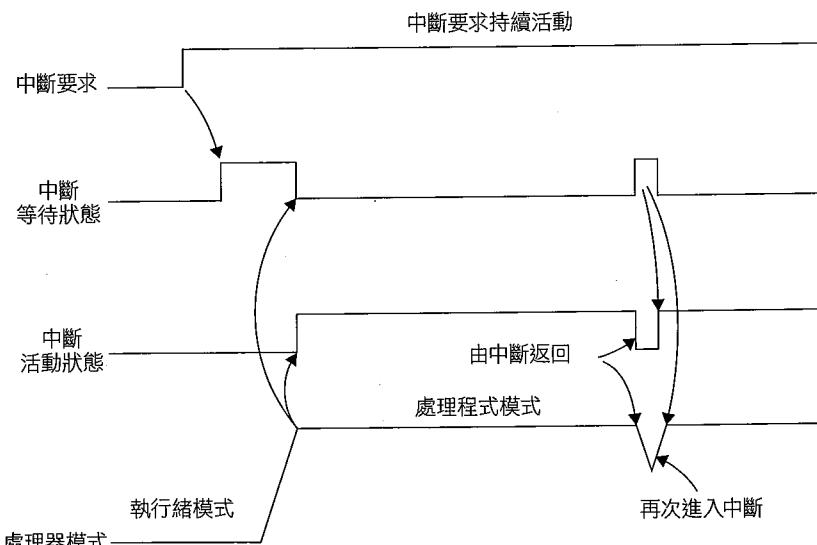


圖 7-12 當離開中斷後，持續的中斷要求再次處於等待狀態

如圖 7-13 所示，如果一個中斷在處理器開始處理它之前被多次要求，則會被當作一個單獨的中斷要求。

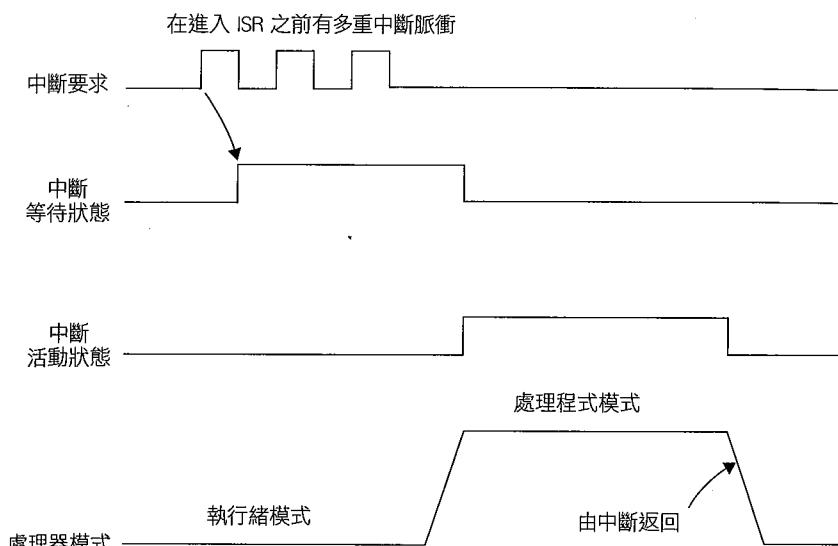


圖 7-13 持續的中斷等待僅發生一次，即使處理程式開始前有多次的中斷脈衝

當一個中斷被停止宣稱，但於中斷服務程式執行期間再被提出要求，它將如圖 7-14 所示，再次處於等待狀態。

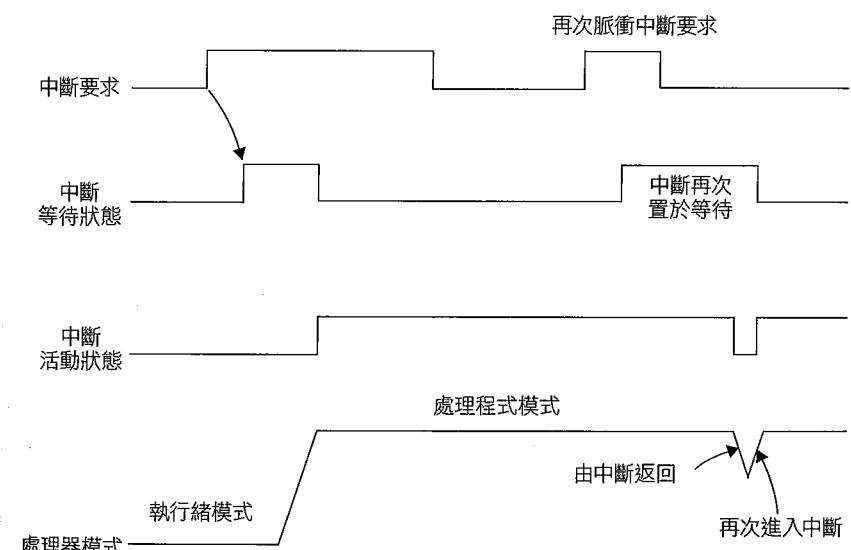


圖 7-14 於處理程式進行中，中斷等待再次發生

即使在中斷被禁能時，其中斷的等待也可能發生；等待的中斷可於後來設定致能時，去觸發中斷序列。因此，在致能中斷前，先檢查等待暫存器是否已經被設定對你可能會幫助，因為中斷來源可能先前曾經被觸發過並設定了等待狀態。在必要時，你可以有幫助，在致能一個中斷前，先清除等待狀態。

## 錯誤例外

一些系統例外對錯誤處理有幫助；錯誤可歸結如下幾個種類：

- ◆ 汇流排錯誤
- ◆ 記憶體管理錯誤
- ◆ 用法錯誤
- ◆ 硬錯誤

## 匯流排錯誤

在 AHB 介面上傳輸期間如果收到錯誤的回應，則產生匯流排錯誤。它可能發生在下列這些階段：

- ◆ 指令擷取，通常稱作預取中止(prefetch abort)
- ◆ 資料讀取/寫入，通常稱作資料中止(data abort)

在 Cortex-M3 裡，匯流排錯誤也可能於下列動作發生：

- ◆ 在中斷處理開始的堆疊 PUSH，稱為堆疊錯誤
- ◆ 在中斷處理結束的堆疊 POP，稱為去堆疊(unstacking)錯誤
- ◆ 在處理器開始中斷處理程序時，中斷向量位址(向量擷取)的讀取(被歸類為硬錯誤的特例)

### 什麼情形下會造成 AHB 錯誤回應？

當 AHB 汇流排上接收到錯誤回應，則發生了匯流排錯誤。常見造成的原因如下：

- ◆ 嘗試去存取一個不合法的記憶體區域(例如，沒有連結記憶體的記憶體位置)
- ◆ 設備尚未準備好接受傳輸(例如，在未初始化 SDRAM 控制器時，試著去存取 SDRAM)
- ◆ 嘗試去執行一個傳輸，但其傳輸的大小並不被目標設備支援(例如，對一個應用以 word 存取的周邊暫存器作 byte 的存取)
- ◆ 因為各種不同的理由，設備不接受傳輸(例如，某個周邊僅可以在特權存取等級裡透過程式碼存取)

當這些型態的匯流排錯誤(除了向量擷取外)產生時，如果匯流排處理程式被致能，且沒有相同或更高優先權的其它例外正在執行，則匯流排錯誤處理程式會被執行。如果匯流排錯誤處理程式被致能，但同時核心收到其它更高優先權的例外處理程式，則此匯流排錯誤例外會處於等待狀態。最後，如果匯流排錯誤處理程式未被致能，或者匯流排錯誤發生在一個例外處理程式執行時，且此例外處理程式比起匯流排錯誤處理程式有著相同或更高的優先權，則硬錯誤處理程式將代之執行。當執行硬錯誤處理程式時，如果有其它的匯流排錯誤發生，核心將會進入鎖住(lockup)狀態<sup>2</sup>。

欲致能匯流排錯誤處理程式，你需要設定 NVIC 裡系統處理程式控制與狀態暫存器中的 BUSFAULTENA 位元。在做這個動作之前，如果向量表被重新定位到 RAM 中，則要確定匯流排錯誤處理程式的起始位址已於向量表中設定。

所以，當處理器進入匯流排錯誤處理程式，你如何找出錯誤所在？在 NVIC 裡有一些錯誤狀態暫存器，其中之一為匯流排錯誤狀態暫存器(Bus Fault Status Register, BCSR)，匯流排錯誤處理程式可藉著此暫存器，去決定錯誤是由資料/指令存取，或是中斷的堆入堆疊或去堆疊運算造成。

確切的匯流排錯誤，可藉著被堆入堆疊的程式計數器以得到惡意指令所在；如果設定了 BCSR 裡的 BFARVALID 位元，則也可能找出造成匯流排錯誤的記憶體錯誤的位置所在，此可藉由讀取另一個稱作匯流排錯誤位址暫存器(Bus Fault Address Register, BFAR)的 NVIC 暫存器來達成。然而，不確切的匯流排錯誤，並無法得到上面相同的資訊，因為在處理器收到錯誤之前，可能已經執行了一些其它的指令。

<sup>2</sup> 更多有關鎖住狀態的資訊於第十二章中討論。

### 確切的與不確切的匯流排錯誤

資料存取造成的匯流排錯誤可進一步歸類為確切的與不確切的。不確切的匯流排錯誤裡，錯誤由可能發生於幾個時脈週期前已經完成的指令(例如被緩衝的寫入)所造成。確切的匯流排錯誤為最後完成的運算所造成，例如，在 Cortex-M3 上記憶體讀取就是確切的，因為在它接收到資料之前，指令就還未完成。

BFSR 的程式設計師模型如下：其為 8 bits 寬，並且可經 byte 傳輸或者 word 傳輸到位址 0xE000ED28 來作存取，其中 BFSR 為第二個 byte(見表 7-8)。當寫進 1 到錯誤指示位元時，該位元就會被清除。

表 7-8 匯流排錯誤狀態暫存器(0xE000ED29)

位元	名稱	類型	重置值	描述
7	BFARVALID	-	0	顯示 BFAR 為合法
6:5	-	-	-	-
4	STKERR	R/Wc	0	堆入堆疊錯誤
3	UNSTKERR	R/Wc	0	去堆疊錯誤
2	IMPRECISERR	R/Wc	0	不確切的資料存取違法
1	PRECISERR	R/Wc	0	確切的資料存取違法
0	IBUSERR	R/Wc	0	指令存取違法

## 記憶體管理錯誤

造成記憶體管理錯誤的原因包括：違反 MPU 設定的記憶體存取，和某些不合法的存取(例如，試著由不可執行的記憶體區域執行程式)，即使沒有 MPU 的存在，這些不合法的存取也能觸發錯誤。

一些常見的 MPU 錯誤包括下列：

- ◆ 存取未定義在 MPU 設定的記憶體區域
- ◆ 寫入唯讀區域
- ◆ 於用戶狀態下存取一個定義為僅能被特權存取的區域

當記憶體管理錯誤發生，且記憶體管理錯誤處理程式被致能，則將執行記憶體管理錯誤處理程式。如果此錯誤發生的同時也有更高優先權例外發生，則將先行處理其它例

外且置記憶體管理錯誤於等待狀態。如果處理器正在執行具相同或更高優先權的例外處理程式，或者記憶體管理錯誤處理程式沒有被致能，則反而會執行硬錯誤處理程式。如果記憶體管理錯誤發生於硬錯誤處理程式內或 NMI 處理程式內，處理器將進入鎖住狀態。

相同於匯流排錯誤處理程式，記憶體管理錯誤處理程式需要被致能，這可由設定 NVIC 裡系統處理程式控制與狀態暫存器中的 MEMFAULTENA 位元來達成。如果向量表已經被重新定位到 RAM，記憶體管理錯誤處理程式的起始位址就需要先在向量表中設定好。

NVIC 包括了一個記憶體管理錯誤狀態暫存器(Memory Management Fault Status Register, MFSR)，來顯示記憶體管理錯誤的原因。如果此狀態暫存器顯示了錯誤為資料存取違規(DACCVIOL 位元)或者指令存取違規(IACCVIOL 位元)，則可藉著堆入堆疊的程式計數器找出導致錯誤的程式碼。如果設定了 MFSR 裡的 MMARVALID 位元，則可能由 NVIC 裡的記憶體管理位址暫存器(Memory Management Address Register, MMAR)來決定造成錯誤的記憶體位置所在。

表 7-9 顯示了程式師的 MFSR 模型。其為 8 bits 寬，並且可經由 byte 傳輸或者 word 傳輸到位址 0xE000ED28 來作存取，其中 MFSR 為最低 byte。同於其它錯誤狀態暫存器，當寫進 1 的時候，錯誤狀態位元會被清除。

表 7-9 記憶體管理錯誤狀態暫存器(0xE000ED28)

位元	名稱	類型	重置值	描述
7	MMARVALID	-	0	顯示 MMAR 合法
6:5	-	-	-	-
4	MSTKERR	R/Wc	0	堆入堆疊錯誤
3	MUNSTKERR	R/Wc	0	去堆疊錯誤
2	-	-	-	-
1	DACCVIOL	R/Wc	0	資料存取違法
0	IACCVIOL	R/Wc	0	指令存取違法

## 用法錯誤

造成用法錯誤的情形如下：

- ◆ 未定義的指令
- ◆ 幫助處理器指令(Cortex-M3 處理器並無支援輔助處理器, 但可以用法錯誤例外機制, 藉由輔助處理器模擬, 來執行為了其它 Cortex 處理器編譯的軟體)
- ◆ 試著切換到 ARM 狀態(軟體可以使用此錯誤機制來測試目前執行軟體的處裡器是否支援 ARM 程式; 因為 Cortex-M3 並不支援 ARM 狀態, 如果嘗試切換將造成用法錯誤產生)
- ◆ 不合法的中斷回傳(連結暫存器包含不合法/不正確的值)
- ◆ 使用多個載入或儲存指令來作未對齊的記憶體存取
- 藉著設定 NVIC 裡的某些控制位元, 亦可能產生如下用法錯誤:
- ◆ 除以零
- ◆ 任何未對齊的記憶體存取

當用法錯誤出現而且用法錯誤處理程式被致能, 正常情形下, 將執行用法錯誤處理程式。然而, 如果同時也有更高優先權例外發生, 則將置用法錯誤於等待狀態。如果處理器正在執行具相同或更高優先權的例外處理程式, 或者用法錯誤處理程式未被致能, 則反而會執行硬錯誤處理程式。如果用法錯誤發生於硬錯誤處理程式內或 NMI 處理程式內, 處理器將進入鎖住狀態。

欲致能用法錯誤處理程式, 可藉著設定 NVIC 裡系統處理程式控制與狀態暫存器中的 USGFAULTENA 位元來達成。如果向量表已經被重新定位到 RAM, 用法錯誤處理程式的起始位址就需要先在向量表中設定好。

NVIC 為用法錯誤處理程式提供了用法錯誤狀態暫存器(Usage Fault Status Register, UFSR)來決定錯誤造成原因。於處理程式內, 造成錯誤的程式碼也可由堆入堆疊的程式計數器找出。

### (意外地切換至 ARM 狀態)

最常造成用法錯誤的原因之一, 是意外地試著切換處理器到 ARM 模式。如果你把一個 LSB 為 0 的值載入到 PC, 就會造成此意外。例如: 如果你試著跳躍到暫存器裡的一個位址(BX LR), 但未設定其 LSB; 或是在例外向量表裡某個向量的 LSB 為 0; 或以 POP[PC]讀出堆入堆疊的 PC 值, 被手動修改而清除了 LSB。當這些情形發生時, 將出現用法錯誤例外, 且設定了 UFSR 裡的 INVSTATE 位元。

表 7-10 列出了 UFSR。其佔用 2 bytes, 並且可經由 half word 傳輸或者 word 傳輸到位址 0xE000ED28 來作存取, 其中 UFSR 為上半 word。同於其它錯誤狀態暫存器, 錯誤狀態位元可藉著寫進 1 至此位元來清除。

表 7-10 用法錯誤狀態暫存器(0xE000ED2A)

位元	名稱	類型	重置值	描述
9	DIVBYZERO	R/Wc	0	顯示發生了除以 0(僅可在 DIV_0_TRP 被設定時設定)
8	UNALIGNED	R/Wc	0	顯示發生了未對齊的存取錯誤
7:4	-	-	-	-
3	NOCP	R/Wc	0	試圖執行輔助處理器指令
2	INVPC	R/Wc	0	試圖在 EXC_RETURN 號碼裡以不正確的數值去執行例外
1	INVSTATE	R/Wc	0	試圖切換到不合法的狀態(例如, ARM 狀態)
0	UNDEFINSTR	R/Wc	0	試圖執行未定義的指令

## 硬錯誤

硬錯誤處理程式成因在於用法錯誤、匯流排錯誤、和記憶體管理錯誤等處理程式無法執行時。此外, 其成因也可能是向量擷取時(於例外處理時對向量表的讀取)的匯流排錯誤。在 NVIC 裡有一個硬錯誤狀態暫存器, 可用來決定是否因向量擷取造成錯誤, 如果不是的話, 硬錯誤處理程式將需要檢查其它的錯誤狀態暫存器, 以決定造成硬錯誤的原因。

硬錯誤狀態暫存器(Hard Fault Status Register, HFSR)的細節列於表 7-11。與其它錯誤狀態暫存器相同, 錯誤狀態位元可藉著寫進 1 至此位元來清除。

表 7-11 硬錯誤狀態暫存器(0xE000ED2C)

位元	名稱	類型	重置值	描述
31	DEBUGEVT	R/Wc	0	顯示硬錯誤被除錯事件所觸發
30	FORCED	R/Wc	0	顯示因為匯流排錯誤、記憶體管理錯誤、或用法錯誤而採取了硬錯誤
29:2	-	-	-	-
1	VECTBL	R/Wc	0	顯示因為向量擷取失敗而造成硬錯誤
0	-	-	-	-

## 應付錯誤的方式

在軟體開發過程，我們可以利用錯誤狀態暫存器(Fault Status Registers, FSRs)以決定程式中造成錯誤的原因並加以更正。本書附錄 E 包含了對各類錯誤常見原因的判斷指引。在實際執行中的系統，情況則不同。在決定了造成錯誤的原因後，軟體需要決定下一步該如何進行。系統若執行作業系統，則可中止惡意的工作或應用；否則，系統可能需要重置。至於是否需要由錯誤復原，則隨目標應用而定。處理得當的話可使產品更為強健，但最好從一開始就能避免錯誤產生。下列為一些處理錯誤的方法：

- ◆ **重置：**此可藉著 NVIC 裡應用中斷與重置控制暫存器中的 VECTRESET 控制位元來進行，此將重置處理器而非整個晶片。取決於晶片的重置設計，某些 Cortex-M3 晶片可藉著上述相同的暫存器中的 SYSRESETREQ 來重置，如此則可以提供一個全系統的重置。
- ◆ **復原：**某些情形下，或許可能解決造成錯誤例外的問題。例如，如果是輔助處理器指令的情形，則可藉由使用輔助處理器軟體來解決。
- ◆ **工作中止：**系統若執行作業系統，極可能中止造成錯誤的工作，必要時也可重新啟動它。

FSRs 會保留狀態直到被手動清除。錯誤處理程式應當清除他們動用過的錯誤狀態位元。否則，當下一次其他的錯誤產生時，錯誤處理程式會再次被啟動並誤認先前錯誤仍存在，因而重新處理錯誤。FSRs 使用了一個寫入以清除的機制(藉著寫入 1 到需要被清除的位元以清除狀態)。

晶片製造廠商可能也會在晶片裡包括了一個輔助 FSR(即 AFSR)來顯示其他錯誤情形。AFSR 的建置隨個別晶片設計需求而定。

## SVC 與 PendSV

系統服務呼叫(System Service Call, SVC)與等待系統呼叫(Pended System Call, PendSV)為以軟體與作業系統為目標的兩個例外。SVC 用來產生系統功能呼叫；例如，作業系統可以經由 SVC 提供對硬體作存取，以取代允許用戶程式直接存

取硬體的行為。所以當用戶程式欲使用某個硬體時，它藉著 SVC 指令以產生 SVC 例外，接著作業系統裡的軟體例外處理程式會被執行並提供用戶應用要求的服務。如此，硬體的存取會受到作業系統控制，此做法可以避免用戶應用程式直接存取硬體，而提供了一個更強健的系統。

SVC 也能使軟體更具可移植性，因為用戶應用並不需要了解硬體的程式細節。用戶程式將僅需要去了解應用程式介面(Application Programming Interface, API)功能 ID 與參數；實際的硬體等級程式由設備驅動程式來處理。

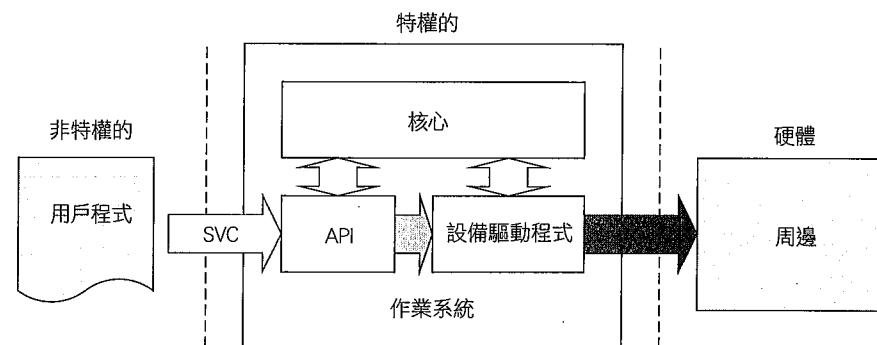


圖 7-15 SVC 作為作業系統功能的閘道

使用 SVC 指令以產生 SVC；此指令需要使用立即值，加上利用傳參數的方式來達成，SVC 例外處理程式接著取出參數，並決定它需要採取的行動。例如：

```
SVC 0x3 ; 呼叫 SVC 功能 3
```

當 SVC 處理程式被執行時，你可以找出 SVC 指令裡的立即資料值，其步驟如下述：先讀取被堆入堆疊的程式計數器的值，再由此位址讀指令，並遮罩其用不到的位元。如果系統的用戶應用程式使用了 PSP，你可能需先決定使用了哪個堆疊，此可由進入處理程式時的連結暫存器來作決定。(第八章會對此主題進一步解說)。

受限於 Cortex-M3 裡的中斷優先權模型，在 SVC 處理程式內，你不能再使用 SVC(因其優先權與進行中的優先權相同)，若再使用，將造成用法錯誤。同理，你不能在 NMI 處理程式內與硬錯誤處理程式內，再使用 SVC。

### 框(SVC 與 SWI(ARM7))

如果你使用過傳統的 ARM 處理器(例如 ARM7), 你可能知道有一個軟體中斷指令(Software Interrupt Instruction, SWI)。SVC 有一個類似功能, 並且事實上 SVC 指令的二進位編碼與 ARM7 裡的 SWI 相同。然而, 因為中斷模型已被更改, 故更名此指令, 以保證程式設計師能適當地從 ARM7 移植軟體程式到 Cortex-M3。

在作業系統中 PendSV 與 SVC 共同運作。雖然 SVC(藉由 SVC 指令)不能等待(呼叫 SVC 的應用, 期望其需求的工作能立即完成), 但 PendSV 可以等待, 所以作業系統可以利用它來把例外置於等待狀態, 故可於其他重要工作完成後再執行其動作。PendSV 可藉由寫入 1 至 NVIC PendSV 的等待暫存器來產生。

PendSV 的一個典型使用為環境切換(於工作之間切換)。例如, 系統可能會有兩個活動中的工作, 環境切換以下列方式觸發：

◆ 呼叫 SVC 功能

◆ 系統計時器(SYSTICK)

讓我們看一個簡單的例子：系統僅有兩個工作, 且 SYSTICK 例外觸發了一個環境切換(參見圖 7-16)。

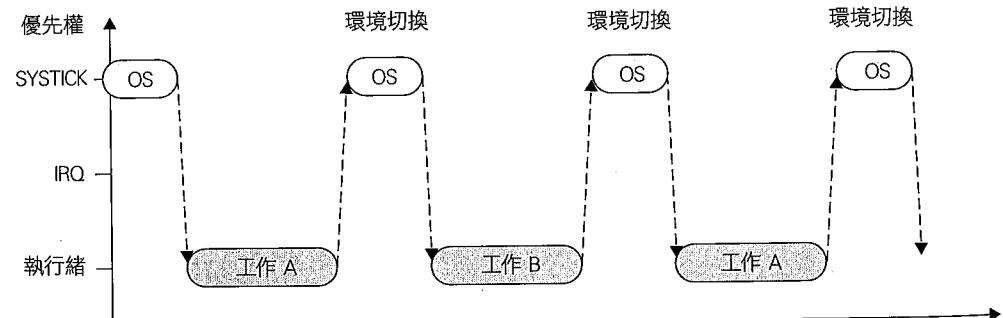


圖 7-16 使用 SYSTICK 在兩個工作間切換的一個簡單的情節

如果在 SYSTICK 例外發生之前出現了一個中斷要求, SYSTICK 例外將強佔 IRQ 處理程式。在此情形下, 作業系統不應當執行環境切換, 否則 IRQ 處理程式的程序將被延遲, 且對 Cortex-M3 來說, 若作業系統於中斷活動期間試著切換到執行緒模式, 則可能會產生用法錯誤。

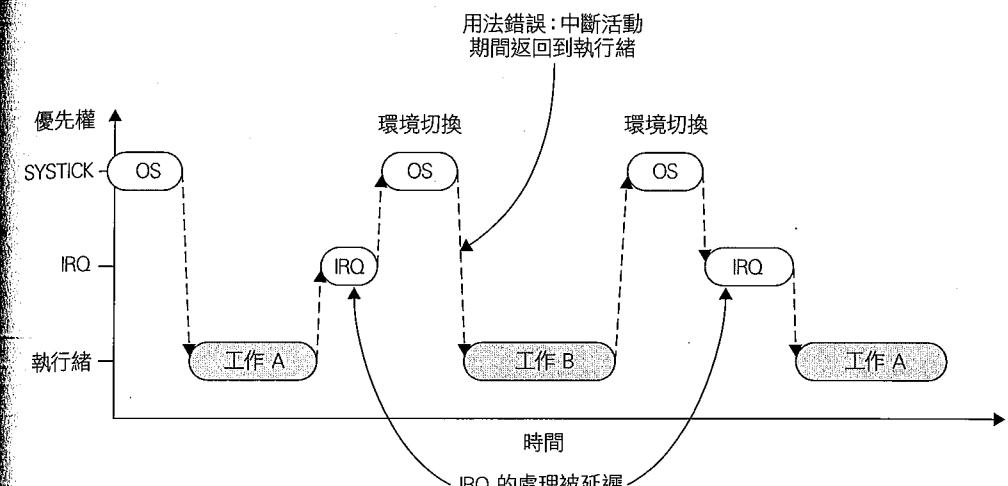


圖 7-17 在 IRQ 期間的環境切換的問題

為了避免延遲 IRQ 處理的問題, 一些作業系統僅在偵測到目前沒有 IRQ 處理程式在執行時才會做環境切換。然而, 這可能導致工作切換非常長的延遲, 特別當中斷來源與 SYSTICK 例外的頻率靠近的時候。

PendSV 例外藉著延遲環境切換的要求, 直到所有其他的 IRQ 處理程式完成其處理, 而解決了問題。為此, PendSV 被設定為最低優先權的例外。如果作業系統偵測到一個正活動中的 IRQ(IRQ 處理程式正執行且會被 SYSTICK 強佔), 它會把 PendSV 例外置於等待狀態, 以延遲環境的切換。

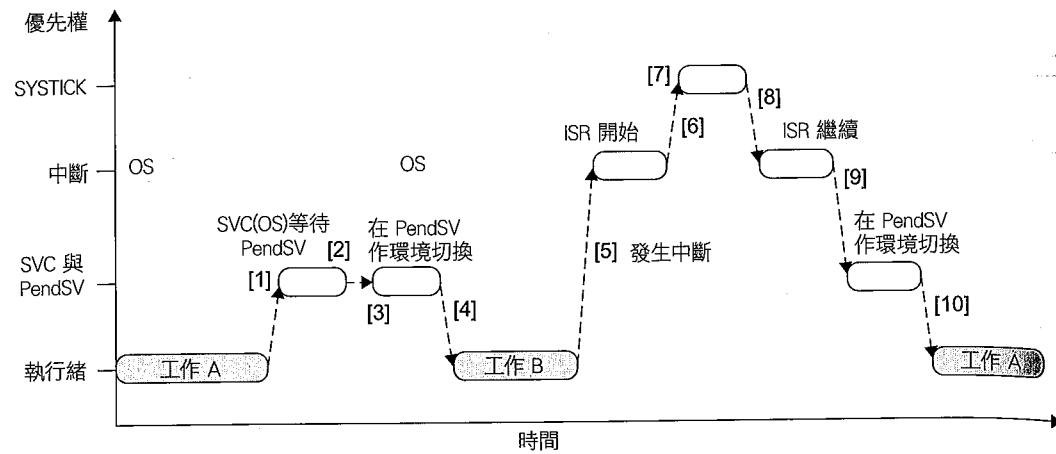


圖 7-18 使用 PendSV 作環境切換的例子

1. 工作 A 呼叫 SVC 以切換工作(例如, 等待某工作完成)。
2. 作業系統接到請求, 為環境切換作準備, 置 PendSV 例外於等待。
3. 當 CPU 離開 SVC, 它立刻進入 PendSV 並作環境切換。
4. 當 PendSV 完成並返回執行緒層級, 於是執行工作 B。
5. 中斷出現並進入中斷處理程式。
6. 當執行中斷處理程式時, SYSTICK 例外(OS tick)發生。
7. 作業系統執行必要的運算, 接著置 PendSV 例外於等待, 並做好環境切換的準備。
8. 當離開 SYSTICK 例外, 於是返回中斷服務程式。
9. 當中斷服務程式完成時, PendSV 啟動並實際執行環境切換運算。
10. 當 PendSV 完成, 程式返回執行緒層級;此時, 它返回工作 A, 並且繼續進行處理。

Chapter

8

## NVIC 與中斷控制

本章內容包括：

- ✓ NVIC 總觀
- ✓ 基本中斷組態
- ✓ 中斷致能與清除致能
- ✓ 中斷等待與清除等待
- ✓ 設定中斷的範例程序
- ✓ 軟體中斷
- ✓ SYSTICK 計時器

### NVIC 總觀

如我們已經看到的, 巢狀向量中斷控制器(Nested Vectored Interrupt Controller, NVIC) 為完整的 Cortex-M3 的一部分。它緊密地連結到 Cortex-M3 CPU 的核心邏輯。它的控制暫存器以記憶體映射裝置的方式存取。除了作中斷處理的控制暫存器與控制邏輯外, NVIC 也包含作 MPU、SYSTICK 計時器、以及除錯控制等的控制暫存器。於此章我們將檢驗作中斷處理的控制邏輯, MPU 與除錯的控制邏輯將於往後的章節裡討論。

NVIC 支援 1 到 240 個外部中斷輸入(通常稱之為 IRQs), 精確的數目由晶片製造廠商於開發他們的 Cortex-M3 晶片時決定。此外, NVIC 也有一個不可遮罩中斷(Nonmaskable Interrupt, NMI)輸入。NMI 的實際功能亦由晶片製造廠商決定, 在某些情形下, 此 NMI 不能由外部的來源控制。

NVIC 可用記憶體位址 0xE000E000 來存取。大部分的中斷控制/狀態暫存器僅可於特權模式作存取，但軟體觸發中斷暫存器則例外，經由設定後，它可於用戶模式作存取。中斷控制/狀態暫存器能以 word、half word、或 byte 作傳輸。

此外，一些其它中斷-遮罩暫存器也參與了中斷，也就是第三章提及的特殊暫存器，可藉由 MRS 與 MSR 指令來存取。

## 基本中斷組態

每一個外部中斷都與幾個暫存器相關聯：

- ◆ 致能與清除致能暫存器
- ◆ 設定-等待與清除-等待暫存器
- ◆ 優先權限等級
- ◆ 活動狀態

此外，其它一些暫存器也影響了中斷處理：

- ◆ 例外-遮罩暫存器(PRIMASK、FAULTMASK、以及 BASEPRI)
- ◆ 向量表位移暫存器
- ◆ 軟體觸發中斷暫存器
- ◆ 優先權限群組

## 中斷致能與清除致能

中斷致能暫存器經由兩個位址設定：欲設定致能位元，你需寫入 SETENA 暫存器位址；欲清除致能位元，你需寫入 CLRENA 暫存器位址。如此，致能或者除能中斷，將不會影響其它中斷致能狀態。SETENA/CLRENA 暫存器為 32 位元寬；每一位元代表一個中斷輸入。

因為 Cortex-M3 處理器裡可能有多於 32 個外部中斷，你可能會找到多於一個的 SETENA 和 CLRENA 暫存器，例如：SETENA0、SETENA1 等等(參見表 8-1)。僅有存在的中斷致能位元才有實作，所以如果你只有 32 個中斷輸入，你將只會有 SETENA0 與 CLRENA0。SETENA 與 CLRENA 暫存器能以 word、half word、或 byte 作存取。因為最先的 16 個例外種類為系統例外，故外部中斷#0 其起始例外號碼為 16(參見表 7-2)。

表 8-1 中斷設定致能暫存器與中斷清除致能暫存器 (0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C)

位址	名稱	類型	重置值	描述
0xE000E100	SETENA0	R/W	0	作為致能外部中斷#0-31 使用 Bit(0)為中斷#0 (例外#16) Bit(1)為中斷#1 (例外#17) ... Bit(31)為中斷#31 (例外#47) 寫入 1 以設定位元為 1；寫入 0 則無影響 讀取值顯示現行狀態
0xE000E104	SETENA1	R/W	0	作為致能外部中斷#32-63 使用 寫入 1 以設定位元為 1；寫入 0 則無影響 讀取值顯示現行狀態
0xE000E108	SETENA2	R/W	0	作為致能外部中斷#64-95 使用 寫入 1 以設定位元為 1；寫入 0 則無影響 讀取值顯示現行狀態
...	-	-	-	-
0xE000E180	CLRENA0	R/W	0	用以清除外部中斷#0-31 之致能 Bit(0)為中斷#0 (例外#16) Bit(1)為中斷#1 (例外#17) ... Bit(31)為中斷#31 (例外#47) 寫入 1 以清除位元為 0；寫入 0 則無影響 讀取值顯示現行狀態
0xE000E184	CLRENA1	R/W	0	用以清除外部中斷#32-63 之致能 寫入 1 以清除位元為 0；寫入 0 則無影響 讀取值顯示現行狀態
0xE000E188	CLRENA2	R/W	0	用以清除外部中斷#64-95 之致能 寫入 1 以清除位元為 0；寫入 0 則無影響 讀取值顯示現行狀態
...	-	-	-	-

## 中斷等待與清除等待

當中斷要求出現但不能立即執行(例如, 正執行其它更高優先權中斷處理程式), 它會處於等待狀態。中斷等待狀態可藉著中斷設定等待(Interrupt Set Pending, SETPEND)與中斷清除等待(Interrupt Clear Pending, CLRPEND)兩個暫存器來作存取。與致能暫存器類似, 如果有多於 32 個外部中斷輸入, 則等待狀態控制可能會包含兩個以上的暫存器。

由於等待狀態暫存器可以更改, 故你可以取消正在等待的例外, 或者經由 SETPEND 暫存器以產生軟體中斷(參見表 8-2)。

表 8-2 中斷設定等待暫存器與中斷清除等待暫存器 (0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C)

位址	名稱	類型	重置值	描述
0xE000E200	SETPEND0	R/W	0	設定外部中斷#0-31 的等待 Bit[0]為中斷#0 (例外#16) Bit[1]為中斷#1 (例外#17) ... Bit[31]為中斷#31 (例外#47) 寫入 1 以設定位元為 1; 寫入 0 則無影響 讀取值顯示現行狀態
0xE000E204	SETPEND1	R/W	0	設定外部中斷#32-63 的等待 寫入 1 以設定位元為 1; 寫入 0 則無影響 讀取值顯示現行狀態
0xE000E208	SETPEND2	R/W	0	設定外部中斷#64-95 的等待 寫入 1 以設定位元為 1; 寫入 0 則無影響 讀取值顯示現行狀態
...	-	-	-	-
0xE000E280	CLRPEND0	R/W	0	用以清除外部中斷#0-31 之等待 Bit[0]為中斷#0 (例外#16) Bit[1]為中斷#1 (例外#17) ... Bit[31]為中斷#31 (例外#47) 寫入 1 以清除位元為 0; 寫入 0 則無影響 讀取值顯示現行狀態
0xE000E284	CLRPEND1	R/W	0	用以清除外部中斷#32-63 之等待 寫入 1 以清除位元為 0; 寫入 0 則無影響 讀取值顯示現行狀態
0xE000E288	CLRPEND2	R/W	0	用以清除外部中斷#64-95 之等待 寫入 1 以清除位元為 0; 寫入 0 則無影響 讀取值顯示現行狀態
...	-	-	-	-

## 優先權等級

每一個外部中斷有與其相關的優先權等級暫存器(最大為 8 位元, 最小為 3 位元)。如前章所描述, 每一個暫存器根據優先權群組設定, 可進一步區分為強佔優先權等級與次優先權等級。優先權等級暫存器能以 byte、half word、或 word 作存取。隨著晶片包含的外部中斷而決定了優先權等級暫存器的數目(參見表 8-3)。優先權等級組態暫存器的細節可參考附錄 D 的表 D-18。

表 8-3 中斷優先權等級暫存器(0xE000E400-0xE000E4EF)

位址	名稱	類型	重置值	描述
0xE000E400	PRI_0	R/W	0 (8-bit)	優先權等級外部中斷#0
0xE000E401	PRI_1	R/W	0 (8-bit)	優先權等級外部中斷#1
...	-	-	-	-
0xE000E41F	PRI_31	R/W	0 (8-bit)	優先權等級外部中斷#31
...	-	-	-	-

## 活動狀態

每一外部中斷有其活動狀態位元。當處理器啟動中斷處理程式, 設定此位元為 1; 當執行了中斷返回時, 則清除之。然而, 在中斷服務程式執行過程中, 可能出現更高優先權的中斷而造成強佔; 在此時期, 雖然事實上處理器正執行其它中斷處理程式, 先前的中斷仍然被定義為活動狀態。活動暫存器為 32 位元但也可以利用 half word 或 byte 大小來存取。如果, 外部中斷超過 32 個, 則會有一個以上的活動暫存器。外部中斷的活動狀態暫存器為唯讀的(參見表 8-4)。

表 8-4 中斷活動狀態暫存器(0xE000E300-0xE000E31C)

位址	名稱	類型	重置值	描述
0xE000E300	ACTIVE0	R	0	設定外部中斷#0-31 的活動狀態 Bit[0]為中斷#0 Bit[1]為中斷#1 ... Bit[31]為中斷#31
0xE000E304	ACTIVE1	R	0	設定外部中斷#32-63 的活動狀態
...	-	-	-	-

## PRIMASK 與 FAULTMASK 特殊暫存器

PRIMASK 暫存器是用來對 NMI 或硬錯誤之外的所有例外作除能。它有效地將現行優先權等級改變為 0(最高可程式的等級)。此暫存器可藉著 MRS 與 MSR 指令來設定。例如：

```
MOV R0, #1  
MSR PRIMASK, R0 ; 將 1 寫入 PRIMASK 以除能所有中斷
```

```
MOV R0, #0  
MSR PRIMASK, R0 ; 將 0 寫入 PRIMASK 以允許中斷
```

因為 PRIMASK 可暫時地除能所有中斷，因此它對關鍵性的工作有幫助。當 PRIMASK 被設定時，若錯誤產生，硬錯誤處理程式將被執行。

FAULTMASK 作用類似 PRIMASK，但它更改有效的現行優先權等級至-1，故甚至硬錯誤處理程式亦被擋住。當設定了 FAULTMASK 時，僅 NMI 可被執行。

在離開例外處理程式時，FAULTMASK 會被自動清除。FAULTMASK 與 PRIMASK 兩個暫存器皆不可在用戶狀態裡設定。

## BASEPRI 特殊暫存器

在某些情形下，你可能希望去除能優先權低於特定等級的中斷。此時，你可以使用 BASEPRI 暫存器。為此，僅需將所要的遮罩優先權等級，寫入 BASEPRI 暫存器。例如，如果要阻擋優先權等級小於或等於 0x60 的所有例外，可把此值寫入 BASEPRI：

```
MOV R0, #0x60  
MSR BASEPRI, R0 ; 除能優先權介於 0x60 ~ 0xFF 的中斷
```

要取消遮罩，僅需寫 0 到 BASEPRI 暫存器：

```
MOV R0, #0x0  
MSR BASEPRI, R0 ; 取消對 BASEPRI 的遮罩
```

BASEPRI 暫存器也可以使用 BASEPRI\_MAX 暫存器名稱來存取，實際上，它意指相同的暫存器，但當你使用此名稱時，將會有條件式寫入保護。(僅與硬體牽涉時，BASEPRI 與 BASEPRI\_MAX 為相同暫存器，但於組繹器程式裡，他們使用不同的暫存器名稱編碼)。當你使用 BASEPRI\_MAX 為暫存器，僅可改為更高的優先權等級，而不能改為較低的優先權等級。例如，考慮下列指令序列：

```
MOV R0, #0x60  
MSR BASEPRI_MAX, R0 ; 除能優先權為 0x60、0x61、等等的中斷  
  
MOV R0, #0xF0  
MSR BASEPRI_MAX, R0 ; 此寫入指令會被忽略，  
                      ; 因它的等級低於 0x60  
  
MOV R0, #0x40  
MSR BASEPRI_MAX, R0 ; 此寫入指令會被允許  
                      ; 而更改遮罩等級至 0x40
```

欲更改至較低的遮罩等級或除能遮罩，則需使用 BASEPRI 暫存器名稱。BASEPRI/BASEPRI\_MAX 暫存器不可於用戶狀態裡設定。

與其它優先權等級暫存器一樣，BASEPRI 暫存器的格式，受到建置的優先權暫存器欄位的數目影響。例如，如果優先權等級暫存器僅建置了 3 個位元，BASEPRI 可規劃為 0x00, 0x20, 0x40, …0xC0, 0xE0。

## 其它例外的組態暫存器

用法錯誤、記憶體管理錯誤、匯流排錯誤例外等，是由系統處理程式控制與狀態暫存器所致能(0xE000ED24)。錯誤的等待狀態與大部分系統例外的活動狀態，也可於此暫存器獲得(參見表 8-5)。

表 8-5 系統處理程式控制與狀態暫存器(0xE000ED24)

位元	名稱	類型	重置值	描述
18	USGFAULTENA	R/W	0	致能用法錯誤處理程式
17	BUSFAULTENA	R/W	0	致能匯流排錯誤處理程式
16	MEMFAULTENA	R/W	0	致能記憶體管理錯誤
15	SVCALLPENDED	R/W	0	SVC 被置於等待；SVCall 被啟動但又被更高的優先權之例外所取代
14	BUSFAULTPENDED	R/W	0	匯流排錯誤被置於等待；匯流排錯誤處理程式被啟動但又被更高的優先權之例外所取代
13	MEMFAULTPENDED	R/W	0	記憶體管理錯誤被置於等待；記憶體管理錯誤被啟動但又被更高的優先權之例外所取代
12	USGFAULTPENDED	R/W	0	用法錯誤被置於等待；用法錯誤被啟動但又被更高的優先權之例外所取代
11	SYSTICKACT	R/W	0	如果 SYSTICK 例外處於活動中，則讀出值為 1
10	PENDSVACT	R/W	0	如果 PendSV 例外處於活動中，則讀出值為 1
8	MONITORACT	R/W	0	如果除錯監視例外處於活動中，則讀出值為 1
7	SVCALLACT	R/W	0	如果 SVCall 例外處於活動中，則讀出值為 1
3	USGFAULTACT	R/W	0	如果用法錯誤例外處於活動中，則讀出值為 1
1	BUSFAULTACT	R/W	0	如果匯流排錯誤例外處於活動中，則讀出值為 1
0	MEMFAULTACT	R/W	0	如果記憶體管理錯誤處於活動中，則讀出值為 1

注意：在 Cortex-M3 的 revision 0 中，並無 bit 12 (USGFAULTPENDED)可用。

當寫入此暫存器時需小心，以確保不會意外地更改系統例外的活動狀態位元。否則，當意外地清除了活動中系統例外的活動狀態，會在系統例外處理程式產生例外離開時，導致錯誤例外。

經由中斷控制與狀態暫存器，可規劃 NMI、SYSTICK 計時器、與 PendSV 的等待狀態。此暫存器有相當多的位元欄位是作為除錯用途，在大部分的情形下，僅有等待位元對開發應用程式有幫助(參見表 8-6)。

表 8-6 中斷控制與狀態暫存器(0xE000ED04)

位元	名稱	類型	重置值	描述
31	NMIPENDSET	R/W	0	NMI 被置於等待
28	PENDSVSET	R/W	0	寫入 1 以將系統呼叫置於等待 讀取值顯示等待狀態
27	PENDSVCLR	W	0	寫入 1 以清除 PendSV 之等待狀態 讀取值顯示等待狀態
26	PENDSTSET	R/W	0	寫入 1 以將 SYSTICK 例外置於等待 讀取值顯示等待狀態
25	PENDSTCLR	W	0	寫入 1 以清除 SYSTICK 之等待狀態 讀取值顯示等待狀態
23	ISRPREEMPT	R	0	指示等待中的中斷在下一步(除錯用)將會活動
22	ISR PENDING	R	0	外部中斷置於等待(但系統例外，例如 NMI 的錯誤，則除外)
21:12	VECTPENDING	R	0	ISR 號碼被置於等待
11	RETOBASE	R	0	當處理器執行例外處理程式時被設定為 1；如果由中斷返回並且無其他等待中之例外，則會返回執行級層級
9:0	VECTACTIVE	R	0	現在正執行的中斷服務程式

## 設定中斷的範例程序

此處為設定中斷的一個簡單範例程序：

1. 當系統啟動時，優先權群組暫存器可能需要被設定，經由預定，使用了優先權群組 0(故優先權等級 bit[7:1]為強佔等級，bit[0]為次優先權等級)。
2. 如果需要重新定位向量表，則需要將硬中斷與 NMI 處理程式複製到新的向量表位置。(簡單的應用可能不需如此)
3. 向量表位移暫存器也應設置好，以完成向量表的準備(可選擇性的)
4. 設定中斷的中斷向量。因為向量表可能已經被重新定位，你可能需要去讀向量表位移暫存器，再去計算你的中斷處理程式的正確記憶體位置。如果此向量被硬編碼(hardcoded)在 ROM 裡則此步驟可省略。
5. 設定中斷的優先權等級。
6. 致能中斷。

此程式以組合語言表示可能如下：

# Jason 嘴書—EETOP 世界唯一貼

```

LDR R0, = 0xE000ED0C ; 應用中斷與重置控制暫存器
LDR R1, = 0x05FA0500 ; 優先權群組 5 (2/6)
STR R1, [R0] ; 設定優先權群組

...
MOV R4, #8 ; ROM 中的向量表
LDR R5, = (NEW_VECT_TABLE+8)
LDMIA R4!, {R0 - R1} ; 讀 NMI 與硬錯誤的向量位址
STMIA R5!, {R0 - R1} ; 複製向量至新的向量表

...
LDR R0, = 0xE000ED08 ; 向量表位移暫存器
LDR R1, = NEW_VECT_TABLE
STR R1, [R0] ; 設定向量表到新位置
...
LDR R0, = IRQ7_Handler ; 取得 IRQ#7 處理暫存器的起始位址
LDR R1, = 0xE000ED08 ; 向量表位移暫存器
LDR R1, [R1]
ADD R1, R1, #(4*(7+16)) ; 計算 IRQ#7 處理暫存器向量位址
STR R0, [R1] ; 設定 IRQ#7 的向量
...
LDR R0, = 0xE000E400 ; 外部 IRQ 優先權基底
MOV R1, #0xC0
STRB R1, [R0, #7] ; 設定 IRQ#7 優先權到 0xC0
...
LDR R0, = 0xE000E100 ; SETEN 暫存器
MOV R1, #(1<<7) ; IRQ#7 致能位元(0x1 的值位移 7 個位元)
STR R1, [R0] ; 致能中斷

```

此外，如果你允許大量的巢狀中斷等級，則需確定你有足夠的堆疊記憶體。因為例外處理程式通常使用 MSP，故為了應付大量的巢狀中斷，主要堆疊記憶體應當包含足夠的空間。

如果應用程式存放在 ROM 裡面，且不需要更改例外處理程式，我們可把全部向量表編碼在程式區 ROM 的開始處(0x00000000)。這樣，向量表位移將永遠為 0，並且中斷向量已在 ROM 中。設定中斷的步驟僅需如下：

1. 若需要時)設定優先權群組。
2. 設定中斷的優先權。
3. 致能中斷。

在軟體需能於多種硬體設備上執行的情形下，則可能需要決定：

◆ 設計中支援的中斷數量

◆ 優先權等級暫存器裡位元的數量

Cortex-M3 有一個中斷控制型態暫存器，可以得知支援的中斷輸入的數量(以 32 個為一階，參考表 8-7)。另外，你可對中斷組態暫存器(例如 SETEN 或優先權暫存器)作讀取/寫入測試，以偵測外部中斷的確切數量。

表 8-7 中斷控制型態暫存器(0xE000E004)

位元	名稱	類型	重置值	描述
4:0	INTLINESNUM	R	-	中斷輸入數量每一階為 32 0 = 1 到 32 1 = 33 到 64 ...

欲決定中斷優先權等級暫存器所建置的位元數目，你可將 0xFF 寫入優先權等級暫存器其中一個，再把它讀回以得知設定的位元數。其最小數目為 3，於此情形下，你應該得到的讀回值為 0xE0。

## 軟體中斷

軟體中斷可能以多於一種的方法產生。第一個方法是利用 SETPEND 暫存器，第二個則是利用軟體觸發中斷暫存器(Software Trigger Interrupt Register, STIR)，參見表 8-8 所列。

表 8-8 軟體觸發中斷暫存器(0xE000EF00)

位元	名稱	類型	重置值	描述
8:0	INTID	W	-	寫入中斷號碼以設定中斷的等待中位元；例如，寫入 0 以將外部中斷#0 置於等待

系統例外(NMI、錯誤、PendSV 等等)不能以此暫存器來作等待，在預設下，用戶程式不能寫入 NVIC；然而，如果用戶程式需要寫進此暫存器，NVIC 組態控制暫存器(0xE000ED14)的 bit 1 (USERSETMPEND)能被設定，以允許用戶存取 NVIC 的 STIR。

# SYSTICK 計時器

SYSTICK 計時器整合在 NVIC 並可用以產生 SYSTICK 例外(例外型態#15)。在很多作業系統裡, 藉著硬體計時器產生中斷, 故作業系統能執行工作管理。例如, 欲允許多個工作在不同時段執行, 且確保沒有任何一個工作鎖住整個系統。為此目的, 計時器需要能夠產生中斷, 且儘可能地保護以免用戶工作影響它, 故用戶應用程式不會改變計時器的行為。

Cortex-M3 處理器包含一個簡單的計時器。因為所有 Cortex-M3 晶片有相同的計時器, 在 Cortex-M3 產品之間移植軟體極為便利。此計時器是 24-bits 倒數計時器, 可使用內部時脈(FCLK, 在 Cortex-M3 處理器上自行動作的時脈信號)或外部時脈(Cortex-M3 處理器上的 STCLK 信號)。然而, STCLK 的來源為晶片設計師所決定, 故產品之間的時脈頻率可能不同, 當選擇時脈來源時, 你應該小心地檢查晶片的資料表。

SYSTICK 計時器可用來產生中斷, 它有指定的例外型態與例外向量。它使得移植作業系統與軟體變得簡易, 因為在不同的 Cortex-M3 產品之間, 其程序皆相同。

SYSTICK 計時器被 4 個暫存器控制, 如表 8-9-表 8-12 顯示。

表 8-9 SYSTICK 控制與狀態暫存器(0xE000E010)

位元	名稱	類型	重置值	描述
16	COUNTFLAG	R	0	若計時器自從上次此暫存器被讀取之後計數至 0 則讀值為 1; 在讀取或清除現行計數器之值時, 將自動清除為 0
2	CLKSOURCE	R/W	0	0 = 外部參考時脈 (STCLK) 1 = 使用核心時脈
1	TICKINT	R/W	0	當 SYSTICK 計時器計數至 0 1 = 啟動 SYSTICK 中斷的產生 0 = 並不產生中斷
0	ENABLE	R/W	0	啟動 SYSTICK 計時器

表 8-10 SYSTICK 重載值暫存器(0xE000E014)

位元	名稱	類型	重置值	描述
23:0	RELOAD	R/W	0	計時器計數至 0 時重新載入數值

表 8-11 SYSTICK 現行值暫存器(0xE000E018)

位元	名稱	類型	重置值	描述
23:0	CURRENT	R/W	0	讀取以回傳計時器現行值。 寫入以清除計數器為 0。 清除現有值亦清除了 SYSTICK 控制與狀態暫存器裡的 COUNTFLAG。

表 8-12 SYSTICK 校正值暫存器(0xE000E01C)

位元	名稱	類型	重置值	描述
31	NOREF	R	-	1 = 無外部參考時脈 (無 STCLK 可用) 0 = 有外部參考時脈可用
30	SKEW	R	-	1 = 校正值並非正好為 10 ms 0 = 校正值為精確的
23:0	TENMS	R/W	0	10 ms 的校正值；晶片設計師需要經由 Cortex-M3 輸入信號提供此值。若此值讀出為 0，則表示無校正值可用

校正值(Calibration Value)暫存器提供一個解決的方法, 使得應用程式執行在各種 Cortex-M3 產品上時, 能產生相同的 SYSTICK 中斷區間。欲使用它, 僅需把 TENMS 的值寫入重載值(reload value)暫存器。這樣得到大約 10 毫秒的中斷區間。欲使用其它中斷計時區間, 軟體程式將需要藉著校正值來計算一個新的適合值。然而, TENMS 欄位可能並非所有 Cortex-M3 皆有提供(到 Cortex-M3 的校正輸入信號可能被固定為低), 故在使用此特性前, 需要檢查一下你的製造廠商的資料表。

除了作為作業系統的系統滴答計時器外, SYSTICK 計時器可用於多種用途：警告計時器、定時測量等等。注意, 當處理器於除錯期間暫停時, SYSTICK 計時器會停止計數。

## 中斷行為

本章內容包括：

- |           |            |
|-----------|------------|
| ✓ 中斷/例外序列 | ✓ 最後到達例外   |
| ✓ 例外出口    | ✓ 再論例外返回值  |
| ✓ 巢狀中斷    | ✓ 中斷延遲     |
| ✓ 未尾連鎖中斷  | ✓ 與中斷相關的錯誤 |

### 中斷/例外序列

當例外產生，會發生下列事情：

- ◆ 堆入堆疊(8 個暫存器的內容被推進堆疊)
- ◆ 向量擷取
- ◆ 更新堆疊指標、連結暫存器、與程式計數器

### 堆疊

當例外產生，暫存器 PC、PSR、R0-R3、R12、LR 等會被 push 進堆疊。如果執行中的程式使用 PSP，則會用到程序堆疊；如果執行中的程式使用 MSP，則會用到主要堆疊。接下來，在處理程式期間通常使用主要堆疊，故所有的巢狀中斷將使用主要堆疊。

MEMO.

# Jason 嘴書—EETOP 世界唯一貼

堆疊裡的次序如圖 9-1 所示(假設 SP 值在例外出現前為 N)。因為 AHB 介面的管線性質，位址與資料位移了一個管線狀態。

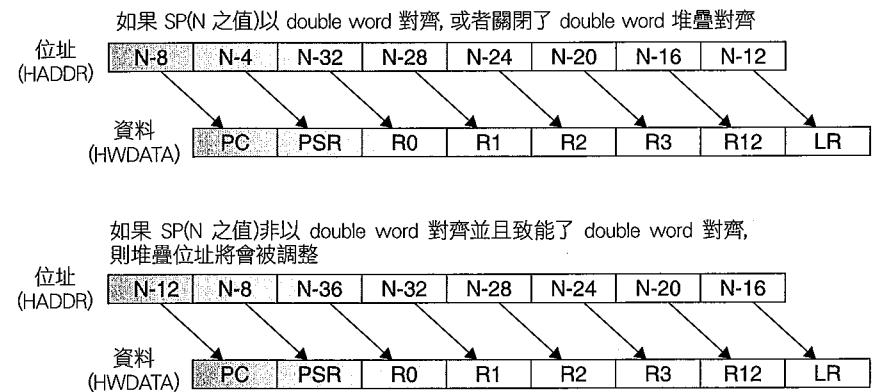


圖 9-1 堆疊的順序

被推進堆疊的 8 個 words 資料之區塊通常被稱作例外的堆疊框。在 Cortex-M3 revision 2 之前, 堆疊框預設為可以由任意 word 的位址開始。而在 Cortex-M3 revision 2 裡, 堆疊框預設以 double word 的位址做對齊。此堆疊框的對齊對 AAPCS (Procedure Call Standard for the ARM Architecture)而言是必須的。此特性在 Cortex-M3 revision 1 裡為可用的但在預設下並沒有被啟能。欲於 revision 1 中使用則需以軟體去設定 NVIC 裡組態控制暫存器的 STKALIGN 位元。在必要時此特性可藉由清除 STKALIGN 位元以在 Cortex-M3 revision 裡將它除能。關於此暫存器更多的細節可以在第 12 章裡找到(以 Double-Word 做堆疊對齊)。

先將 PC 與 PSR 之值置於堆疊, 故指令擷取(此需修改 PC 之值)與 IPSR 的更新可以較早開始。在堆入堆疊之後, SP 將被更新為 N-32 (如果 SP 以 double word 對齊, 或者 double word 對齊特性被關閉), 或者為 N-36 (如果 SP 非以 double word 對齊並且開啟了 double word 對齊之特性)。

表 9-1a 當 SP 以 double word 位址對齊或者當關掉了 double word 堆疊對齊的特性時, 堆入堆疊之後的堆疊記憶體內容與堆疊次序

位址	資料	推進的次序
舊的 SP (N) ->	(先前被推進的資料)	-
(N - 4)	PSR (bit 9 等於 0)	2
(N - 8)	PC	1
(N - 12)	LR	8
(N - 16)	R12	7
(N - 20)	R3	6
(N - 24)	R2	5
(N - 28)	R1	4
新的 SP (N - 32)	R0	3

表 9-1b 當 SP 非以 double word 位址對齊並且開啟了 double word 堆疊對齊的特性時, 堆入堆疊之後的堆疊記憶體內容與堆疊次序

位址	資料	推進的次序
舊的 SP (N) ->	(先前被推進的資料)	-
(N - 4)	並無使用	-
(N - 8)	PSR (bit 9 等於 1)	2
(N - 12)	PC	1
(N - 16)	LR	8
(N - 20)	R12	7
(N - 24)	R3	6
(N - 28)	R2	5
(N - 32)	R1	4
新的 SP (N - 36)	R0	3

R0-R3、R12、LR、PC、PSR 等被堆入的原因, 在於根據 C 標準, 這些是被呼叫程式保留的暫存器(C/C++ standard Procedure Call Standard for the ARM Architecture, AAPCS, Ref5)。這樣的安排允許中斷處理程式成為正常的 C 函數, 因為會被例外處理程式改變的暫存器已被堆疊保留。

通用暫存器(R0-R3 與 R12)置於堆疊框架的底部, 所以可容易地藉著 SP 相關的定址法存取。因此, 利用被堆疊進的暫存器, 就可以方便地傳參數給軟體中斷。

## 向量擷取

當資料匯流排忙碌地堆入暫存器時，指令匯流排執行另一個中斷序列的重要工作：它從向量表擷取例外向量(例外處理程式的起始位址)。因為堆疊動作與向量擷取在分開的匯流排介面執行，所以可同時進行。

## 暫存器更新

在堆疊動作與向量擷取完成後，例外向量會開始執行。進入例外處理程式後，一些暫存器會被更新：

- ◆ **SP**: 在堆疊過程中，堆疊指標(MSP 或 PSP)會被更新到新的位置。在中斷服務程式執行期間，如果存取了堆疊，則 MSP 會被使用。
- ◆ **PSR**: IPSR(PSR 的最低部分)將被更新為新的例外編號。
- ◆ **PC**: 當向量擷取完成時，PC 將更改至向量處理程式，並且開始從例外向量擷取指令。
- ◆ **LR**: LR 將更新至稱作 EXC\_RETURN<sup>1</sup> 的一個特殊值。此特殊值驅動中斷返回運算。LR 裡面最後 4 個位元有特殊意義，將於本章稍後論及。

一些其它的 NVIC 暫存器也將被更新。例如，例外的等待狀態將被清除，而例外的活動位元將被設定。

## 例外出口

在例外處理程式結尾，例外出口(在某些處理器裡叫做中斷返回)需要恢復系統狀態，以便被中斷的程式能再正常執行。觸發中斷返回序列有三種方式，且皆會使用到處理程式開端 LR 裡的一個特殊值(參見表 9-2)。

表 9-2 可用來觸發例外返回的指令

返回指令	描述
BX<reg>	如果 EXC_RETURN 值仍然存於 LR，我們可以使用 BX LR 指令去執行中斷返回。
POP {PC}, 或 POP {.., PC}	常常 LR 的值在進入例外處理程式之後會被推入堆疊。我們可以使用 POP 指令，單獨的 POP 或多重的 POPs，以將 EXC_RETURN 值放到程式計數器。此會使得處理器執行中斷返回。
LDR, 或 LDM	有可能藉由 LDR 指令，以 PC 為目的暫存器，而產生中斷返回

某些微處理器結構使用特殊指令以作中斷返回(例如 8051 中的 reti)。在 Cortex-M3 裡，使用了正常的返回指令，故整個中斷處理程式可以 C 副程式的方式來寫。

當執行中斷返回指令時，會進行下列程序：

1. **去堆疊**: 被推進堆疊的暫存器會被復原。POP 動作的次序會與推進堆疊的動作一致。堆疊指標也會被更改回來。
2. **NVIC 暫存器更新**: 例外的活動位元將被清除。對外部中斷來說，如果中斷輸入仍被設定，則等待位元將再一次被設定，使得再次進入中斷處理程式。

## 巢狀中斷

巢狀中斷的支援是內建於 Cortex-M3 處理器核心與 NVIC 裡，並不需要使用組譯器包裝程式(wrapper code)以效能巢狀中斷，事實上，除了對每一個中斷來源設定適當的優先權等級外，你不需要做其它事情。首先，Cortex-M3 裡的 NVIC 把解碼後的優先權做排序，所以，當處理器正處理一個例外時，其它具相同或更低優先權的例外會被阻擋。接著，自動的硬體堆疊與去堆疊，允許在不損失暫存器中資料下，執行巢狀中斷處理程式。

然而，需要留意一件事情：如果允許了許多巢狀中斷，則需確定主要堆疊裡的空間足夠。因為每一個例外等級將使用 8 words 的堆疊空間，並且例外處理程式也可能需要額外的堆疊空間，故使用到的堆疊記憶體可能多於預料。

Cortex-M3 並不允許重入(reentrant)例外。因為對每個例外指定了一個優先權等級，而且，在例外處理期間，具相同或更低優先權的例外會被阻擋，故在處理程式終了前，不能執行與其相同的例外。據此，SVC 指令不可用於 SVC 處理程式內，否則會造成一個錯誤例外。

<sup>1</sup> EXC\_RETURN 其 bit[31:4]之值皆為 1(意即 0xFFFFFFFF)；其最低 4 位元定義了回傳的資訊。本章後面就 EXC\_RETURN 值做了更多的討論。

## 末尾連鎖中斷

Cortex-M3 使用許多方法以改善中斷延遲，我們將看的第一個方法為末尾連鎖。

當例外發生，但處理器正處理其它具相同或更高優先權的例外，發生的例外將會進入等待狀態。當處理器完成執行現行例外處理程式後，接著就可以處理被置於等待的中斷。不過並不是從堆疊回復暫存器(去堆疊)並再次將之推進堆疊(進堆疊)，反之處理器會跳過去堆疊與進堆疊的步驟並且以最快的可能進入被置於等待例外的例外處理程式。如此，兩個例外處理程式的時間間隔將大為縮短。

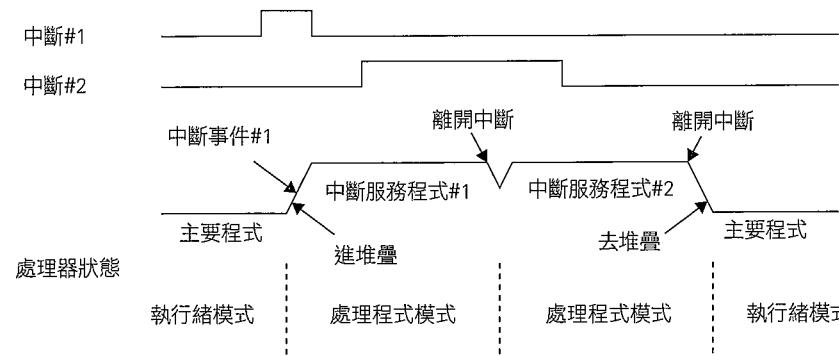


圖 9-2 例外的末尾連鎖

## 最後到達例外

另一個增強中斷表現的方法為最後到達例外處理。當一個例外產生並且處理器開始堆疊程序，如果具有更高強佔優先權的新例外於此延遲期間到達，此最後到達的例外將被最先處理。

例如，如果例外#1(優先權較低)比例外#2(優先權較高)提前幾個週期出現，處理器的行為將如圖 9-3 所顯示，其中處理程式#2 在堆疊完成後即被執行。

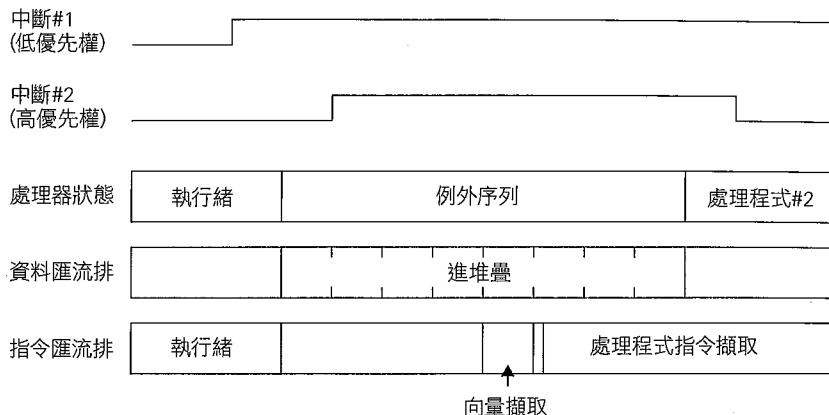


圖 9-3 最後到達例外的行為

## 再論例外返回值

當進入例外處理程式時，LR 更新為一個被稱做 EXC\_RETURN 的特殊值，其最高 28 位元皆被設定為 1。此值於例外處理程式終了時載入 PC，將造成處理器執行一個例外返回序列。

可以用以產生例外返回的指令如下：

◊ / POP/LDM      ◊ / LDR (以 PC 為目的的)      ◊ / BX(對任何暫存器)

EXC\_RETURN 的 bit[31:4]皆設定為 1，且 bit[3:0]提供了例外返回運算所需的資訊(參見表 9-3)。當進入例外處理程式時，LR 的值會被自動更新，所以不必手動地產生這些值。

表 9-3 EXC\_RETURN 值裡位元欄位的描述

位元	31:4	3	2	1	0
描述	0xFFFFFFFF	返回模式(執行緒/ 處理程式)	返回堆疊	保留的；需為 0	程序狀態 (Thumb/ARM)

Bit 0 顯示在例外返回後要使用的程序狀態。因為 Cortex-M3 僅支援 Thumb 狀態，bit 0 應當為 1。

表 9-4 顯示了(對 Cortex-M3 來講)合法的值。

表 9-4 在 Cortex-M3 上, 允許的 EXC\_RETURN 值

值	情況
0xFFFFFFFF1	返回至處理程式模式
0xFFFFFFFF9	返回至執行緒模式並於返回時使用主要堆疊
0xFFFFFFF9	返回至執行緒模式並於返回時使用程序堆疊

如圖 9-4 所顯示, 若執行緒正使用 MSP(main stack, 主要堆疊), 當進入例外時, LR 的值會被設定為 0xFFFFFFFF9；當進入巢狀例外時, LR 的值會被設定為 0xFFFFFFFF1。如圖 9-5 所顯示, 若執行緒正使用 PSP(process stack, 程序堆疊), 當進入第一個例外時, LR 的值會被設定為 0xFFFFFFF9；當進入巢狀例外時, LR 的值會被設定為 0xFFFFFFF1。

因為 EXC\_RETURN 數字格式的緣故, 你不能讓中斷返回到記憶體範圍 0xFFFFFFFF0-0xFFFFFFFFF 內的位址。然而, 既然此位址位於不可執行的區域, 所以這並不會成為問題。

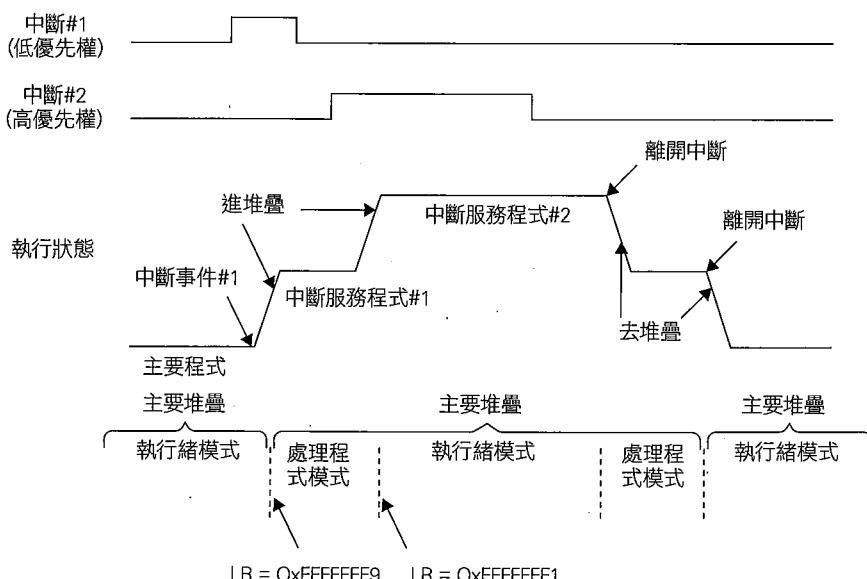


圖 9-4 在例外時設定 LR 為 EXC\_RETURN(當執行緒模式用的是主要堆疊時)

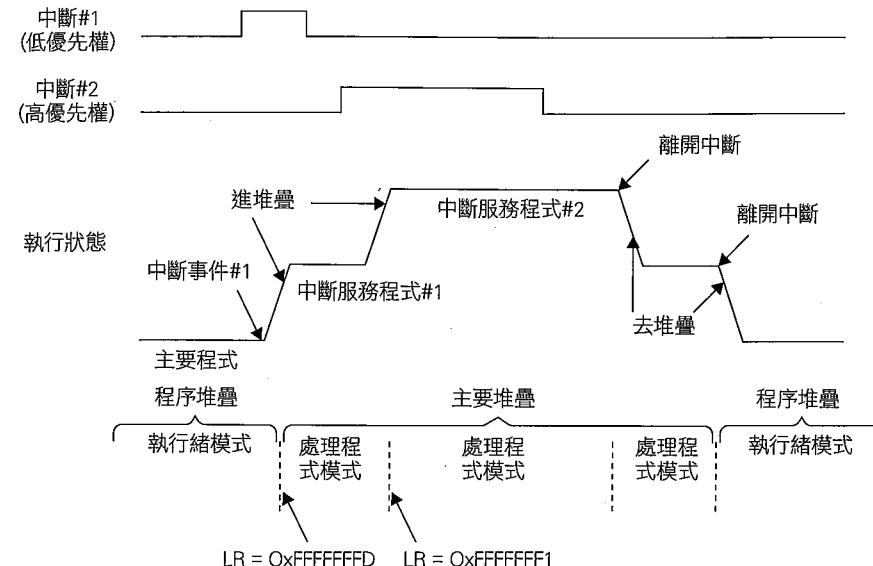


圖 9-5 在例外時設定 LR 為 EXC\_RETURN(當執行緒模式用的是程序堆疊時)

## 中斷延遲

中斷延遲意指從中斷要求至中斷處理程式開始的延遲時間。在 Cortex\_M3 裡, 如果記憶體系統為 0 延遲, 並且假設匯流排系統設計允許同時進行向量擷取與堆疊, 則中斷延遲最低為 12 個週期, 其中包括了把暫存器的值推進堆疊、向量擷取、擷取中斷處理程式的指令等。然而, 延遲時間隨記憶體存取等待狀態與一些其他的因素而定。

對末尾連鎖中斷而言, 因為不必執行堆疊運算, 故從一個例外處理程式, 轉換到另一個例外處理程式, 造成的延遲最低可為六個週期。

當程式執行多週期指令(例如除法指令), 指令可能被丟棄並於中斷處理程式完成後重新執行, 這種做法也可應用於載入雙字組(load double, LDRD)與儲存雙字組(store double, STRD)指令。

為減低例外延遲, Cortex-M3 處理器允許於多重載入與儲存指令(LDM/STM)間的例外。如果正執行 LDM/STM 指令, 會先完成進行中的記憶體存取, 並把下一個暫存器號碼保留在被堆入堆疊的 xPSR (ICI 位元)裡；且在例外處理程式完成後, 由傳輸停

止點再繼續做多重載入/儲存。有一個例外的情形：如果被中斷的多重載入/儲存指令為 IF-THEN(IT) 指令區塊的一部分，載入/儲存指令將被取消，並於中斷完成後重新執行。這是因為 ICI 位元與 IT 執行狀態位元分享了 EPSR 裡相同的空間的緣故。

此外，如果於匯流排介面有未完成的傳輸(例如被緩衝的寫入)，處理器會等待傳輸完成，以確保匯流排錯誤處理程式能強佔正確的程序。

當然，如果處理器正執行其他相同或更高優先權的例外處理程式，或者若中斷請求為中斷遮罩暫存器所遮罩，則中斷可能會被阻擋。在這些情形下，中斷會處於等待，並且在移開阻擋前不會被處理。

## 與中斷相關的錯誤

例外處理會造成各種錯誤，現在讓我們來看一下：

### 進堆疊

如果匯流排錯誤於進堆疊時產生，則進堆疊序列會被中止，而匯流排錯誤例外將被觸發或處於等待。如果匯流排錯誤被除能，將執行硬錯誤處理程式。否則，如果匯流排錯誤處理程式比原先例外的優先權高，匯流排錯誤處理程式將被執行；如果不是的話，它將處於等待直到原先例外完成。這個情節被稱作進堆疊錯誤，並藉著匯流排錯誤狀態暫存器(0xE000ED29)裡的 STKERR(bit 4) 顯示。

如果進堆疊錯誤是因為違反 MPU 所造成，則記憶體管理錯誤處理程式會被執行，並且會設定記憶體管理錯誤狀態暫存器(0xE000ED28)裡的 MSTKERR (bit 4)，以指出問題所在。如果記憶體管理錯誤被除能，則會執行硬錯誤處理程式。

### 去堆疊

如果匯流排錯誤於去堆疊(中斷返回)時產生，則去堆疊序列會被中止，而匯流排錯誤例外將被觸發或處於等待。如果匯流排錯誤被除能，將執行硬錯誤處理程式。否則，如果匯流排錯誤處理程式優先權比進行工作的現行優先權更高(核心可能於巢狀中斷情形下，正執行其他例外)，匯流排錯誤處理程式將被執行。這個情節被稱作去堆疊錯誤，並藉著匯流排錯誤狀態暫存器(0xE000ED29)裡的 UNSTKERR(bit 3) 顯示。

同樣地，如果去堆疊錯誤是因為違反 MPU 所造成，則記憶體管理錯誤處理程式會被執行，並設定記憶體管理錯誤狀態暫存器(0xE000ED28)裡的 MUSTKERR (bit 3)，以指出顯示問題所在。如果記憶體管理錯誤被除能，則會執行硬錯誤處理程式。

### 向量擷取

如果於向量擷取期間產生了匯流排錯誤或記憶體管理錯誤，硬錯誤處理程式將被執行。此以硬錯誤狀態暫存器(0xE000ED2C)裡的 VECTTBL(bit 1)來顯示。

### 不合法返回

如果 EXC\_RETURN 數值不合法，或者與處理器的狀態不合(例如使用 0xFFFFFFF1 以返回到執行緒模式)，將觸發用法錯誤。如果用法錯誤處理程式未被啟能，會執行硬錯誤處理程式以取代之。視錯誤的實際造成原因而定，以設定用法錯誤狀態暫存器(0xE000ED2A)裡的 INVPC 位元(bit 2)或 INVSTATE(bit 1)。

## 10

Chapter

# Cortex-M3 程式設計

本章內容包括：

- |                 |                   |
|-----------------|-------------------|
| ✓ 綜觀            | ✓ 使用資料記憶體         |
| ✓ 組合語言與 C 之間的介面 | ✓ 使用獨占存取作號誌       |
| ✓ 典型開發流程        | ✓ 使用 bit-band 作號誌 |
| ✓ 第一步           | ✓ 使用位元欄位取出與表格跳躍   |
| ✓ 產生輸出          |                   |

## 綜觀

Cortex-3 可藉由組合語言或 C 來撰寫程式，雖然可能也存在其它語言的編譯器，但是大多數的人會在他們的專案裡，使用組合語言、C、或兩者之組合。因為有關進行程式設計中間的許多資訊會隨你使用的工具鏈(tool chain)與矽晶片而定，所以本書並不會專注在編譯程式的細節上、或是如何下載程式到你的電路板上。第 19 章與第 20 章提及了一些這方面的資訊。

## 使用組合語言

對小的專案來說，可能會以組合語言開發整個應用程式。使用組譯器，你可能可以達成所希望的最佳化，但卻會增加開發時間，也比較容易出錯。此外，在組譯器裡處理複雜的資料結構與函數庫管理也可能極其困難。然而，即使專案採用 C 語言，在許多情形下，部分程式還是會以組合語言來實現：

MEMO.

- ◆ 不能於 C 裡實現的功能，例如特殊暫存器存取與獨佔存取
- ◆ 時間是關鍵點的副程式
- ◆ 記憶體需求非常吃緊，故部分程式以組語寫作來得到最小的記憶體大小

## 使用 C

與組合語言比較，C 的優點為可移植性與易於實作複雜的運算。因為 C 為通用的電腦語言，所以不會特別指定如何對處理器初始化。在這個領域，工具鏈可能會有不同的對策。閱讀範例程式是開始學習的最佳方式。對 ARM C 編譯器產品(例如 RealView Development Suite, RVDS 或者 KEIL RealView 微控制器開發套件)的使用者來說，安裝時就已經包含了一些 Cortex-M3 程式範例。對 GNU 工具鏈使用者來說，本書第十九章提供了基於 CodeSourcery GNU ARM 工具鏈的簡單的 C 範例。

使用 C 語言經常可以加速應用程式開發，但是在非常多的情形下，低階的系統控制仍會需要組譯器程式。大部分 ARM C 編譯器允許你去含括組譯器程式，稱作行內組譯器(inline assembler)，這樣的程式對許多專案通常是必須的。

在 ARM 編譯器裡，你可於 C 程式內加入組譯器程式。傳統上，這會使用行內組譯器，但是 RealView C Compiler 的行內組譯器並不支援 Thumb-2 指令。從 RealView C Compiler version 3.0 開始，包括了稱作嵌入式組譯器的新特性，支援了 Thumb-2 指令。例如，你可以利用如下方式將組合語言功能插入你的 C 程式裡：

```
_asm void SetFaultMask (unsigned int new_value)
{
    // 此處為組合語言程式
    MSR FAULTMASK, new_value // 把新的值寫入 FAULTMASK
    BX LR // 返回呼叫程式
}
```

RealView C 編譯器裡嵌入式組譯器的詳細描述可於 RVCT 3.0 Compiler and Library Guide (Ref 6)找到。

就 Cortex-M3 而言，嵌入式組譯器對下述工作有用，例如存取特殊暫存器(MRS 及 MSR 指令；譬如設定堆疊記憶體)，或當需要使用無法以 C 產生的指令時(例如，休眠[WFI 及 WFE]、獨佔存取、記憶體障礙運算等)。

於先前的 ARM 處理器裡，因為擁有 Thumb 狀態與 ARM 狀態，不同狀態的程式需個別編譯。於 Cortex-M3 裡則無此需求，因為所有程式皆為 Thumb 狀態，故專案檔案管理大為容易。

當你於 C 中開發應用程式，建議你採用 double word 堆疊對齊功能(以 NVIC 組態控制暫存器裡的 STKALIGN 來組態)。這可於啟動程式設定。例如：

```
#define NVIC_CCR ((volatile unsigned long *) (0xE00ED14))
*NVIC_CCR = *NVIC_CCR | 0x200; /* 設定 STKALIGN */
```

使用此特性可確保系統合於 ARM 結構的副程式呼叫標準(AAPCS)。此主題的其他資訊將於第十二章討論。

## 組合語言與 C 之間的介面

各種情形下，組合語言程式會與 C 程式互動。例如：

- ◆ 當在 C 程式碼中使用了嵌入式組合語言(或於 GNU 工具鏈裡的行內組譯器)
- ◆ 當 C 程式碼呼叫建置於分開檔案裡的組譯器的函數或副程式
- ◆ 當組合語言程式呼叫 C 函數或副程式

於上述情況下，了解參數與回傳結果，以及如何於呼叫程式與被呼叫函數之間傳遞，就顯得非常重要。這些互動的機制於 ARM Architecture Procedure Call Standard (AAPCS, Ref 5)之中陳述。

對簡單的例子，當呼叫程式需要傳參數到副程式或函數時，它將使用暫存器 R0 至 R3，其中 R0 為第一個參數，R1 為第二個，以此類推。相似地，R0 也於副程式結束時，用以回傳值。R0-R3 以及 R12 可被副程式或函數所改變，但 R4-R11 需要恢復至進入副程式之前的狀態，此經常藉由堆疊 PUSH 與堆疊 POP 來處理。

為了容易了解，本書的範例沒有嚴格地遵循 AAPCS 的做法：如果 C 函數被組合語言程式叫用，則暫存器 R0-R3 與 R12 可能會改變的影響需加以考慮，如果往後的階段需要這些暫存器的內容，則需把它們保留在堆疊，並於 C 函數完成後回復。因為這些範例程式大多僅呼叫影響少數暫存器、或會在結束時恢復暫存器內容的組合語言函數或副程式，故並不一定需要保留暫存器 R0-R3 與 R12。

## 典型開發流程

有各種軟體程式可用來開發 Cortex-M3 的應用程式。以這些工具描述的程式產生流程，有著相似的概念。就最基本的用途而言，你將需要組譯器、C 編譯器、連結程式、二進位檔案產生工具等。就 ARM 解決方案而言，RealView Development Suite (RVDS)或 RealView Compiler Tools (RVCT)提供了如圖 10-1 所示的檔案產生流程，其中散佈載入腳本(scatter-loading script)為選擇性的，但當記憶體映射變為複雜時，就經常會用到。

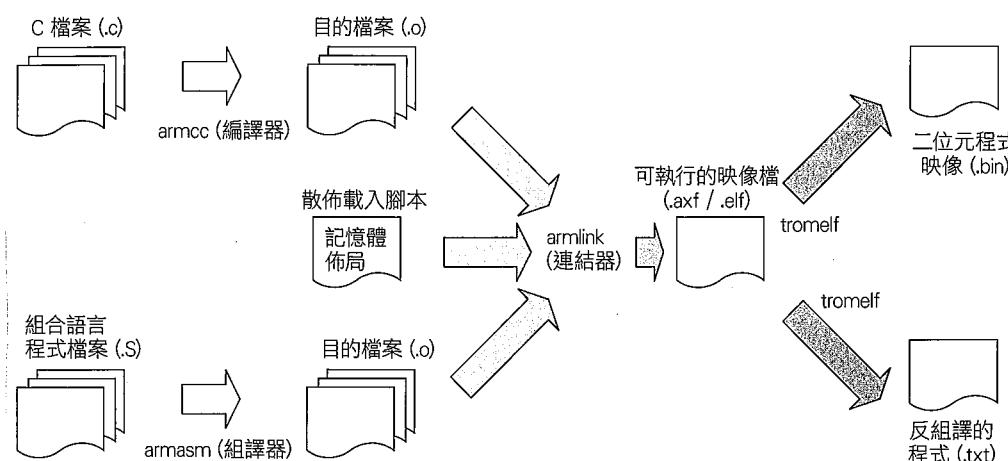


圖 10-1 使用 ARM 開發工具的範例流程

除了這些基本工具外，RVDS 也包括許多工具：例如整合開發環境(IDE)與除錯程式。細節部分請造訪 ARM 網站：[www.arm.com](http://www.arm.com)。

## 第一步

本章會檢視一些組合語言的範例。大部分的情形下，你將以 C 來撰寫程式，但藉由觀察一些組譯器範例，對如何使用 Cortex-M3 處理器，可得到更好的了解。此處的範例是根據 ARM 組譯器工具(armasm)；若是使用其他組譯器工具，則檔案格式以及指令語法可能需要修改。此外，一些開發工具事實上會為你產生啟動程式，所以你並不需要煩惱如何產生你的組合語言啟動程式。

第一個簡單程式可能如下：

```

STACK_TOP EQU 0x20002000 ; SP 起始值的常數
AREA |Header Code|, CODE
DCD STACK_TOP ; 堆疊頂端
DCD Start ; 重置向量
ENTRY
; 主程式開始
; 初始化暫存器
MOV r0, #10 ; 迴圈計數器起始值
MOV r1, #0 ; 結果起始值
; 計算 10+9+8+...+1

loop
ADD r1, r0 ; R1 = R1 + R0
SUBS r0, #1 ; 減 R0，更新旗標(後綴字"S")
BNE loop ; 如果上個指令計算結果不為 0
          ; 則跳躍至 loop
          ; 結果存放於 R1

deadloop
B deadloop ; 無窮迴圈
END
          ; 檔案結束
  
```

此簡單程式包括 SP 初始值、PC 初始值、設定暫存器，接著以迴圈做所需的計算。

假設你正使用 ARM 工具，此程式可用下面的指令組譯：

```
$> armasm --cpu cortex-m3 -o test1.o test1.s
```

-o 選項指定輸出檔案名，test1.o 為一個目的檔，我們接著需要使用連結程式以產生可執行的映像(ELF)，這可用下面的指令來做：

```
$> armlink --rw_base 0x20000000 --ro_base 0x0 --map -o test1.elf test1.o
```

此處, `--ro_base 0x0` 指定了唯讀區域(程式 ROM)於位址 0x0 開始; `--rw_base` 指定了讀取/寫入區域(資料記憶體)於位址 0x20000000 開始。(在這個 test1.s 例子, 我們並未定義任何 RAM 資料。) `--map` 選項產生映像映射, 此有助於了解編譯的映像之記憶體佈局。

最後, 我們需要產生二進位映像:

```
$> fromelf --bin --output test1.bin test1.elf
```

為了檢查映像確如我們所需要, 可以產生反組譯程式列表的檔案:

```
$> fromelf -c --output test1.list test1.elf
```

如果一切順利, 你可將你的 ELF 映像或二進位映像, 載入硬體或指令集模擬器加以測試。

## 產生輸出

如果可以讓微控制器與外在世界連繫, 會帶來更多樂趣。最簡單的做法是去開/關 LEDs。然而, 此做法相當有限, 因為它僅能代表非常有限的資訊。最常用的輸出方式之一是把文字訊息傳送到控制臺。在嵌入式產品開發裡, 此工作通常由連結至個人電腦的 JART 介面來處理。例如, 使用具有 Hyper-Terminal 程式的 Windows<sup>1</sup> 系統電腦做為控制台, 就能極其方便地產生輸出。

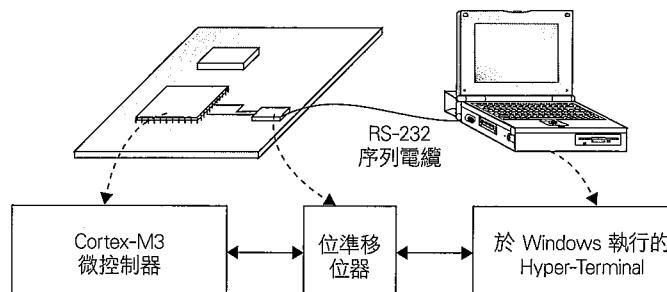


圖 10-2 用來輸出文字訊息的低成本測試環境

Windows 與 Hyper-Terminal 為微軟公司的商標。

Cortex-M3 處理器並沒有包含 UART 介面, 但大多數 Cortex-M3 微控制器伴隨著由晶片製造廠商提供的 UART。不同的元件具不同的 UART 的規格, 所以在本書中我們不會嘗試去討論這個主題。我們在下一個範例裡假設 UART 為可用的, 並且有以顯示傳輸緩衝器是否已經準備好送出新的資料的一個狀態旗標。因為 RS-232 的電位準位與微控制器 I/O 接腳不同, 故於連結中需要使用一個準位移位器。

UART 並不是輸出文字訊息的唯一解決之道, 在 Cortex-M3 處理器上亦建置了一些有助於輸出除錯訊息的功能:

- ◊ **Semihosting:** 隨著除錯程式與程式庫支援而定, semihosting(藉著除錯探查設備以輸出 printf 之訊息)可藉由 NVIC 裡的除錯暫存器來達成。(此主題的更多資訊於第十五章討論。)在這些情形下, 你可在你的 C 程式內使用 printf, 並且輸出將會被顯示於除錯程式軟體的控制台/標準輸出(STDOUT)上。
- ◊ **設備追蹤:** 如果 Cortex-M3 微控制器提供了追蹤埠, 且外部的追蹤埠分析儀(Trace Port Analyzer, TPA)為可用, 則可以使用設備追蹤模組(�器化 Trace Module, ITM)取代 UART 以輸出訊息。追蹤埠比 UART 速度快且能提供更多的資料頻道。
- ◊ **經由序列線檢視器作設備追蹤:** Cortex-M3 處理器(revision 1 以後)亦於追蹤埠介面單元(TPIU)提供了序列線檢視器(Serial Wire Viewer, SWV)運算模式作為替代, 此介面允許藉著低成本硬體來取代 TPA, 以捕捉 ITM 輸出。然而, SWV 模式提供的頻寬有限, 所以不適用於大量的資料。

## “Hello World”範例

在我們試著寫“Hello World”程式之前, 我們應當設法經由 UART 輸出一個字元。輸出字元的程式可以副程式來實現, 故可為其它的訊息輸出程式呼叫。如果輸出設備改變, 我們僅需要改變此副程式, 則所有的文字訊息可利用不同設備作輸出, 這樣的修改方式通常稱作重設目標(retargeting)。

輸出字元的簡單程式可能如下:

<code>UART0_BASE</code>	<code>EQU</code>	<code>0x4000C000</code>
<code>UART0_FLAG</code>	<code>EQU</code>	<code>UART0_BASE+0x018</code>
<code>UART0_DATA</code>	<code>EQU</code>	<code>UART0_BASE+0x000</code>

接下頁

```
Putc
    ; 經由 UART 輸出一個字元的副程式
    ; 輸入 R0 = 欲輸出的字元
    PUSH {R1, R2, LR}           ; 保留暫存器值
    LDR   R1, =UART0_FLAG
```

```
PutcWaitLoop
    LDR   R2, [R1]              ; 取得狀態旗標
    TST   R2, #0x20             ; 檢查傳輸緩衝器的滿旗標位元
    BNE   PutcWaitLoop          ; 若忙碌中則作迴圈
    LDR   R1, =UART0_DATA       ; 否則
    STRB R0, [R1]               ; 輸出資料至傳輸緩衝器
    POP   {R1, R2, PC}          ; 返回
```

此處的暫存器位址與位元的定義僅為舉例，你可能需要就你的設備更改其值。此外，某些 UART 在將字元輸出至傳輸緩衝器之前，可能需要作更複雜的狀態檢查程序。再者，需要以另一個副程式呼叫(下個例子裡的 Uart0Initialize)來初始化 UART，但這會隨 UART 規格而定，故不在此處討論。(第二十章討論在 Luminary Micro LM3S811 設備上，以 C 作 UART 初始化的範例。)

現在我們可以使用上述副程式來建立一些功能以顯示訊息：

```
Puts ; 輸出字串至 UART 的副程式
    ; 輸入 R0 = 字串的起始位址
    ; 字串需要以 null 作結尾
    PUSH {R0, R1, LR}           ; 保留暫存器值
    MOV   R1, R0                ; 因為會使用到 R0，故將字串的起始位址複製至 R1
    PutsLoop
        LDRB R0, [R1], #1         ; 作為 Putc 的輸入
        CBZ  R0, PutsLoopExit    ; 讀一個字元並遞增位址
        BL   Putc                ; 如果字元為 null，則結束
        B    PutsLoop             ; 輸出字元至 UART
        B    PutsLoop             ; 下一個字元
    PutsLoopExit
        POP  {R0, R1, PC}          ; 返回
```

利用上述副程式，我們已經準備好第一個"Hello world"程式：

```
STACK_TOP      EQU 0x20002000 ; 作為 SP 起始值的常數
UART0_BASE     EQU 0x4000C000
UART0_FLAG     EQU UART0_BASE+0x018
UART0_DATA     EQU UART0_BASE+0x000
AREA | Header Code |, CODE
DCD  STACK_TOP ; 堆疊指標初始值
```

```
DCD  Start ; 重置向量
ENTRY
Start
    MOV  r0, #0 ; 主程式開始
    MOV  r1, #0 ; 初始化暫存器
    MOV  r2, #0
    MOV  r3, #0
    MOV  r4, #0
    BL   Uart0Initialize ; 初始化 UART0
    LDR  r0, =HELLO_TXT ; 設定 R0 值為字串的起始位址
    BL   Puts

    B    deadend ; 無窮迴圈
; -----
;副程式
; -----
Puts
    ; 輸出字串至 UART 的副程式
    ; 輸入 R0 = 字串的起始位址
    ; 字串需要以 null 作結尾
    PUSH {R0, R1, LR}           ; 保留暫存器值
    MOV  R1, R0                ; 因為會使用到 R0，故將字串的起始位址複製至 R1
    PutsLoop
        LDRB R0, [R1], #1         ; 作為 Putc 的輸入
        CBZ  R0, PutsLoopExit    ; 讀一個字元並遞增位址
        BL   Putc                ; 如果字元為 null，則結束
        B    PutsLoop             ; 輸出字元至 UART
        B    PutsLoop             ; 下一個字元
    PutsLoopExit
        POP  {R0, R1, PC}          ; 返回
; -----
Putc
    ; 經由 UART 輸出一個字元的副程式
    ; 輸入 R0 = 欲輸出的字元
    PUSH {R1, R2, LR}           ; 保留暫存器值
    LDR  R1, =UART0_FLAG
    PutcWaitLoop
        LDR   R2, [R1]              ; 取得狀態旗標
        TST   R2, #0x20             ; 檢查傳輸緩衝器的滿旗標位元
        BNE   PutcWaitLoop          ; 若忙碌中則作迴圈
        LDR   R1, =UART0_DATA       ; 否則
        STRB R0, [R1]               ; 輸出資料至傳輸緩衝器
        POP   {R1, R2, PC}          ; 返回
; -----
Uart0Initialize
    ; 隨設備而定，故不於此列出
    BX   LR ; 返回
; -----
Hello_TXT
    DCB  "Hello world\n", 0 ; Null 結尾的 Hello world 字串
    END ; 檔案結束
```

唯一需要你增加到程式的是 Uart0Initialize 副程式的細節。

能夠輸出暫存器值的副程式亦相當有用。我們可基於之前完成的 Putc 與 Puts 副程式，以更輕易地撰寫上述副程式。下列第一個副程式用來顯示十六進位的值：

```

PutHex ; 以十六進位格式輸出暫存器值
; 輸入 R0 = 欲顯示的值
PUSH {R0-R3, LR}
MOV R3, R0      ; 保留暫存器值於 R3, 因為 R0 用來
; 傳輸入參數
MOV R0, # '0'   ; 以 0x 作顯示的開端
BL Putc
MOV R1, #8       ; 設定迴圈計數器
MOV R2, #28      ; 旋轉位移
PutHexLoop
ROR R3, R2      ; 資料值左旋轉 4 位元
; (即右旋轉 28 位元)
AND R0, R3, #0xF ; 取出最低 4 位元
CMP R0, #0xA     ; 作 ASCII 轉換
ITE GE
ADDGE R0, #55    ; 若大於 10，則轉換為 A-F
ADDLT R0, #48    ; 否則轉換為 0-9
BL Putc
SUBS R1, #1       ; 遲減迴圈計數器
BNE PutHexLoop  ; 若顯示了所有的 8 個字元，
; 則返回，否則繼續處理下面 4 個位元
POP {R0-R3, PC}

```

此副程式可用來輸出暫存器值；然而，有時候我們也會想要以十進位表示方式來輸出暫存器值。聽起來，這像是相當複雜的運算，但是在 Cortex-M3 裡相當容易進行，因為它擁有硬體乘法與除法指令。在此運算中另一個主要問題，在於我們是以相反的次序得到所欲輸出的字元，因而我們需要把欲輸出的結果先放於一個文字緩衝器裡，並等到整個文字已經準備好作顯示時，再藉著 Puts 功能將整個結果一起顯示。在下例，是以部分的堆疊記憶體當作文字緩衝器使用：

```

PutDec ; 以十進位顯示暫存器值的副程式
; 輸入 R0 = 欲顯示的數值
; 因為有 32 位元，故若以十進位格式表示加上 null 的終結
; 則最大所需位元為 11 個
PUSH {R0-R5, LR} ; 保留暫存器值

```

MOV R3, SP	; 將現在進行的堆疊指標複製至 R3
SUB SP, SP, #12	; 保留 12 bytes 以作文字緩衝器
MOV R1, #0	; Null 字元
STRB R1, [R3, #-1]!	; 把 null 字元置於文字緩衝器
;	的結尾，預先索引好
MOV R5, #10	; 設定除數之值
<b>PutDecLoop</b>	
UDIV R4, R0, R5	; R4 = R0 / 10
MUL R1, R4, R5	; R1 = R4 * 10
SUB R2, R0, R1	; R2 = R0 - (R4 * 10) = 餘數
ADD R2, #48	; 作 ASCII5 轉換(R2 代表字元介於 0-9)
STRB R2, [R3, #-1]	; 把 ASCII 字元置於文字緩衝器
;	，預先索引好
MOVS R0, R4	; 設定 R0 = 相除的結果
BNE PutDecLoop	; 若 R4 = 0 則設定 Z 旗標
MOV R0, R3	; 把 R0 置於文字緩衝器的起始位置
BL Puts	; 使用 Puts 來顯示結果
ADD SP, SP, #12	; 恢復堆疊位置
POP {R0-R5, PC}	; 返回

藉著 Cortex-M3 指令集的各種功能，將數值轉換為十進位格式作顯示的處理，可以上面非常短的副程式來實作。

## 使用資料記憶體

重新回到第一個範例：當執行連結階段時，我們指定了讀取/寫入的記憶體區域。如何把資料放在那裡？其方法是在你的組合語言檔案裡定義一個資料區。利用與一開始相同的例子，我們把資料存放於 0x2000000(SRAM 區域)的資料記憶體裡。資料區段的位置以執行連結程式的命令列選項來控制：

```

STACK_TOP EQU 0x20002000 ; SP 起始值的常數
AREA |Header Code|, CODE
DCD STACK_TOP ; SP 初始值
DCD Start ; 重置向量
ENTRY ; 主程式開始
Start ; 初始化暫存器
MOV r0, #10 ; 迴圈計數器起始值
MOV r1, #0 ; 結果的起始值
; 計算 10+9+8+...+1
loop

```

接下頁

```

ADD    r1, r0          ; R1 = R1 + R0
SUBS   r0, #1          ; 遲減 R0, 更新旗標(後綴字"S")
BNE    loop             ; 如果上個指令計算結果不為 0
                           ; 則跳躍至 loop

; 結果現在存放於 R1

deadloop
B      deadloop        ; 無窮迴圈
AREA  | Header Data |, DATA
ALIGN 4
MyData1 DCD 0
MyData2 DCD 0          ; 計算結果的目的地
END
                           ; 檔案結束

```

在連結的階段，連結程式會把 DATA 區放在讀取/寫入的記憶體區域，故於此情形下，上例 MyData1 的位址將會是 0x20000000。

## 使用獨占存取作號誌

獨占存取指令是用來作號誌運算，例如，用以確認資源僅為一個工作所用。舉例來說，假設記憶體裡的資料變數 DeviceALocked 可以用來顯示設備 A 是否被使用。如果一個工作欲使用設備 A，它需讀取變數 DeviceALocked 的值以檢查其狀態。若狀態為 0，則把 1 寫入 DeviceALocked 以鎖住設備；並在使用設備完成後，把 DeviceALocked 清為 0，讓其他工作可使用設備。

如果有兩個工作同時試著存取設備 A，會發生什麼事情？在此情形下，可能兩個工作會讀取變數 DeviceALocked，並且兩者的讀值皆為 0，接著他們兩個將會嘗試把 1 回寫至變數 DeviceALocked 以鎖住設備，故我們會有兩個工作皆認為他們對設備 A 具獨占存取。此刻獨占存取可派上用場。STREX 指令有一個回傳狀態，可用來顯示獨占儲存是否已經成功；如果有兩個工作同時試著鎖住一個設備，回傳值將為 1(獨占失敗)，讓工作可以得知它需要再次去嘗試鎖住的動作。

第 5 章提供一些使用獨占存取的背景知識，其討論所用的流程圖如圖 10-3 所示。

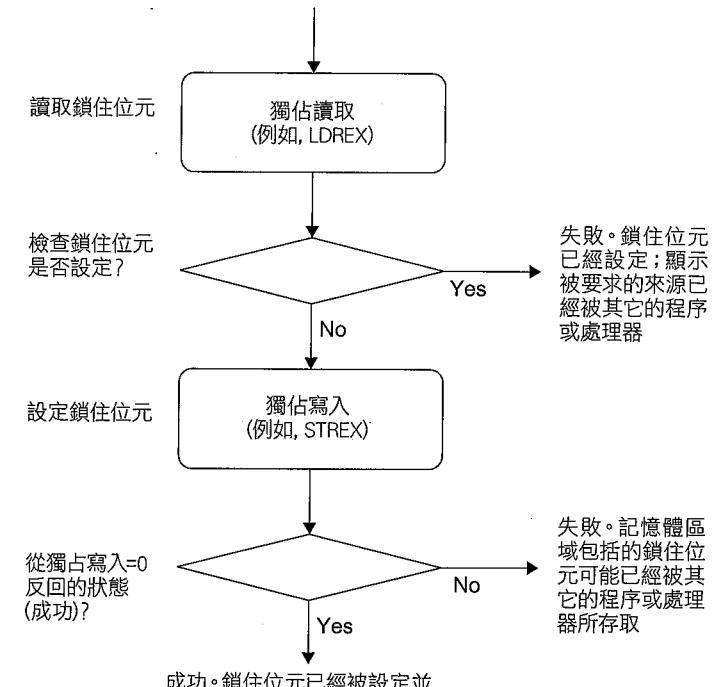


圖 10-3 使用獨占存取作號誌運算

其運算可以下列組合語言程式來執行。注意，如果獨占監視器回傳了失敗的狀態，將不會執行 STREX 的資料寫入運算，此可用來避免在獨占存取失敗時去設定鎖住位元：

```

LockDeviceA
; 嘗試鎖住設備 A 的簡單副程式
; 輸出 R0 : 0 = 成功, 1 = 失敗
; 如果成功了, 1 的值將被寫到變數 DeviceALocked
PUSH {R1, R2, LR}

TryToLockDeviceA
LDR R1, =DeviceALocked           ; 取得鎖住狀態
LDREX R2, [R1]                   ; 檢查它是否被鎖住
CMP R2, #0
BNE LockDeviceAFailed
DeviceAIsNotLocked
MOV R0, #1                         ; 嘗試把 1 寫入
                                    ; DeviceALocked
                                    ; 獨占寫入

```

接下頁

```

    CMP    R2, #0
    BNE    LockDeviceAFailed ; STREX 失敗
LockDeviceASucceed
    MOV    R0, #0           ; 回傳成功狀態
    POP    {R1, R2, PC}     ; 返回
LockDeviceAFailed
    MOV    R0, #1           ; 回傳失敗狀態
    POP    {R1, R2, PC}     ; 返回

```

如果此副程式的回傳狀態為 1(獨占失敗), 應用程式的工作應當等待片刻, 之後再重試。在單處理器系統, 獨占存取失敗通常的原因, 在於獨占載入與獨占儲存之間出現中斷。如果程式處於特權的模式, 欲避免此情形, 可藉著短暫地設定中斷遮罩暫存器(例如 PRIMASK), 以增加成功鎖住資源的機會。

在多處理器系統, 除了中斷之外, 如果其他處理器亦存取相同的記憶體區域, 也會造成獨占儲存失敗。為了偵測從不同的處理器的記憶體存取, 匯流排基礎結構需要獨占存取監視器硬體, 以偵測是否於兩個獨占存取之間, 有來自不同匯流排 master 對記憶體的存取。然而, 大多數低成本的 Cortex-M3 微控制器僅有單一處理器, 故並無需要此監視器硬體。

依此機制, 我們可以確定僅有一個工作可存取某些資源。如果應用程式經過一些時間仍無法得以鎖住資源, 它可能會因為逾時錯誤而需放棄。例如, 鎖住一資源的工作可能已經當掉, 但其鎖住狀態仍維持設定。在這些情形下, 作業系統需要檢查哪一個工作正在使用資源, 如果工作已經完成或終止但未清除鎖定, 作業系統可能需要去解除資源的鎖定。

如果處理器已使用 LDREX 開始了獨占存取, 然後發現不再需要獨占存取, 則可使用 CLREX 指令以清除獨占存取監視器裡的局部紀錄, 其語法如下：

CLREX.W

就 Cortex-M3 處理器而言, 所有獨占的記憶體傳輸動作需要循序地執行, 然而, 如果獨占存取的控制程式碼需要在其他的 ARM Cortex 處理器裡重複使用, 則可能需要在獨占傳輸之間插入資料記憶體障礙(Data Memory Barrier, DMB)指令, 以確保記憶體存取正確的次序。

## 使用 Bit-Band 作號誌

假使記憶體系統支援鎖定傳輸, 或者於記憶體匯流排裡僅存在一個匯流排 master, 則可能使用 bit-band 特性以執行號誌運算。藉由 bit band, 則可在 C 程式裡執行號誌, 但此運算有別於使用獨占存取。欲以 bit band 作資源的配置控制, 需使用 bit-band 記憶體區域的一個記憶體位置(例如一個 word 資料), 並且此變數的每一個位元顯示了該資源正為特定的工作所用。

因為 bit-band alias 寫入是鎖住的 READ-MODIFY-WRITE 傳輸(匯流排 master 在傳輸期間不能切換到另一個匯流排), 假設所有的工作僅能改變代表他們自己的鎖定位元, 即使有兩個工作同時嘗試寫入相同的記憶體位置, 其他工作的鎖定位元將不會遺失。有別於使用獨占存取, 一個資源可能會短暫地同時被兩個工作鎖住, 直到他們其一偵測到衝突並釋放鎖定。

使用 bit band 作為號誌的先決條件, 為系統裡的所有工作僅能藉由 bit-band alias 改變其被分配的鎖定位元。如果有任何一個工作藉由正常的寫入改變了鎖定變數, 號誌可能會失敗, 因為如果另一個工作恰好於寫入鎖定變數之前設定了鎖定位元, 先前其他工作設定的鎖定位元將會遺失。

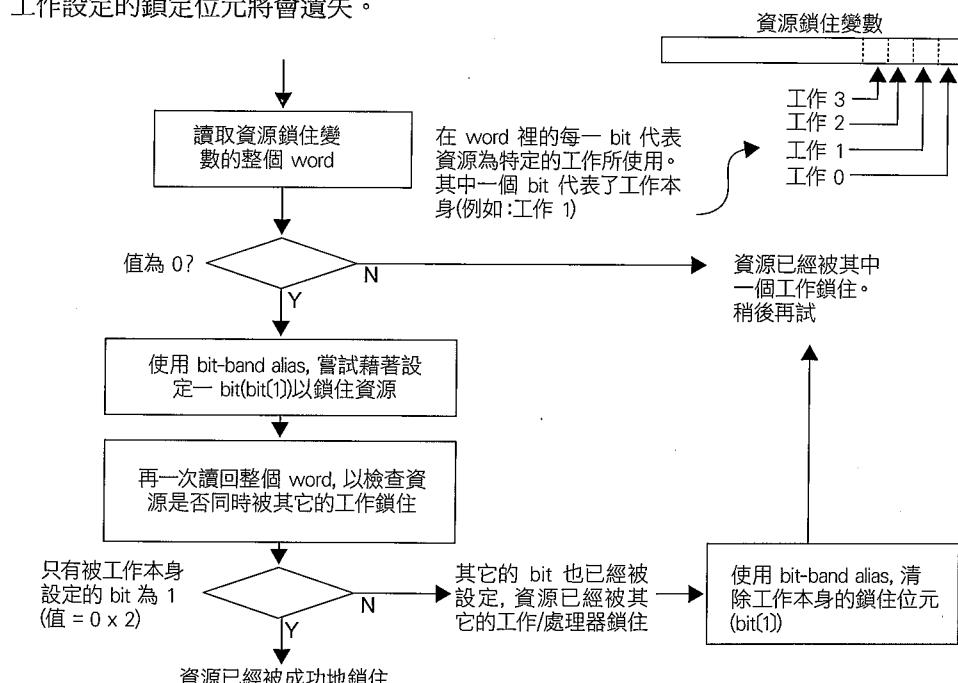


圖 10-4 使用 Bit-band 作為號誌控制

## 使用位元欄位取出與表格跳躍

在第四章中我們研究了無號數位元欄位取出(Unsigned Bit Field Extract, UBFX)與表格跳躍(Table Branch, TBB/TBH)，同時使用此兩指令可行成非常有威力的分支樹。此功能在資料通信應用中非常有用，資料通信應用裡資料序列因不同的標頭檔而有不同的意義。例如，假設下面根據輸入 A 的決策樹，並以組譯器來作編程(參見圖 10-5)：

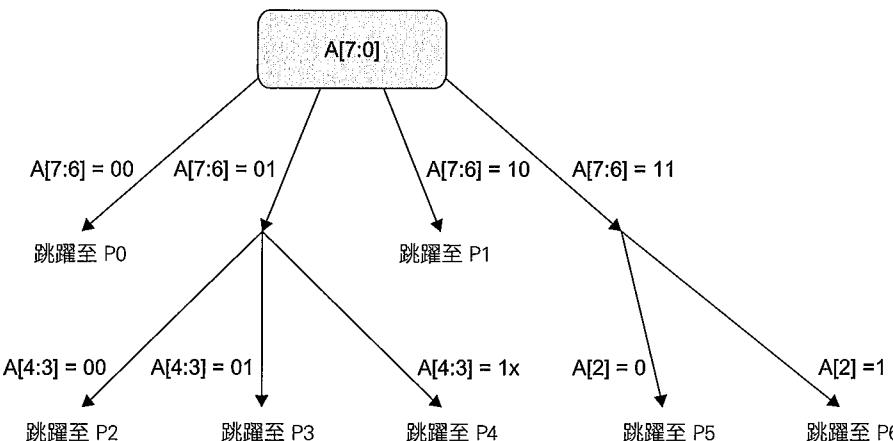


圖 10-5 位元欄位解碼器：使用位元欄位取出與表格跳躍指令的範例

```

DecodeA
LDR R0, =A           ; 由記憶體取得 A 的值
LDR R0, [R0]
UBFX R1, R0, #6, #2  ; 取出 bit[7:6] 值至 R1
TBB [PC, R1]

BrTable1
DCB ((P0 -BrTable1)/2) ; 若 A[7:6] = 00，則跳躍至 P0
DCB ((DecodeA1-BrTable1)/2) ; 若 A[7:6] = 01，則跳躍至 DecodeA1
DCB ((P1 -BrTable1)/2) ; 若 A[7:6] = 10，則跳躍至 P1
DCB ((DecodeA2-BrTable1)/2) ; 若 A[7:6] = 11，則跳躍至 DecodeA2

DecodeA1
UBFX R1, R0, #3, #2  ; 取出 bit[4:3] 值至 R1
TBB [PC, R1]

BrTable2
DCB ((P2 -BrTable2)/2) ; 若 A[4:3] = 00，則跳躍至 P2
DCB ((P3 -BrTable2)/2) ; 若 A[4:3] = 01，則跳躍至 P3
DCB ((P4 -BrTable2)/2) ; 若 A[4:3] = 10，則跳躍至 P4
  
```

DCB	((P4 -BrTable2)/2)	; 若 A[4:3] = 11，則跳躍至 P4
DecodeA2		
TST	R0, #4	; 僅測試一位元，故不必使用 UBX
BEQ	P5	
B	P6	
P0 ...		; 程序 0
P1 ...		; 程序 1
P2 ...		; 程序 2
P3 ...		; 程序 3
P4 ...		; 程序 4
P5 ...		; 程序 5
P6 ...		; 程序 6

此程式以簡短的組譯器程式序列完成決策樹，如果跳躍的目標較大，則需使用 TBH 指令來取代 TBB。

# 例外程式設計

- ✓ 中斷的使用
- ✓ 例外/中斷處理程式
- ✓ 軟體中斷
- ✓ 例外處理程式範例
- ✓ 使用 SVC
- ✓ SVC 範例：作為輸出函數
- ✓ 以 C 來使用 SVC

## 中斷的使用

幾乎所有的嵌入式應用程式都會使用到中斷。於 Cortex-M3 處理器裡，中斷控制器 NVIC 為你處理了一些工作，包括優先權檢查與暫存器的進堆疊/去堆疊。然而，當行使中斷時，需要準備好一些工作：

◆ 堆疊設定

◆ 中斷優先權設定

◆ 向量表設定

◆ 致能中斷

### 堆疊設定

若是開發簡單的應用程式，你可以整個程式都使用 MSP；這樣僅需保留足夠的記憶體並把 MSP 設定於堆疊頂端。當決定所需的堆疊大小時，除了需檢查軟體使用的堆疊層級外，你亦需要檢查可能出現的巢狀中斷的層數；對每一個巢狀中斷層級，你至少需要 8 words 的堆疊，而在中斷處理程式裡的處理過程裡也可能需要額外的堆疊空間。

MEMO.

因為 Cortex-M3 裡堆疊運算為全遞減的，故經常把堆疊的起始值放在靜態記憶體的底部，使得 SRAM 的剩餘空間不會成為分散的區塊。

若應用程式的用戶程式與核心程式使用不同的堆疊，則主要堆疊需要有足夠的空間，讓巢狀中斷處理程式與核心程式用到的堆疊記憶體來使用。程序堆疊應當有足夠的記憶體，以容納用戶應用程式加上一層的堆疊空間(8 words)；這是因為從用戶執行緒到第一層的中斷處理程式的堆疊使用了程序堆疊的緣故。

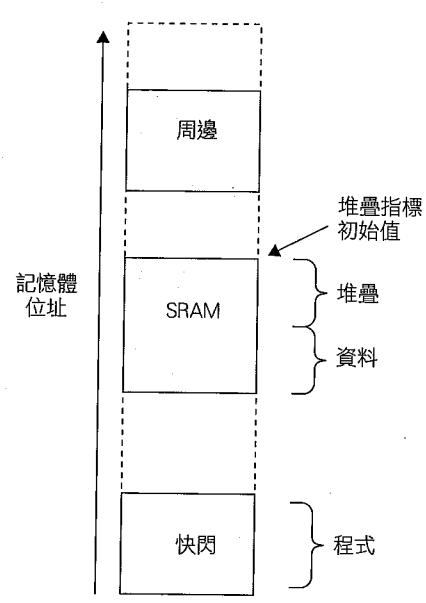


圖 11-1 一個簡單的記憶體使用例子

## 向量表設定

因為簡單的應用程式使用固定的中斷處理程式，所以向量表可於 ROM 裡面編程，在此情形下，並不需要於執行期間設定向量表。然而，在許多應用中，你需要因應不同的情形來更改中斷處理程式，然後再把向量表重新配置到一個可寫入的記憶體。

在重新配置向量表時，你可能需要複製現行的向量表內容到新的相量表位置。此包含除錯處理程式、NMI、系統呼叫等向量位址。否則，如果例外在向量表重新配置之後發生，處理器會擷取不到合法的向量位址。

在必要的向量表項目設定好且重新定位向量表之後，我們可加入新向量到向量表裡。例如：

```
; 根據例外型態以設定例外向量的副程式
; ( For IRQs 加 6: IRQ #0 = 例外型態 16 )
SetVector
    ; 輸入 R0 = 例外型態
    ; 輸入 R1 = 向量位址值
    PUSH {R2, LR}
    LDR R2, =0xE000ED08      ; 向量表位移暫存器
    LDR R2, [R2]
```

## Jason 嘴書—EETOP 世界唯一貼

```
STR R1, [R2, R0, LSL, #2] ; 把向量寫至
                            ; VectTblOffset + ExcpType * 4
    POP {R2, PC}           ; 返回
```

## 中斷優先權設定

根據預設，重置之後所有可程式優先權的例外皆位於優先權等級 0；而硬錯誤例外的優先權等級為-1，NMI 的優先權等級為-2。規劃優先權等級暫存器時，我們可以利用暫存器能以 byte 定址的事實，使得程式更易撰寫。例如：

```
; 設定 IRQ #4 的優先權為 0xC0
LDR R0, =0xE000E400        ; 外部中斷優先權暫存器起始位址
LDR R1, =0xC0               ; 優先權等級
STRB R1, [R0, #4]           ; 設定 IRQ #4 優先權 (Byte 寫入)
```

在 Cortex-M3 裡，中斷優先權組態暫存器的大小，由晶片製造廠商來設定；其最小為 3 個位元，最大為 8 個位元。你可藉著把 0xFF 寫入一個優先權組態暫存器，再把它讀回，以決定建置的優先權暫存器的大小。例如：

```
; 決定建置的優先權大小
LDR R0, =0xE000E400        ; 外部中斷#0 的優先權組態暫存器
LDR R1, =0xFF
STRB R1, [R0]
LDRB R1, [R0]
RBIT R2, R1
CLZ R1, R2
MOV R2, #8
SUB R2, R2, R1
MOV R1, #0x0
STRB R1, [R0]
; 寫入 0xFF (注意：byte 大小的寫入)
; 讀回(例如，3-bits 則為 0xE0)
; 把 R2 作位元顛倒
; (例如，3-bits 則為 0x07000000)
; 計數開始 0 的數目(例如，3-bits 則為 0x5)
; 算出建置的優先權大小
; (例如：3-bits 則為 8-5=3)
; 回復到重置的值 (0x0)
```

如果你的應用程式需為可移植的，最好只使用優先權等級 0x00、0x20、0x40、0x60、0x80、0xA0、0xC0、0xE0 等，因為所有的 Cortex-M3 設備應該都會有上述優先權等級。

# Jason 嘴書—EETOP 世界唯一貼

不要忘記設定系統例外與錯誤處理例外的優先權。如果必須使得某些重要的中斷擁有比其他的系統例外或錯誤處理程式更高的優先權，就要先降低其他的系統例外或錯誤處理程式的優先權，讓重要的中斷可以強佔這些處理程式。

## 致能中斷

設定了向量表與中斷優先權之後，此時可以致能中斷。然而，在你能夠真正致能中斷前，還需要下面兩個步驟：

1. 如果向量表位於寫入緩衝的記憶體位置，則需要資料同步化障礙(Data Synchronization Barrier, DSB)指令，以確定更新了向量表記憶體。大多數的情形下，記憶體寫入在幾個時脈週期內就會完成。然而，如果你的軟體需要能夠在不同的Cortex-M3 產品之間移植，此步驟可確保即使中斷在致能之後立刻發生，核心也可以取得更新的向量。
2. 中斷可能早已處於等待或之前已被宣稱，所以也需要清除等待狀態。例如，電源啟動時的信號突波(glitch)可能會意外地觸發一些中斷產生邏輯。此外，於某些週邊：例如 UART 裡，在 UART 接收器連結之前的雜訊，可能被誤認為資料而造成中斷等待，因此，可能需要在致能一個中斷前，先檢查並清除中斷的等待狀態。視周邊設計而定，如果等待狀態已經被設定，則周邊可能需要再次初始化。

在 NVIC 內，有兩個分開的暫存器位址用來致能與除能中斷。此對偶性質可確保每一中斷的致能或除能，不會影響或失去其他的中斷致能狀態。否則，經由基於軟體的READ-MODIFY-WRITE 動作，就可能會造成由中斷處理程式執行改變的致能暫存器狀態被覆蓋。為了設定致能，軟體需要計算 NVIC 裡 SETEN 暫存器的正確位元位置，以將 1 寫入；同樣的，為了除能中斷，軟體需要將 0 寫入 CLREN 暫存器的相關位元：

```
; 根據 IRQ 數值以致能 IRQ 的副程式
EnableIRQ
; 輸入 R0 = IRQ 數值
PUSH {R0-R2, LR}
AND.W R1, R0, #0x1F ; 為 IRQ 產生致能 bit pattern
MOV R2, #1 ; Bit pattern = (0x1<< (N & 0x1F))
LSL R2, R2, R1 ; 若 IRQ 數值大於 31
AND.W R1, R0, #0xE0
```

LSR R1, R1, #3	; 則產生位址位移 ; 位址位移 = (N/32)*4 ; (每一 word 有 32 IRQ enable)
LDR R0, = 0xE000E100	; 外部中斷#31 - #0 的 SETEN 暫存器
STR R2, [R0, R1]	; 把 bit pattern 寫入 SETEN 暫存器
POP {R0-R2, PC}	; 回復暫存器值並返回

同樣地，我們可以寫另一個除能 IRQ 的副程式：

```
; 根據 IRQ 數值以除能 IRQ 的副程式
DisableIRQ
; 輸入 R0 = IRQ 數值
PUSH {R0-R2, LR} ; 為 IRQ 產生除能 bit pattern
AND.W R1, R0, #0x1F ; Bit pattern = (0x1<< (N & 0x1F))
MOV R2, #1 ; 若 IRQ 數值大於 31
LSL R2, R2, R1 ; 則產生位址位移
AND.W R1, R0, #0xE0 ; 位址位移 = (N/32)*4
LSR R1, R1, #3 ; (每一 word 有 32 IRQ enable)
LDR R0, = 0xE000E180 ; 外部中斷#31 - #0 的 CLREN 暫存器
STR R2, [R0, R1] ; 把 bit pattern 寫入 CLREN 暫存器
POP {R0-R2, PC} ; 回復暫存器值並返回
```

也可以開發類似的副程式以設定或清除 IRQ 等待狀態暫存器。

### 存取 NVIC 中斷暫存器

NVIC 裡的大多數暫存器可藉著 word、half word 或 byte 來作存取。選擇正確的大小，可使得你的程式開發更加容易。例如，優先權等級暫存器最好以 byte 傳輸來撰寫程式，這樣做的話，可不必擔心會意外改變其他例外的優先權。

## 例外/中斷處理程式

在 Cortex-M3 裡，中斷處理程式可完全以 C 來撰寫。但在 ARM7 裡，常需要用組合語言處理程式以確定保留了所有的暫存器值，且在具有巢狀中斷支援的系統裡，處理器還需切換到不同的模式以避免遺失資訊。在 Cortex-M3 裡，因為不需要上述步驟，故程式撰寫更為容易。

# Jason 嘴書—EETOP 世界唯一貼

在組譯程式裡，一個簡單的例外處理程式可能如下：

```
irq1_handler
    ; 處理 IRQ 要求
    ...
    ; 去除宣稱週邊內 IRQ 要求
    ...
    ; 中斷返回
    BX      LR
```

在中斷服務程式內去除宣稱 IRQ 要求需視周邊設計而定。如果周邊以脈衝的形式產生中斷要求，此步驟則非必要。在某些情形下，如果周邊可在一個短時間內產生多個的中斷要求，則需要條件式地去除宣稱周邊裡的 IRQ 要求，以保證我們不會遺失新到的中斷。

在大多數的情形下，中斷處理程式不會只用到 R0-R3 與 R12 暫存器，因此也需要保留一些其他的暫存器值。下列程式保留了所有未在堆疊程序期間保留的暫存器值；但是如果一些暫存器並不被例外處理程式所用，則可於保留的暫存器列表裡將其省略：

```
irq1_handler
    PUSH   {R4-R11, LR}      ; 保留所有未於堆疊過程
                                ; 保留的暫存器
    ; 處理 IRQ 要求
    ...
    ; 去除週邊內 IRQ 要求的宣稱（選擇性的）
    ...
    POP    {R4-R11, PC}      ; 回復暫存器值並由中斷返回
```

因為 POP 是可啟動中斷返回的指令之一，所以我們可於相同的指令裡，結合暫存器復原與中斷返回。

隨著周邊的設計而定，可能會需要例外處理程式去程式化周邊，以去除例外要求的宣告。如果從周邊至 NVIC 的例外要求是一個脈衝信號，則例外處理程式無需去清除例外要求；否則，例外處理程式需要清除例外要求，以免在離開例外之後，又立即再次被置於等待。在傳統的 ARM 處理器裡，在得到服務之前，周邊需要維持其中斷要求，原因在於為先前的 ARM 核心設計的例外控制器，並沒有等待記憶體。

就 Cortex-M3 而言，如果周邊以脈衝的型態產生中斷要求，則 NVIC 會把要求儲存為等待要求的狀態，一旦處理器進行例外處理程式，將自動地清除等待狀態。藉著這個方式，例外處理程式並不需要對周邊編程以清除中斷要求。

## 軟體中斷

中斷可藉著各種方式觸發：

- ◆ 外部的中斷輸入
- ◆ 設定 NVIC 裡的中斷等待暫存器(詳見第八章)
- ◆ 經由 NVIC 裡的軟體觸發中斷暫存器(Software Trigger Interrupt Register, STIR) (詳見第八章)

在大部分的例子中，一些中斷未被使用，可以用來作軟體中斷。軟體中斷的動作類似 SVC，允許對系統服務做存取。然而，在預設下，用戶程式不能存取 NVIC，唯一例外情形是：當設定了 NVIC 組態控制暫存器裡的 USERSETMPEND 位元時，用戶程式可去存取 NVIC 的 STIR(參見附錄 D2 的表 D.17)。

與 SVC 不一樣，軟體中斷不是準確的，此意味即使不受中斷遮罩暫存器或其他中斷服務副程式阻擋之下，中斷強佔並不必然會立刻發生。因此，如果緊接在對 NVIC STIR 寫入之後的指令需要依賴軟體中斷的結果，則運算可能會失敗，其原因是軟體中斷可能會在緊接的指令執行後才會被觸動。

您可以使用 DSB 指令解決此問題。例如：

```
MOV    R0, #SOFTWARE_INTERRUPT_NUMBER
LDR    R1, =0xE000EF00      ; NVIC 中斷觸發暫存器位址
STR    R0, [R1]              ; 觸發軟體中斷
DSB                            ; 資料同步化障礙
...
```

然而，依然有另一個可能的問題：若中斷遮罩暫存器被設定，或者若產生軟體中斷的程式碼本身為例外處理程式，則有可能無法執行軟體中斷。因此產生軟體中斷的程式碼應當要檢查軟體中斷是否已被執行，此可藉由軟體中斷處理程式設定的軟體旗標來達成。

最後，設定 USERSETPEND 可能導致另外一個問題。作此設定之後，除了系統例外之外，用戶程式可以觸發任意的軟體中斷。因而，如果使用了 USERSETPEND，且系統包含了不受信任的用戶程式，例外處理程式需要檢查例外是否被允許，因為此例外可能為用戶程式所觸發。理想的做法是：如果系統包含了不受信任的用戶程式，則最好僅經由 SVC 來提供系統服務。

## 例外處理程式範例

在第 7 章裡，我們提到啟始向量表應當包括重置向量、NMI 向量、以及硬錯誤向量，其原因是 NMI 與硬錯誤處理程式在未致能任何例外時，仍可能發生。於程式開始之後，接著我們可以將向量表重置到 SRAM 裡一個不同的地方。隨著應用程式而定，向量表的重置並非必要。在下面的例子中，我們把新配置的向量表放在 SRAM 的開始處，然後資料變數接在它後面：

```

STACK_TOP EQU 0x20002000 ; SP 常數起始值
NVIC_SETEN EQU 0xE000E100 ; 設定致能暫存器的基底位址
NVIC_VECTTBL EQU 0xE000ED08 ; 向量表位移暫存器
NVIC_AIRCR EQU 0xE000ED0C ; 應用中斷與重置控制暫存器
NVIC_IRQPRI EQU 0xE000E400 ; 中斷優先權等級暫存器

AREA | Header Code |, CODE
DCD STACK_TOP ; SP 初始值
DCD Start ; 重置向量
DCD Nmi_Handler ; NMI 處理程式
DCD Hf_Handler ; 硬錯誤處理程式
ENTRY
Start ; 主程式開始
; 初始化暫存器
MOV r0, #0 ; 初始化暫存器
MOV r1, #0
...
; 複製舊向量表至新向量表
LDR r0, =0
LDR r1, =VectorTableBase
LDMIA r0!, {r2-r5} ; 複製 4 words
STMIA r1!, {r2-r5}

DSB ; 資料同步化障礙
; 設定向量表位移暫存器
LDR r0, =NVIC_VECTTBL

```

```

LDR r1, =VectorTableBase
STR r1, [r0]
...
; 設定優先權群組暫存器
LDR r0, =NVIC_AIRCR
LDR r1, =0x05FA0500 ; 第五優先權群組
STR r1, [r0]
; 設定 IRQ 0 向量
MOV r0, #0 ; IRQ#0
LDR r0, =Irq0_Handler
BL SetupIrqHandler
; 設定優先權
LDR r0, =NVIC_IRQPRI
LDR r1, =0xC0
STRB r1, [r0, #0] ; IRQ#0 優先權
; 設定 IRQ0 優先權於位移 = 0
; 注意：為 byte 儲存
; (IRQ#1 之位移 = 1)
DSB ; 資料同步化障礙
; 確定致能中斷之前已做好所有準備
MOV r0, #0 ; 選擇 IRQ#0
BL EnableIRQ
...
; -----
; 函數
SetupIrqHandler
; 輸入 R0 = IRQ 編號
; R1 = IRQ 處理程式
PUSH {R0, R2, LR}
LDR R2, =NVIC_VECTTBL ; 取得向量表位移
LDR R2, [R2]
ADD R0, #16 ; 例外號碼 = IRQ 號碼+16
LSL R0, R0, #2 ; 乘以 4 (每一向量為 4 bytes)
ADD R2, R0 ; 計算向量位置
STR R1, [R2] ; 儲存向量處理程式
POP {R0, R2, PC} ; 返回
EnableIRQ
; 輸入 R0 = IRQ 編號
PUSH {R0 - R3, LR}
AND R1, R0, #0x1F ; 取最低 5 個位元以求位元圖案
MOV R2, #1
LSL R2, R2, R1 ; R2 裡的位元圖案
BIC R0, #0x1F
LSR R0, #3 ; word 位移 (IRQ 編號可大於 32)
LDR R1, =NVIC_SETEN
STR R2, [R1, R0] ; 設定致能位元
POP {R0 - R3, PC} ; 返回
; -----
; 例外處理程式

```

接下頁

```

Hf_Handler
...
    BX    LR      ; 將你的程式置於此處
; 返回
Nmi_Handler
...
    BX    LR      ; 將你的程式置於此處
; 返回
Irq0_Handler
...
    BX    LR      ; 將你的程式置於此處
; 返回
;
AREA   | Header Data |, DATA
ALIGN 4
; 重置的向量表
VectorTableBase SPACE 256           ; bytes 數量
VectorTableEnd          ; (256/4 = 最多 64 個例外)
MyData1    DCD    0                 ; 變數
MyData2    DCD    0
;
END          ; 檔案結束

```

此範例稍微長一點，讓我們從結尾處，即資料區開始說起：

在資料記憶體區(於程式後段)，我們將 256 bytes 的空間定義為一個向量表(SPACE 256)。於此處可存放 64 個例外向量，如果你需要更多或更少的向量表空間，則可以改變其大小。其他的軟體變數接在向量表空間之後，故此時變數 MyData1 位於位址 0x20000100。

在程式開始處，我們定義一些位址常數以在其後的程式中使用，因此，我們可以使用這些常數名稱以取代數值，這樣可使得程式易於明瞭。

此時初始的向量表包括重置向量、NMI 向量、硬錯誤處理程式向量等。上述範例顯示如何設定例外向量，但並沒有包括實際的 NMI、硬錯誤、以及 IRQ 處理程式。這些處理程式的開發，隨實際的應用來決定。範例中使用了 BX LR 作例外返回，但可用其他合法的例外返回指令來取代它。

在初始化暫存器之後，我們將向量處理程式複製到 SRAM 裡新的向量表，此可藉著一個多重載入與一個多重儲存指令來完成。如果有更多的向量需要複製，我們僅需加上額外的載入/儲存指令，或者增多每一對載入與儲存指令所複製的 words 數。

當向量表準備好之後，我們可以將向量表重新配置到 SRAM 裡的新向量表。然而，為了確定向量處理程式的傳輸已完成，故使用了 DSB 指令。

接下來我們需要設置其餘的中斷設定。最先的一個是優先權群組的設定，此僅需要設定一次。於範例中，開發了稱作 SetupIrqHandler 與 EnableIRQ 的兩個副程式，以便於設定中斷；使用相同的程式碼，並僅將 NVIC\_SETEN 改為 NVIC\_CLREN，我們也可以加上被稱作 DisableIRQ 的類似副程式。在設定了處理程式與優先權等級之後，接著可以致能 IRQ。

## 使用 SVC

SVC 為允許用戶應用程式存取 OS 裡的 API 的通用方式，這是因為用戶應用程式僅需知道將哪些參數傳給 OS，而並不需要去知道 API 副程式的記憶體位址。

SVC 指令包括了一個參數，其為指令裡的一個 8-bit 立即值資料，在使用 SVC 指令時需要此值。例如：

```
SVC    3      ; 呼叫編號為 3 的系統服務
```

於 SVC 處理程式內，將需要從指令中把參數取回，此可藉由顯示於圖 11-2 的程序以達成。

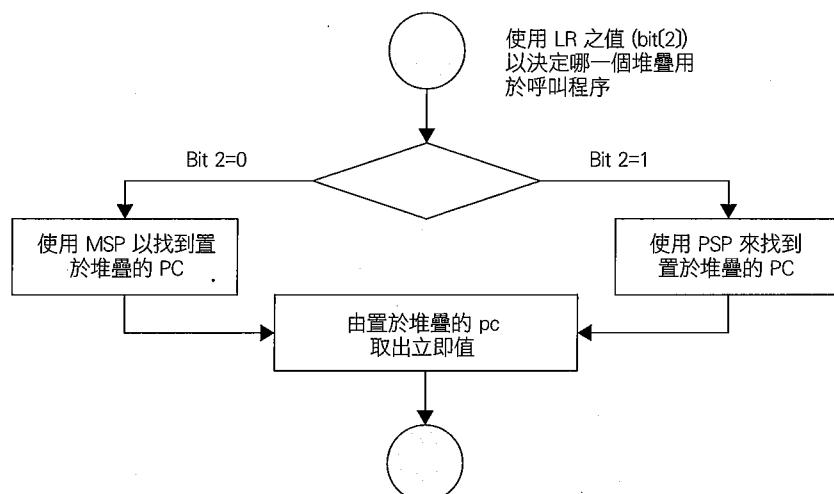


圖 11-2 取得 SVC 參數的一個方式

# Jason 嘴書—ETOP 世界唯一貼

此處是取得 SVC 參數的一個簡單的組合語言程式：

```

svc_handler
TST    LR, #0x4      ; 測試 LR 裡 bit 2 的 EXC_RETURN 數值
ITE    EQ              ; 如果為 0(相等), 則
MRSEQ   R0, MSP       ; 使用主要堆疊, 將 MSP 置於 R0
MRSNE   R0, PSP       ; 否則, 使用程序堆疊, 將 PSP 置於 R0
LDR    R1, [R0, #24]   ; 從堆疊取出置於堆疊的 PC
LDRB R0, [R1, #-2]    ; 從指令取出立即值資料
; 此時立即值資料位於 R0
...
BX     LR              ; 返回呼叫程式

```

一旦決定了 SVC 的呼叫參數，則可執行相關的 SVC 函數。欲跳躍到正確的 SVC 服務程式，一種有效率的方法是使用表格跳躍指令，例如 TBB 與 TBH。然而，如果使用了表格跳躍指令，除非已確定 SVC 呼叫參數包含正確數值，否則就應該檢查參數數值，以避免因不合法的 SVC 呼叫而導致系統衝突。

因為 SVC 呼叫不能經由例外機制要求另一個 SVC 服務，SVC 處理程式應當直接呼叫另外的 SVC 函數(例如，BL)。

## SVC 範例：作為輸出函數

先前我們開發了不同的副程式作為輸出函數。但有時候，會遇到不能使用 BL 去呼叫副程式的情況，例如，當函數置於不同的目的檔案而不能找到副程式的位址，或者當跳躍的位址範圍太大的時候。在這些情形下，我們可能想要使用 SVC 作為輸出函數的進入點。例如：

```

LDR    R0, =HELLO_TXT
SVC    0                  ; 顯示 R0 指向的字串
MOV    R0, #'A'
SVC    1                  ; 顯示 R0 裡的字元
LDR    R0, =0xC123456
SVC    2                  ; 顯示 R0 裡的十六進位數
MOV    R0, #1234
SVC    3                  ; 顯示 R0 裡的十進位數

```

欲使用 SVC，我們需設定 SVC 處理程式，我們可以把為 IRQ 所作的函數，稍加修改，其唯一不同處在於此函數接受一個例外型態的輸入(SVC 為例外型態 11)。此外，這一次我們進一步地最佳化程式碼以利用 Thumb-2 的特性：

```

; 輸入 R0 = 例外號碼
;          R1 = 例外處理程式
PUSH   {R0, R2, LR}
LDR    R2, =NVIC_VECTTBL           ; 取得向量表位移
LDR    R2, [R2]
STR.W  R1, [R2, R0, LSL #2]       ; 儲存 [R2+R0<<2] 裡的向量處理程式
POP    {R0, R2, PC}               ; 返回

```

就 svc\_handler 而言，SVC 呼叫號碼可以如先前的例子加以取出，而且傳至 SVC 的參數可經由讀取堆疊來存取，此外，增加了到達不同函數的決定分支：

```

TST    LR, #0x4      ; 測試 LR 中 bit 2 的 EXC_RETURN
ITTEE  EQ              ; 如果為 0 (相同), 則
MRSEQ  R1, MSP       ; 使用主要堆疊, 將 MSP 置於 R1
MRSNE  R1, PSP       ; 否則, 使用程序堆疊, 將 PSP 置於 R1
LDR    R0, [R1, #0]   ; 從堆疊取出置於堆疊的 R0
LDR    R1, [R1, #24]   ; 從堆疊取出置於堆疊的 PC
LDRB  R1, [R1, #-2]    ; 從指令取出立即值資料
; 此時立即值資料位於 R1，輸入參數位於 R0
PUSH   {LR}            ; 儲存 LR 至堆疊
CBNZ   R1, svc_handler_1
BL     Puts             ; 跳躍至 Puts
B     svc_handler_end

svc_handler_1
CMP   R1, #1
BNE  svc_handler_2
BL   Putc              ; 跳躍至 Putc
B   svc_handler_end

svc_handler_2
CMP   R1, #2
BNE  svc_handler_3
BL   PutHex             ; 跳躍至 PutHex
B   svc_handler_end

svc_handler_3
CMP   R1, #3
BNE  svc_handler_4
BL   PutDec              ; 跳躍至 PutDec
B   svc_handler_end

svc_handler_4
B   error              ; 輸入為未知
...
svc_handler_end
POP   {PC}              ; 返回

```

# Jason 嘴書—EETOP 世界唯一貼

需要將 svc\_handler 程式與輸出函數放在一起，以讓我們可以確定他們位於允許的跳躍範圍之內。

注意，我們以置於堆疊的暫存器內容而非暫存器現行的內容來傳遞參數。這是因為如果在執行 SVC 時，出現了更高優先權的中斷，則 SVC 將會接在其他中斷處理程式之後開始(未尾連鎖)，而 R0-R3 與 R12 的內容就可能會被中斷處理程式所改變，這是因為若存在中斷的末尾連鎖時，並不會執行去堆疊的特性之緣故。例如：

1. 以放在 R0 裡的參數作為參數
2. 與執行 SVC 同時間，出現更高優先權中斷
3. 堆疊被執行，並且 R0-R3、R12、LR、PC、以及 xPSR 被保留到堆疊。
4. 執行中斷處理程式。R0-R3 與 R12 可被處理程式更改，這是可接受的，其原因是這些暫存器將藉著硬體去堆疊來回復。
5. SVC 處理程式接在中斷處理程式後，作未尾連鎖。當進入 SVC 時，R0-R3 與 R12 裡的內容可能與 SVC 被呼叫時不同，然而，正確的參數存放於堆疊裡，並且可以被 SVC 處理程式存取。

## 善加利用定址模式

從 SetupIrqHandler 與 SetupExcHandler 副程式的程式範例，我們發現如果善加利用 Cortex\_M3 裡的定址模式的特性，程式可大為縮短。在 SetupIrqHandler 裡，計算了 IRQ 向量的目的位址之後，再執行儲存動作：

```
SetupIrqHandler
PUSH {R0, R2, LR}
LDR R2, =NVIC_VECTTBL ; 取得向量表位移 ; 步驟 1
LDR R2, [R2] ; ; 步驟 2
ADD R0, #16 ; 例外號碼 =IRQ 號碼+16 ; 步驟 3
LSL R0, R0, #2 ; 乘以 4 (每一向量為 4 bytes) ; 步驟 4
ADD R2, R0 ; 計算向量位置 ; 步驟 5
STR R1, [R2] ; 儲存向量處理程式 ; 步驟 6
POP {R0, R2, PC} ; 返回
```

在 SetupExcHandler，運算步驟 4-6 簡化，僅為一個步驟：

```
PUSH {R0, R2, LR}
LDR R2, =NVIC_VECTTBL ; 取得向量表位移
LDR R2, [R2]
STR.W R1, [R2, R0, LSL #2] ; 儲存 [R2+R0<<2] 裡的向量處理程式
POP {R0, R2, PC} ; 返回
```

通常，如果資料位址如下所示時，我們可以減少所需的指令數量：

- Rn + 2N\*Rm
- Rn +/- immediate\_offset

就 SetupIrqHandler 副程式而言，我們可得到的最簡短的程式如下：

```
SetupIrqHandler
PUSH {R0, R2, LR}
LDR R2, =NVIC_VECTTBL ; 取得向量表位移 ; 步驟 1
LDR R2, [R2] ; ; 步驟 2
ADD R2, #(16*4) ; 取得 IRQ 向量開始 ; 步驟 3
STR.W R1, [R2, R0, LSL #2] ; 儲存向量處理程式 ; 步驟 4
POP {R0, R2, PC} ; 返回
```

## 以 C 來使用 SVC

在大部分的情形下，SVC 函式的參數傳遞會需要一段組譯器處理程式碼，如先前所解釋，這是因為傳遞參數需經由堆疊，而非暫存器。如果要以 C 來開發 SVC 處理程式，可以用一個簡單的組合語言包裹器程式來取得置於堆疊的暫存器位置，並將它傳至 SVC 處理程式。接著 SVC 處理程式可以從堆疊指標資訊取得 SVC 號碼與參數。假設使用 RealView Development Suite (RVDS) 或 KEIL RealView Microcontroller Development Kit，則組譯器包裹器可藉由內嵌的組譯器來建構：

# Jason 嘴書—EETOP 世界唯一貼

```
// 用來取得堆疊框架起始位置的組譯器包裹器
// 把堆疊框架起始位置放在 R0，再跳躍到實際的 SVC 處理程式
__asm void svc_handler_wrapper (void)
{
    IMPORT      svc_handler
    TST         LR, #4
    ITE         EQ
    MRSEQ      R0, MSP
    MRSNE      R0, PSP
    B          svc_handler
} // 因為作 svc_handler 返回，故無需加上 BX LR 返回指令
// 應該把執行直接返回到 SVC 呼叫程式
```

剩下的 SVC 處理程式可於 C 撰寫，並以 R0 為輸入(堆疊框架起始位置)，其作用為取得 SVC 號碼並傳參數(R0-R3)：

```
// 以 C 撰寫 SVC 處理程式，堆疊框架位置為輸入參數
// 並把它當作記憶體指標以指向一個引數陣列
// svc_args[0] = R0, svc_args[1] = R1
// svc_args[2] = R2, svc_args[3] = R3
// svc_args[4] = R12, svc_args[5] = LR
// svc_arg[6] = 返回位址(置於堆疊的 PC)
// svc_args[7] = xPSR
void svc_handler(unsigned int * svc_args)
{
    unsigned int svc_number;
    unsigned int svc_r0;
    unsigned int svc_r1;
    unsigned int svc_r2;
    unsigned int svc_r3;

    // 記憶體[(被堆疊的 PC)-2]
    svc_number = ((char *) svc_args[6])[-2];
    svc_r0     = ((unsigned long) svc_args[0]);
    svc_r1     = ((unsigned long) svc_args[1]);
    svc_r2     = ((unsigned long) svc_args[2]);
    svc_r3     = ((unsigned long) svc_args[3]);
    printf ("SVC number = %x\n", svc_number);
    printf("SVC parameter 0 = %x\n", svc_r0);
    printf("SVC parameter 1 = %x\n", svc_r1);
    printf("SVC parameter 2 = %x\n", svc_r2);
    printf("SVC parameter 3 = %x\n", svc_r3);

    return;
}
```

注意，SVC 不能以與正常 C 相同的方式將結果回傳至呼叫程式。正常 C 函數藉著將函數以資料型態來定義，例如 `unsigned int func()`，並使用 `return` 來傳遞回傳值，但實際上回傳值放在暫存器 R0 裡。當離開處理程式時，如果 SVC 處理程式將回傳值放在暫存器 R0 至 R3，則暫存器的值可能會被去堆疊序列所覆蓋，因此，如果 SVC 必須回傳結果至呼叫程式，它需要直接修改堆疊框架，以便於去堆疊期間將回傳值載入暫存器。

對 ARM RealView Development Suite (RVDS)或 KEIL RealView Microcontroller Development Kit (RV-MDK)來說，要在 C 程式之內呼叫 SVC，可以使用 `_svc` compiler 關鍵字。例如，如果四個變數將被傳遞到 SVC 函數號碼 3，一個稱作 `call_svc_3` 的 SVC 可以如此宣告：

```
Void __svc(0x03) call_svc_3 (unsigned long svc_r0,
                               unsigned long svc_r1, unsigned long svc_r2,
                               unsigned long svc_r3);
```

如此將允許 C 程式碼這樣進行系統呼叫：

```
Int main(void)
{
    Unsigned long p0, p1, p2, p3; // 傳至 SVC 處理程式的參數
    ...
    call_svc_3 (p0, p1, p2, p3); // 呼叫 SVC 號碼 3，參數 p0, p1, p2, p3 傳至 SVC
    ...
    return;
}
```

於 RealView Development Suite 或 RealView C Compiler 裡使用 `_svc` 關鍵字的詳細資訊可在 RVCT 3.0 Compiler and Library Guide (Ref 6)中找到。

對於 GNU 工具鏈的用戶，因為 GCC 並無 `_svc` 關鍵字，故 SVC 需要藉著行內組譯器來存取。例如，如果 SVC 呼叫號碼 3 需要一個輸入變數，並且經由暫存器 R0 來回傳一個變數(根據 AAPCS, Ref 5，第一個傳遞參數將會使用到暫存器 R0)，則下述的行內組譯器程式可用來呼叫 SVC：

```
Int MyDataIn = 0x123;
__asm __volatile ("mov R0, %0 \n"
                  "svc 3 \n" : "" : "r" (MyDataIn) );
```

此行內組譯器程式可細分為下述部分，其輸入資料由 r(MyDataIn)指定，並且沒有輸出欄位(於先前程式裡以""指定)：

```
__asm (assembler_code : output_list : input_list)
```

有關 GNU 工具鏈使用行內組譯器的更多範例可於本書第十九章中找到，傳遞參數進或出行內組譯器的完整細節，請參考 GNU 工具鏈文件。

# 12

## Chapter

# 進階程式設計特性與系統行為

本章內容包括：

- ✓ 使用兩個不同堆疊來執行系統
- ✓ Double-Word 堆疊對齊
- ✓ Nonbase 執行緒致能
- ✓ 效能考慮
- ✓ 鎖住情形
- ✓ 錯誤遮罩

## 使用兩個不同堆疊來執行系統

v7-M 結構的一個重要特性是允許用戶應用程式堆疊與特權的/核心堆疊分開的能力。如果建置了可選用的 MPU，則可藉著它來阻擋用戶應用程式去存取核心堆疊記憶體，故應用程式不會破壞記憶體而與核心起衝突。

基於 Cortex-M3 的典型強健系統具有下列特性：

- ◆ 例外處理程式使用 MSP
- ◆ 於規則的時間間隔引發的 SYSTICK 例外所啟動的核心程式碼，會執行於特權的存取等級，以作工作排程與系統管理
- ◆ 以執行緒執行的用戶應用程式會處於用戶存取等級(非特權的)，並使用 PSP
- ◆ MSP 會指向核心與例外處理程式所用的堆疊記憶體；且僅在有 MPU 時，才會限制堆疊記憶體為特權的存取
- ◆ PSP 會指向用戶應用程式所用的堆疊記憶體

假設系統記憶體有一個 SRAM 記憶體，我們可以設定 MPU，將 SRAM 分為用戶以及特權存取兩個區域，而每一個區域都可做為應用程式資料與堆疊記憶體空間使用。因 Cortex-M3 裡的堆疊運算為全遞減的，故堆疊指標的初始值需要指向區域頂端。

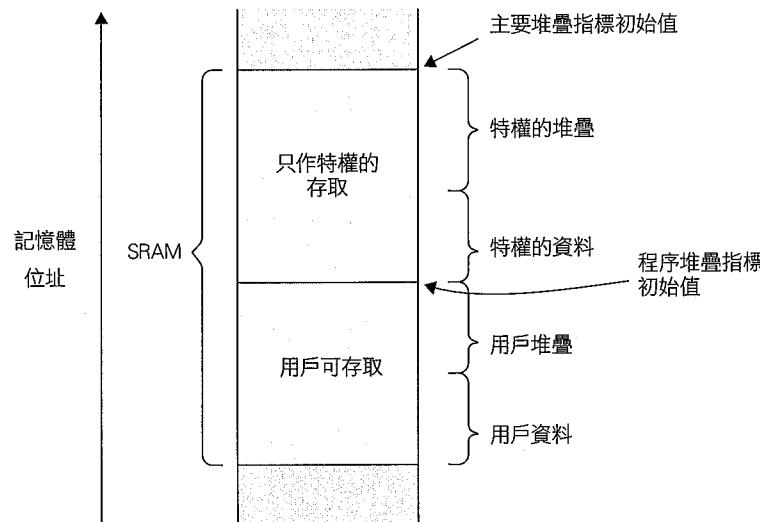


圖 12-1 作為特權的資料與用戶應用資料的記憶體使用範例

在啟動之後，僅有 MSP 被初始化(於啟動序列中，取得位址 0x0)。設定一個完整、強健的雙堆疊系統需要額外的步驟，如果是以組合語言撰寫的應用程式，可以如下這樣簡單：

```
; 以特權的等級開始程式(此程式碼位於用戶存取記憶體)
BL MpuSetup           ; 設定 MPU 區域並啟能記憶體保護
LDR R0, =PSP_TOP      ; 設定程序 SP 至程序堆疊頂端
MSR PSP, R0            ;
BL SystickSetup       ; 設定 Systick 與 Systick 例外
; 在一定時間的間隔啟動 OS 核心
MOV R0, #0x3           ; 設定 CONTROL 暫存器以讓用戶程式使用 PSP
MSR COBTROL, R0        ; 切換現行存取等級至用戶
B UserApplicationStart ; 此時我們進入用戶存取等級
; 用戶程式開始
```

這樣的安排適用於組譯器，但對於 C 程式而言，於 C 函數中途切換堆疊指標可能造成區域變數的遺失(因為在 C 函數或副程式中，區域變數可能放在堆疊記憶體)。Cortex-M3 TRM (Ref 1)建議我們使用 ISR(例如 SVC)去啟動核心，再藉著修改 EXC\_RETURN 值以改變堆疊指標。

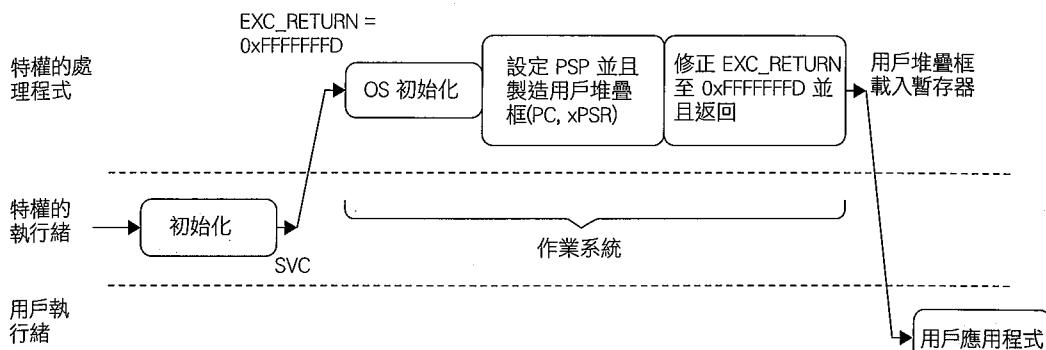


圖 12-2 於簡單 OS 中，初始化多重堆疊

在大多數的情形下，EXC\_RETURN 修改以及堆疊切換包含於作業系統裡。在用戶應用程式開始以後，SYSTICK 例外可在固定的時間間隔啟動作業系統，以作系統的管理，並且可能在必要時，安排環境的切換。

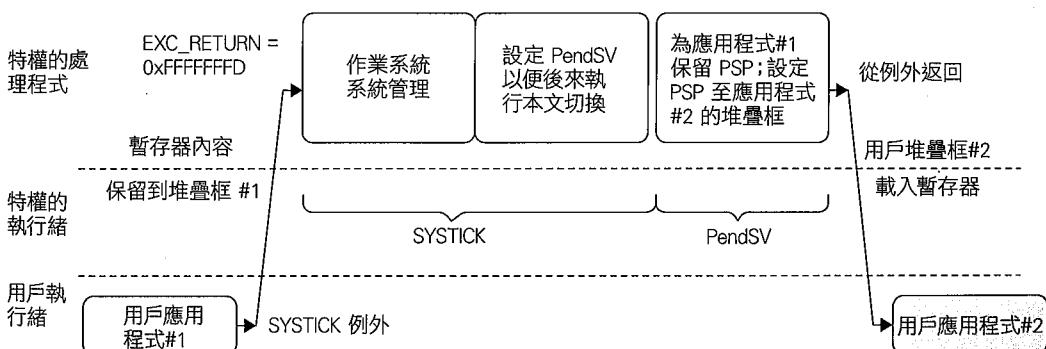


圖 12-3 於簡單 OS 中，環境的切換

注意，環境切換於 PendSV 裡執行(一個低優先權例外)，以防止在中斷處理程式的中途作環境的切換。

然而，許多應用程式並不需要作業系統，但仍然需要在應用程式不同的區段，使用不同的堆疊，因為這是增進可靠度的一個方式。要這樣處理的可能方法是在啟動 Cortex-M3 時，使用 MSP 指向程序堆疊區。這樣做，可以利用 MSP 以完成程序堆疊區的初始化。在開始用戶應用程式之前，會執行下述程式：

```
; 以特權的等級開始程式，MSP 指向用戶堆疊
MPUSetup( ); // 設定 MPU 區域並且致能記憶體保護
SystickSetup( ); // 設定 Systick 與 systick 例外，// 以作例行的系統管理程式
SwitchStackPointer( ); // 呼叫組合語言副程式以切換 SP
/*; -----於 SwitchStackPointer 內-----
PUSH {R0, R1, LR}
MRS R0, MSP ; 保留現行的堆疊指標
LDR R1, =MSP_TOP ; 變更 MSP 到新的位置
MSR MSP, R1
MSR PSP, R0 ; 儲存現行堆疊指標於 PSP
MOV R0, #0x3
MSR CONTROL, R0 ; 切換到用戶模式，並使用 PSP 為現行堆疊
POP {R0, R1, PC} ; 返回
; -----回到 C 程式-----*/
; 此時我們位於用戶模式，使用了 PSP 與區域變數
; 依然位於此處
UserApplicationStart( ); // 於用戶模式開始應用程式
```

## Double-Word 堆疊對齊

於依據 AAPCS<sup>1</sup> 的應用程式中，必須確認在例外處理時暫存器進堆疊時，是以原始資料大小作對齊(1、2、4、8 bytes)。此為在 Cortex-M3 處理器的組態選項。欲致能此特性，需要設定 NVIC 組態控制暫存器裡的 STKALIGN 位元(參考附錄 D 表 D.17)。例如，此可藉由下列組合語言來達成：

```
LDR R0, =0xE000ED14 ; 設定 R0 為 NVIC CCR 的位址
LDR R1, [R0]
ORR.W R1, R1, #0x200 ; 設定 STKALIGN 位元
STR R1, [R0] ; 寫到 NVIC CCR
```

<sup>1</sup> AAPCS 即 Procedure Call Standard for the ARM Architecture(Ref 5)。在 ARM 的網頁就 SP 對齊與 AAPCS 做了建議的告示，參見 [www.arm.com/pdfs/ABI-Advisory-1.pdf](http://www.arm.com/pdfs/ABI-Advisory-1.pdf)。

或者使用 C 語言來達成：

```
#define NVIC_CCR ((volatile unsigned long *) (0xE000ED14))
NVIC_CCR = *NVIC_CCR | 0x200; /* 於 NVIC 中，設定 STKALIGN */
```

當 STKALIGN 位元在進行例外進堆疊時被設定，則進堆疊的 xPSR 的 bit 9 會被用來指出是否做了對齊堆疊的堆疊指標調整。在去堆疊時，SP 調整會檢查堆疊中的 xPSR 的 bit 9，並且依檢查結果來調整 SP。

為了防止堆疊資料破壞，在例外處理程式中 STKALIGN 位元不可以被更改，因為若被更改，則會造成例外之前與之後的堆疊指標位置不符。

此項特性是自 Cortex-M3 revision 1 以後才有；基於 revision 0 的早期 Cortex-M3 產品並沒有此特性。在 revision 2 時此特性預設是被致能的，但在 revision 裡則需要藉由程式以開啟它。如果需要遵循 AAPCS 則應該使用此特性，而當應用程式(或其中一部分)以 C 開發，並且其程式包括 double-word 大小的資料時，也推薦使用此特性。

## Nonbase 執行緒致能

於 Cortex-M3 裡可以把執行中的中斷處理程式，從特權等級切換到用戶存取等級；這在中斷處理程式為用戶應用程式的一部分，因而不應允許擁有特權存取時，是必要的。此特性可藉由 NVIC 組態控制暫存器裡的 Nonbase 執行緒致能(NONBASETHRDENA)的位元來加以致能。

### 使用此特性需謹慎

因為需要手動調整堆疊並修改被堆疊的資料，在撰寫正常的程式應用時應儘量避免使用此特性。如果必須使用此特性時，要非常小心，並且系統設計師需要確認正確地終結中斷服務副程式。否則，它可能會造成與其相同或更低優先權等級的中斷被遮罩。

欲使用此特性，會牽涉到例外處理程式的重新指向；向量表裡的向量指向一個處理程式，此處理程式執行於特權模式但卻位於用戶模式可存取的記憶體裡：

```
directive_handler
PUSH {LR}
SVC 0           ; 由特權模式更改變為用戶模式的 SVC 函數
BL  User_IRQHandler
SVC 1           ; 由用戶模式更改變回特權模式的 SVC 函數
POP {PC}         ; 返回
```

SVC 處理程式可分作三個部分：

- ◆ 當呼叫 SVC 時，決定參數。
- ◆ SVC service #0 啟動了 nonbase 執行緒致能，調整用戶堆疊與 EXC\_RETURN 的值，並且使用程序堆疊，返回到用戶模式中的重新指向處理程式。
- ◆ SVC service #1 除能了 nonbase 執行緒致能，恢復用戶堆疊指標的位置，並且使用主要堆疊，返回到特權模式中的重新指向處理程式。

```
svc_handler
TST LR, #0x4          ; 測試 EXC_RETURN bit 2
ITE EQ                ; 如果為 0，則
MRS R0, MSP            ; 取正確的堆疊指標至 R0
MRSNE R0, PSP
LDR R1, [R0, #24]      ; 取被堆疊的 PC 值
LDRB R0, [R1, #-2]      ; 於被堆疊的 PC-2 位置取參數
CBZ r0, svc_service_0
CMP r0, #1
BEQ svc_service_1
B.W Unknown_SVC_Request

svc_service_0          ; 從特權模式切換處理程式至用戶模式的服務
MRS R0, PSP            ; 調整 PSP
SUB R0, R0, #0x20 ; PSP = PSP + 0x20
MSR PSP, R0
MOV R1, #0x20          ; 從主要堆疊複製堆疊框架至程序堆疊

svc_service_0_copy_loop
SUBS R1, R1, #4
LDR R2, [SP, R1]
STR R2, [R0, R1]
```

```
CMP R1, #0
BNE svc_service_0_copy_loop
STRB R1, [R0, #0x1C]      ; 清除堆疊於用戶堆疊的 IPSR 至 0
LDR R0, =0xE000ED14      ; 設定 CCR 裡的 non-base 執行緒致能
LDR r1, [r0]
ORR r1, #1
STR r1, [r0]
ORR LR, #0xC              ; 使用 PSP 變更 LR 以返回執行緒
BX LR

svc_service_1
MRS R0, PSP
LDR R1, [R0, #0x18]        ; 從用戶模式切換處理程式回到特權模式的服務
LDR R1, [SP, #0x18]        ; 更新特權堆疊中被堆疊的 PC 值
ADD R0, R0, #0x20          ; 故其可於重新指向處理程式
MSR PSP, R0
LDR R0, =0xE000ED14      ; 第 2 次 SVC 之後返回指令
LDR r1, [r0]
BIC r1, #1
STR r1, [r0]
BIC LR, #0xC              ; 調整 PSP 回到第 1 次 SVC 之前的值
BX LR

; 清除 CCR 中的 non-base 執行緒致能
; 使用主要堆疊，返回處理程式模式
```

使用 SVC 服務的原因在於改變 IPSR 的唯一方式是經由例外返回。雖然也可以使用其它的例外，例如軟體觸發中斷，但並不推薦，因為它們並不精確且可能會被遮罩，此意味需要的堆疊複製與切換運算可能不會被立即執行。圖 12-4 顯示了程式的順序，其中包括堆疊指標的改變與現行的例外優先權。

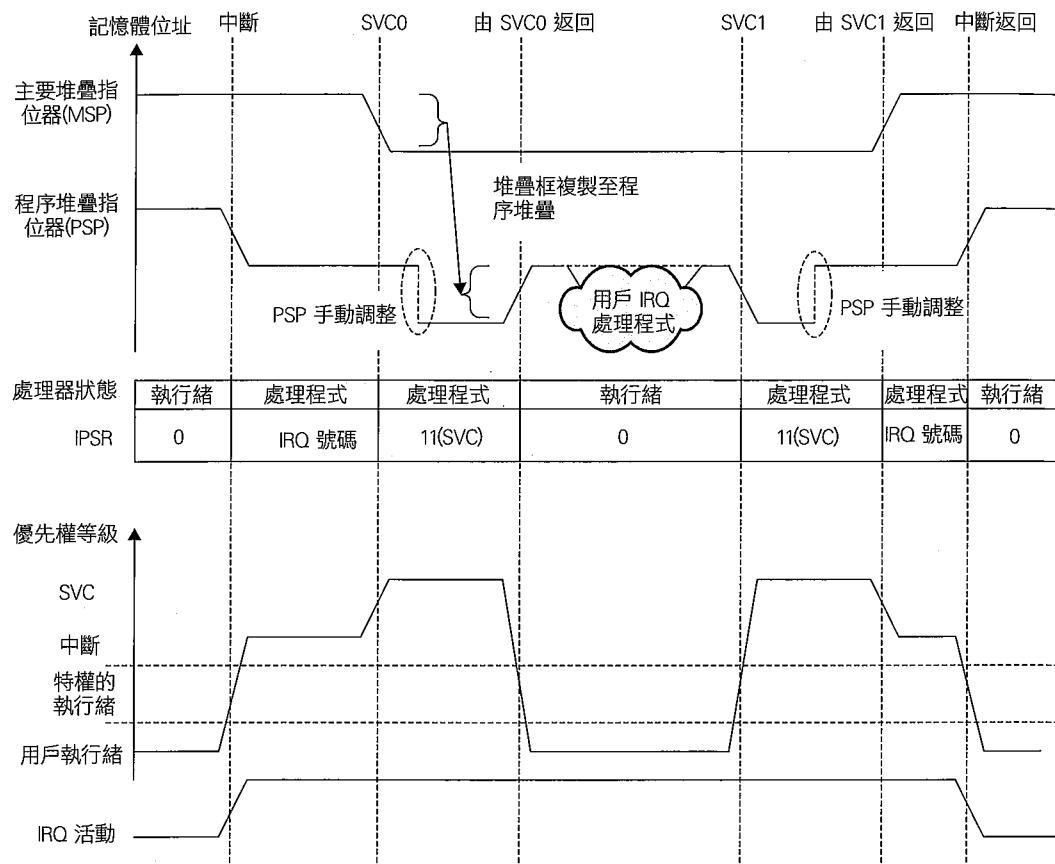


圖 12-4 non-base 執行緒致能的運算

於示意圖裡，以虛線的圓圈標示手動調整 SVC 服務內的 PSP。

## 效能考慮

欲發揮 Cortex-M3 最佳效能，需要考慮一些事情。第一，我們需要避免記憶體等待狀態。在微控制器或 SoC 設計階段，設計師應該最佳化記憶體系統設計，以允許同時執行指令與資料存取，並且儘量使用 32-bit 記憶體。就開發者而言，安排好記憶體映射，故程式由程式區執行，並且大多數的資料存取經由系統匯流排。如此安排，資料存取與指令擷取可同時進行。

第二，中斷向量表也應該儘量地放入程式區。如此則向量擷取與堆疊可以同時進行。如果向量表位於 SRAM 中，因為向量擷取與堆疊可能分享相同的系統匯流排，所以額外的時脈週期可能會造成中斷延遲(除非堆疊位於使用 D-Code 匯流排的程式區)。

如果可能的話，避免使用無對齊的傳輸，因為無對齊的傳輸可能會使用兩個以上的 AHB 傳輸來完成，降低程式的效能，所以要謹慎地規劃你的資料結構。如果使用 ARM 工具的組合語言，可以利用 ALIGN 假指令以確認資料位置是否對齊。

大部分的讀者會使用 C 語言作開發，但對使用組合語言的人，則可利用一些技巧加快部分程式：

1. 藉位移來使用記憶體存取指令。當存取小區域的多個記憶體位置，可用下列指令：

```

LDR    R0, =0xE000E400 ; 設定中斷優先權 #3, #2, #1, #0
LDR    R1, =0xE0C02000 ; 優先權等級
STR   R1, [R0]
LDR    R0, =0xE000E404 ; 設定中斷優先權 #7, #6, #5, #4
LDR    R1, =0xE0E0E0E0
STR   R1, [R0]

```

你可以簡化上述程式如下：

```

LDR    R0, =0xE000E400 ; 設定中斷優先權 #3, #2, #1, #0
LDR    R1, =0xE0C02000 ; 優先權等級
STR   R1, [R0]
LDR    R1, =0xE0E0E0E0 ; 設定中斷優先權 #7, #6, #5, #4
STR   R1, [R0, #4]      ; 優先權等級

```

上述第二個儲存指令使用了第一個位址的位移，並因此減少了指令的數量。

2. 結合多個記憶體存取成為載入/儲存多重指令(LDM/STM)。上面的程式可藉著 STM 指令，進一步簡化：

```

LDR    R0, =0xE000E400 ; 設定中斷優先權基底
LDR    R1, =0xE0C02000 ; 優先權等級#3, #2, #1, #0
LDR    R2, =0xE0E0E0E0 ; 優先權等級#7, #6, #5, #4
STR   R0, [R1, R2]

```

3. 使用 IT 指令區塊以取代小段的條件跳躍。因為 Cortex-M3 為管線處理器，故進行跳躍運算會有跳躍的代價，如果條件式跳躍運算被用來越過一小段指令，則可使用 IT 指令區塊來取代之，這樣可能節省一些時脈週期。
4. 如果運算可藉著兩個 Thumb 指令或者一個 Thumb-2 指令來執行，則採用 Thumb-2 指令的方法，雖然他們佔用的記憶體大小相同，但 Thumb-2 指令的方式有較短的執行時間。

## 鎖住情形

錯誤狀況出現時，相關的錯誤處理程式將會被觸發。如果在用法錯誤/匯流排錯誤/記憶體管理錯誤處理程式之內又發生了另一個錯誤，將會觸發硬錯誤處理程式。然而，如果我們在硬錯誤處理程式之內又得到另外一個錯誤？此時，則會出現鎖住的情形。

### 鎖住時會發生哪些事情？

鎖住期間，程式計數器將會被強制到 0xFFFFFFF，並且一再地存取此位址。此外，來自 Cortex-M3 的輸出信號(LOCKUP)會被宣稱以顯示其處境。晶片設計師可能使用此信號，於系統重置產生器觸發重置。

鎖住於下列情況發生：

1. 在硬錯誤處理程式之內又有錯誤產生(雙重錯誤)
2. 在 NMI 處理程式之內有錯誤產生
3. 在重置序列之內有匯流排錯誤產生(初始的 SP 或 PC 摷取)

在雙重錯誤情形下，核心依然可能回應 NMI 並執行 NMI 處理程式，但在完成處理程式之後，它將返回鎖住狀態，且程式計數器恢復為 0xFFFFFFF。在這個情形下，系統被鎖住並且現行優先權等級留在-1。如果 NMI 產生，處理器將依然作強佔並執行 NMI 處理程式，其原因是 NMI 比現行的優先權等級(-1)有更高的優先權(-2)。當 NMI 完成並返回鎖住狀態，現行的例外優先權恢復為-1。

通常，離開鎖住的最好方式是執行重置。或者，若系統有附加的除錯器，則可用來暫停核心，更改 PC 到不同的值，並由此處開始程式的執行。在大多數的情形下，這不是

一個好主意，因為一些暫存器，包括中斷系統，在系統能夠回復正常運算前，需要再次初始化。

你可能猜想鎖住發生時為何不就簡單地重置核心，當然在使用中的系統，你可能會如此做，但在軟體開發階段，我們應該首先試著找出造成的原因。如果我們即刻重置核心，因為暫存器被重置且硬體狀態被改變，就可能會無法分析錯誤所在。在大多數 Cortex-M3 微控制器，看守狗計時器可以在進入鎖住狀態時，用來重置核心。

注意：當進入硬錯誤處理程式或 NMI 處理程式的進堆疊期間，若出現匯流排錯誤，並不會造成鎖住，但匯流排錯誤處理程式會被置於等待。

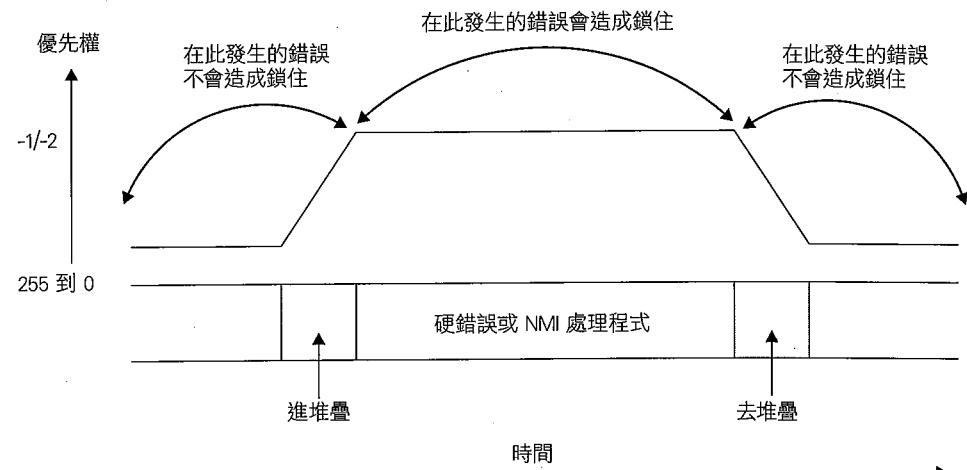


圖 12-5 只要在硬錯誤或 NMI 處理程式期間出現一個錯誤，就會造成鎖住

### 避免鎖住

在你開發 NMI 或硬錯誤處理程式時，多加留心以避免鎖住的問題是非常重要的。例如，在硬錯誤處理程式裡，除了確定記憶體工作正常並且堆疊指標依然合法，我們應該避免掉不必要的堆疊存取。開發複雜的系統時，造成匯流排錯誤或記憶體錯誤的一個可能原因是堆疊指標被破壞了。如果我們以下列指令作為硬錯誤處理程式開始：

```
hard_fault_handler
PUSH {R4-R7, LR}
; 除非你確定可安全地使用堆疊
; 否則這會是個壞主意
...
```

並且錯誤原因為堆疊錯誤，那就可能馬上在硬錯誤處理程式內進入鎖住。通常，當撰寫硬錯誤、匯流排錯誤、以及記憶體管理錯誤處理程式時，在執行更多的堆疊運算之前，值得去檢查堆疊指標是否處於合法的範圍。當撰寫 NMI 處理程式時，使用 R0-R3 與 R12 可以減低堆疊運算造成的風險，因為他們早已被推進堆疊。

開發硬錯誤與 NMI 處理程式的一個對策，是在處理程式之內僅執行重要的工作；而其餘工作，例如錯誤報告，可藉由不同的例外(例如 PendSV 或軟體中斷)把它們置於等待。這對撰寫小而強健的硬錯誤處理程式或 NMI 有幫助。

再者，我們應該確定 NMI 與硬錯誤處理程式碼不會去試著使用 SVC 指令；因為 SVC 通常有比硬錯誤與 NMI 更低的優先權，故在這些處理程式裡使用 SVC 會造成鎖住。這雖然看來簡單，但當你的應用程式變為複雜，而且在 NMI 與硬錯誤處理程式內會呼叫函數，就可能會不小心地呼叫到內含 SVC 指令的函數。因此，在開發軟體前，需要謹慎地計畫 SVC 的建置。

## 錯誤遮罩

錯誤遮罩是用來將可組態錯誤處理程式(匯流排錯誤、用法錯誤或者記憶體管理錯誤)升級到硬錯誤等級，而不需要藉著真正的錯誤來喚起硬錯誤。此允許將可組態錯誤管理程式假裝作硬錯誤處理程式，使得錯誤處理程式可以：

1. 藉著設定組態控制暫存器裡的 HFHFNMIGN 以遮罩匯流排錯誤。這可以在不造成鎖住的前提下，偵查匯流排系統。例如，用來檢察匯流排橋是否正確地工作。
2. 略過 MPU。這在僅僅需要幾次傳輸就能修正錯誤的情況下，可以不需對 MPU 重新寫程式的前提下，讓錯誤處理程式存取受 MPU 保護的記憶體位置。

FAULTMASK 的用法異於 PRIMASK。通常 PRIMASK 用在計時為關鍵的程式，但它並沒有能力去遮罩匯流排錯誤或是略過 MPU。在設定了 PRIMASK 之後，所有的可組態錯誤將會被升級為硬錯誤處理程式。而 FAULTMASK 藉著使用在正常情形下僅有硬錯誤處理程式可用的特性，用來允許可組態錯誤處理程式去解決與記憶體相關的問題。然而，當設定了 FAULTMASK 後，像是不正確定義的指令等錯誤，若在錯誤的優先權等級裡使用 SVC，依然可能會造成鎖住。

Chapter

# 13

## 記憶體保護單元

本章內容包括：

- ✓ 綜觀
- ✓ MPU 暫存器
- ✓ 設定 MPU
- ✓ 典型設定

## 綜觀

Cortex-M3 設計包括了一個選用的記憶體保護單元(MPU)。在微控制器或 SoC 產品中包含 MPU 可提供記憶體保護特性，這可增進開發產品的強韌性。MPU 在使用之前需要編程與致能。如果 MPU 未被致能，記憶體系統的行為會與並無 MPU 存在的系統行為相同。

MPU 可以就以下方式增強嵌入式系統的可靠性：

- ◆ 預防用戶程式破壞作業系統使用的資料
- ◆ 藉著阻擋工作去存取其它工作的資料，故能分隔處理工作之間的資料
- ◆ 允許定義記憶體區域為唯讀的，因而保護了重要的資料
- ◆ 偵測非預期的記憶體存取(例如，堆疊的破壞)

除此之外, MPU 也可以用來定義記憶體存取的特性, 例如對不同區域的快取與緩衝的行為。

MPU 設定保護的方式是藉著定義記憶體映射為一些區域。最多可設定 8 個區域, 但也可能定義一個預設的背景記憶體映射以作為特權的存取。存取未定義於 MPU 區域, 或者區域設定不允許的記憶體位置, 將會造成記憶體管理錯誤例外。

MPU 區域可以重疊；如果記憶體位置在兩個區域出現, 則記憶體存取屬性與權限將會根據最高號碼的區域。例如, 如果傳輸的位址落於定義為區域 1 與區域 4 的位址範圍內, 就會採用區域 4 的設定。

## MPU 暫存器

MPU 包括一些暫存器。第 1 個是 MPU Type 暫存器(見表 13-1)。

表 13-1 MPU Type 暫存器(0xE000ED90)

位元	名稱	類型	重置值	描述
23:16	IREGION	R	0	此 MPU 支援的指令區域數量；因為 ARM v7-M 架構使用了統合的 MPU, 此值通常為 0
15:8	DREGION	R	0 或 8	此 MPU 支援的區域數量；在 Cortex-M3 中此值不是 0(沒有 MPU)就是 8(有 MPU)
0	SEPARATE	R	0	因為 MPU 為統合的, 故此值通常為 0

MPU Type 暫存器可以用來決定 MPU 是否存在。如果 DREGION 欄位讀值為 0, 則 MPU 未被建置(見表 13-2)。

表 13-2 MPU 控制暫存器(0xE000ED94)

位元	名稱	類型	重置值	描述
2	PRIVDEFENA	R/W	0	致能特權的預設記憶體映射。當被設定為 1 且若 MPU 被致能, 則預設之記憶體映射將會被當作背景區域而用來作特權的存取。如果此位元沒有被設定, 則背景區域會被除能, 並且對未被致能之區域覆蓋的任何存取將會造成錯誤。
1	HFNMIENA	R/W	0	如果設定為 1, 則在硬錯誤處理程式與 NMI 處理程式期間會致能 MPU; 否則 MPU 不會因為硬錯誤處理程式與 NMI 而致能。
0	ENABLE	R/W	0	如果設定為 1, 則致能 MPU。

藉著使用 PRIVDEFENA 且無設定其它的區域, 特權程式將能夠存取所有的記憶體位置, 僅有用戶程式會被阻擋。然而, 如果其它的 MPU 區域被編程且致能, 他們可能會覆蓋背景區域。例如, 若兩個系統具有相似區域設定但僅有一個系統其 PRIVDEFENA 設定為 1(圖 13-1 裡右半部), PRIVDEFENA 設定為 1 的系統將會允許對背景區域的特權存取。

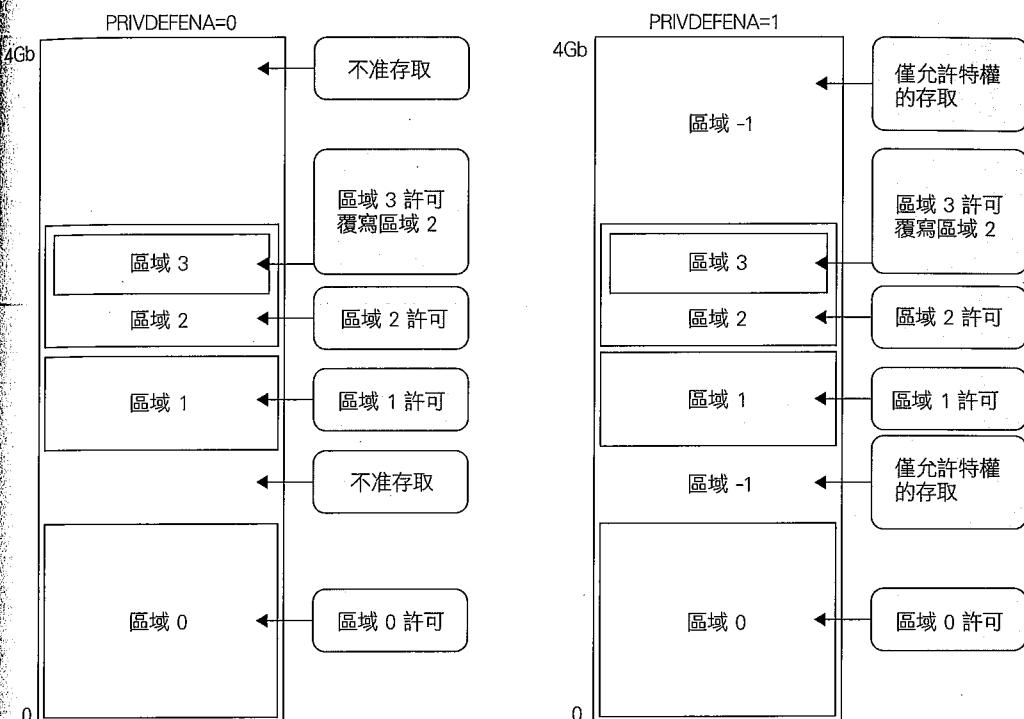


圖 13-1 PRIVDEFENA 的影響

設定 MPU 控制暫存器的致能位元通常是 MPU 設定程式的最後步驟。否則, 在區域組態完成之前, MPU 可能會意外的產生錯誤。在某些情形下, 在 MPU 組態程式開始時, 可能得將 MPU 除能, 確保在設定 MPU 區域時, 不會觸發 MPU 錯誤。

表 13-3 MPU 區域號碼暫存器(0xE000ED98)

位元	名稱	類型	重置值	描述
7:0	REGION	R/W	-	選擇正被編程的區域。因為在 Cortex-M3 MPU 裡支援 8 個區域, 故此區域僅實作了 bit(2:0)。

設定每一個區域前，先寫入這個暫存器以選擇編程區域(見表 13-3)。

利用 MPU 區域基底位址暫存器(見表 13-4)裡的 VALID 與 REGION 欄位，我們可以略過編程 MPU 區域號碼暫存器。這樣可能會減少程式的複雜性，特別是如果整個 MPU 設定是定義在查看表時。

表 13-4 MPU 區域基底位址暫存器(0xE000ED9C)

位元	名稱	類型	重置值	描述
31:N	ADDR	R/W	-	區域的基底位址；N 依照區域大小而定，例如，64k 大小之區域將會有[31:16]的基底位址欄位。
4	VALID	R/W	-	若此為 1，則 bit[3:0]定義的 REGION 將會在此程式步驟裡使用；否則，會使用由 MPU 區域號碼暫存器選擇的區域。
3:0	REGION	R/W	-	若 VALID 為 1，此欄位會蓋過 MPU 區域號碼暫存器的選擇，否則它會被忽略。因為在 Cortex-M3 MPU 裡支援 8 個區域，故若 REGION 欄位的值大於 7 則將會忽略區域號碼的覆蓋。

我們也需要定義每一個區域的記憶體位址與特性，此為 MPU 區域基底屬性與大小暫存器所控制(見表 13-5)。

表 13-5 MPU 區域基底屬性與大小暫存器(0xE000EDAO)

位元	名稱	類型	重置值	描述
31:29	保留的	-	-	-
28	XN	R/W	-	指令存取除能(1 = 除能由此區域的指令擷取；試圖擷取將會造成記憶體管理錯誤)
27	保留的	-	-	-
26:24	AP	R/W	-	資料存取許可欄位
23:22	保留的	-	-	-
21:19	TEX	R/W	-	類型擴展欄位
18	S	R/W	-	可共享的
17	C	R/W	-	可快取的
16	B	R/W	-	可緩衝的
15:8	SRD	R/W	-	次區域除能
7:6	保留的	-	-	-
5:1	REGION SIZE	R/W	-	MPU 保護區域大小
0	SZENABLE	R/W	-	區域致能

MPU 區域基底屬性與大小暫存器裡的 REGION SIZE 欄位(5-bit)決定了區域的大小(見表 13-6)。

表 13-6 對不同記憶體區域大小的 REGION 欄位之編碼

REGION SIZE	大小
b00000	保留的
b00001	保留的
b00010	保留的
b00011	保留的
b00100	32 Byte
b00101	64 Byte
b00110	128 Byte
b00111	256 Byte
b01000	512 Byte
b01001	1 KB
b01010	2 KB
b01011	4 KB
b01100	8 KB
b01101	16 KB
b01110	32 KB
b01111	64 KB
b10000	128 KB
b10001	256 KB
b10010	512 KB
b10011	1 MB
b10100	2 MB
b10101	4 MB
b10110	8 MB
b10111	16 MB
b11000	32 MB
b11001	64 MB
b11010	128 MB
b11011	256 MB
b11100	512 MB
b11101	1 GB
b11110	2 GB
b11111	4 GB

次區域除能欄位(MPU 區域基底屬性與大小暫存器的 bit[15:8])可以用來分割區域為八個相同大小的次區域，並定義其致能或除能。若一除能之次區域與其它區域重疊，則會套用其它區域的存取規則；若一除能之次區域並未與其它區域重疊，則對此記憶體範圍的存取會造成記憶體管理錯誤。小於 128 bytes 的區域不可使用次區域。

資料存取許可(Access Permission, AP)欄位位元(bit[26:24])定義了區域的存取許可(見表 13-7)。

表 13-7 AP 欄位對不同存取許可組態的編碼

AP 值	特權的存取	用戶存取	描述
000	不可存取	不可存取	不可存取
001	R/W	不可存取	僅作特權的存取
010	R/W	唯讀	在用戶程式裡的寫入會產生錯誤
011	R/W	R/W	全存取
100	不可預測	不可預測	不可預測
101	唯讀	不可存取	僅作特權的讀取
110	唯讀	唯讀	唯讀
111	唯讀	唯讀	唯讀

XE (Execute Never, 決不可執行)欄位(bit[28])決定是否允許由此區域作指令擷取。當此欄位設定為 1 時,所有從此區域擷取的指令,會在其進入執行階段時,產生記憶體管理錯誤。

TEX、S、B、以及 C 欄位(bit[21:16])更為複雜。雖然事實上 Cortex-M3 處理器並沒有快取,但因為它遵循 ARM v7-M 架構來建構,故它支援外部快取與更高階的記憶體系統。因而,可以編程區域存取特性以支援不同種類的記憶體管理模型。

在 v6 與 v7 結構中,記憶體系統可以有兩個快取等級:內部快取與外部快取。他們可以有不一樣的快取政策。由於 Cortex-M3 處理器本身並沒有快取控制器,故快取規則僅會影響內部匯流排矩陣與(可能是)記憶體控制器的寫入緩衝(見表 13-8)。

表 13-8 其中(S)表示可共享性(shareability)是由 S 位元欄位決定(被多個處理器所分享)

TEX	C	B	描述	區域可共享性
b000	0	0	強烈順序的(傳輸的執行與完成依照編程的次序)	可共享的
b000	0	1	共享的設備(寫入可以被緩衝)	可共享的
b000	1	0	外部與內部 寫穿(write-through); 無寫入配置(no write allocate)	(S)
b000	1	1	外部與內部回寫( write-back); 無寫入配置	(S)
b001	0	0	外部與內部不可快取的	(S)
b001	0	1	保留的	保留的
b001	1	0	由實作定義	-
b001	1	1	外部與內部回寫; 寫入及讀取配置	(S)
b010	0	0	不共享的設備	不共享的
b010	0	1	保留的	保留的
b010	1	x	保留的	保留的
b1BB	A	A	快取的記憶體; BB = 外部政策, AA = 內部政策	(S)

當 TEX[2]為 1, 則外部快取與內部快取的快取政策如表 13-9 所顯示。

表 13-9 當 TEX 最高位元設定為 1 的內部與外部快取政策之編碼

記憶體屬性編碼 (AA 與 BB)	快取政策
00	不可快取的
01	回寫(Write back), 寫入及讀取配置
10	寫穿(Write through), 無寫入配置
11	回寫, 無寫入配置

更多快取行為與快取政策的資訊可參考 ARM Architecture Application Level Reference Manual (Ref 2)。

## 設定 MPU

MPU 暫存器可能看起來很複雜,但只要對你的應用所需要的記憶體區域有清楚的想法,其實並不困難。一般而言,你將需要下列記憶體區域:

- ◆ 特權程式的程式碼(例如, OS 核心與例外處理程式)
- ◆ 用戶程式的程式碼
- ◆ 位於程式區之內的特權程式之資料記憶體(資料+堆疊)
- ◆ 位於程式區之內的用戶程式之資料記憶體(資料+堆疊)
- ◆ 在其他的記憶體區域(例如, SRAM)作為特權的與用戶程式的資料記憶體
- ◆ 系統設備區域(通常僅是特權存取;例如, NVIC 與 MPU 暫存器)
- ◆ 其它的周邊

對 Cortex-M3 產品而言,大多數記憶體區域可設定為: TEX = b000, C = 1, B = 1。如 NVIC 之系統設備應該要有高度的順序,而週邊區域可編程為分享的設備 (TEX = b000, C = 0, B = 1)。然而,如果你希望確定於其區域發生的任意匯流排錯誤確實是匯流排錯誤,你應當使用高度順序(TEX = b000, C = 0, B = 0),讓寫入緩衝被除能。可是,這樣的作法會降低系統效能。

MPU 設定程式的簡單流程圖如圖 13-2 的方塊圖所示。

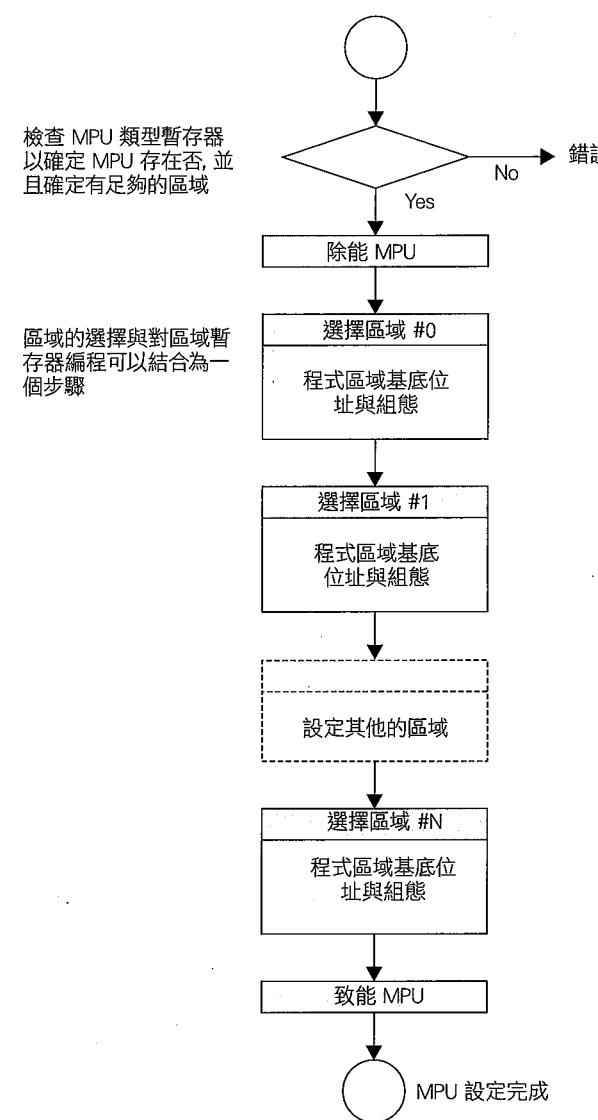


圖 13-2 設定 MPU 的簡單步驟

MPU 被致能之前與向量表若被重新配置到 RAM，請記得設定向量表裡記憶體管理錯誤的錯誤處理程式，並且致能系統處理程式控制與狀態暫存器裡的記憶體管理錯誤。這樣在 MPU 違例產生時，才會執行記憶體管理錯誤處理程式。

對於一個需要四個區域的簡單例子，一個簡單的 MPU 設定程式(不作區域檢查與致能)可能看來如下：

```

LDR R0, =0xE000ED98           ; 區域號碼暫存器
MOV R1, #0                      ; 選擇區域 0
STR R1, [R0]
LDR R1, =0x00000000             ; 基底位址 = 0x00000000
STR R1, [R0, #4]                ; MPU 區域基底位址暫存器
LDR R1, =0x0307002F             ; R/W, TEX=0, S=1, C=1, B=1, 16MB, Enable=1
STR R1, [R0, #8]                ; MPU 區域屬性與大小暫存器
MOV R1, #1                      ; 選擇區域 1
STR R1, [R0]
LDR R1, =0x08000000             ; 基底位址=0x08000000
STR R1, [R0, #4]                ; MPU 區域基底位址暫存器
LDR R1, =0x0307002B             ; R/W, TEX=0, S=1, C=1, B=1, 4MB, Enable=1
STR R1, [R0, #8]                ; MPU 區域屬性與大小暫存器
MOV R1, #2                      ; 選擇區域 2
STR R1, [R0]
LDR R1, =0x40000000             ; 基底位址=0x40000000
STR R1, [R0, #4]                ; MPU 區域基底位址暫存器
LDR R1, =0x03050039             ; R/W, TEX=0, S=1, C=0, B=1, 512MB, Enable=1
STR R1, [R0, #8]                ; MPU 區域屬性與大小暫存器
MOV R1, #3                      ; 選擇區域 3
STR R1, [R0]
LDR R1, =0xE0000000             ; 基底位址=0xE0000000
STR R1, [R0, #4]                ; MPU 區域基底位址暫存器
LDR R1, =0x03040027             ; R/W, TEX=0, S=1, C=0, B=0, 1MB, Enable=1
STR R1, [R0, #8]                ; MPU 區域屬性與大小暫存器
MOV R1, #1                      ; 致能 MPU
STR R1, [R0, #-4]               ; MPU 控制暫存器
(0xE000ED98-4=0xE000ED94)
    
```

此提供了四個區域：

- ◆ 特權程式區：0x00000000-0x00FFFFFF (16MB), 全存取, 可快取
- ◆ 特權資料區：0x080000000-0x0803FFFF (4MB), 全存取, 可快取
- ◆ 周邊區：0x40000000-0x5FFFFFFF (0.5GB), 全存取, 分享設備
- ◆ 系統控制區：0xE0000000-0xE00FFFFF (1MB), 特權存取, 高度順序的, XN

藉著結合區域選擇並且寫入基底位址暫存器，我們可將程式簡化如下：

```

LDR R0, =0xE000ED9C ; 區域基底位址暫存器
LDR R1, =0x00000010 ; 基底位址=0x00000000, 區域 0, 合法=1
STR R1, [R0, #0] ; MPU 區域基底位址暫存器
LDR R1, =0x0307002F ; R/W, TEX=0, S=1, C=1, B=1, 16MB, Enable=1
STR R1, [R0, #4] ; MPU 區域屬性與大小暫存器
LDR R1, =0x08000011 ; 基底位址=0x08000000, 區域 1, 合法=1
STR R1, [R0, #0] ; MPU 區域基底位址暫存器
LDR R1, =0x0307002B ; R/W, TEX=0, S=1, C=1, B=1, 4MB, Enable=1
STR R1, [R0, #4] ; MPU 區域屬性與大小暫存器
LDR R1, =0x40000012 ; 基底位址=0x40000000, 區域 2, 合法=1
STR R1, [R0, #0] ; MPU 區域基底位址暫存器
LDR R1, =0x03050039 ; R/W, TEX=0, S=1, C=0, B=1, 512MB, Enable=1
STR R1, [R0, #4] ; MPU 區域屬性與大小暫存器
LDR R1, =0xE0000013 ; 基底位址=0xE0000000, 區域 3, 合法=1
LDR R1, =0x03040027 ; R/W, TEX=0, S=1, C=0, B=0, 1MB, Enable=1
STR R1, [R0, #4] ; MPU 區域屬性與大小暫存器
MOV R1, #1 ; 致能 MPU
STR R1, [R0, #-8] ; MPU 控制暫存器
; (0xE000ED9C-8=0xE000ED94)

```

我們把程式簡化許多。然而，你可以進一步改善而更快速的設定程式；此可藉著使用 MPU aliased 暫存器位址來達成(見附錄 D 裡的表 D.33)。aliased 暫存器位址跟隨在 MPU 區域屬性與大小暫存器之後，並且 aliased 至 MPU 基底位址暫存器與 MPU 區域屬性與大小暫存器，共佔連續的 8 words 的位址，故可以使用多重的下載/儲存指令(LDM 與 STM)：

```

LDR R0, =0xE000ED9C ; 區域基底位址暫存器
LDR R1, =MPUconfig ; 預先定義的 MPU 設定變數表
LDMIA R1!, {R2, R3, R4, R5} ; 從表格讀取 4 words
STMIA R0!, {R2, R3, R4, R5} ; 寫入 4 words 至 MPU
LDMIA R1!, {R2, R3, R4, R5} ; 從表格讀取下面 4 words
STMIA R0!, {R2, R3, R4, R5} ; 寫入下面 4 words 至 MPU
B MPUconfigEnd
ALIGN 4 ; 為確保下面的表格為 word 對齊，故用此指令
; 因而底下我們可以使用多重載入指令
MPUconfig
  DCD 0x00000010 ; 基底位址=0x00000000, 區域 0, 合法=1
  DCD 0x0307002F ; R/W, TEX=0, S=1, C=1, B=1, 16MB, Enable=1
  DCD 0x08000011 ; 基底位址=0x08000000, 區域 1, 合法=1
  DCD 0x0307002B ; R/W, TEX=0, S=1, C=1, B=1, 4MB, Enable=1
  DCD 0x40000012 ; 基底位址=0x40000000, 區域 2, 合法=1
  DCD 0x03050039 ; R/W, TEX=0, S=1, C=0, B=1, 512MB, Enable=1
  DCD 0xE0000013 ; 基底位址=0xE0000000, 區域 3, 合法=1
  DCD 0x03040027 ; R/W, TEX=0, S=1, C=0, B=0, 1MB, Enable=1
MPUconfigEnd
  LDR R0, =0xE000ED94 ; MPU 控制暫存器

```

```

MOV R1, #1 ; 致能 MPU
STR R1, [R0]

```

當然，上述的寫法僅於預先知道所需資訊時方可使用，否則，應當使用更有組織的方法：其中之一為使用副程式(MpuRegionSetup)，其可根據一些輸入參數來設定區域，並可藉由多次呼叫以設定不同的區域：

```

MpuSetup ; 藉著呼叫設定區域的副程式以設定 MPU 的副程式
PUSH {R0-R6, LR}
LDR R0, =0xE000ED94 ; MPU 控制暫存器
MOV R1, #0
STR R1, [R0] ; 除能 MPU
; ---區域 #0---
LDR R0, =0x00000000 ; 區域 0：基底位址=0x00000000
MOV R1, #0x0 ; 區域 0：區域號碼=0
MOV R2, #0x17 ; 區域 0：大小=0x17 (16MB)
MOV R3, #0x3 ; 區域 0：AP=0x3 (全存取)
MOV R4, #0x7 ; 區域 0：MemAttrib=0x7
MOV R5, #0x0 ; 區域 0：Sub R 除能=0
MOV R6, #0x1 ; 區域 0：{XN, 致能}=0, 1
BL MpuRegionSetup
; ---區域 #1---
LDR R0, =0x08000000 ; 區域 1：基底位址=0x08000000
MOV R1, #0x1 ; 區域 1：區域號碼=1
MOV R2, #0x15 ; 區域 1：大小=0x15 (4MB)
MOV R3, #0x3 ; 區域 1：AP=0x3 (全存取)
MOV R4, #0x7 ; 區域 1：MemAttrib=0x7
MOV R5, #0x0 ; 區域 1：Sub R 除能=0
MOV R6, #0x1 ; 區域 1：{XN, 致能}=0, 1
BL MpuRegionSetup
; ... ; 區域#2 與#3 的設定
; ---區域 #4-#7 的除能---
MOV R0, #4
BL MpuRegionDisable
MOV R0, #5
BL MpuRegionDisable
MOV R0, #6
BL MpuRegionDisable
MOV R0, #7
BL MpuRegionDisable
LDR R0, =0xE000ED94 ; MPU 控制暫存器
MOV R1, #1
STR R1, [R0] ; 致能 MPU
POP {R0-R6, PC} ; 返回
MpuRegionSetup ; MPU 區域設定副程式

```

接下頁

```

; 輸入 R0 : 基底位址
; R1 : 區域號碼
; R2 : 大小
; R3 : AP(存取許可)
; R4 : MemAttrib ({TEX[2:0], S, C, B})
; R5 : 次區域除能
; R6 : {XN, 致能}
PUSH {R0-R1, LR}
BIC R0, R0, #0x1F
BFI R0, R1, #0, #4
ORR R0, R0, #0x10
LDR R1, =0xE000ED9C
STR R0, [R1]

; 清除位址裡面未使用之位元
; 插入區域號碼至 R0[3:0]
; 設定合法位元
; MPU 區域基底位址暫存器
; 設定基底位址暫存器

AND R0, R6, #0x01
UBFX R1, R6, #1, #1
BFI R0, R1, #28, #1
BFI R0, R2, #1, #5
BFI R0, R3, #24, #3
BFI R0, R4, #16, #6
BFI R0, R5, #8, #8
LDR R1, =0xE000EDA0
STR R0, [R1]
POP {R0-R1, PC}

; 取得致能位元
; 取得 XN 位元
; 插入 XN 至 R0[28]
; 插入區域大小欄位(R2[4:0])至 R0[5:1]
; 插入 AP 欄位(R3[2:0])至 R0[26:24]
; 插入 memattrib 欄位(R4[5:0])至 R0[21:16]
; 插入次區域除能(SRD)欄位至 R0[15:8]
; MPU 區域基底屬性與大小暫存器
; 設定基底屬性與大小暫存器
; 返回

MpuRegionDisable
; 除能未使用區域的副程式
; 輸入 R0 : 區域號碼
PUSH {R1, LR}
AND R0, R0, #0xF
ORR R0, R0, #0x10
LDR R1, =0xE000ED9C
STR R0, [R1]
MOV R0, #0
LDR R1, =0xE000EDA0
STR R0, [R1]
POP {R1, PC}
; 清除區域號碼裡面未使用之位元
; 設定合法位元
; MPU 區域基底位址暫存器
; MPU 區域基底屬性與大小暫存器
; 設定基底屬性與大小暫存器為 0
; (被除能)
; 返回

```

於此例子中，包括了一個用來除能未使用區域的一個副程式。如果你不確定一個區域是否先前已被設定，則需要此副程式；如果一個未使用的區域先前被設定為致能，則需要先將它除能，才不致影響新的組態設定。

此外，此例子顯示在 Cortex-M3 裡位元欄位插入(Bit Field Insert, BFI)指令的應用，可大為簡化位元欄位合併運算。

## 典型設定

在典型的應用中，當需要避免用戶程式去存取特權程序資料與程式區域時，則會使用 MPU。在開發 MPU 的設定副程式的時候，你需要考慮下列一些區域：

### 1. 程式區：

- ◆ 特權程式，包括起始向量表 ◆ 用戶程式

### 2. SRAM 區：

- ◆ 特權資料，包括主要堆疊 ◆ 用戶資料，包括程序堆疊
- ◆ 特權 bit-band alias 區 ◆ 用戶 bit-band alias 區

### 3. 周邊

- ◆ 特權周邊 ◆ 用戶周邊
- ◆ 特權周邊 bit-band alias 區 ◆ 用戶周邊 bit-band alias 區

### 4. 系統控制空間(NVIC 與除錯元件)：

- ◆ 僅可特權存取

由此列表，我們識別了 11 個區域，多於 Cortex-M3 MPU 支援的 8 個區域。然而，我們可使用背景區域(PRIVDEFENA 設定為 1)來定義特權區域，故僅需設定 5 個用戶區域，因而尚有 3 個未使用的 MPU 區域。這些未使用到的區域可能仍會用來設定外部記憶體中的額外區域，以保護唯讀資料，或者在必要時，全然阻絕了某部分的記憶體。

## 使用次區域除能的範例

在某些情形下可能會有一些周邊可為用戶程式存取，另一些周邊卻需要保護以作特權存取，這樣造成了用戶存取周邊記憶體空間的碎裂(fragmentation)。這時，可採用下述對策：

- ◆ 定義多重的用戶區域 ◆ 於用戶周邊區域內定義特權區域
- ◆ 在用戶區域內使用次區域除能

# Jason 嘴書—EETOP 世界唯一貼

前兩個方法很容易就會消耗掉可用的區域，而藉著第三種對策，即使用次區域除能的特性，不需假借額外的區域，就能輕易地對個別的周邊區塊設定存取許可。例如圖 13-3 所示：

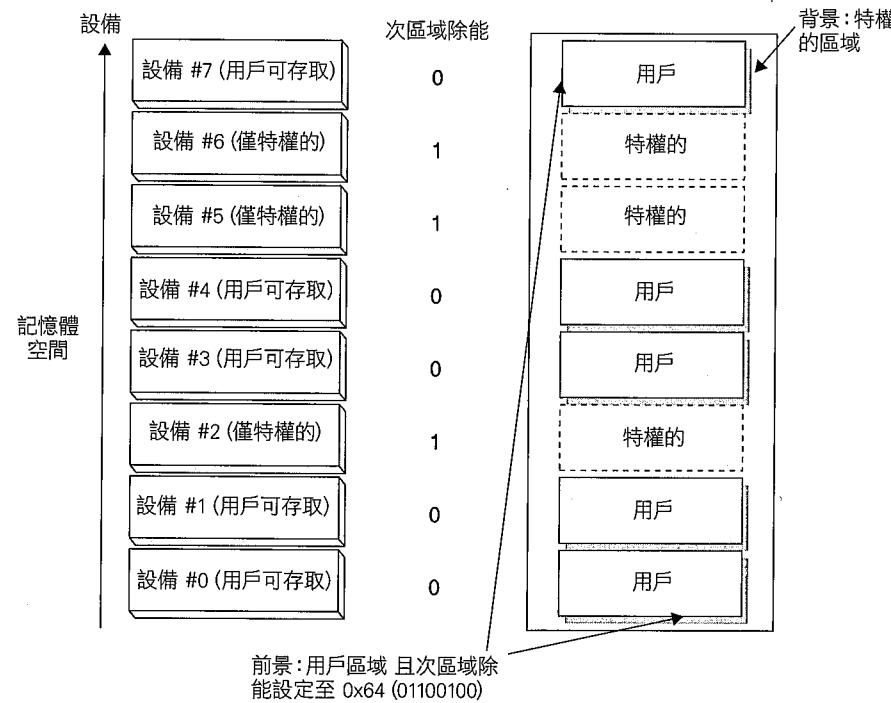


圖 13-3 使用次區域除能來控制個別周邊的存取權限

相同的技術也可應用在記憶體區域，然而，周邊比較可能會有片段的特權設定。

讓我們假設將會使用表 13-10 所列的記憶體區域。在定義了需要的區域後，我們可以產生 MPU 設定程式，為了使得程式易於了解與修改，我們利用了先前製造的函數，以開發完整的 MPU 設定的程式範例：

```

MpuSetup          ; 藉著呼叫設定區域的副程式以設定 MPU 的副程式
PUSH {R0-R6, LR}
LDR R0, =0xE000ED94 ; MPU 控制暫存器
MOV R1, #0
STR R1, [R0]        ; 除能 MPU
; ---區域 #0--- 用戶程式
LDR R0, =0x00004000 ; 區域 0 : 基底位址=0x00004000
MOV R1, #0x0          ; 區域 0 : 區域號碼=0

```

```

MOV R2, #0x0D          ; 區域 0 : 大小=0x0D (16KB)
MOV R3, #0x3            ; 區域 0 : AP=0x3 (全存取)
MOV R4, #02              ; 區域 0 : MemAttrib=0x2 (TEX=0, S=0, C=1, B=0)
MOV R5, #0x00            ; 區域 0 : Sub R 除能=0
MOV R6, #0x1             ; 區域 0 : {XN, 致能}=0, 1
BL MpuRegionSetup

; ---區域 #1--- 用戶資料
LDR R0, =0x20000000    ; 區域 1 : 基底位址=0x20000000
MOV R1, #0x1            ; 區域 1 : 區域號碼=1
MOV R2, #0x0B           ; 區域 1 : 大小=0x0B (4KB)
MOV R3, #0x3            ; 區域 1 : AP=0x3 (全存取)
MOV R4, #0xB             ; 區域 1 : MemAttrib=0xB (TEX=1, S=0, C=1, B=1)
MOV R5, #0x00            ; 區域 1 : Sub R 除能=0
MOV R6, #0x1             ; 區域 1 : {XN, 致能}=0, 1
BL MpuRegionSetup

; ---區域 #2--- 用戶 bit band
LDR R0, =0x22000000    ; 區域 2 : 基底位址=0x22000000
MOV R1, #0x2            ; 區域 2 : 區域號碼=2
MOV R2, #0x10           ; 區域 2 : 大小=0x10 (128KB)
MOV R3, #0x3            ; 區域 2 : AP=0x3 (全存取)
MOV R4, #0xB             ; 區域 2 : MemAttrib=0xB (TEX=1, S=0, C=1, B=1)
MOV R5, #0x00            ; 區域 2 : Sub R 除能=0
MOV R6, #0x1             ; 區域 2 : {XN, 致能}=0, 1
BL MpuRegionSetup

; ---區域 #3--- 用戶周邊
LDR R0, =0x40000000    ; 區域 3 : 基底位址=0x40000000
MOV R1, #0x3            ; 區域 3 : 區域號碼=3
MOV R2, #0x13           ; 區域 3 : 大小=0x13 (1MB)
MOV R3, #0x3            ; 區域 3 : AP=0x3 (全存取)
MOV R4, #0x1             ; 區域 3 : MemAttrib=0x1 (TEX=0, S=0, C=0, B=1)
MOV R5, #0x9B            ; 區域 3 : Sub R 除能=0x9B (依據先前例子)
MOV R6, #0x3             ; 區域 3 : {XN, 致能}=1, 1
BL MpuRegionSetup

; ---區域 #4--- 用戶周邊 bit band
LDR R0, =0x42000000    ; 區域 4 : 基底位址=0x42000000
MOV R1, #0x4            ; 區域 4 : 區域號碼=4
MOV R2, #0x18           ; 區域 4 : 大小=0x18 (32MB)
MOV R3, #0x3            ; 區域 4 : AP=0x3 (全存取)
MOV R4, #0x1             ; 區域 4 : MemAttrib=0x1 (TEX=0, S=0, C=0, B=1)
MOV R5, #0x9B            ; 區域 4 : Sub R 除能=0x9B (依據先前例子)
MOV R6, #0x3             ; 區域 4 : {XN, 致能}=1, 1
BL MpuRegionSetup

; ---區域 #5--- 外部 RAM
LDR R0, =0x60000000    ; 區域 5 : 基底位址=0x60000000
MOV R1, #0x5            ; 區域 5 : 區域號碼=5
MOV R2, #0x17           ; 區域 5 : 大小=0x17 (16MB)
MOV R3, #0x3            ; 區域 5 : AP=0x3 (全存取)
MOV R4, #0xB             ; 區域 5 : MemAttrib=0xB (TEX=1, S=0, C=1, B=1)
MOV R5, #0x0             ; 區域 5 : Sub R 除能=0

```

接下頁

# 其它 Cortex-M3 的特性

本章內容包括：

- ✓ SYSTICK 計時器
- ✓ 電源管理
- ✓ 多個處理器之間的通信
- ✓ 自我重置的控制

表 13-10 MPU 設定範例程式的記憶體區域安排

位址	描述	大小	類型	記憶體屬性 (C, B, A, XN)	MPU 區域
0x00000000-0x00003FFF	特權的程式	16k	唯讀	C, -, A, -, -	背景
0x00004000-0x00007FFF	用戶程式	16k	唯讀	C, -, A, -, -	區域 #0
0x20000000-0x20000FFF	用戶資料	4k	全存取	C, B, A, -, -	區域 #1
0x20001000-0x2001FFFF	特權的資料	4k	特權的存取	C, B, A, -, -	背景
0x22000000-0x2201FFFF	用 戶 資 料 bit-band alias	128k	全存取	C, B, A, -, -	區域 #2
0x22020000-0x2203FFFF	特權的資料 bit-band alias	128k	全存取	C, B, A, -, -	背景
0x40000000-0x400FFFFF	用 戶 周 邊	1M	全存取	-, B, -, -, XN	區域 #3
0x40040000-0x4005FFFF	在用 戶 周 邊 區 域 內 的 特 權 的 周 邊	128k	特權的存取	-, B, -, -, N	區域 #3 內被 除能的次區域
0x42000000-0x43FFFFFF	用 戶 周 邊 bit-band alias	32M	全存取	-, B, -, -, XN	區域 #4
0x42800000-0x42BFFFFFF	在用 戶 區 域 內 的 特 權 的 周 邊 bit- band alias	4M	特權的存取	-, B, -, -, XN	區域 #4 內被 除能的次區域
0x60000000-0x60FFFFFF	外部的 RAM	16M	全存取	C, B, A, -, -	區域 #5
0xE0000000-0xF00FFFFF	NVIC, 除 錯, 與私有 周邊匯流排	1M	特權的存取	-, -, -, -, XN	背景

## SYSTICK 計時器

第 8 章簡單提及了 NVIC 中 SYSTICK 單存器, 如我們看到的, SYSTICK 計時器為 24-bit 倒數計時器。一旦它倒數至 0, 計數器由重載入單存器下載重載值; 並且直到 SYSTICK 控制與狀態單存器裡的致能位元被清除為止, 否則不會停止計數。

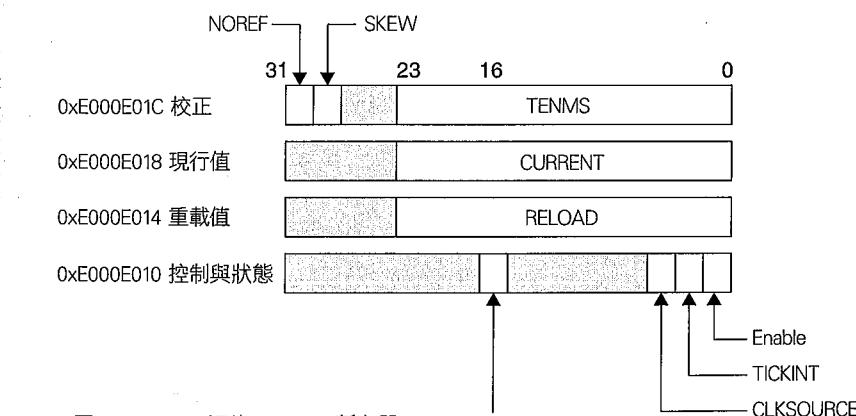


圖 14-1 NVIC 裡的 SYSTICK 單存器

Cortex-M3 處理器允許 SYSTICK 計數器擁有兩個不同的時脈來源。第一個為核心自動時脈(並非來自系統時脈 HCLK, 所以即使系統時脈停止, 它也不會停止)。第二個為外部參考時脈, 此時脈信號需至少比自動時脈緩慢兩倍, 因為此信號被自動時脈取樣。由於晶片設計師可能會決定在其設計裡省略此外部參考時脈, 所以可能並不存在。欲決定是否有外部時脈來源可用, 你應該檢查 SYSTICK 校正暫存器的 bit[31]。晶片設計師應該根據設計將此接角連到適當的值。

當 SYSTICK 計時器由 1 變更至 0 時, 它會設定 SYSTICK 控制與狀態暫存器裡的 COUNTFLAG 位元。COUNTFLAG 會被下列動作所清除：

- ◆ 處理器讀取 SYSTICK 控制與狀態暫存器
- ◆ 藉著寫入任意數值至 SYSTICK 現行值暫存器, 以清除 SYSTICK 計數器之值

SYSTICK 計數器可用來產生規律時間間隔的 SYSTICK 例外, 這對於 OS 經常是需要的, 因其可作為工作與資源的管理。欲致能 SYSTICK 例外產生, 則需設定 TICKINT 位元。此外, 如果向量表已經重新配置到 SRAM, 則可能需要設定向量表裡的 SYSTICK 例外處理程式：

```
; 設定 SYSTICK 例外處理程式
MOV R0, #0xF          ; 例外類別 15
LDR R1, =sysTick_handler ; 例外處理程式的位址
LDR R2, =0xE000ED08 ; 向量表位移暫存器
LDR R2, [R2]
STR R1, [R2, R0, LSL #2] ; 寫向量至 VectTblOffset+ExcpType*4
```

一個設定 SYSTICK 的簡單程式可能如下：

```
; 啟動 SYSTICK 計時器運算並啟動 SYSTICK 中斷
LDR R0, =0xE000E010 ; SYSTICK 控制與狀態暫存器
MOV R1, #0
STR R1, [R0]          ; 停止計數器以避免意外觸發中斷
LDR R1, =0x3FF        ; 每 1024 週期觸發一次(因為計數器由
; 1023 倒數至 0, 總共 1024 週期,
; 故使用數值 0x3FF)。
STR R1, [R0, #4]      ; 寫重載值至重載暫存器位址
STR R1, [R0, #8]      ; 寫任意數值至現行值暫存器以清除
; 現行值至 0 並清除 COUNTFLAG
MOV R1, #0x7          ; 時脈來源 = 核心時脈, 啟動中斷
; 啟動 SYSTICK 計數器
STR R1, [R0]          ; 開始計時器
```

SYSTICK 計數器提供了一個簡單的方式, 允許存取定時校正資訊。Cortex-M3 處理器的頂層有一個 24-bit 輸入, 晶片設計師可以輸入一個重載值至其內, 以產生 10 ms 的時間間隔；而此數值可被 SYSTICK 校正暫存器存取。然而, 這並不是必然會有的選項, 你需要檢查設備的資料列表以確定是否能使用此特性。

SYSTICK 計數器亦可以當作警告計時器使用, 在一定數量的時脈週期後, 開始某項工作。例如, 若某工作需於 300 時脈週期後開始執行, 則我們可以在 SYSTICK 例外處理程式裡設定此工作, 並設定好 SYSTICK 計時器, 故此工作將會在 300 個時脈週期到達時被執行：

```
LDR r0, =15           ; 設定 SYSTICK 處理程式
LDR r1, =SysTickAlarm ; SYSTICK 例外處理程式名稱
BL SetupExcpHandler

LDR R0, =0xE000E010   ; SYSTICK 基底
MOV R1, #0             ; 於程式中, 除能 SYSTICK
STR R1, [R0]
STR R1, [R0, #0x8]     ; 清除現行值
LDR R1, =(300-12)      ; 設定重載值：考慮例外延遲故減 12
STR R1, [R0, #0x4]

LDR R4, =SysTickFired ; RAM 裡的資料變數
MOV R5, #0             ; 設定軟體旗標至 0
STR R5, [R4]
MOV R1, #0x7            ; 使用內部時脈, 啟動 SYSTICK 例外
STR R1, [R0]            ; 開始計數
LDR R4, =SysTickFired

WaitLoop
LDR R5, [R4]          ; 等待直到軟體旗標
; 被 SYSTICK 處理程式設定
CMP R5, #0
BEQ WaitLoop
...
; SysTickFired 被設定,
; 主程式進行其它工作
```

上述範例程式使用稱作 SetupExcpHandler 的副程式設定 SYSTICK 向量, 此僅可於向量表為可寫入的情況下使用(例如, 重新配置至 SRAM)：

```

SetupExcpHandler          ; 設定例外向量之副程式
; 輸入R0：例外號碼
; R1：例外向量
PUSH   {R2, LR}
LDR    R2, =0xE000ED08      ; 向量表位移
LDR    R2, [R2]
STR    R1, [R2, R0, LSL #2] ; 位址=向量表位移+4x 例外號碼
POP    {R2, PC}

```

因為計數器從主程式被手動清除，故其起始值為 0，接著它立即重載至 288 (即 300-12)。減去 12 是因為此數值為最小的例外延遲之時脈週期。然而，如果當 SYSTICK 計數器到達 0 時有相同或更高優先權的其它例外正在執行，例外將可能會延後開始。

注意，此例子中把重載值減去 12 個週期數，僅適用於一次使用的警告計時器的用途；而在週期性計數的用途裡，重載值則需是每周期的時脈數減 1。

因為 SYSTICK 計數器並不會自動停止，我們需要在 SYSTICK 處理程式內將其停止。再者，如果 SYSTICK 例外為其它例外的處理而延遲，它可能會再度被置於等待。所以若 SYSTICK 例外是僅會執行一次，就必須執行一些步驟：

```

SysTickAlarm           ; SYSTICK 例外處理程式
PUSH   {LR}
LDR    R0, =0xE000E010      ; SYSTICK 基底
MOV    R1, #0
STR    R1, [R0]              ; 除能後來的 SYSTICK 例外
LDR    R0, =0xE000ED04
LDR    R1, =0x02000000      ; 如果它再次被置於等待
                           ; 就清除 SYSTICK 等待位元
STR    R1, [R0]
...
LDR    R2, =SysTickFired    ; 設定軟體旗標，以使得主程式知道
                           ; 工作已經被執行
LDR    R1, [R2]
ADD    R1, #1
STR    R1, [R2]
POP    {PC}                  ; 例外返回

```

在 SYSTICK 例外處理程式的最後步驟，我們設定了稱作 SysTickFired 的軟體變數，以使得主程式知道所需工作已經執行完畢。

## 電源管理

Cortex-M3 提供了睡眠模式以作電源管理特性。在睡眠模式期間，系統時脈可以被停止，但是自動時脈的輸入必須持續執行，故允許中斷去喚醒處理器。兩個睡眠模式為：

睡眠：由 Cortex-M3 處理器的 SLEEPING 信號來指定

深度睡眠：由 Cortex-M3 處理器的 SLEEPDEEP 信號來指定

NVIC 系統控制暫存器有一個稱作 SLEEPDEEP 的位元欄位(參見表 14-1)，用來決定使用了哪個睡眠模式。SLEEPING 與 SLEEPDEEP 的動作隨著特定的 MCU 實作而定；在某些實作裡，兩種模式的動作相同。

表 14-1 系統控制暫存器(0xE000ED10)

位元	名稱	類型	重置值	描述
4	SEVONPEND	R/W	0	等待期間送出事件；如果新的中斷被置於等待則會從 WFE 喚醒，而不管此中斷擁有的優先權是否高於現行的等級。
3	保留的	-	-	-
2	SLEEPDEEP	R/W	0	當進入睡眠模式時致能 SLEEPDEEP 輸出信號
1	SLEEPONEXIT	R/W	0	致能 SleeponExit 特性
0	保留的	-	-	-

WFI 或 WFE 指令會引發睡眠模式；WFI 意為 Wait-For-Interrupt(等待中斷)而 WFE 意為 Wait-For-Events(等待事件)。事件可能是：中斷、先前觸發的中斷、或經由 RXEV 信號而來的外部事件信號脈波。在處理器之內有一個事件的門鎖，故過去的事件可從 WFE 來喚醒處理器。

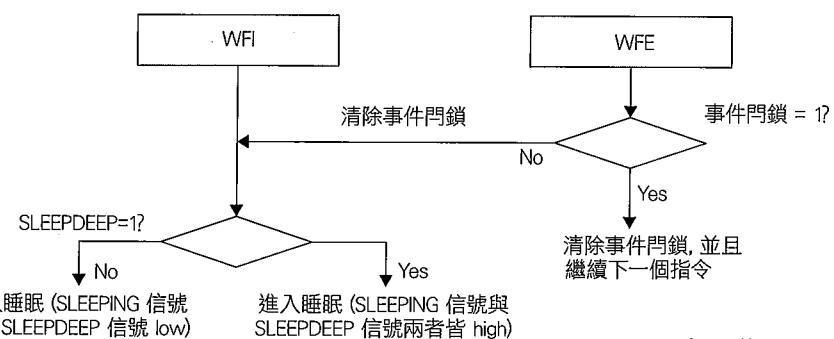


圖 14-2 睡眠運算

當處理器進入睡眠模式時真正發生的動作隨晶片設計而定。通常的情形是一些時脈信號可以被停止以減少功率消耗。然而，也可以設計晶片去關掉部分晶片以更進一步減少功率消耗，或者甚至可能把整個晶片關掉並停止所有時脈信號。當晶片被完整地關掉時，僅可藉由系統重置以從睡眠中喚醒系統。

要從 WFI 睡眠中喚醒處理器，中斷優先權等級需高於現行優先權等級(如果其為執行中的中斷)，並且高於 BASEPRI 或遮罩暫存器(PRIMASK 或 FAULTMASK)設定的等級。如果因為優先權等級的緣故而使得中斷不被接受，將不會喚醒由 WFI 造成的睡眠。

而 WFE 的情況稍微不同；如果睡眠中觸發的中斷，其優先權低於或等於遮罩暫存器或 BASEPRI 暫存器，並且若設定了 SEVONPEND，則將會從睡眠中喚醒處理器。由睡眠中喚醒 Cortex-M3 處理器的規則歸納在表 14-2 裡。

表 14-2 WFI 與 WFE 喚醒的行為

WFI 行為	喚醒	IRQ 的執行
IRQ with BASEPRI		
IRQ 優先權 > BASEPRI	Y	Y
IRQ 優先權 <= BASEPRI	N	N
IRQ with BASEPRI 與 PRIMASK		
IRQ 優先權 > BASEPRI	Y	N
IRQ 優先權 <= BASEPRI	N	N
WFE 行為	喚醒	IRQ 的執行
IRQ with BASEPRI, SEVONPEND = 0		
IRQ 優先權 > BASEPRI	Y	Y
IRQ 優先權 <= BASEPRI	N	N
IRQ with BASEPRI, SEVONPEND = 1		
IRQ 優先權 > BASEPRI	Y	Y
IRQ 優先權 <= BASEPRI	Y	N
IRQ with BASEPRI, PRIMASK, 且 SEVONPEND = 0		
IRQ 優先權 > BASEPRI	N	N
IRQ 優先權 <= BASEPRI	N	N
IRQ with BASEPRI, PRIMASK, 且 SEVONPEND = 1		
IRQ 優先權 > BASEPRI	Y	N
IRQ 優先權 <= BASEPRI	Y	N

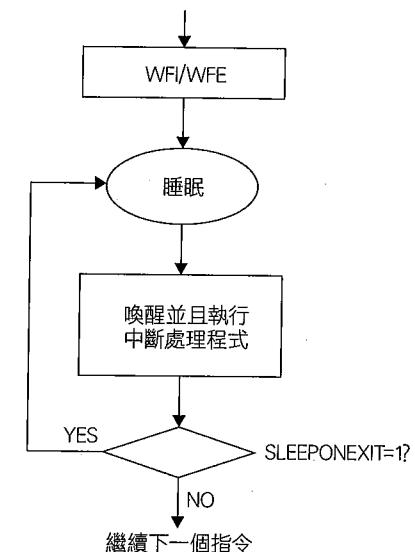


圖 14-3 使用 SleepOnExit 特性的範例

睡眠模式的另一個特性是當離開中斷程式時，可設定主動回復至睡眠；這樣我們可以使核心除了需要作中斷服務之外，經常處於睡眠。欲使用此特性，我們需要設定系統控制暫存器的 SLEEPONEXIT 位元。

請注意在啟能了 SLEEPONEXIT 特性時，處理器在離開例外後可以在未執行 WFI/WFE 的情形下進入睡眠模式。因此，SLEEPONEXIT 特性通常會在期望進行睡眠運算處的 WFI/WFE 指令之前被啟能。

在 Cortex-M3 revision 2 (released in 2008)，已經加上了額外的低功率特性。從軟體的觀點來看，WFI 與 WFE 依然與之前相同，但是在 revision 2 裡深度睡眠允許時脈信號進入欲停止的處理器核心，而以一個叫做喚醒-中斷-控制器的分開的單元去掌管處理器核心的喚醒。在此安排下，處理器核心可以被置於功率降低模式，並且將處理器狀態資訊儲存在特殊的邏輯晶格上，達到閒置功率最小化的設計。

新的功率降低能力需要一個外部的電源管理單元以控制功率昇降序列。此單元由矽供應商開發，並且可能需要在使用功率降低特性之前先做編程。更多的資訊請參考矽供應商的文件。有兩點需要留意：功率降低特性在深度睡眠期間停止了 SYSTICK 計時器；以及當連接了除錯器時，功率降低特性會被除能(這是因為除錯器需要存取除錯暫存器)。

## 多個處理器之間的通信

Cortex-M3 擁有一個簡單的多重處理器通信介面以同步工作。處理器有一輸出信號，稱作 TXEV(Transmit Event, 傳輸事件) 以送出事件；並且有一輸入信號，稱作 RXEV(Receive Event, 接收事件) 以接收事件。若一個系統有兩個處理器，則可以依圖 14-4 所示的事件通信信號連接來實作。

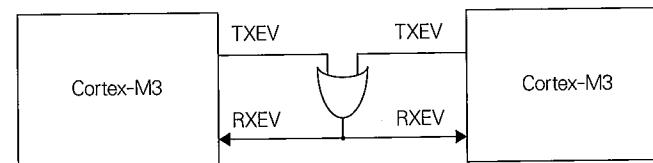


圖 14-4 擁有兩個處理器系統的事件通信連接

如前節電源管理所提到的，執行了 WFE 指令後，處理器可進入睡眠；並在接收了外部事件後，繼續指令的執行。若我們使用了稱作 SEV(Send Event 送出事件)的指令，其一處理器可喚醒另一睡眠中的處理器，並且確定兩個處理器同時間開始執行工作。

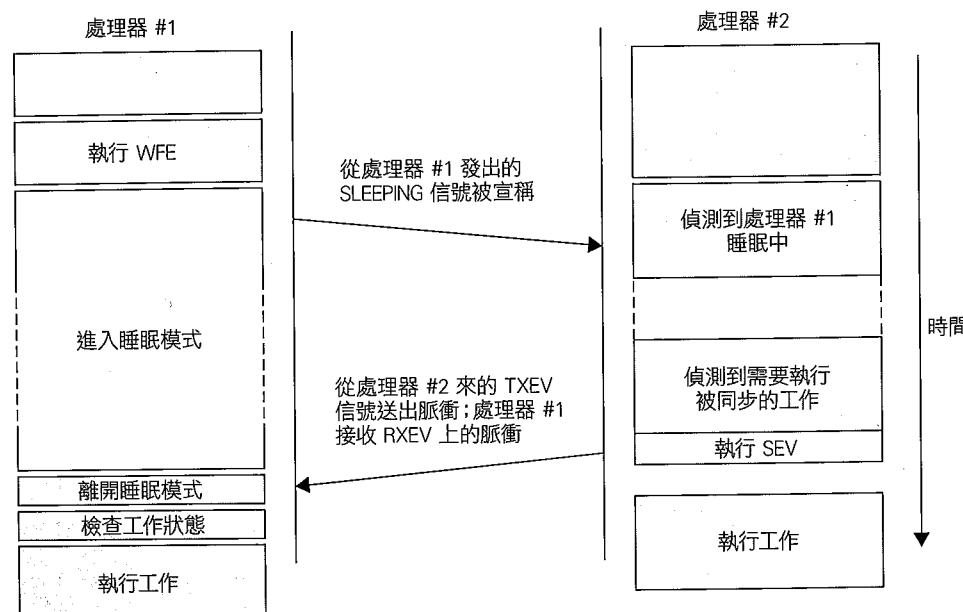


圖 14-5 使用事件信號以同步工作

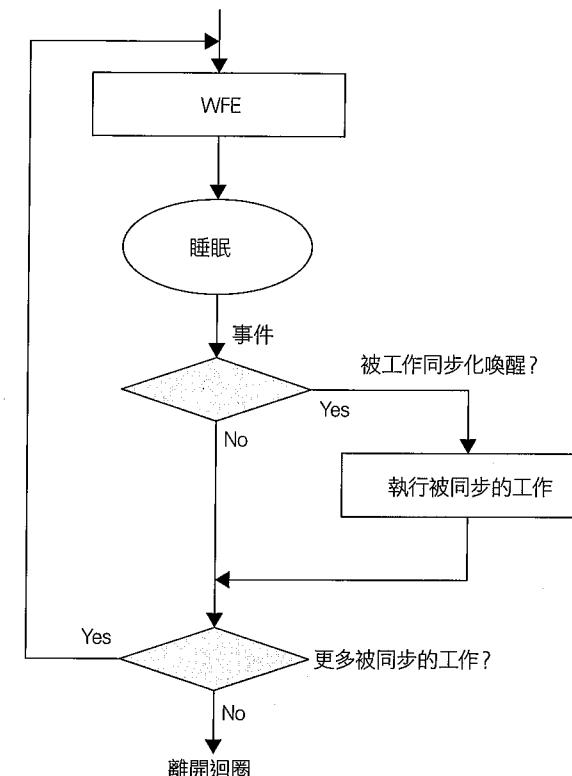


圖 14-6 使用 WFE 特性的範例

使用此特性，我們可使得兩個處理器於同時間開始執行一個工作(會隨著實際的晶片實作而可能有一些時脈週期的誤差)。可喚醒的處理器數目並沒有限制，唯一的要求是，其中必須有一個處理器作為 master，以對其它處理器產生事件脈波。

當執行 WFE 指令時，會先檢查區域事件門鎖，若門鎖未被設定，核心將進入睡眠模式；若門鎖被設定，它將被清除並繼續執行指令而不會進入睡眠模式。區域事件門鎖可被先前發生的例外或 SEV 指令設定。因此，如果你執行了一個 SEV 並接著執行一個 WFE，處理器將不會進入睡眠，僅只是繼續下一個指令並以 WFE 清除事件門鎖。

只有在簡單的情境下，WFE 本身才可以用來作工作同步。在複雜的應用中，需要額外的程式以正確的處理同步。這是因為處理器也可能被其它的事件喚醒，例如中斷與除錯事件；而且因為內部事件暫存器的現行狀態可能未知，故執行 WFE 並不一定保證進入睡眠。因此，WFE 通常會跟迴圈(以減低系統功率消耗)與狀態檢查碼同時使用，以檢查是否在 WFE 之後，執行了所需要的同步工作。

此用法的一個例子是在多重處理器系統裡的號誌。在典型的情境下，會使用系統層級獨佔存取監視器與獨佔存取指令來作迴旋栓鎖(spin lock)，以便存取共享的記憶體或周邊。當一個程序需要資源時，需要呼叫一段組合語言程式碼以得到“栓鎖”：

```

get_lock           ; 取得栓鎖的組合語言函數
    LDR    r0, =Lock_Variable
    MOVS   r2, #1
get_lock_loop
    LDREX  r1, [r0]
    CMP    r1, #0
    BNE    get_lock_loop
    STREX  r1, r2, [r0]
    CMP    r1, #0
    BNE    get_lock_loop
    DMB
    BX     LR
; 用來鎖住 STREX
; 它被鎖住了，再次重試
; 使用 STREX 以試著設定 Lock_Variable 為 1
; 檢查 STREX 的返回狀態
; STREX 並未成功，重試
; 資料記憶體障礙 DMB
; 返回

```

另一方面，使用資源的程序在它不再需要時應該釋放資源：

```

free_lock ; 釋放栓鎖的組合語言函數
    LDR    r0, =Lock_Variable
    MOVS   r1, #0
    DMB
    STR    r1, [r0]
    BX     LR
; 資料記憶體障礙
; 清除栓鎖
; 返回

```

當處理器閒置時，迴旋栓鎖可能會造成不必要的功率消耗。因此，我們加入 WFE 到這些運算以減低功率消耗，並且同時允許處理器等待栓鎖在資源被釋放後，儘快被喚醒。

```

get_lock_with_WFE      ; 取得栓鎖的組合語言函數
    LDR    r0, =Lock_Variable
    MOVS   r2, #1
get_lock_loop
    LDREX  r1, [r0]
    CBNZ   r1, lock_is_set
    STREX  r1, r2, [r0]
    CMP    r1, #0
    BNE    get_lock_loop
    DMB
    BX     LR
lock_is_set
    WFE    ; 等待事件
    B      get_lock_loop
; 用來鎖住 STREX
; 如果栓鎖被設定，睡眠並且以後再試
; 使用 STREX 以試著設定 Lock_Variable 為 1
; 檢查 STREX 的返回狀態
; STREX 並未成功，重試
; 資料記憶體障礙
; 返回
; 醒來，再次重試

```

為了讓函數釋放栓鎖，使用 SEV 指令以喚醒其它的正等待栓鎖的處理器。

```

free_lock_with_SEV      ; 釋放栓鎖的組合語言函數
    LDR    r0, =Lock_Variable
    MOVS   r1, #0
    DMB
    STR    r1, [r0]
    SEV
    BX     LR
; 資料記憶體障礙
; 清除栓鎖
; 送出事件以喚醒其他的處理器
; 返回

```

藉著結合事件通信界面與必要的號誌程式，在迴旋栓鎖期間的功率消耗可以被降低。類似的技巧可以用在訊息傳遞與工作同步上。

在大部分基於 Cortex-M3 的產品裡，都僅有單一個處理器，因此 RXEV 輸入可能會固定為 0，或者連結到產生事件的周邊上。

## 自我重置的控制

Cortex-M3 提供了兩個自我重置控制的特性。第一個為 NVIC 應用中斷與重置控制( Application Interrupt and Reset Control)暫存器(bit [0])的 VECTRESET 控制位元：

```

LDR    R0, =0xE000ED0C      ; NVIC AIRCR 位址
LDR    R1, =0x05FA0001      ; 設定 VECTRESET 位元
                                ; (05FA 為寫入存取鑰匙)
STR    R1, [R0]
deadloop
B     deadloop
; deadloop 用來確保並無其它指令
; 跟隨在重置執行之後

```

寫進此位元將會重置 Cortex-M3 處理器，但不包括除錯邏輯。此寫入動作不會重置 Cortex-M3 處理器之外的電路。例如，若單晶片包含一個 UART，寫進此位元並不會重置 UART 或位於 Cortex-M3 之外的任意周邊。

第二個重置特性為相同的 NVIC 暫存器中的 SYSRESETREQ 位元。它允許了 Cortex-M3 處理器對系統重置產生器宣稱一個重置要求信號。因為系統重置產生器並非 Cortex-M3 設計的一部分，此重置特性的實作會隨晶片設計而定。因此某些晶片可能沒有提供此功能，請小心謹慎地檢查晶片規格。

此用法的一個例子是在多重處理器系統裡的號誌。在典型的情境下，會使用系統層級獨佔存取監視器與獨佔存取指令來作迴旋栓鎖(spin lock)，以便存取共享的記憶體或周邊。當一個程序需要資源時，需要呼叫一段組合語言程式碼以得到“栓鎖”：

```
get_lock          ; 取得栓鎖的組合語言函數
    LDR    r0, =Lock_Variable
    MOVS   r2, #1
get_lock_loop
    LDREX  r1, [r0]
    CMP    r1, #0
    BNE    get_lock_loop
    STREX  r1, r2, [r0]
    CMP    r1, #0
    BNE    get_lock_loop
    DMB
    BX     LR
                                ; 用來鎖住 STREX
                                ; 它被鎖住了，再次重試
                                ; 使用 STREX 以試著設定 Lock_Variable 為 1
                                ; 檢查 STREX 的返回狀態
                                ; STREX 並未成功，重試
                                ; 資料記憶體障礙 DMB
                                ; 返回
```

另一方面，使用資源的程序在它不再需要時應該釋放資源：

```
free_lock        ; 釋放栓鎖的組合語言函數
    LDR    r0, =Lock_Variable
    MOVS   r1, #0
    DMB
    STR    r1, [r0]
    BX     LR
                                ; 資料記憶體障礙
                                ; 清除栓鎖
                                ; 返回
```

當處理器閒置時，迴旋栓鎖可能會造成不必要的功率消耗。因此，我們加入 WFE 到這些運算以減低功率消耗，並且同時允許處理器等待栓鎖在資源被釋放後，儘快被喚醒。

```
get_lock_with_WFE      ; 取得栓鎖的組合語言函數
    LDR    r0, =Lock_Variable
    MOVS   r2, #1
get_lock_loop
    LDREX  r1, [r0]
    CBNZ   r1, lock_is_set
    STREX  r1, r2, [r0]
    CMP    r1, #0
    BNE    get_lock_loop
    DMB
    BX     LR
lock_is_set
    WFE    ; 等待事件
    B      get_lock_loop
                                ; 用來鎖住 STREX
                                ; 如果栓鎖被設定，睡眠並且以後再試
                                ; 使用 STREX 以試著設定 Lock_Variable 為 1
                                ; 檢查 STREX 的返回狀態
                                ; STREX 並未成功，重試
                                ; 資料記憶體障礙
                                ; 返回
                                ; 醒來，再次重試
```

為了讓函數釋放栓鎖，使用 SEV 指令以喚醒其它的正等待栓鎖的處理器。

free_lock_with_SEV	; 釋放栓鎖的組合語言函數
LDR    r0, =Lock_Variable	
MOVS   r1, #0	
DMB	
STR    r1, [r0]	; 資料記憶體障礙
SEV	; 清除栓鎖
BX    LR	; 送出事件以喚醒其他的處理器
	; 返回

藉著結合事件通信界面與必要的號誌程式，在迴旋栓鎖期間的功率消耗可以被降低。類似的技巧可以用在訊息傳遞與工作同步上。

在大部分基於 Cortex-M3 的產品裡，都僅有單一個處理器，因此 RXEV 輸入可能會固定為 0，或者連結到產生事件的周邊上。

## 自我重置的控制

Cortex-M3 提供了兩個自我重置控制的特性。第一個為 NVIC 應用中斷與重置控制(Application Interrupt and Reset Control)暫存器(bit [0])的 VECTRESET 控制位元：

```
LDR    R0, =0xE000ED0C      ; NVIC AIRCR 位址
LDR    R1, =0x05FA0001      ; 設定 VECTRESET 位元
                            ; (05FA 為寫入存取鑰匙)
STR    R1, [R0]
deadloop
B     deadloop
                                ; deadloop 用來確保並無其它指令
                                ; 跟隨在重置執行之後
```

寫進此位元將會重置 Cortex-M3 處理器，但不包括除錯邏輯。此寫入動作不會重置 Cortex-M3 處理器之外的電路。例如，若單晶片包含一個 UART，寫進此位元並不會重置 UART 或位於 Cortex-M3 之外的任意周邊。

第二個重置特性為相同的 NVIC 暫存器中的 SYSRESETREQ 位元。它允許了 Cortex-M3 處理器對系統重置產生器宣稱一個重置要求信號。因為系統重置產生器並非 Cortex-M3 設計的一部分，此重置特性的實作會隨晶片設計而定。因此某些晶片可能沒有提供此功能，請小心謹慎地檢查晶片規格。

下面為使用 SYSRESETREQ 的範例程式：

```

LDR R0, =0xE000ED0C      ; NVIC AIRCR 位址
LDR R1, =0x05FA0004      ; 設定 SYSRESETREQ 位元
                          ; (05FA 為寫入存取鑰匙)

STR R1, [R0]
deadloop
B    deadloop            ; deadlock 用來確保並無其它指令
                          ; 跟隨在重置執行之後

```

在大部分的情形下，當 SYSRESETREQ 位元被設定，會藉著重置產生器去宣稱 Cortex-M3 處理器(SYSRESETn)的系統重置信號。隨著晶片設定，可能會也可能不會重置晶片的其它部分(例如周邊)。正常的情形下，此重置宣告不會重置 Cortex-M3 的除錯邏輯。

注意，從 SYSRESETREQ 宣告到重置產生器的真正重置之間的延遲，也可能值得留意。因為重置產生器內的延遲，你可能會發現在重置要求設定後，處理器依然接受中斷。如果想要核心在執行此程式之前停止接受中斷，你可以藉由 MSR 指令去設定 FAULTMASK。

# 15

## Chapter

# 除錯架構

本章內容包括：

- ✓ 除錯特性概觀
- ✓ CoreSight 概觀
- ✓ 除錯模式
- ✓ 除錯事件
- ✓ Cortex-M3 內的中斷點
- ✓ 在除錯時存取暫存器內容
- ✓ 其它核心除錯的特性

## 除錯特性概觀

Cortex-M3 提供了全面的除錯環境，根據運算的性質，除錯特性可以歸類為兩個群組：

### 1. 侵入性的除錯：

- |           |                          |
|-----------|--------------------------|
| ○ 程式暫停與步進 | ○ 暫存器值的存取(讀取與寫入兩者)       |
| ○ 中斷點指令   | ○ 存取資料位址、位址範圍、資料值等的資料觀察點 |
| ○ 硬體中斷點   | ○ 基於 ROM 的除錯(快閃補丁)       |
| ○ 除錯監視器例外 |                          |

## 2. 非侵入性的除錯：

- 記憶體存取(甚至可於核心執行期間存取記憶體內容)
- 經由選擇性的嵌入式追蹤模組 (Embedded Trace Module) 進行指令追蹤
- 資料追蹤
- 經由儀器追蹤模組 (Instrumentation Trace Module) 進行軟體追蹤
- 經由資料觀察點 (Data Watchpoint) 與追蹤模組 (Trace Module) 進行效能分析 (Profiling)

Cortex-M3 處理器包括了一些除錯的元件。除錯系統則根據 CoreSight 除錯架構，允許以標準的方案，去存取除錯控制、蒐集追蹤資訊、與偵測除錯系統組態。

## CoreSight 概觀

CoreSight 除錯架構涵蓋範圍廣大，包括除錯介面協定、除錯匯流排協定、除錯元件之控制、安全特性、追蹤資料介面及其它等等。CoreSight Technology System Design Guide (Ref 3)為獲得此架構綜觀的一份有用的文件。除此之外，Cortex-M3 Technical Reference Manual (Ref 1)內的一些章節對 Cortex-M3 設計的除錯元件作了一些敘述；這些元件通常被除錯程式軟體所用，而非應用程式。然而，簡短地檢視這些項目依然有用，因此我們會對除錯系統的運作更加了解。

## 處理器除錯介面

不同於傳統的 ARM7 或 ARM9，Cortex-M3 處理器的除錯系統是基於 CoreSight Debug Architecture。傳統上，ARM 處理器提供了 JTAG 介面，以允許存取暫存器與控制記憶體介面。在 Cortex-M3 裡，處理器上對除錯邏輯的控制是經由稱作除錯存取埠(Debug Access Port, DAP, 相似於 AMBA 中的 APB)的匯流排介面來執行。DAP 受到轉換 JTAG 或者 Serial-Wire(序列線)至 DAP 汇流排介面協定的其他元件控制。

因為內部除錯匯流排相似於 APB，因此非常容易連接多個除錯元件，是相當具有彈性的除錯系統。除此之外，藉著分開除錯介面與除錯控制硬體，使用在晶片上的實際介面類型能夠具有通透性；因此，不管你使用的除錯介面為何，皆能執行相同的除錯工作。

Cortex-M3 處理器核心內的實際除錯功能，受到 NVIC 與一些其它除錯元件，例如 FPB、DWT 以及 ITM 等等所控制。NVIC 包括一些暫存器以控制核心除錯動作，例如暫停與步進；而其它的區塊支援了觀察點、中斷點、與除錯訊息輸出等特性。

## 除錯 Host 介面

CoreSight 技術支援了一些在除錯 host 與 SoC 之間作聯繫的介面類型，傳統上，這通常是 JTAG 介面。如今既然處理器的除錯介面已被更改為標準匯流排介面，故在除錯 host 與處理器除錯介面之間放置不同的介面模組，即可得到具有不同除錯 host 介面的不同晶片，而不需要重新設計處理器上的除錯介面。

Cortex-M3 系統目前支援兩種類型的除錯 host 介面：第一種是非常為人熟知的 JTAG 介面，第二種是新的介面協定，即序列線(Serial-Wire, SW)。SW 介面把信號減少為兩個。ARM 提供了數個類型的除錯 host 介面模組(稱作除錯埠, Debug Port, DP)。除錯硬體連接到 DP 的一端，DP 另一端則連接到處理器上的 DAP 介面。

### 為何是序列線？

Cortex-M3 的目標是低成本微控制器的市場，而其中大部分的元件有著非常低的接腳數。例如，一些低階的版本為 28 接腳的包裝。雖然事實上 JTAG 是個相當受歡迎的協定，但使用 4 個接腳以除錯，對 28 接腳的元件負擔太大，所以，除錯接腳減低為 2 的 SW 就成為具有吸引力的解決方案。

## DP 模組、AP 模組、與 DAP

在 Cortex-M3 處理器裡，外部除錯硬體與除錯介面的連接被區分為多個階段(參見圖 15-1)。

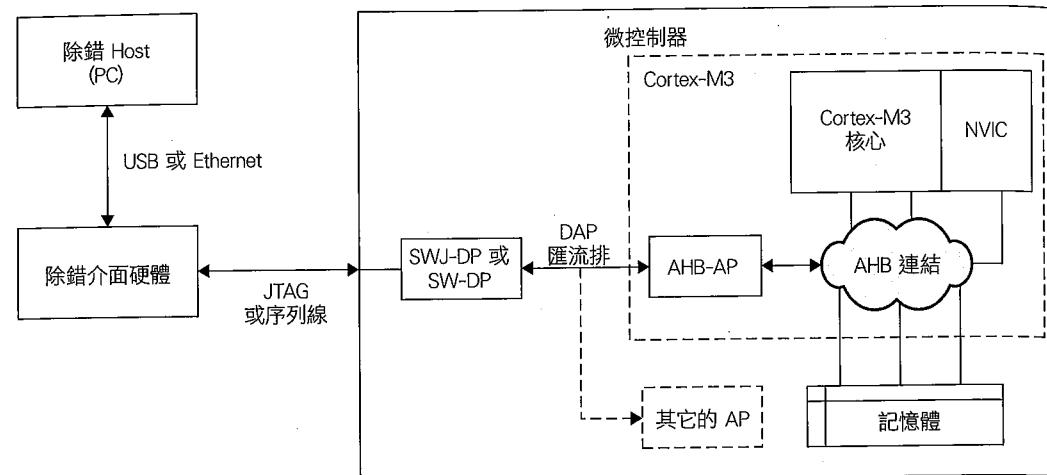


圖 15-1 除錯 host 與 Cortex-M3 之間的連接

DP 介面模組(通常為 SWJ-DP 或 SW-DP)首先轉換外部信號到標準 32-bit 除錯匯流排(方塊圖中的 DAP 匯流排), SWJ-DP 支援 JTAG 與 SW 兩者, 而 SW-DP 僅支援 SW。ARM CoreSight 產品系列中亦有僅支援 JTAG 協定的 JTAG-DP。晶片製造廠商可選擇任何一個 DP 模組以配合他們的需求。DAP 匯流排的位址為 32 位元, 位址匯流排最高 8 位元用來選擇被存取的設備, 最多 256 個設備可連接到 DAP 匯流排。於 Cortex-M3 處理器內, 僅使用了一個設備位址, 所以必要時你還可以連接 255 個存取埠(Access Port, AP)設備至 DAP 匯流排。

於 Cortex-M3 處理器內, 在通過了 DAP 介面之後, 連接了稱為 AHB-AP 的 AP 元件；其作用是充作匯流排橋以轉換命令為 AHB 傳輸, 此元件被插入 Cortex-M3 中的內部匯流排網路。因此允許去存取 Cortex-M3 的記憶體映射, 包括 NVIC 裡的除錯控制暫存器。

於 CoreSight 產品系列中, 可選擇數種類型的 AP 設備, 包括 APB-AP 與 JTAG-AP。APB-AP 可用來產生 APB 傳輸；JTAG-AP 可用來控制傳統基於 JTAG 的測試介面, 例如 ARM7 的除錯介面。

## 追蹤介面

CoreSight 架構的另外一部分與追蹤有關。在 Cortex-M3 裡有三個類型的追蹤來源：

- ◆ 指令追蹤：由嵌入式追蹤巨集格(Embedded Trace Macrocell, ETM)產生
- ◆ 資料追蹤：由 DWT 產生
- ◆ 除錯訊息：由 ITM 產生(提供了除錯 GUI 內的訊息輸出, 例如 printf)

在追蹤期間, 追蹤的結果會以資料封包的型態, 藉著被稱作進階追蹤匯流排(Advanced Trace Bus, ATB)的追蹤資料匯流排, 從追蹤來源(例如 ETM)輸出。根據 CoreSight 架構, 若 SoC 包括了多個追蹤來源(例如多重處理器), 則 ATB 資料串流可藉由 ATB 合併硬體(於 CoreSight 架構此硬體被稱為 ATB 漏斗, funnel)來加以合併。接著, 晶片上最後的資料串流可以被連結到追蹤埠介面單元(Trace Port Interface Unit, TPIU)並輸出到外部的追蹤硬體。一旦資料到達除錯 host(例如, 一部個人電腦), 資料串流可以再轉換回多重的資料串流。

雖然 Cortex-M3 有多個追蹤來源, 其除錯元件本來就設計成可以處理追蹤合併, 故無需增加 ATB 漏斗模組。追蹤輸出介面可以直接連接到為 Cortex-M3 設計的 TPIU 特別版本。接著, 追蹤資料被外部硬體所擷取, 並且被除錯 host(例如, 個人電腦)收集後加以分析。

## CoreSight 特性

基於 CoreSight 的設計有下述優勢：

- ◆ 在處理器執行中, 可以檢查記憶體內容與周邊暫存器。
- ◆ 多個處理器除錯介面可以單一除錯硬體控制。例如, 若使用了 JTAG, 即使晶片上有許多個處理器, 也需要一個 TAP 控制器。
- ◆ 內部除錯介面皆根據一個簡單的匯流排設計, 使得它具擴展性, 並且易於開發晶片或 SoC 其它部分的額外測試邏輯。

◆ 它允許多個追蹤資料串流由單一追蹤擷取元件所收集，並在除錯 host 上分離回多個串流。

使用在 Cortex-M3 中的除錯系統與標準的 CoreSight 建構稍微不同：

追蹤元件在 Cortex-M3 中被特別設計。某些 ATB 介面於 Cortex-M3 裡為 8 位元；而於 CoreSight 裡則為 32 位元。

在 Cortex-M3 裡，除錯的實作並不支援 TrustZone<sup>1</sup>。

除錯元件為系統記憶體映射的一部分；而在標準的 Coresight 系統，是用一個分開的匯流排(具有分開的記憶體映射)來控制除錯元件。例如，於 CoreSight 系統裡概念上系統的連接可以如圖 15-2 所示：

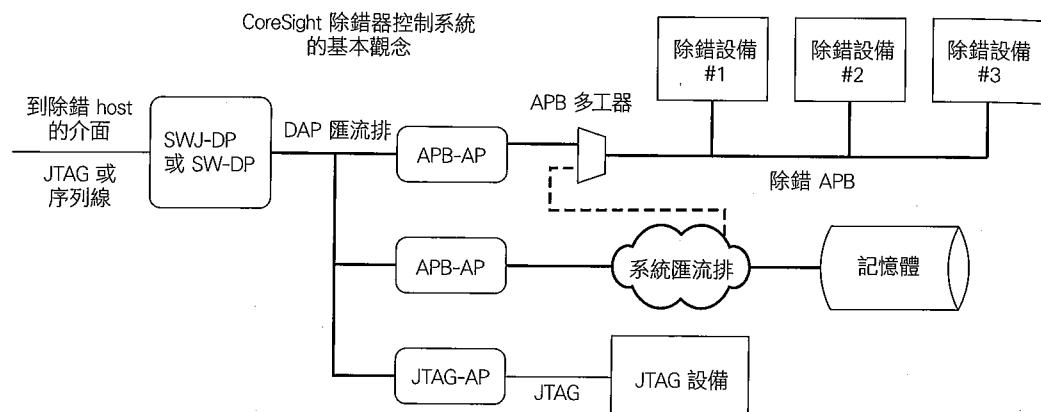


圖 15-2 CoreSight 系統的設計概念

在 Cortex-M3 裡，除錯元件分享了相同的系統記憶體映射(參見圖 15-3)。

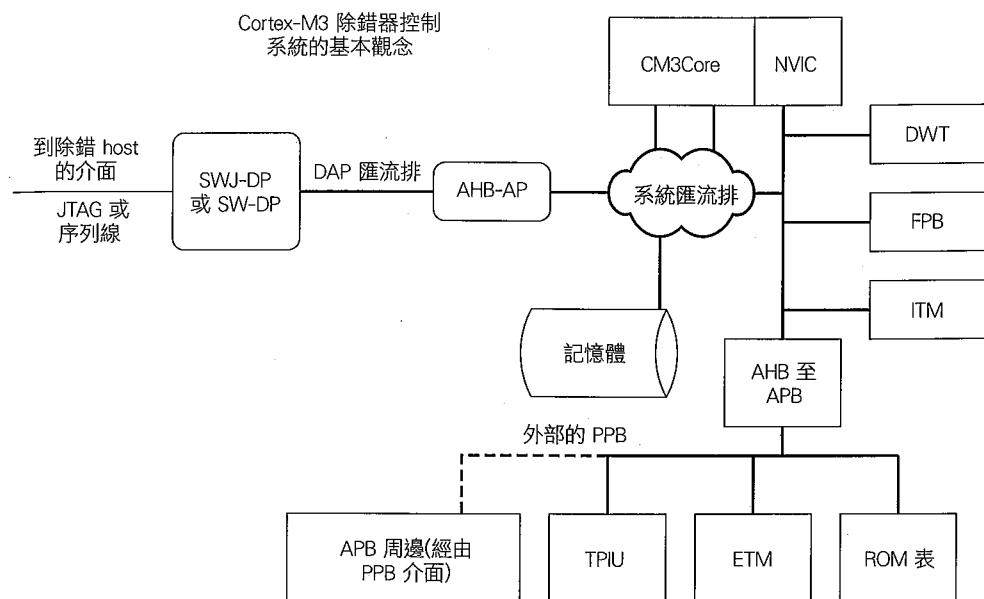


圖 15-3 Cortex-M3 中的除錯系統

雖然在 Cortex-M3 中建立的除錯元件不同於平常的 CoreSight 系統，但在 Cortex-M3 中的通信界面與協定相容於 CoreSight 架構，並且可以直接被連結到 CoreSight 系統。例如，CoreSight 除錯元件，像是 CoreSight TPIU、除錯埠、與追蹤基礎架構方塊等可以在 Cortex-M3 使用並可使其擴充為一個多核心的除錯系統。

更多關於 CoreSight 除錯架構的資訊，可於 CoreSight Technology System Design Guide (Ref 3)中找到。

## 除錯模式

Cortex-M3 裡有兩類型的除錯運算模式。第一個為暫停，此時處理器完全地停止程式的執行；第二個為除錯監視例外，此時處理器執行了一個例外處理程式以遂行除錯工作，但仍然允許更高優先權例外之出現。除錯監視為例外類型 12 並且其優先權可加以設定；它可藉除錯事件來啟動，或者藉手動以設定等待位元。總結如下：

<sup>1</sup> TrustZone 為 ARM 為嵌入式產品提供了安全的特性的技術。

# Jason 嘴書—EETOP 世界唯一貼

## 1. 暫停模式：

- 指令的執行被停止
- SYSTICK 計數器被停止
- 支援單步的運算
- 中斷可被置於等待，且可於單步動作中被啟動，或者被遮罩以忽略單步期間的外部中斷

## 2. 除錯監視模式：

- 處理器執行例外處理程式類型 12(除錯監視)
- SYSTICK 計數器繼續執行
- 隨除錯監視的優先權與新中斷的優先權而定，新來的中斷可能會也可能不會被強佔
- 如果除錯事件發生時，正執行更高優先權的中斷，則除錯事件將會被忽略
- 支援單步的運算
- 記憶體內容(例如，堆疊記憶體)在堆疊與處理程式執行期間，可能會被除錯監視處理程式所改變

除錯監視存在的理由是因為在某些電子系統，停止處理器以作除錯運算可能是不可行的。例如，在汽車引擎控制或硬碟控制器應用中，於除錯期間，處理器應該繼續服務中斷的請求，以確保操作的安全或避免破壞被測試的設備。藉由除錯監視，除錯器可停止並除錯執行緒等級應用程式與更低優先權的中斷處理程式，而同時更高優先權的中斷與例外依然能被執行。

要進入暫停模式，必須設定 NVIC 裡除錯暫停控制與狀態暫存器(Debug Halting Control and Status Register, DHCSR)的 C\_DEBUGEN 位元。此位元僅可藉由 DAP 來設定，故若不藉由除錯器，就無法暫停 Cortex-M3 處理器。在設定 C\_DEBUGEN 之後，藉著設定 DHCSR 中的 C\_HALT 位元，核心可被暫停。C\_HALT 位元可經由除錯器或在處理器本身執行的軟體來加以設定。

做讀取運算與寫入運算時，DHCSR 的位元欄位的定義方式不同。做寫入運算時，除錯鑰值需使用在 bit 31 至 bit 16 之間；做讀取運算，則並沒有除錯鑰，且傳回值的上半 word 包含了狀態位元(詳見表 15-1)。

表 15-1 除錯暫停控制與狀態暫存器(0xE000EDF0)

位元	名稱	類型	重置值	描述
31:16	KEY	W	-	除錯鑰：欲寫值進此暫存器則需將 0xA05F 之值寫入此欄位，否則寫入將會被忽略
25	S_RESET_ST	R	-	核心已被重置或正被重置；讀取將清除此位元
24	S_RETIRE_ST	R	-	自先前讀取之後的指令已完成；讀取將清除此位元
19	S_LOCKUP	R	-	當此位元為 1，核心會處於鎖住狀態
18	S_SLEEP	R	-	當此位元為 1，核心會處於睡眠狀態
17	S_HALT	R	-	當此位元為 1，核心會被暫停
16	S_REGRDY	R	-	暫存器讀取/寫入運算已完成
15:6	保留的	-	-	保留的
5	C_SNAPSTALL	R/W	0*	用來打斷暫停的記憶體存取
4	保留的	-	-	保留的
3	C_MASKINTS	R/W	0*	當步進時遮罩中斷；僅可在處理器被暫停時作修改
2	C_STEP	R/W	0*	處理器單步運算；僅在 C_DEBUGEN 被設定時有效
1	C_HALT	R/W	0*	暫停處理器核心；僅在 C_DEBUGEN 被設定時有效
0	C_DEBUGEN	R/W	0*	致能暫停模式除錯

\*DHCSR 內的控制位元藉著啟動電源重置以作重置。系統重置(例如，藉由 NVIC 的應用中斷與重置控制暫存器)並不會重置除錯控制。

在正常的情形下，DHCSR 僅為除錯器所使用。應用程式不應該更改 DHCSR 內容，以避免造成除錯工具的問題。

◆ DHCSR 中的控制位元被電源啟動重置以作重置。系統重置(例如，藉著 NVIC 的應用中斷與重置控制暫存器)並不會重置除錯控制。使用除錯監視作除錯時，另一不同的 NVIC 暫存器，即 NVIC 的除錯例外與監視控制暫存器，被用來控制除錯的活動(詳見表 15-2)。除了除錯監視控制之位元外，除錯例外與監視控制暫存器包含追蹤系統致能位元(TRCENA)與一些向量捕捉(Vector Catch, VC)控制位元。VC 特性僅可以用於暫停模式除錯。當錯誤(或核心重置)產生且相關的 VC 控制位元被設定，則會設定暫停的請求，並且核心會在現行指令完成時，立即停止。

# Jason 嘴書—EETOP 世界唯一貼

表 15-2 除錯例外與監視控制暫存器(0xE000EDFC)

位元	名稱	類型	重置值	描述
24	TRCENA	R/W	0*	追蹤系統致能；欲使用 DWT、ETM、ITM、與 TPIU 則此位元必須被設定為 1
23:20	保留的	-	-	保留的
19	MON_REQ	R/W	0	指出除錯監視器是因手動等待要求而造成，而非硬體除錯事件
18	MON_STEP	R/W	0	處理器單步運算；僅在 MON_EN 被設定時有效
17	MON_PEND	R/W	0	將監視器例外要求置於等待；在優先權允許之下，核心將進入監視器例外
16	MON_EN	R/W	0	致能除錯監視器例外
15:11	保留的	-	-	保留的
10	VC_HARDERR	R/W	0*	在硬錯誤上的除錯陷阱
9	VC_INTERR	R/W	0*	在中斷/例外服務錯誤的除錯陷阱
8	VC_BUSERR	R/W	0*	在匯流排錯誤的除錯陷阱
7	VC_STATERR	R/W	0*	在用法錯誤狀態下錯誤的除錯陷阱
6	VC_CHKERR	R/W	0*	在用法錯誤致能檢查時錯誤的除錯陷阱(例如，未對齊的，除以 0 等)
5	VC_NOCPERR	R/W	0*	在用法錯誤的除錯陷阱，並無輔助處理器的錯誤
4	VC_MMERR	R/W	0*	在記憶體管理錯誤的除錯陷阱
3:1	保留的	-	-	保留的
0	VC_CORERESET	R/W	0*	在核心重置的除錯陷阱

\*DHCSR 內的控制位元藉著啟動電源重置以作重置。系統重置(例如，藉由 NVIC 的應用中斷與重置控制暫存器)並不會重置除錯控制。

◆ TRCENA 控制位元與 DEMCR 中的 VC 控制位元由電源啟動重置以作重置；系統重置並不會重置這些位元。然而，作監視模式除錯的控制位元，會被電源啟動重置與系統重置加以重置。

## 除錯事件

Cortex-M3 會因為一些可能的理由而進入除錯模式(暫停與除錯監視例外兩者)。就暫停模式除錯而言，如果發生類似圖 15-4 所示的情況，處理器將進入暫停模式。

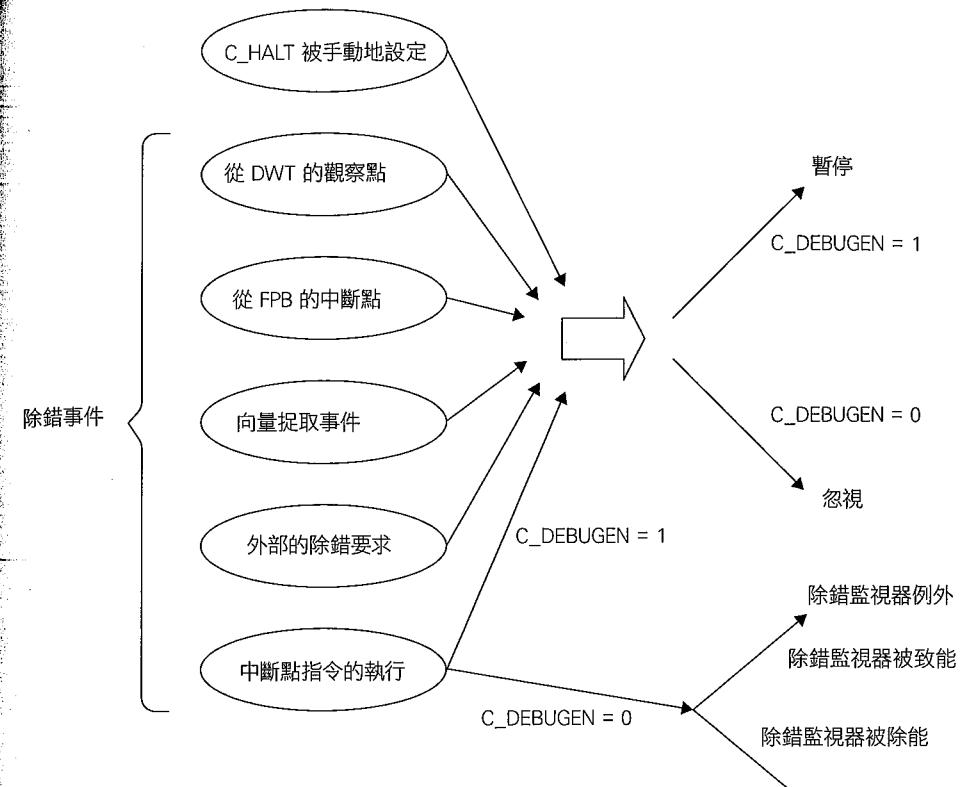


圖 15-4 以暫停模式除錯的除錯事件

外部的除錯要求由 Cortex-M3 處理器上被稱作 EDBGREQ 的信號提出，此信號的實際連接隨著微控制器或 SoC 設計而定。在某些情形下，此信號被定在低電位並且永遠不會有作用。然而，它也可以連接去接受來自外加除錯元件的除錯事件(晶片製造廠可以增加額外的除錯元件到 SoC)，或者，在一個多處理器的設計裡，它可以連結到來自其它處理器的除錯事件。

在完成除錯之後，藉著清除 C\_HALT 位元，程式的執行可返回至正常。

同樣地，就以除錯監視例外作除錯而言，一些除錯事件會造成除錯監視的發生(詳見圖 15-5)。

# Jason 嘴書—ETOP 世界唯一貼

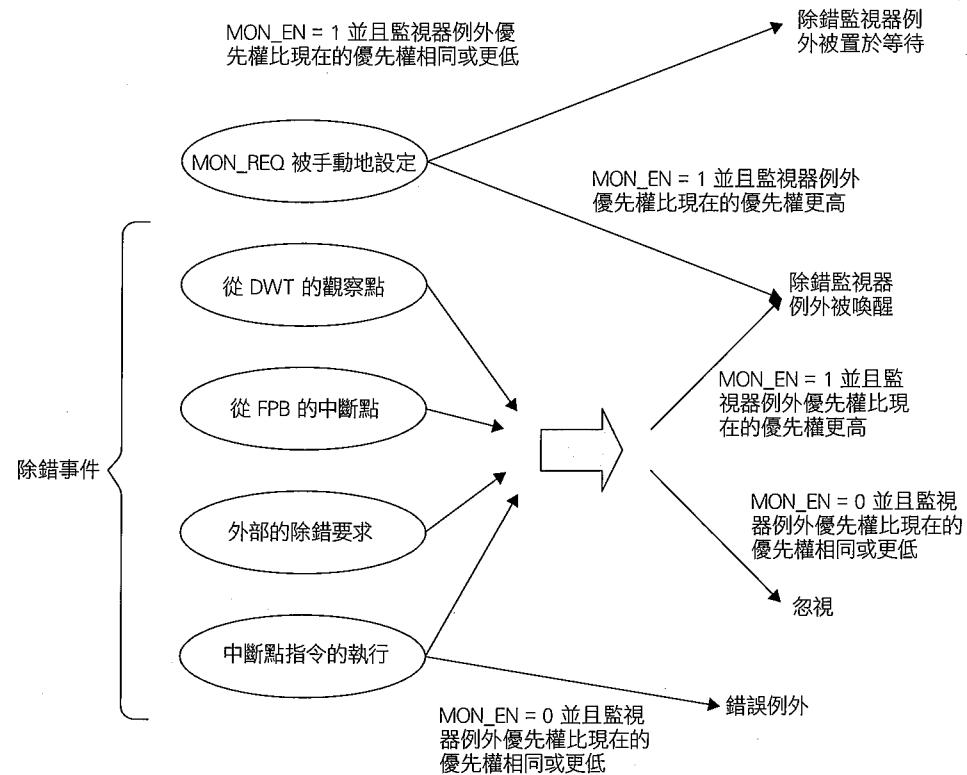


圖 15-5 除錯監視例外的除錯事件

除錯監視的行為與暫停模式除錯稍有不同，其原因是除錯監視例外本身為一種例外類型，若處理器正執行其它例外處理程式，則可能會受到處理器現行優先權的影響。

在完成除錯之後，藉著執行例外返回，程式的執行可返回至正常。

## Cortex-M3 內的中斷點

在大多數微控制器中最常被使用的除錯特性為中斷點特性。Cortex-M3 裡，支援了兩個類型的中斷點機制：

◆ 中斷點指令

◆ 使用 FPB 內的位址比較器之中斷點

中斷點指令(BKPT immed8)是一個 16-bit Thumb 指令，其編碼為 0xBExx，當中最低 8 位元依照指令後面給定的立即值資料來決定。當此指令被執行後，會產生除錯事件，如果 C\_DBGEN 被設定，則可用來暫停處理器核心；如果除錯監視被啟用，則可用來觸發除錯監視例外。因為除錯監視為一個可程式化優先權的例外類型，所以只能使用在執行緒層級，或者優先權低於它本身的例外處理程式中。因此，如果除錯監視被用來除錯，則 BKPT 指令就不該使用於例如 NMI 或硬錯誤等例外處理程式中，而且除錯監視只能置於等待狀態，並於例外處理程式完成之後執行。

當除錯監視例外返回時，會返回到 BKPT 指令所在的位址，而非 BKPT 之後的位址。這是因為正常使用中斷點指令的情形下，BKPT 先被用來取代一個正常的指令，在遇到中斷點並執行除錯動作之後，指令記憶體會回復成原先的指令，不會影響到其後的指令記憶體。

如果 BKPT 指令被執行時，C\_DEBUGEN=0 且 MON\_EN=0，則會造成處理器進入硬錯誤例外，硬錯誤狀態暫存器(Hard Fault Status Register, HFSR)中的 DEBUGEVT 被設定為 1，並且除錯錯誤狀態暫存器(Debug Fault Status Register, DFSR)中的 BKPT 也被設定為 1。

即使程式記憶體不能被改變，也可編程 FPB 單元以產生中斷點事件。然而，這種作法僅限於六個指令位址與兩個常數位址。更多有關 FPB 的資訊於下一章討論。

## 在除錯時存取暫存器內容

於 NVIC 裡還包含兩個暫存器以提供除錯功能，即是除錯核心暫存器選擇暫存器(Debug Core Register Selector Register, DCRSR)與除錯核心暫存器資料暫存器(Debug Core Register Data Register, DCRDR)(詳見表 15-3 與表 15-4)。此兩個暫存器允許除錯器去存取處理器之暫存器。暫存器傳輸特性僅適用於處理器暫停時。欲使用上述暫存器以讀取暫存器內容，需遵循下列程序：

1. 確定處理器已暫停。
2. 寫入 DCRSR，設定其 bit 16 為 0 以顯示讀取運算。
3. 一直詢問，直到 DHCSR (0xE000EDF0)裡的 S\_REGRDY 位元為 1。
4. 讀取 DCRDR 以取得暫存器內容。

# Jason 嘴書—EETOP 世界唯一貼

要寫入暫存器也需要類似的程序：

- 確定處理器已暫停。
- 寫資料值至 DCRDR。
- 寫入 DCRSR, 設定其 bit 16 為 1 以顯示寫入運算。
- 一直詢問, 直到 DHCSR (0xE000EDF0)裡的 S\_REGRDY 位元為 1。

DCRSR 與 DCRDR 暫存器只可於暫停模式除錯時, 傳輸暫存器值。對使用除錯監視處理程式做的除錯, 一些暫存器的內容可從堆疊記憶體存取, 其餘則可在監視例外處理程式之內直接存取。

如果有適當的函數庫與除錯的支援, DCRDR 也可以用來當做 semihosting。例如, 當應用程式執行 printf 敘述, 本文輸出可藉由一些 putc (put character)函數呼叫來產生。Putc 函數呼叫可儲存輸出字元與狀態至 DCRDR, 接著並觸發除錯模式。除錯器可偵測核心暫停並收集輸出字元以做顯示。這項運算原本需要暫停核心, 然而使用 ITM 的 semihosting 方案就沒有這樣的限制。

表 15-3 除錯核心暫存器選擇暫存器(0xE000EDF4)

位元	名稱	類型	重置值	描述
16	REGWnR	W	-	資料傳輸的方向： 寫入 = 1, 讀取 = 0
15:5	保留的	-	-	-
4:0	REGSEL	W	-	被存取的暫存器： 00000 = R0 00001 = R1 ... 01111 = R15 10000 = xPSR/旗標 10001 = MSP (主要堆疊指標) 10010 = PSP (程序堆疊指標) (31:24)控制 (23:16)FAULTMASK (15:8)BASEPRI (7:0)PRIMASK 其他被保留

表 15-4 除錯核心暫存器資料暫存器(0xE000EDF4)

位元	名稱	類型	重置值	描述
31:0	Data	R/W	-	資料暫存器保留暫存器讀取結果或者要寫入被選定暫存器的資料

## 其它核心除錯的特性

NVIC 亦包括一些其它除錯特性。這些特性包括了：

- ◆ **外部的除錯要求信號:** NVIC 提供了一個外部除錯要求信號, 使得 Cortex-M3 處理器可以藉著外部的事件(例如多處理器系統中其它處理器的除錯狀態)進入除錯模式。此特性對多重處理器系統的除錯非常有用；在簡單的微控制器裡, 此信號極有可能會定在低電位。
- ◆ **除錯錯誤狀態暫存器(DFSR):** 因為在 Cortex-M3 上有各種可得的除錯事件, 除錯器可利用 DFSR 來判斷發生的除錯事件。
- ◆ **重置控制:** 在除錯的期間, 處理器核心可使用 NVIC 應用中斷與重置控制暫存器(0xE000ED0C)的 VECTRESET 控制位元來重新啟動。使用此重置控制, 處理器可以在不影響系統中除錯元件的情形下作重置。
- ◆ **中斷遮罩:** 此特性在步進動作時非常有用。例如, 如果你需要除錯一個應用程式, 但並不希望程式在步進動作時進入中斷服務程式, 則可以遮罩中斷要求。此可藉由設定除錯暫停控制與狀態暫存器(Debug Halting Control and Status register)(0xE000EDF0)裡的 C\_MASKINTS 位元來達成。
- ◆ **暫停的匯流排傳輸終止:** 如果一個匯流排傳輸被暫停了相當長的時間, 則可能藉由 NVIC 控制暫存器以終止暫停傳輸。此可藉由設定除錯暫停控制與狀態暫存器(Debug Halting Control and Status register)(0xE000EDF0)裡的 C\_SNAPSTALL 位元來達成。此特性僅可在暫停時由除錯器使用。

## 除錯的元件

本章內容包括：

- ✓ 簡介
- ✓ 追蹤元件：資料觀察點與追蹤
- ✓ 追蹤元件：儀器追蹤巨集格
- ✓ 追蹤元件：嵌入式追蹤巨集格
- ✓ 追蹤元件：追蹤埠界面單元
- ✓ 快閃補丁與中斷點單元
- ✓ AHB 存取埠
- ✓ ROM 表

### 簡介

Cortex-M3 處理器伴隨一些除錯元件，以提供除錯功能，例如中斷點、觀察點、快閃補丁、以及追蹤等等。如果你是一位應用程式開發者，你可能永遠不需要知道這些除錯元件的細節，因為通常只有除錯器工具會使用到他們。本章將介紹給你每一個除錯元件的基本常識。如果你希望了解細節，例如實際的程式設計師的模型，可參考 Cortex-M3 Technical Reference Manual (Ref 1)。

所有的除錯追蹤元件與 FPB 皆可經由 Cortex-M3 私用周邊匯流排(PPB)來加以編程。在大多數的情形下，元件只會由除錯 host 作編程。並不推薦應用程式去嘗試存取除錯元件(ITM 裡的激勵埠暫存器除外)，因為這可能會干擾除錯器的運算。

MEMO.

## Cortex-M3 內的追蹤系統

Cortex-M3 追蹤系統是基於 CoreSight 架構。追蹤結果以封包的形式產生，其大小不一(以 bytes 數為單位)。追蹤元件使用進階追蹤匯流排(ATB)以傳輸封包到追蹤埠介面單元(TPIU)，TPIU 格式化封包為追蹤介面協定。接著，資料被外部的追蹤擷取設備，例如追蹤埠分析儀(Trace Port Analyzer, TPA)，加以擷取。

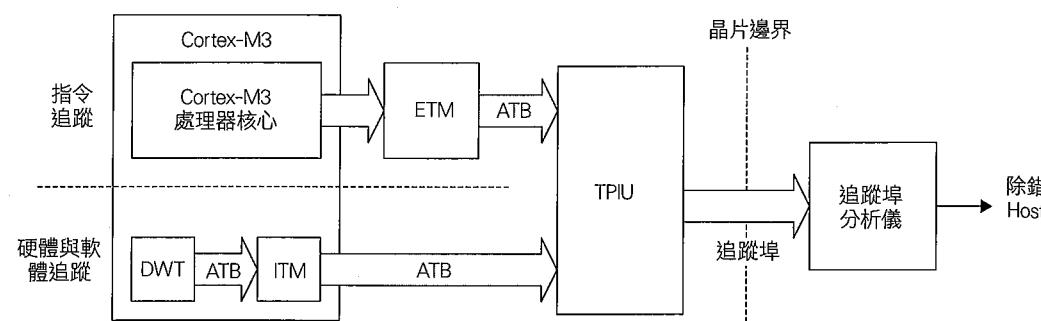


圖 16-1 Cortex-M3 追蹤系統

在標準的 Cortex-M3 處理器內，最多有三個追蹤來源：ETM、ITM、DWT 等。注意在 Cortex-M3 中 ETM 為選擇性的，故某些 Cortex-M3 產品並沒有指令追蹤能力。在運算時，每一個追蹤來源被指定了一個 7-bit 的 ID 值(ATID)，此值於 ATB 會合時期與追蹤封包一起傳送，所以當封包到達除錯 host 時，可以被分開為多個追蹤串流。

不同於許多其它的標準 CoreSight 元件，Cortex-M3 處理器中的除錯元件包括合併 ATB 串流的功能；然而在標準的 CoreSight 系統裡，被稱作 ATB 漏斗的 ATB 封包合併器是獨立的區塊。

在使用追蹤系統之前，除錯例外與監視控制暫存器(DEMCR)裡的追蹤致能(TRCENA)位元需要被設定為 1(參考表 15-2 或 D.37)。否則追蹤系統會被禁能；在不需要做追蹤的正常運算當中，清除 TRCENA 位元，可以禁能某些追蹤邏輯，並且減低功率消耗。

## 追蹤元件：資料觀察點與追蹤 (Data Watchpoint and Trace)

DWT 擁有一些追蹤功能：

1. 它有四個比較器，且可被組態如下：
  - 硬體觀察點(對處理器產生觀察點事件以引發除錯模式，例如暫停或者除錯監視)
  - ETM 觸發器(造成 ETM 將觸發器封包送進指令追蹤串流)
  - PC 取樣器事件觸發器
  - 資料位址取樣器觸發器
  - 第一個比較器也可以與時脈週期計數器(CYCCNT)比較，以取代與資料位址的比較

2. 計數器可作下述計數：

- 時脈週期(CYCCNT)
- 摺疊的指令
- 下載儲存單元(Load Store Unit, LSU)運算
- 睡眠週期
- 指令週期(Cycles per instruction, CPI)
- 中斷成本

3. 以規律時間間隔作 PC 取樣

4. 中斷事件追蹤

當使用為硬體觀察點或 ETM 觸發器時，可對比較器編寫程式去比較資料位址或者程式計數器；當作其它功能使用時，它會比較資料位址。

每一個比較器有三個相關的暫存器：

◆ COMP(比較)暫存器

◆ MASK(遮罩)暫存器

◆ FUNCTION 控制暫存器

COMP 暫存器為 32-bit 的暫存器，可用來與資料位址(或者程式計數器值，即 CYCCNT)作比較。MASK 暫存器決定作比較時，是否忽略資料位址內的任何位元(詳見表 16-1)。

表 16-1 DWT MASK 暫存器的編碼

MASK	忽略的位元
0	比較所有的位元
1	忽略 bit[0]
2	忽略 bit[1:0]
3	忽略 bit[2:0]
...	...
15	忽略 bit[14:0]

比較器的 FUNCTION 暫存器決定其功能。為了避免預料之外的行為，在設定此暫存器之前，應該先設定好 MASK 暫存器與 COMP 暫存器。如果要更改比較器的功能，你需要藉著設定 FUNCTION 為 0 以除能比較器，然後程式化 MASK 與 COMP 暫存器，再於最後的步驟中致能 FUNCTION 暫存器。

其餘的 DWT 計數器典型的用途是當作分析應用程式碼效能 (profiling)。它們可被編程，使得計數器溢位時會送出事件(其形式為追蹤封包)。一個典型的應用是使用 CYCCNT 暫存器來計算一個特定工作的時脈週期數，以作效能評比。

在 DWT 使用之前，DEMCR 中的 TRCENA 位元必須設定為 1。如果使用 DWT 以產生追蹤，也應該致能 ITM 控制暫存器的 DWTFEN 位元。

## 追蹤元件：儀器追蹤巨集格 (Instrumentation Trace Macrocell, ITM)

ITM 擁有下列功能：

- ◆ 軟體可以直接將控制台的訊息寫進 ITM 激勵埠，並且把它們當作追蹤資料以輸出。
- ◆ DWT 可以產生追蹤封包並將其經由 ITM 輸出。
- ◆ ITM 可以產生時戳封包，並將其插入追蹤串流以幫助除錯器去找出事件的時點。

因為 ITM 使用追蹤埠以輸出資料，如果微控制器或 SoC 並沒有 TPIU 的支援，可能無法輸出所追蹤的資訊。因此，在你使用 ITM 之前，需要檢查是否微控制器或 SoC 擁有所有需要的功能。在最差的情形下，如果沒有這些功能可用，你依然可以使用 NVIC 除錯暫存器，或 UART 去輸出控制台訊息。

欲使用 ITM，DEMCR 中的 TRCENA 位元需要被設定為 1；否則，ITM 將會被除能，並且無法存取 ITM 暫存器。

除此之外，ITM 中也有一個鎖暫存器。在編程 ITM 之前，你需要寫入存取鑰 0xC5ACCE55 (CoreSight ACCESS)至此暫存器。否則，所有對 ITM 作寫入的運算將被忽略。

最後，ITM 本身是控制致能特有功能的另一個控制暫存器。此控制暫存器也包括了 ATID 欄位，其為 ATB 裡 ITM 的 ID 值。此 ID 值必須不同於其它追蹤來源的 IDs，故接收到追蹤封包的除錯 host，能夠把 ITM 的追蹤封包與其它追蹤封包分開。

### 使用 ITM 作軟體追蹤

ITM 的一個主要的用途是支援除錯訊息的輸出(例如 printf)。ITM 包含了 32 個激勵埠，允許不同的軟體程序對不同的埠作輸出，並可於後來在除錯 host 裡將訊息分開。每一個埠可以由追蹤致能暫存器去致能或者除能，並且能被編程(八個埠為一群)以允許或拒絕用戶程式對它作寫入。

與基於 UART 的文字輸出不同，使用 ITM 作輸出並不會造成應用程式太多延遲。ITM 內使用了 FIFO 緩衝器，所以寫下的輸出訊息可以被緩衝。然而，在你寫進它之前，仍然需要檢查 FIFO 緩衝器是否已滿。

輸出的訊息可以在追蹤埠介面或 TPIU 上的序列線介面(SWV)加以收集。你並不需要由最後完成的程式中移除產生除錯訊息的程式碼，因為若 TRCENA 控制位元為低電位，ITM 將不會動作並且除錯訊息不會被輸出。你也可以在一個活動中的系統打開錯誤訊息，並使用 ITM 中的追蹤致能暫存器去限制致能哪些暫存器，使得其中僅有部分的訊息可被輸出。

## 以 ITM 與 DWT 作硬體追蹤

ITM 可用來輸出硬體追蹤封包。封包由 DWT 產生，而 ITM 作為追蹤封包合併單元。欲使用 DWT 追蹤，你需要致能 ITM 控制暫存器內的 DWTFEN 位元；其餘的 DWT 追蹤設定依然需要在 DWT 上撰寫程式。

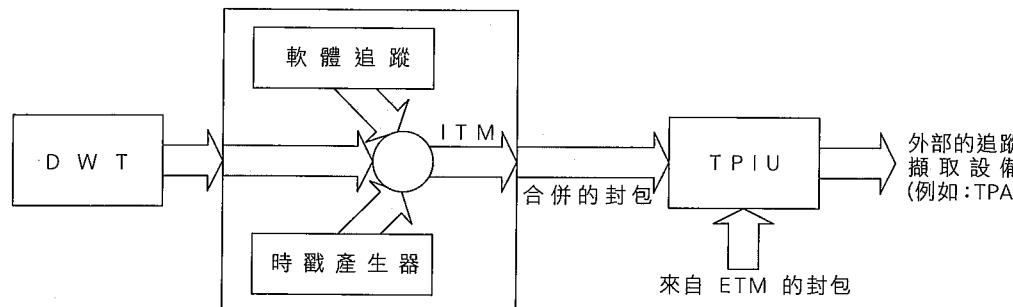


圖 16-2 在 ITM 與 TPIU 上追蹤封包的合併

## ITM 時戳

ITM 有一個時戳特性，會在新的追蹤封包進入 ITM 裡面的 FIFO 時，藉著插入 delta 時戳封包進入追蹤，讓追蹤擷取工具發現時點資訊。當時戳計數器發生溢位時，也會產生時戳封包。

時戳封包提供了與先前事件之間的時間差(即 delta)。藉著使用 delta 時戳封包，追蹤擷取工具可以建立每一個封包產生的時間，重建不同除錯事件的時點。

## 追蹤元件：嵌入式追蹤巨集格 (Embedded Trace Macrocell, ETM)

ETM 區塊是用來提供指令的追蹤。它為選擇性的，所以在一些 Cortex-M3 產品上可能不會有。當被致能並且開始追蹤運算後，它會產生指令追蹤封包。ETM 內提供了 FIFO 緩衝器以爭取足夠的時間來擷取追蹤串流。

為了減少 ETM 產生的資料量，它並不會永遠輸出處理器已到達/執行的真正位址。通常輸出的是程式流程的資訊，並且僅在必要時輸出完整的位址（例如，如果作了跳躍）。因為除錯 host 應該有二位元映像的副本，所以它可以重建處理器已執行的指令序列。

ETM 也會與其它的除錯元件(例如 DWT)相互影響，DWT 內的比較器可用來產生 ETM 裡的觸發事件，或者控制追蹤之開始/停止。

不同於傳統的 ARM 處理器內的 ETM，Cortex-M3 的 ETM 並沒有自己的位址比較器，其理由是 DWT 能夠替 ETM 執行比較運算。更且，因為資料追蹤的功能由 DWT 執行，Cortex-M3 內的 ETM 設計與其它 ARM 核心的傳統 ETM 設計有相當大的差別。

欲使用 Cortex-M3 內的 ETM，需要作下列的設定（由除錯工具來處理）：

1. 除錯例外與監視控制暫存器 (DEMCR) 內的 TRCENA 位元應該被設定為 1（詳見表 15-2 或 D.37）。
2. 需要解除 ETM 的鎖定，故其控制暫存器可以被編程。此可藉由寫數值 0xC5ACCE55 進入 ETM 的 LOCK\_ACCESS 暫存器來達成。
3. ATB 的 ID 暫存器 (ATID) 應該以一個獨特的數值作設定，使經由 TPIU 的追蹤封包輸出，可以與其它追蹤來源的封包分開。
4. ETM 的 NIDEN 輸入信號應當設定為高電位。此信號的實作因設備而定。詳細的細節請參考晶片製造廠商的資料規格列表。
5. 編程 ETM 控制暫存器以產生追蹤。

## 追蹤元件：追蹤埠介面單元 (Trace Port Interface Unit, TPIU)

TPIU 用以輸出 ITM、DWT 與 ETM 的追蹤封包到外部的擷取設備(例如：追蹤埠分析儀)。Cortex-M3 的 TPIU 支援兩個輸出模式：

- ◆ 時脈模式：使用至多 4-bit 平行資料輸出埠
- ◆ 序列線檢視器(SWV)模式：使用一個位元的 SWV 輸出<sup>1</sup>

在時脈模式，資料輸出埠上實際使用的位元數，可藉由程式定為不同的大小；這將會隨著晶片封裝與應用的追蹤輸出可用的信號接腳而定。晶片支援的最大追蹤埠的大小，可由 TPIU 裡的一個暫存器來決定。除此之外，也可藉程式設定追蹤資料輸出的速度。

在 SWV 模式中使用了 SWV 協定。這樣可以減少輸出信號的數目到 1 位元，但是追蹤輸出的最大頻寬也會減少。當使用序列線除錯協定時，SWV 模式輸出可以與 TDO 共享。故你可以藉著使用標準 JTAG 連結的低成本除錯器，從 DWT 與 ITM 擷取追蹤資訊。

欲使用 TPIU, DEMCR 內的 TRCENA 位元應該被設定為 1，並且協定(模式)選擇暫存器與追蹤埠大小控制暫存器，需要由追蹤擷取軟體藉程式加以設定。

## 快閃補丁與中斷點單元 (Flash Patch and Breakpoint Unit, FPB)

FPB 有兩種功能：

- ◆ 硬體中斷點(對處理器產生中斷點事件以啟動除錯模式，例如暫停或除錯監視)
- ◆ 從程式記憶體空間補丁指令或是字面值資料至 SRAM

FPB 包含八個比較器：

<sup>1</sup> 基於早期 Cortex-M3 revision 0 的 Cortex-M3 產品的早期版本並無此模式。

六個指令比較器

兩個文字比較器

### 什麼是字面值載入？

我們以組合語言撰寫程式的時候，經常需要在暫存器裡設定立即值；但是如果立即資料的值非常大，則運算不能容納於一個指令空間。例如：

```
LDR R0, =0xE000E400 ; 外部中斷優先權暫存器的起始位址
```

因為並無指令擁有 32 個位元的立即值空間，所以我們需要將立即值資料放置在一個不同的記憶體空間，通常位於程式區之後，並且再藉由 PC 相對載入指令以將立即值資料讀入暫存器。故我們會得到如下編譯的二進位程式：

```
LDR R0, [PC, #<immed_8>*4]
; immed_8 = (字面值之位址 - PC) / 4
...
; 字面值庫
...
DCD 0xE000E400
...
```

或者以 Thumb-2 指令撰寫如下：

```
LDR.W R0, [PC, #+/- <offset_12>]
; offset_12 = 字面值之位址 - PC
...
; 字面值庫
...
DCD 0xE000E400
...
```

因為我們可能在程式中使用一個以上的字面值，組譯器或編譯器經常會產生一區塊的字面值資料，即所謂的字面值庫。

Cortex-M3 中，字面值載入為資料匯流排上執行的資料讀取運算。(依記憶體位置而定，可能是 D-CODE 匯流排或系統匯流排)。

FPB 有一個快閃補丁控制暫存器，此暫存器包含一個致能位元以啟能 FPB。於此之外，每一個比較器，於其比較器控制暫存器裡伴隨著一個獨立的致能位元。欲運作比較器，則此兩個位元皆須設定為 1。

比較器可編寫程式，將程式空間的位址重新映射到 SRAM 記憶體區域。當使用此功能時，需規劃 REMAP 暫存器以提供重新映射內容之基底位址。REMAP 暫存器的最高 3 位元(bit[31:29])被硬體設定為 3' b001，因而限定了重新映射基底位址的位置在 0x20000000 與 0x3FFFFFFF 間，此範圍永遠位於 SRAM 記憶體區域內。

當指令位址或字面值位址符合比較器定義的位址時，讀取存取會被重新映射到 REMAP 暫存器所指定的表格。

使用重新映射功能，就可以建立一些"what if"的測試條件，將原來的指令或字面值以其他的指令或字面值替代；即使程式碼位於 ROM 或快閃記憶體也沒問題。一個使用範例是在程式區裡補丁程式 ROM，使得能夠執行位於 SRAM 區域內的程式或副程式，因而能跳躍至測試程式或副程式。如此則可能為基於 ROM 的設備除錯。

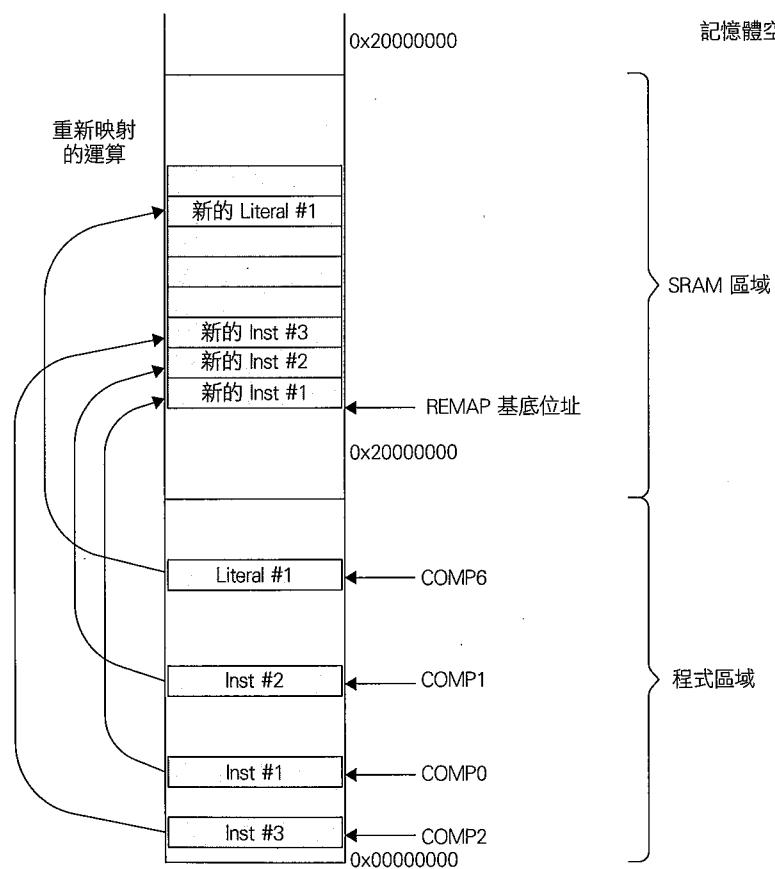


圖 16-3 快閃補丁：指令與字面值讀取的重新映射

你也可以改用六個指令位址比較器以產生中斷點，並引發暫停模式除錯或除錯監視例外。

## AHB 存取埠

AHB-AP 為除錯介面模組(SWJ-DP 或 SW-DP)與 Cortex-M3 記憶體系統之間的橋樑。AHB-AP 內的三個暫存器用來做除錯 host 與 Cortex-M3 系統之間最基本的資料傳輸：

- ◆ 控制與狀態字組(Control and Status Word, CSW)
- ◆ 傳輸位址暫存器(Transfer Address Register, TAR)
- ◆ 資料讀取/寫入(Data Read/Write, DRW)

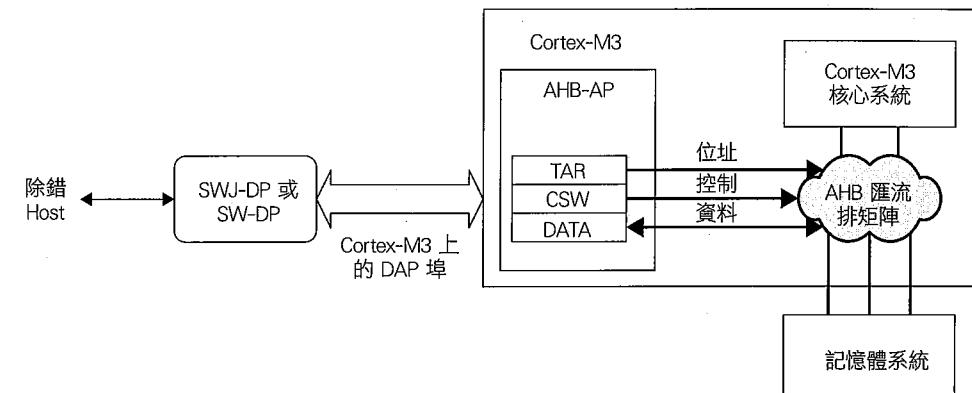


圖 16-4 Cortex-M3 內 AHB-AP 的連接

CSW 暫存器可以控制傳輸方向(讀取/寫入)、傳輸大小、傳輸型態等等。TAR 暫存器用以指定傳輸位址；而 DRW 暫存器用以執行資料傳輸運算(當此暫存器被存取時，傳輸則開始)。

資料暫存器 DRW 代表了確實顯示在匯流排的內容。作 half word 與 byte 傳輸時，所需的資料將需要藉著除錯軟體，手動位移至正確的 byte 通道。例如，如果你希望執行一個 half word 大小的資料傳輸到位址 0x1002，你需要將資料置於 DRW 暫存器 bit[31:16]上。APB-AP 可以產生未對齊的傳輸，但它不會根據位址位移去旋轉獲得的資料，所以除錯器軟體將需要手動旋轉資料，或者於必要時，將一個未對齊的資料存取分開為多個存取動作。

AHB-AP 內的其他暫存器提供了額外的特性。例如，AHB-AP 提供四個成庫的暫存器與一個自動位址遞增功能，故可加速小範圍內的記憶體存取或者循序的傳輸。

在 CSW 暫存器內，有一稱作 MasterType 的位元。此位元正常情形下被設定為 1，故當硬體接收來自 AHB-AP 的傳輸時，會知道傳輸來自除錯器；然而藉著清除此位元，除錯器則可假裝為核心。在此情形下，被連接到 AHB 系統的設備接收到的傳輸，其行為應該會猶如它被處理器存取時一般。這對測試 FIFO 的周邊有用，因為它們被除錯器存取時行為不同。

## ROM 表

ROM 表用來允許自動偵測 Cortex-M3 晶片內的除錯元件。Cortex-M3 處理器是基於 ARM v7-M 架構的第一個產品。它有一個被定義的記憶體映射並且包括一些除錯元件。然而，在更新的 Cortex-M 設備裡，或者若晶片設計師修改了預設的除錯元件，則除錯設備的記憶體映射可能會不同。為了允許除錯工具偵測除錯系統內的元件，故包含了一個 ROM 表；此表提供了 NVIC 與除錯區塊位址上的資訊。

ROM 表位於位址 0xE00FF000。利用 ROM 表裡的內容，可以計算出系統與除錯元件所在的記憶體位置，然後除錯工具可以檢查所發現元件的 ID 暫存器，以判斷系統上有何元件可用。

就 Cortex-M3 而言，ROM 表的第一項(位於 0xE000FF000)應該包含到 NVIC 記憶體位置的位移。(ROM 表內第一項的預設值為 0xFFFF0F003；bit[1:0]表示設備存在並且 ROM 表中此項資料之後還有有下一項資料。則 NVIC 位移可計算如下：  
 $0xE00FF000 + 0xFFFF0F000 = 0xE000E000$ 。)

Cortex-M3 預設的 ROM 表顯示在表 16-2。然而，因為晶片製造廠商可以增加、移除，或者將一些選用的除錯元件取代為其他的 CoreSight 除錯元件，所以在你的 Cortex-M3 設備上發現的值可能會有不同。

表 16-2 Cortex-M3 上 ROM 表預設值

位址	值	名稱	描述
0xE00FF000	0xFFFF0F003	NVIC	指向位於 0xE000E000 的 NVIC 基底位址
0xE00FF004	0xFFFF02003	DWT	指向位於 0xE0001000 的 DWT 基底位址
0xE00FF008	0xFFFF03003	FPB	指向位於 0xE0002000 的 FPB 基底位址
0xE00FF00C	0xFFFF01003	ITM	指向位於 0xE0000000 的 ITM 基底位址
0xE00FF010	0xFFFF41003/0xFFFF41002	TPIU	指向位於 0xE0040000 的 TPIU 基底位址
0xE00FF014	0xFFFF42003/0xFFFF42002	ETM	指向位於 0xE0041000 的 ETM 基底位址
0xE00FF018	0	END	End-of-table 標記
0xE00FFFCC	0x1	MEMTYPE	指出系統記憶體可以在此記憶體映射被存取
0xE00FFFD0	0	PID4	周邊 ID 空間；保留的
0xE00FFFD4	0	PID5	周邊 ID 空間；保留的
0xE00FFFD8	0	PID6	周邊 ID 空間；保留的
0xE00FFFDc	0	PID7	周邊 ID 空間；保留的
0xE00FFFE0	0	PID0	周邊 ID 空間；保留的
0xE00FFFE4	0	PID1	周邊 ID 空間；保留的
0xE00FFFE8	0	PID2	周邊 ID 空間；保留的
0xE00FFFEc	0	PID3	周邊 ID 空間；保留的
0xE00FFFF0	0	CID0	元件 ID 空間；保留的
0xE00FFFF4	0	CID1	元件 ID 空間；保留的
0xE00FFFF8	0	CID2	元件 ID 空間；保留的
0xE00FFFC	0	CID3	元件 ID 空間；保留的

數值的最低兩個位元(LSB)顯示了設備是否存在。在正常的情形下，NVIC、DWT 與 FPB 應該永遠存在，故其相關最低的兩個位元永遠為 1。然而，TPIU 與 ETM 可能被晶片製造廠商移除，並且可能替換為 CoreSight 產品家族的其他除錯元件。

數值的較高部分顯示相較於 ROM 表基底位址的位移位址。例如：

$$\text{NVIC 位址} = 0xE00FF000 + 0xFFFF0F000 = 0xE000E000 \text{ (截為 32-bit)}$$

開發除錯工具時，需要從 ROM 表來決定除錯元件的位址。一些 Cortex-M3 設備可能有不同的除錯元件連接設定，而造成了不同的基底位址。藉著由此 ROM 表計算出正確的設備位址，除錯器可以決定提供的除錯元件之基底位址，然後經由這些元件的元件 ID，除錯器可以判斷可用的除錯元件種類。

# 17

Chapter

## Cortex-M3 開發入門

本章內容包括：

- ✓ 選擇 Cortex-M3 產品
- ✓ Cortex-M3 Revision 0 與 Revision 1 之間的差別
- ✓ Cortex-M3 Revision 1 與 Revision 2 之間的差別
- ✓ 修正版 2 的好處與影響
- ✓ 開發工具

### 選擇 Cortex-M3 產品

除了記憶體、周邊選項、運算速度之外，一些其他的因素使得一個 Cortex-M3 產品與另一個產品不同。ARM 供給的 Cortex-M3 設計包括一些可組態的特性，例如：

- ◆ 外部中斷的數量
- ◆ 中斷優先權等級的數量(優先權等級暫存器的欄寬)
- ◆ 有 MPU 或無 MPU
- ◆ 有 ETM 或無 ETM
- ◆ 除錯介面的選擇(序列線、JTAG 或兩者)

在大多數的專案裡，微控制器的特性與規格一定會影響你的 Cortex-M3 產品的選擇。例如：

- ◆ **周邊**: 對許多的應用而言, 周邊支援為主要的判斷標準。更多的周邊可能不錯, 但這也會影響了微控制器的功率消耗與價格。
- ◆ **記憶體**: Cortex-M3 微控制器可能會有從幾 kB 到幾 MB 的快閃記憶體。除此之外, 內部記憶體的大小可能也重要。通常這些因素將對價格有直接的衝擊。
- ◆ **時脈速度**: 即使使用了 0.18 um 製程, ARM 的 Cortex-M3 設計可輕易地達到 100MHz 以上。然而, 因為記憶體存取速度的限制, 製造廠商可能會指定較低的運算速度。
- ◆ **腳標區域(footprint)**: 隨晶片製造廠商的決定, Cortex-M3 可以有許多不同封裝。許多 Cortex-M3 設備有低接腳數的封裝, 是低成本製造環境的理想選擇。

## Cortex-M3 Revision 0 與 Revision 1 之間的差別

Cortex-M3 產品的早期版本是根據 Cortex-M3 處理器修正版 0。自從 2006 年第三季以後, 開始可得到根據 Cortex-M3 修正版 1 的產品。在本書出版時, 所有基於 Cortex-M3 的新產品應該是根據修正版 1。知道你正使用的晶片是修正版 0 或修正版 1 可能是件重要的事, 因為在第二次版本發表裡有一些更動與改進。

在程式設計師的模型與開發特性中可見的改變包括這些：

- ◆ 從修正版 1 開始, 例外出現時暫存器的堆疊可以藉組態以使得堆疊由 double word 對齊的記憶體位址開始。這可藉著設定 NVIC 組態控制暫存器內的 STKALIGN 位元來達成。
- ◆ 因為上述理由, NVIC 組態控制暫存器擁有 STKALIGN 位元。
- ◆ 修正版 2 包含新的 AUXFAULT (Auxiliary Fault, 輔助錯誤) 狀態暫存器(選用的)。
- ◆ 額外的特性, 包括了加到 DWT 的資料值比對(matching)。
- ◆ 因為修正版欄位的更新而造成 ID 暫存器值的改變。

在終端用戶看不見的改變包括：

- ◆ 核心記憶體空間的記憶體屬性是以硬體線路設死為可快取的、已配置的、不可緩衝的、以及不可共享的。這樣會影響 I-Code AHB 與 D-Code AHB 介面, 但不會影響系統匯流排介面。
- ◆ 支援 I-Code AHB 與 D-Code AHB 之間匯流排多工運算模式。在此運算模式下, 藉著使用一個簡單的匯流排多工器可合併 I-Code 與 D-Code 累積排(先前解決方案係採用一個 ADK 累積排矩陣元件)。這樣可以減低總閘數。
- ◆ 增加新的輸出埠以連接 AHB 追蹤巨集格(即 HTM, 一個由 ARM 提供的 CoreSight 除錯元件)以作複雜的資料追蹤運算。
- ◆ 即使在系統重置期間, 也可以存取除錯元件或者除錯控制暫存器; 僅有在電源啟動的重置期間, 這些暫存器為不可存取的。
- ◆ TPIU 支援 SWV 運算模式; 此允許以低成本硬體去捉取追蹤資訊。
- ◆ 在修正版 1 裡, NVIC 中斷控制與狀態暫存器裡的 VECTPENDING 欄位可以被 NVIC 除錯暫停控制與狀態暫存器裡的 C\_MASKINTS 位元影響。如果 C\_MASKINTS 被設定, 若遮罩正在遮罩一個等待中的中斷, 則 VECTPENDING 之值可能為 0。
- ◆ JTAG-DP 除錯介面模組已經被更改為 SWJ-DP 模組(詳見下一節 "修正版 1 的改變:由 JTAG-DP 傳輸到 SWJ-DP")。晶片製造廠商可以繼續使用 JTAG-DP, 且 JTAG-DP 依然為 CoreSight 產品家族裡的一員。

因為 Cortex-M3 的修正版 0 在它的例外序列裡並沒有 double word 堆疊對齊的特性, 一些編譯器工具, 例如 ARM RealView Development Suite(RVDS)與 KEIL RealView Microcontroller Development Kit, 有特別選項以允許軟體調整堆疊, 這使得開發的應用與 EABI 相容。如果它需要與其他的 EABI-相容的開發工具一起使用, 這可能就很重要。

為了判斷微控制器或 SoC 內使用了哪一個 Cortex-M3 修正版, 你可以利用 NVIC 裡的 CPU ID 基底暫存器。如表 17-1 所示, 修正版 (Revision) 與變異 (Variant) 號碼顯示它是何種 Cortex-M3 版本。

表 17-1 CPU ID 基底暫存器(0xE000ED00)

	Implemented [31:24]	Variant [23:20]	Constant [19:16]	PartNo [15:4]	Revision [3:0]
修正版 0 (r0p0)	0x41	0x0	0xF	0xC23	0x0
修正版 1 (r1p0)	0x41	0x0	0xF	0xC23	0x1
修正版 1 (r1p1)	0x41	0x1	0xF	0xC23	0x1
修正版 2 (r2p0)	0x41	0x2	0xF	0xC23	0x0

在 Cortex-M3 處理器內個別的除錯元件也會有他們自己的 ID 暫存器，其修正版欄位在修正版 0 與修正版 1 之間可能會有不同。

## 修正版 1 的改變：由 JTAG-DP 傳輸到 SWJ-DP

在一些更早期的 Cortex-M3 產品裡提供的 JTAG-DP 被 SWJ-DP 所取代。序列線 JTAG 除錯埠(SWJ-DP)結合了 SW-DP 與 JTAG-DP 的功能，並且有自動的協定偵測。使用此元件，Cortex-M3 設備可以支援藉由 SW 與 JTAG 兩種介面除錯。

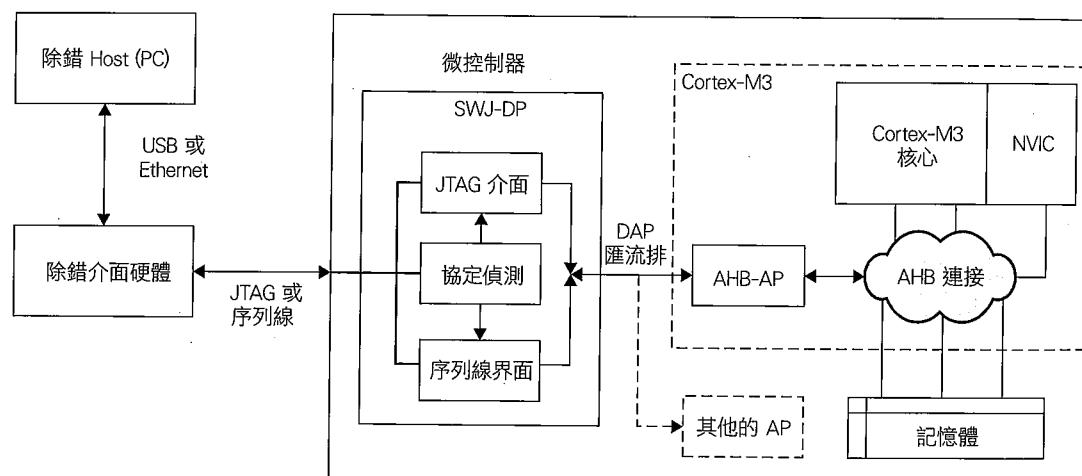


圖 17-1 SWJ-DP：結合了 JTAG-DP 與 SW-DP 的功能

## Cortex-M3 Revision 1 與 Revision 2 之間的差別

於 2008 年的年中，對矽供應商發表了 Cortex-M3 的修正版 2。使用修正版 2 的實際產品將可能於 2008 年的年終到達市場。修正版 2 有一些新的特性，其大多數的目標在減低功率消耗以及在除錯方面有更佳的彈性。

在程式設計師的模型看得到的改變包括：

### 預設組態為 Double word 堆疊對齊

例外堆疊的 Double word 堆疊對齊特性，現在預設為致能。(注意：矽供應商可能會選擇使用修正版 1 的行為)。這樣能減低大部分 C 應用程式的啟動成本(因為不需要設定 NVIC 組態控制暫存器裡的 STKALIGN 位元)。

### 新的輔助控制暫存器

NVIC 中新增了輔助控制暫存器以允許微調處理器行為。例如，為了除錯的目的，可以關掉 Cortex-M3 裡的寫入緩衝器，故匯流排錯誤將會與記憶體存取指令(精確地)同步。如此則錯誤的指令可由被堆疊的返回位址(被堆疊的程式計數器)輕易地找到。輔助控制暫存器的細節如表 17-2 所示：

表 17-2 輔助控制暫存器(0xE000E008)

位元	名稱	類型	重置值	描述
2	DISFOLD	R/W	0	避免 IT 摺疊(預防 IT 指令執行時期與下一個指令重疊)
1	DISDEFWBUF	R/W	0	對預定的記憶體映射，除能寫入緩衝(在 MPU 映射區的記憶體存取不受影響)
0	DISMCYCINT	R/W	0	除能多重週期指令的中斷，例如 LDM、STM、64-bit 乘與除指令等。

## ID 暫存器值的更新

NVIC 與除錯元件內不同的 ID 暫存器已經被更新。例如, NVIC 裡的 CPU ID 暫存器改變如表 17-3：

表 17-3 CPU ID 基底暫存器(0xE000ED00)

	Implemented [31:24]	Variant [23:20]	Constant [19:16]	PartNo [15:4]	Revision [3:0]
Revision 2 (r2p0)	0x41	0x2	0xF	0xC23	0x0

## 除錯特性

在除錯特性方面, 修正版 2 有一些改進：

- ◆ DWT 裡面的觸發資料追蹤的觀察點, 現在支援單獨作讀取傳輸的追蹤, 以及單獨作寫入傳輸的追蹤。這樣可以減少需要的追蹤資料頻寬, 因為你可以指定僅在資料被更改時做追蹤, 或者僅在讀取資料時做追蹤。
- ◆ 在建構除錯特性時有更高的彈性。例如, 在非常低功率設計中, 可以減少中斷點與觀察點的數量, 因而減少了設計的大小。

對多處理器的除錯有更好的支援。引進了一個新的介面以允許同時重新開機和多處理器的單步執行功能(以程式設計師的觀點是不會發現這一點的)。

## 睡眠特性

在系統設計等級方面, 現存的睡眠特性也有改善。在 r2p0 裡處理器的喚醒有可能被延遲。這樣做允許晶片的更多部分處於低功率狀態, 並且在系統準備好的時候, 電源管理系統可以重新進行程式的執行。在一些微控制器設計裡, 系統的某些部分在睡眠時會處於低功率, 這樣做是需要的, 因為在恢復電源之後, 可能需要一些時間電壓供應才會穩定。除了睡眠延長之外, 採用新技術以允許更低的功率消耗。在先前的 Cortex-M3 版本, 為了允許處理器經由中斷以從睡眠中醒來, 核心的自動時脈需要維持活動。雖然自動時脈僅僅驅動系統的一小部分, 但是最好還是把它完全關閉。

為了解決這個問題, 在處理器外面加上一個簡單的中斷控制器。此控制器, 稱作喚醒中斷控制器(Wake-up Interrupt Controller, WIC), 鏡射了深度睡眠期間 NVIC 內的中斷遮罩功能, 並且在需要醒來的時候, 能夠告知電源管理系統。這樣做, 則所有進入 Cortex-M3 處理器的時脈信號皆能被停止。

除了停止時脈之外, 新的設計方法也允許了處理器大多數的部分處於低功率, 並且將處理器的狀態保留在特殊邏輯晶胞(logic cell)裡。當中斷到達時, WIC 送出一個要求到 PMU 以開啟系統電源。在處理器電源開啟之後, 處理器的先前狀態由特殊邏輯晶胞恢復回來, 並且處理器已準備好處理中斷。

此功率降低特性減低了設計在睡眠期間的功率消耗。然而, 這項特性隨使用的矽製程而定, 並且在一些修正版 2 產品上可能會沒有。

## 修正版 2 的好處與影響

就嵌入式產品開發而言, 修正版 2 的好處與影響的意義為何？

首先, 它意指嵌入式產品有較低的功率消耗且其電池壽命較長。

當使用 WIC 模式的深度睡眠, 僅需要產品中非常小的部分保持活動。並且, 在目標為極低功率的設計中, 矽供應商可藉由減低中斷點與觀察點的數量以減少設計的大小。

第二, 它在除錯與解決問題方面提供了更好的彈性。除了除錯器所改善的資料追蹤特性之外, 我們還可以使用新的輔助控制暫存器去強迫寫入傳輸為不可緩衝, 故能找出發生錯誤指令所在, 或者在多重時脈指令執行期間除能中斷, 因而在處理例外前, 會先完成每一個多重載入/儲存指令, 這樣會更容易分析記憶體內容。對擁有多個 Cortex-M3 的系統, 修正版 2 賦予了多核心同時重新啟動與步進的能力。

除此之外, 修正版 2 有一些內部的最佳化以允許更高效率表現與更好的介面特性。這允許矽供應商去開發更快速更多特性的 Cortex-M3。

然而, 有一些嵌入式程式師需要知道的事情：

- 例外堆疊框架的 double word 堆疊對齊：**藉由預設，例外堆疊框架將會依 double word 記憶體位置作對齊。對於專為修正版 0 或修正版 1 寫的一些組合語言應用程式，如果使用了堆疊來傳輸資料到例外處理程式，就可能會受到影響。例外處理程式應該藉著讀取堆疊框架裡被堆疊的 PSR 的 bit 9 以判斷堆疊對齊是否已執行，接著，它可以判斷在例外發生之前被堆疊資料的位址。另一個替代方式是應用程式可以設定 STKALIGN 位元為 0，以得到與修正版 0 及修正版 1 相同的堆疊行為。與 EABI 相容的應用程式（例如使用 EABI 相容的編譯器編譯的 C 程式）不會受到影響。
- SYSTICK 計時器需要在深度睡眠期間停止：**如果 Cortex-M3 微控制器包含了降低功率的特性，或者核心的時脈在深度睡眠期間完全停止，則 SYSTICK 計時器在深度睡眠期間就不能動作。使用 OS 的嵌入式應用需要藉由處理器核心外部的計時器來喚醒處理器以作事件排程。
- 除錯與功率降低的特性：**當處理器連接到除錯器的時候，降低功率的新特性會被除能。這是因為除錯器需要在除錯階段去存取處理器的暫存器。在除錯階段，核心將依然能夠暫停或進入睡眠模式，但即使啟動了降低功率特性也不會觸發降低功率的序列。測試功率降低運算時，測試中的設備不可與除錯器連接。

## 開發工具

在開始使用 Cortex-M3 時，你將需要一些工具。這些工具通常包括：

- ◆ **編譯器及/或者組譯器：**編譯你的 C 或組譯器應用程式的軟體；幾乎所有的 C 編譯器套件都會伴隨著一個組譯器。
- ◆ **指令集模擬器：**在軟體開發初期作為除錯用途之模擬指令執行情形的軟體。
- ◆ **線上模擬器(in-circuit emulator, ICE)或除錯探測工具：**連接你的除錯 host(通常為一台 PC)到目標電路的硬體設備；其介面可以是 JTAG 或者 SW。
- ◆ **開發板：**包括了微控制器的電路板。
- ◆ **追蹤蒐集：**一個選用的硬體與軟體套件，可用來蒐集指令追蹤，或者從 DWT 與 ITM 輸出，並輸出成適合人們閱讀的格式。
- ◆ **一個嵌入式的作業系統：**執行於微控制器的作業系統。此為選用的；許多的應用並不需要一個 OS。

## C 編譯器

已經有一些給 Cortex-M3 用的 C 編譯器套件與開發工具（詳見表 17-4）。

由 CodeSourcery 提供的 GNU C Compiler 為一個免費的解決方案。在撰寫本書時，主要的 GNU C Compiler (GCC) 並沒有支援 Cortex-M3；然而，在不久的未來，Cortex-M3 的支援將會合併於主要的 GCC。你也可以獲得一些商業工具的測試版，例如 RealView-MDK。

表 17-4 支援 Cortex-M3 的開發工具的例子

公司	產品 <sup>1</sup>
ARM ( <a href="http://www.arm.com">www.arm.com</a> )	Cortex-M3 得到了 Realview Development Suite3.0 (RVDS) 的支援。可使用 RealView-ICE (RVI) version 1.5 以連接除錯目標到除錯的環境。注意，較舊一些的產品，例如 ADS 與 SDT 並沒有支援 Cortex-M3。
KEIL (一個 ARM 公司; <a href="http://www.keil.com">www.keil.com</a> )	Cortex-M3 得到了 RealView Microcontroller Development Kit (RealView-MDK) 的支援。可使用 ULINK(TM) USB-JTAG 轉接器以連接除錯目標至除錯 IDE。
CodeSourcery ( <a href="http://www.codesourcery.com">www.codesourcery.com</a> )	為了 ARM 處理器的 GNU Tool Chain 可在下面網址得到： <a href="http://www.codesourcery.com/gnu_toolchains/arm/index.htm">www.codesourcery.com/gnu_toolchains/arm/index.htm</a> 。它基於 GNU C Compiler 4.1.0 並且支援 Cortex-M3。
Rowley Associates ( <a href="http://www.rowley.co.uk">www.rowley.co.uk</a> )	CrossWorks for ARM 為一個支援 Cortex-M3 的 GNU C Compiler-based 開發套件 ( <a href="http://www.rowley.co.uk/arm/index.htm">www.rowley.co.uk/arm/index.htm</a> )。
IAR Systems ( <a href="http://www.iar.com">www.iar.com</a> )	IAR Embedded Workbench for ARM and Cortex 提供了一個 C/C++ 編譯器與除錯環境（v4.40 或更新的版本）。你也可以取得一個基於 Luminary Micro LM3S102 微控制器的 KickStart kit，包括了除錯器與一個 J-Link Debug Probe 去連接目標板以除錯 IDE。
Lauterbach ( <a href="http://www.lauterbach.com">www.lauterbach.com</a> )	JTAG 除錯器與追蹤功能可從 Lauterbach 得到。

## 嵌入式作業系統的支援

許多的應用需要作業系統；而為了嵌入式市場亦開發了許多的 OS。現在 Cortex-M3 上支援了如表 17-5 所示的一些 OS。

<sup>1</sup> 產品名稱為列在表內左邊公司的註冊商標。

表 17-5 支援 Cortex-M3 的嵌入式作業系統的例子

公司	產品 <sup>2</sup>
FreeRTOS ( <a href="http://www.freertos.org">www.freertos.org</a> )	FreeRTOS
Express Logic ( <a href="http://www.expresslogic.com">www.expresslogic.com</a> )	ThreadX(TM)RTOS
Micrium ( <a href="http://www.micrium.com">www.micrium.com</a> )	μC/OS-II
Accelerated Technology ( <a href="http://www.Acceleratedtechnology.com">www.Acceleratedtechnology.com</a> )	Nucleus
Pumpkin Inc. ( <a href="http://www.pumpkininc.com">www.pumpkininc.com</a> )	Salvo RTOS
CMX Systems ( <a href="http://www.cmx.com">www.cmx.com</a> )	CMX-RTX
Keil ( <a href="http://www.keil.com">www.keil.com</a> )	ARTX-ARM
Segger ( <a href="http://www.segger.com">www.segger.com</a> )	embOS
IAR Systems ( <a href="http://www.iar.com">www.iar.com</a> )	IAR PowerPac for ARM

# 18

## Chapter

# 從 ARM7 移植應用程式到 Cortex-M3

本章內容包括：

- ✓ 概觀
- ✓ 系統特色
- ✓ 組合語言檔案
- ✓ C 程式檔案
- ✓ 預先編譯的目的檔
- ✓ 最佳化

## 概觀

對許多工程師而言，移植現有程式到新的架構上是一件常見的工作。隨著 Cortex-M3 產品開始在市場出現，有許多人需要面對移植 ARM7TDMI(於下文中稱之為 ARM7)程式到 Cortex-M3 的挑戰。本章對從 ARM7 移植應用程式到 Cortex-M3 所牽涉到的各方面事情，作了一些評估。

當你從 ARM7 移植到 Cortex-M3 時，要考慮幾個地方：

- ◆ 系統特色
- ◆ 組合語言檔案
- ◆ C 程式檔案
- ◆ 最佳化

2 產品名稱為列在表內左邊公司的註冊商標。

總而言之，低階程式例如硬體控制、工作管理、與例外處理程式等需要最多的改變，而應用程式在正常情形下，稍作修改與再編譯之後，即可移植。

## 系統特色

基於 ARM7 的系統與基於 Cortex-M3 的系統之間有一些系統特色的差異(例如：記憶體映射、中斷、MPU、系統控制、與運算模式等)。

### 記憶體映射

在不同的微控制器之間移植程式最明顯的修改目標是記憶體映射的差異。在 ARM7 裡，記憶體與周邊可能在幾乎任何的位址裡面出現，而 Cortex-M3 處理器有一個預先定義的記憶體映射。記憶體位址的差異通常在編譯與連結的階段解決；周邊程式的移植可能更費時間，原因是周邊的程式師模型可能會全然不同。在這樣的情形下，設備的驅動程式可能需要全部重寫。

許多 ARM7 產品提供了一個記憶體重映射的特性，故向量表在驅動後可以重映射到 SRAM。在 Cortex-M3 裡，可使用 NVIC 暫存器重定位向量表，故不再需要作記憶體重映射。因此，在許多 Cortex-M3 產品中，並沒有記憶體重映射的特性。

ARM7 裡支援的 big endian 與 Cortex-M3 所支援的不同。程式檔案可重新編譯以適合新的 big endian 系統，但硬編碼的查看表可能需要在移植過程中作轉換。

在 ARM720T 裡，以及例如 ARM9 等一些後來的 ARM 處理器裡，有一個稱作高向量 (high vector) 的特性，此特性允許將向量表配置於 0xFFFFF0000。此特性是為了支援 Windows CE，但在 Cortex-M3 裡並沒有這個特性。

### 中斷

第二個修改目標是中斷控制器的差異。用來控制中斷控制器的程式碼，例如致能或者除能中斷，將需要改變。除此之外，需要新的程式以設定各種中斷的中斷優先權等級與向量位址。

中斷返回的方式也改變了。這需要於組譯程式中修改中斷返回，如果使用了 C 語言，則可能需要調整編譯的假指令。

對中斷的致能與除能，先前藉著修改 CPSR 來達成，則以設定中斷遮罩暫存器來取代。

在 Cortex-M3 裡，一些暫存器藉由進堆疊與去堆疊的機制而自動保留。因此，可以減少或是刪去一些軟體的堆疊運算。然而，就 FIQ 處理程式的例子來說，傳統的 ARM 核心有分開的暫存器以作 FIQ(R8-R11)。不需將這些暫存器推進堆疊，FIQ 就可以使用他們。但是，在 Cortex-M3 裡，這些暫存器並沒有自動被堆入堆疊，所以當把 FIQ 處理程式的移植到 Cortex-M3 時，需要改變處理程式的暫存器或者需要加上一個進堆疊步驟。

作巢狀中斷處理的程式可以被去除。在 Cortex-M3 裡，NVIC 有內建的巢狀中斷處理。

在錯誤的處理上也有一些差異。Cortex-M3 提供了各種錯誤狀態暫存器，可以指出造成錯誤的原因。除此之外，Cortex-M3 定義了新的錯誤類型(例如，進堆疊與去堆疊錯誤、記憶體管理錯誤、與硬錯誤等)。因此，將需要重寫錯誤處理程式。

### MPU

MPU 程式模型是另一個需要新的程式設定的系統區塊。基於 ARM7TDMI/ARM7TDMI-S 的微控制器產品並沒有 MPU，故將程式移植到 Cortex-M3 應該不致造成問題。但是，基於 ARM720T 的產品有一個記憶體管理單元(Memory Management Unit, MMU)，其功能有別於 Cortex-M3 上的 MPU。如果應用程式需要使用 MMU(例如在虛擬體系統中)，則應用程式不能移植到 Cortex-M3。

### 系統控制

當你移植應用程式時，系統控制是另一個主要需要留意的地方。Cortex-M3 有一個內建的指令以進入睡眠模式。除此之外，Cortex-M3 產品裡面的系統控制器可能會與 ARM7 產品完全地不同，所以牽涉到系統管理特性的功能將需要重寫。

## 運算模式

在 ARM7 中有 7 種運算模式，這些運算模式在 Cortex-M3 裡已改變為各種例外(參考表 18-1)。

表 18-1 ARM7TDMI 例外與模式與 Cortex-M3 相關的對照

ARM7 裡的模式與例外	Cortex-M3 裡相關的模式與例外
Supervisor (預設的)	特權的, 執行緒
Supervisor(軟體中斷)	特權的, SVC
FIQ	特權的, 中斷
IRQ	特權的, 中斷
Abort (預先擷取)	特權的, 汇流排錯誤例外
Abort (資料)	特權的, 汇流排錯誤例外
未定義的	特權的, 用法錯誤例外
系統	特權的, 執行緒
用戶	用戶存取(無特權的), 執行緒

ARM7 裡的 FIQ 可以當作 Cortex-M3 裡面正常的 IRQ 來移植，因為在 Cortex-M3 裡，我們可以設定特定中斷的優先權為最高，讓它可以強佔其他的例外，恰如 ARM7 裡的 FIQ 一樣。然而，因為 ARM7 內 FIQ 暫存器庫與 Cortex-M3 內被堆疊的暫存器之間的差異，所以需要改變使用在 FIQ 處理程式裡的暫存器，或者將這些暫存器手動地保留到堆疊。

### FIQ 與 NMI

許多工程師可能預期 ARM7 裡的 FIQ 可以直接對照 Cortex-M3 裡的 NMI。在一些應用上這是可能的；但是當你移植把 NMI 作 FIQ 使用的應用程式時，需要特別留心一些 FIQ 與 NMI 之間不同的地方。

首先，NMI 不能被除能；而在 ARM7 上，可設定 CPSR 裡的 F-bit 以除能 FIQ。所以，在 Cortex-M3 裡 NMI 處理程式可能在驅動時期就啟始；而在 ARM7 裡，在重置時 FIQ 會被除能。

第二，在 Cortex-M3 上不能使用 NMI 處理程式裡的 SVC；但在 ARM7 上可以使用 FIQ 處理程式裡的 SWI。在 ARM7 上執行 FIQ 處理程式的時期，可能會有其他的例外發生(IRQ 則除外，因為作 FIQ 服務時，I-bit 會被自動地設定)。然而，在 Cortex-M3 上，於 NMI 處理程式內的錯誤例外，可能會造成處理器出現鎖住。

## 組合語言檔案

移植組合語言檔案時，會隨著檔案是否為 ARM 狀態或 Thumb 狀態而定。

### Thumb 狀態

如果檔案為 Thumb 狀態，這樣子的情形會非常簡單。在大多數的情形之下，可以直接重用檔案而不會有任何問題。然而，在 ARM7 裡的一些 Thumb 指令在 Cortex-M3 裡並不被支援：

- ◆ 任何試圖切換到 ARM 狀態的程式
- ◆ SWI 被 SVC 所取代(注意：其使用模式也同樣改變了)

最後，需要確定程式存取堆疊僅可採取全遞減的堆疊運算。在 ARM7TDMI 裡，雖然不常見，但是有可能實作不同的堆疊模式(例如，全遞增)。

### ARM 狀態

ARM 狀態的情形比較複雜，有如下幾個情境：

- ◆ 向量表：在 ARM7 裡，向量表從位址 0x0 開始，並且由跳躍指令組成。在 Cortex-M3 裡，向量表中包含有堆疊指標的初始值、重置向量位址，跟著是各個例外處理程式的位址。因為這些不同，向量表將需要完全地改寫。
- ◆ 暫存器初始化：在 ARM7 裡，常常需要為了不同的模式去初始化不同的暫存器。例如，在 ARM7 裡有堆疊指標庫(R13)、連結暫存器(R14)、和保留的程式狀態暫存器(Saved Program Status Register, SPSR)等。因為 Cortex-M3 有不同的程式師模型，將需要更改暫存器初始化程式。事實上，Cortex-M3 上的暫存器初始化程式將會更加簡單，因為並不需要切換處理器到不同的模式。
- ◆ 模式切換與狀態切換程式：因為在 Cortex-M3 裡的運算模式定義異於 ARM7，故模式切換程式需要被移除；同樣地這也適用 ARM/Thumb 狀態的切換程式。

- ◆ **中斷致能與除能**: 在 ARM7 裡, 可以藉著清除或設定 CPSR 裡的 I-bit 以致能或除能中斷。在 Cortex-M3 裡, 這可以藉由清除或設定 PRIMASK 或 FAULTMASK 等中斷遮罩暫存器。再者因為並無 FIQ 輸入, 故在 Cortex-M3 裡並無 F-bit。
- ◆ **輔助處理器的存取**: Cortex-M3 上並沒有支援輔助處理器, 故這一類的運算不可以被移植。
- ◆ **中斷處理程式與中斷返回**: 在 ARM7 裡, 中斷處理程式的第一個指令在向量表裡, 通常就是跳到實際中斷處理程式的一個跳躍指令。在 Cortex-M3 裡, 不再需要此步驟。ARM7 依賴手動去調整返回的程式計數器以作中斷返回。在 Cortex-M3 裡, 正確調整好的程式計數器會保留在堆疊中, 並且藉著載入 EXC\_RETURN 到程式計數器來觸發中斷返回。在 Cortex-M3 上不可使用 MOVS 與 SUBS 等指令作中斷返回。因為這些不同, 在移植時需要修改中斷處理程式與中斷返回程式碼。
- ◆ **巢狀中斷支援程式**: 在 ARM7 裡, 當需要巢狀中斷時, 通常 IRQ 處理程式將需要切換處理器到系統模式並且再次致能中斷。在 Cortex-M3 裡並不需要這樣。
- ◆ **FIQ 處理程式**: 如果移植了 FIQ 處理程式, 你可能需要增加額外的步驟以將 R8-R11 的內容保留到堆疊記憶體。在 ARM7 裡, R8-R12 會被庫存(banked), 故 FIQ 處理程式可以略過把這些暫存器推入堆疊的動作。然而, 在 Cortex-M3 上, R0-R3 與 R12 自動地被保留在堆疊, 但 R8-R11 則否。
- ◆ **軟體中斷(Software interrupt, SWI)處理程式**: SWI 被 SVC 所取代。然而, 當移植 SWI 處理程式到 SVC 時, 為 SWI 指令提取傳遞參數的程式碼需要被更新。呼叫的 SVC 指令的位址可以在置於堆疊的 PC 裡找到, 此不同於 ARM7 裡的 SWI, 其程式計數器的位址需要由連結暫存器來決定。
- ◆ **互換(swap)指令(SWP)**: 在 Cortex-M3 裡並沒有互換指令。如果互換指令被用在號誌, 則需要以獨佔存取指令來取代。這將需要重寫號誌程式。如果指令僅純粹作為資料傳輸使用, 則可以使用多重記憶體存取指令來取代。

◆ **存取 CPSR、SPSR**: ARM7 裡的 CPSR 被 Cortex-M3 裡的 xPSR 所取代;而 SPSR 則被去除。如果應用程式想要存取處理器旗標的現有值, 則可以讀取 APSR 來取代。如果例外處理程式想要取得例外發生之前的 PSR, 可以在堆疊記憶體裡找到, 因為在接受一個中斷時, xPSR 的值會被自動保留在堆疊。故在 Cortex-M3 裡並不需要 SPSR。

◆ **條件式執行**: 在 ARM7 裡, 大多數的 ARM 指令支援條件式執行;而大多數的 Thumb-2 指令在指令碼內並沒有條件欄位。當將這些程式移植到 Cortex-M3 的時候, 在某些情形下我們可以使用 IF-THEN 指令區塊, 否則我們可能需要插入跳躍指令以產生條件式執行的程式。以 IT 指令區塊取代條件式執行程式的潛在問題, 就是可能會增加程式大小, 並且因此造成一些小問題, 例如在程式某些部分的載入/儲存運算可能會超出指令的存取範圍。

◆ **在牽涉到使用現在程式計數器作計算的程式裡去使用程式計數器之值**: 在 ARM7 執行 ARM 程式時, 指令執行期間 PC 的讀取值為指令位址值加 8。這是因為 ARM7 有三個管線階段, 故在執行階段去讀取 PC 之時, 程式計數器已經增加了兩次, 每次 4 bytes。當移植處理了 PC 值的程式到 Cortex-M3, 因為將會是 Thumb 程式, 故程式計數器的位移值僅只為 4。

◆ **使用 R13 之值**: 在 ARM7 裡, 堆疊指標 R13 有 32 位元;而在 Cortex-M3 處理器裡, 堆疊指標的最低 2 個位元永遠會被強迫設為 0。因此, 雖然可能性很低, 但若是將 R13 當作資料暫存器使用, 就需要修改程式, 否則將會遺失最低兩位元的資料。

對其餘的 ARM 程式碼, 我們可以試著將它編譯為 Thumb/Thumb-2, 並看是否需要進一步修改。例如, ARM7 裡一些前索引(pre-index)與後索引(post-index)記憶體存取指令, 在 Cortex-M3 裡並不支援, 故需要以多重指令來重新編碼;一些程式可能會有過大的跳躍範圍或過大的立即資料值, 而不能編譯為 Thumb 程式, 因此需要手動地修改為 Thumb-2 程式。

## C 程式檔案

移植 C 程式檔案遠比移植組合語言檔案容易。在大部分的例子中，以 C 撰寫的應用程式可以 Cortex-M3 重新編譯而不會有問題。然而，有幾個潛在需要修正的地方：

- ◆ **行內組譯器**: 一些 C 程式碼可能含有需要修改的行內組合語言程式。此程式碼可以經由關鍵字 `_asm` 輕易地找出。如果使用了 RVDS/RVCT 3.0 或更新的版本，則應該改變為嵌入式組譯器(Embedded Assembler)。
- ◆ **中斷處理程式**: 在 C 程式裡你可以使用 `_irq` 去產生可用於 ARM7 的中斷處理程式。因為 ARM7 與 Cortex-M3 之間中斷行為(例如保留的暫存器與中斷返回)的差異，隨著使用的開發工具而定，可能需要移除關鍵字 `_irq`。(然而，於 RVDS 3.0 與 RVCT 3.0 中，`_irq` 增加了對 Cortex-M3 的支援，因此，為了能夠清楚標示，推薦使用假指令 `_irq`。)

## 預先編譯的目的檔

大部分 C 編譯程式會為了各種函數庫與啟動程式，提供預先編譯的目的檔。因為運算模式與狀態的不同，一些預先編譯的檔案(例如傳統 ARM 核心的啟動程式)不能在 Cortex-M3 上使用。其中有些可以取得原始程式，並且以 Thumb-2 程式加以重新編譯。細節請參考你的工具供應商的文件。

## 最佳化

讓程式可以工作在 Cortex-M3 之後，你可能可以更進一步地改良它，以獲得更佳表現與更少量記憶體的使用。有些地方應該加以探討：

- ◆ **Thumb-2 指令的使用**: 例如，如果一個 16 位元 Thumb 指令從一個暫存器傳輸資料到另一個暫存器，並在其上進行資料處理的運算，則有可能以單一 Thumb-2 指令來取代此指令序列。這樣可以減少運算所需的時脈週期。
- ◆ **Bit band**: 如果把周邊置於 bit-band 區域裡，則對控制暫存器位元的存取，可以藉由 bit-band alias 存取位元而大為簡化。

- ◆ **乘法與除法**: 需要除法運算的程式，例如轉換數值為十進位作輸出，可以修改以使用 Cortex-M3 裡的除法指令。作更大數值資料的乘法時，Cortex-M3 裡的多重指令如 UMULL、SMULL、MLA、MLS、UMLAL、與 SMLAL 等可用來減低程式的複雜度。
- ◆ **立即值資料**: 一些不能在 Thumb 指令編碼的立即值資料，可以使用 Thumb-2 指令來產生。
- ◆ **跳躍**: 一些在 Thumb 程式裡不能編碼的較長距離的跳躍(通常最後會採用多重的跳躍步驟)可以使用 Thumb-2 指令來編碼。
- ◆ **布林資料**: 多重的布林資料(是 0 或 1)可以在 bit-band 區域包裝為單一 byte/half word/word 以節省記憶體空間；後續可以經由 bit-band alias 來存取。
- ◆ **位元欄位的處理**: Cortex-M3 提供一些指令作為位元欄位的處理，其中包括了：UBFX、SBFX、BFI、BFC、RBIT 等。它們可以簡化許多周邊編程、資料封裝的形成與提取、序列資料通信等的程式碼。
- ◆ **IT 指令區塊**: 一些短距離跳躍可能可以 IT 指令區塊來取代。藉此做法，我們可以避免跳躍期間清除管線時浪費的時脈週期。
- ◆ **ARM/Thumb 狀態切換**: 在一些情形下，ARM 開發者會把程式分開為各種檔案，其中一些編譯為 ARM 程式，另外一些則編譯為 Thumb 程式。在執行速度並非緊要而想要增加程式密度時，通常必須這樣做。因為 Cortex-M3 裡的 Thumb-2 程式，不再需要這個步驟，所以可以節省一些狀態切換的成本，產生的程式更短，更少成本，並可能有更少的程式檔案。

Chapter

# 19

## 使用 GNU 工具鏈開始 Cortex-M3 的開發

本章內容包括：

- ✓ 背景
- ✓ 取得 GNU 工具鏈
- ✓ 開發流程
- ✓ 範例
- ✓ 存取特殊暫存器
- ✓ 使用未支援的指令
- ✓ GNU C 編譯器裡的行內組譯器

### 背景

很多人使用 GNU 工具鏈作 ARM 產品的開發，並且有一些 ARM 的開發工具是根基於 GNU 工具鏈。GNU 工具鏈支援 Cortex-M3，並且現在可以免費由 CodeSourcery 取得([www.codesourcery.com](http://www.codesourcery.com))。在不久的未來，主要的 GNU C 編譯器開發將會包括對 Cortex-M3 的支援。

本章僅只介紹使用 GNU 工具鏈的最基本步驟。工具鏈詳細的使用可在網際網路取得，並且超出了本書的範圍。

GNU 組譯器的組譯器語法(在 GNU 工具鏈裡的 AS)與 ARM 組譯器稍微不同。這些差異包括了宣告、編譯假指令、註解等。因此，以 ARM RealView Development 工具撰寫的組合語言程式，在使用於 GNU 工具鏈之前需要修改。

MEMO.

## 取得 GNU 工具鏈

編譯好的 GNU 工具鏈版本可以從 [www.codesourcery.com/gnu\\_toolchains/arm/](http://www.codesourcery.com/gnu_toolchains/arm/) 下載。有一些二進位檔可以取得；以最簡單的使用而言，讓我們選擇適合 EABI 但無特定嵌入式 OS 的二進位檔為目標平台。你可取得在各種開發平台如 Windows 與 Linux 的工具鏈。本章範例適用於上述兩版本。

## 開發流程

跟 ARM 工具相同，GNU 工具鏈包括一個編譯器、一個組譯器，與一個連結器。工具程式允許包含了 C 與組合語言撰寫的原始程式的專案(見圖 19-1)。

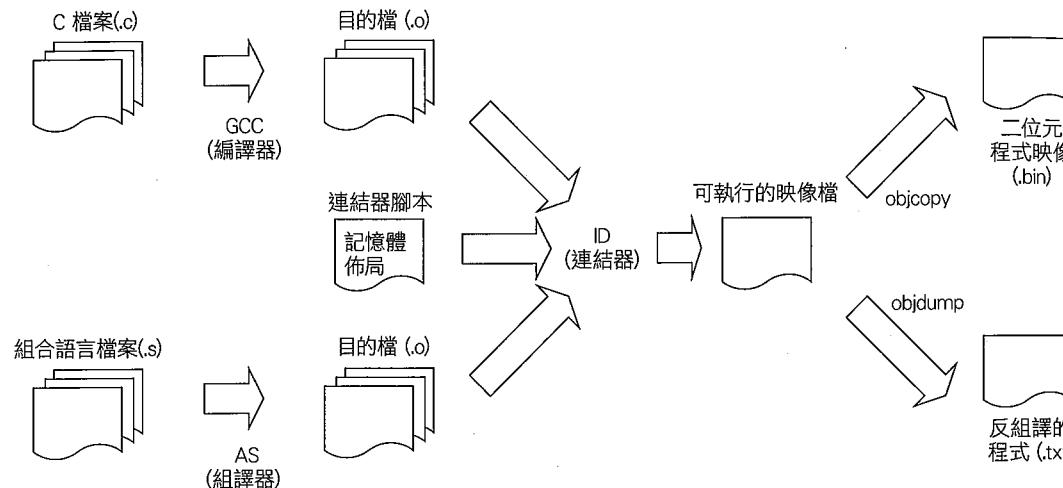


圖 19-1 根據 GNU 工具鏈的開發流程範例

有各種工具鏈版本可適用在不同的應用環境(Symbian, Linux, EABI<sup>1</sup> 等)。程式的檔案名稱通常有一個前綴字，隨著你的工具鏈目標選擇而定。例如，如果使用了 EABI 環境，則 GCC 命令可能為 arm-xxxx-eabi-gcc。下面的範例使用表 19-1 所示來自 CodeSourcery GNU ARM Tool Chain 的命令。

<sup>1</sup> ARM 架構的嵌入式應用二進制介面(Embedded Application Binary Interface, EABI)，可執行檔需要合乎此規格才可使用在不同的開發工具集裡。

表 19-1 CodeSourcery Tool Chain 的命令名稱

功能	命令 (EABI 版本)
組譯器	arm-none-eabi-as
C 編譯器	arm-none-eabi-gcc
連結器	arm-none-eabi-ld
二進位元映像產生器	arm-none-eabi-objcopy
反組譯器	arm-none-eabi-objdump

注意來自其他供應商的工具鏈的命令名稱差異。

開發流程中的連結器腳本(script)是選擇性的，但是當記憶體映射變為更複雜時通常就會是必須的。

## 範例

讓我們看一下幾個使用 GNU 工具鏈的例子。

### 範例 1：第一個程式

作為開始，讓我們嘗試第十章討論的計算  $10+9+8\dots+1$  的簡單組合語言程式：

```

=====
example1.s =====
/* 定義常數 */
.equ      STACK_TOP, 0x20000800
.text
.global _start
.code 16
.syntax unified
/* .thumbfunc */
/* .thumbfunc 僅需用在 2006Q3-26 之前
   CodeSourcery GNU 工具鏈*/
_start:
.word    STACK_TOP, start
.type    start, function
/* 主程式開始*/
start:
    movs    r0, #10
    movs    r1, #0
/* 計算 10 + 9 + 8 . . . +1 */
loop:
    adds    r1, r0

```

```

subs    r0 , #1
bne    loop
/* 現在結果在 R1 中 */
deadloop:
    b      deadloop
.end
=====end of file =====

```

- ◆ 此處.word 假指令幫助我們定義堆疊指標初始值為 0x20000800、重置向量為 start。
- ◆ .text 為預先定義的假指令，顯示其為需要組譯的程式區域。
- ◆ .global 允許標籤\_start 在必要時與其它目的檔共用。
- ◆ .code 16 顯示為 Thumb 的程式碼。
- ◆ .syntax unified 顯示使用了統一的組合語言語法。
- ◆ \_start 為一個標籤，顯示程式區域的起始點
- ◆ start 為一個分開的標籤，顯示了重置處理程式
- ◆ .type start, function 告知符號 start 為一個函數。對於向量表裡所有的例外向量這 是需要的；否則，組譯器會設定向量的 LSB 為 0。
- ◆ .end 顯示程式的結束。

與 ARM 組譯器不同，GNU 裡的標籤跟隨著一個冒號(:)。註解以/\*與\*/來引用，假指令以句點(.)作前綴。

注意，重置向量(start)被定義為 thumb 程式內(.code 16)的一個函數(.type start function)。這樣做則可迫使重置向量的 LSB 為 1 以顯示它以 Thumb 狀態開始；否則，處理器將試圖以 ARM 模式開始，而造成硬錯誤。要組譯這個檔案，我們可以下面的命令一樣，使用 as。

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
```

這樣會產生目的檔 example1.o。選項-mcpu 與-mthumb 定義所使用的指令集。連結階段可如下所示以 ld 來達成。

```
$> arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
```

接著可以如下使用 Object Copy (objcopy)製造二進位檔：

```
$> arm-none-eabi-objcopy -Obinary example1.out example1.bin
```

我們可以使用 Object Dump (objdump)製造一個反組譯的程式列表檔以檢查輸出：

```
$> arm-none-eabi-objdump -S example1.out > example1.list
```

程式列表檔看起來如下：

```

example1.out:          file format elf32-littlearm
Disassembly of section .text:
00000000 <_start>:
 0: 0800    lsrs    r0, r0, #32
 2: 2000    movs    r0, #0
 4: 0009    lsls    r1, r1, #0
 ...
00000008 <start>:
 8: 200a    movs    r0, #10
 a: 2100    movs    r1, #0
0000000C <loop>:
 c: 1809    adds    r1, r1, r0
 e: 3801    subs    r0, #1
 10: dffc    bne.n   c <loop>
00000012 <deadloop>:
 12: e7fe    b.n    12 <deadloop>

```

## 範例 2：連結多個檔案

如之前所提及，我們可以製造多個目的檔並且將他們連結在一起。此處的範例裡，我們有兩個組合程式檔案：example2a.s 與 example2b.s；example2a.s 僅包含向量表，而 example2b.s 包含程式碼。.global 用在檔案之間傳位址：

```
===== example2a.s =====
/* 定義常數 */
.equ STACK_TOP, 0x20000800
.global vectors_table
.global _start
.global nmi_handler
.code 16
.syntax unified

vector_table:
.word STACK_TOP, start, nmi_handler, 0x00000000
.end

===== end of file =====
===== example2b.s =====
/* 主程式 */
.text
.global _start
.global start
.global nmi_handler
.code 16
.syntax unified
.type start, function
.type nmi_handler, function

_start:
/* 主程式開始 */
.start:
    movs r0, #10
    movs r1, #0
    /* 計算 10+9+8+...+1 */
loop:
    adds r1, r0
    subs r0, #1
    bne loop
    /* 現在結果存放於 R1 */
deadloop:
    b deadloop
    /* 作為說明例子的假 NMI 處理程式 */
    bx lr
.end

===== end of file =====
```

使用下列步驟以製造可執行的映像檔：

1. 組譯 example2a.s：

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2a.s -o example2a.o
```

2. 組譯 example2b.s：

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2b.s -o example2b.o
```

3. 連結目的檔為一個映像檔。注意，命令列的目的檔的次序，將會影響在最後可執行映像中物件的次序：

```
$> arm-none-eabi-ld -Ttext 0x0 -o example2.out example2a.o example2b.o
```

4. 接著產生了二進位檔案：

```
$> arm-none-eabi-objcopy -Obinary example2.out example2.bin
```

5. 與先前例子相同，我們產生了一個列表檔以檢查組譯的映像是否正確：

```
$> arm-none-eabi-objdump -S example2.out > example2.list
```

當檔案的數量增加時，可使用 UNIX 的 makefile 來簡化編譯程序。個別的開發套件可能也有內建的功能，使得編譯程式更為簡易。

### 範例 3：一個簡單的"Hello World"程式

在這裡我們嘗試一個更有點野心的"Hello World"程式。(注意：在此我們跳過了 UART 的初始化；你需要加上自己的 UART 初始化程式才能測試此程式。在第二十章裡提供了以 C 語言撰寫的 UART 初始化的例子。)

```
===== example3a.s =====
/* 定義常數 */
.equ STACK_TOP, 0x20000800
.global vectors_table
.global _start
.code 16
.syntax unified
vectors_table:
.word STACK_TOP, _start
.end
```

```

=====
example3b.s =====
.text
.global _start
.code 16
.syntax unified
.type _start, function
_start:
/* 主程式開始 */
movs r0, #0
movs r1, #0
movs r2, #0
movs r3, #0
movs r4, #0
movs r5, #0

ldr r0, =hello
bl puts
movs r0, #0x4
bl putc
deadloop:
b deadloop
hello:
.ascii "Hello\n"
.byte 0
.align

puts:/* 傳送字串到 UART 的副程式 */
/* 輸入 r0 = 字串的起始位址 */
/* 字串需要以空字元(null)作結束 */
push {r0, r1, lr} /* 保留暫存器值 */
mov r1, r0 /* 複製位址到 R1, 因為 */
/* R0 將會作為 putc 的輸入 */

putloop:
ldrb.w r0, [r1], #1 /* 讀進一個字元並且增加位址值 */
cbz r0, putsloopexit /*如果為空字元, 則結束 */
bl putc
b putsloop
putsloopexit:
pop {r0, r1, pc} /* 返回 */

.equ UART0_DATA, 0x4000C000
.equ UART0_FLAG, 0x4000C018

putc:/* 經由 UART 傳送字元的副程式 */
/* 輸入 R0 = 欲傳送的字元 */
push {r1, r2, r3, lr} /* 保留暫存器值 */
LDR r1, =UART0_FLAG
putcwaitloop:

```

```

ldr r2, [r1] /* 取得狀態旗標 */
tst.w r2, #0x20 /* 檢查傳送緩衝器滿溢旗標位元 */
bne putcwaitloop /* 若忙碌則回到迴圈 */
ldr r1, =UART0_DATA /* 否則輸出資料以傳送緩衝器內容 */
str r0, [r1]
pop {r1, r2, r3, pc} /* 返回 */
.end

=====

```

在這個例子裡，我們使用了.ascii 與.byte 以製造一個空字元結尾的字串。在定義了字串之後，我們使用了.align 以確定下一個指令會在正確的位置開始；否則，組譯器可能將下一個指令放在一個未對齊的位置。

下列步驟可用來編譯程式、製造二進位影像、反組譯輸出檔：

```

$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example3a.s -o example3a.o
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example3b.s -o example3b.o
$> arm-none-eabi-ld -Ttext 0x0 -o example3.out example3a.o example3b.o
$> arm-none-eabi-objcopy -Obinary example3.out example3.bin
$> arm-none-eabi-objdump -S example3.out > example3.list

```

## 範例 4：RAM 裡面的資料

我們經常會把資料存放在 SRAM 裡。下面簡單的例子顯示了需要的步驟：

```

=====
example4.s =====
.equ STACK_TOP, 0x20000800
.text
.global _start
.code 16
.syntax unified
_start:
.word STACK_TOP, start
.type start, function
/* 主程式開始 */
start:
movs r0, #10
movs r1, #0
/* 計算 10+9+8+...+1 */
loop:
adds r1, r0
subs r0, #1

```

```
bne      loop
/* 現在結果存放於 R1 */
ldr      r0, =Result
str      r1, [r0]
deadloop:
b       deadloop
/* 資料區 */
.data
Result:
.word 0
.end
===== end of file =====
```

在程式裡，使用假指令.data 以產生資料區。在此區域內，使用假指令.word 以保留一個標籤為 Result 的空間。接著，程式可以定義的標籤名 Result 來存取這個空間。

欲連結此程式，我們需要告知連結器 RAM 所在處。這個可以使用-Tdata 選項來達成，此選項設定資料區到所需的位置：

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example4.s -o example4.o
$> arm-none-eabi-ld -Ttext 0x0 -Tdata 0x20000000 -o example4.out example4.o
$> arm-none-eabi-objcopy -Obinary -R .data example4.out example4.bin
$> arm-none-eabi-objdump -S example4.out > example4.list
```

亦請注意，在此例子裡在執行 objcopy 時，使用了-R .data 選項。這樣可以避免將資料記憶體區包括在二進位的輸出檔裡。

## 範例 5：僅有 C 而無組合語言的檔案

GNU 工具鏈裡的一個主要的元件是 C 編譯器。在這個例子裡，全部可執行的程式，甚至連重置向量與堆疊指標初始值，都以 C 來撰寫。除此之外，需要一個連結器腳本以將各區域放置於適當的地方。首先，讓我們看 C 程式檔：

```
===== example5.C =====
#define STACK_TOP 0x20000800
#define NVIC_CCR ((volatile unsigned long *) (0xE000ED14))
// 宣告函數
void myputs(char *string1);
void myputc(char mychar);
int main(void);
void nmi_handler(void);
```

```
void hardfault_handler(void);
// 定義向量表
__attribute__((section("vectors")))
void (* const VectorArray[])(void) = {
    STACK_TOP,
    main,
    nmi_handler,
    hardfault_handler
};

// 主程式開始
int main(void)
{
    const char *helloworld[] = "Hello world\n";
    *NVIC_CCR = *NVIC_CCR | 0x200; /* 設定 NVIC 裡的 STKALIGN */
    myputs(*helloworld);
    while (1);
    return (0);
}

// 函數
void myputs(char *string1)
{
    char mychar;
    int j;
    j = 0;
    do {
        mychar = string1[j];
        if (mychar != 0) {
            myputc(mychar);
            j++;
        }
    } while (mychar != 0);
    return;
}
void myputc (char mychar)
{
#define UART0_DATA ((volatile unsigned long *) (0x4000C000))
#define UART0_FLAG ((volatile unsigned long *) (0x4000C018))

    // 等待忙碌旗標清除為止
    while ((*UART0_FLAG & 0x20) != 0);
    // 輸出字元到 UART
    *UART0_DATA = mychar;
    return;
}

// 假的處理程式
void nmi_handler (void)
```

接下頁

```
{
    return;
}

void hardfault_handler (void)
{
    return;
}

=====
end of file =====
```

其中向量表是使用 \_\_attribute\_\_ 程式碼來定義。此檔案並沒有說明向量表在何處，此為連結器腳本的工作。一個簡單的連結器腳本可能如下面的 simple.ld：

```
===== simple.ld =====
/* MEMORY 命令：定義可用的記憶體區域 */
/* 此部分定義了允許連結器放進資料 */
/* 的各種記憶體區域。此為選擇性的特性， */
/* 但是有用，因為連結器可以在程式過大 */
/* 而放不下的時候警告你 */
MEMORY
{
    /* ROM 為一個可讀的(r)，可執行的區域(x) */
    rom (rx)      : ORIGIN = 0, LENGTH = 2M

    /* RAM 為一個可讀的(r)，可寫的(w)，並可執行的區域(x) */
    ram (rwx)     : ORIGIN = 0x20000000, LENGTH = 4M
}

/* SECTION 命令：定義輸入節區到輸出節區的映射 */
SECTIONS
{
    . = 0x0;           /* 從 0x00000000 */
    .text : {
        *(vectors)    /* 向量表 */
        *(.text)       /* 程式碼 */
        *(.rodata)     /* 唯讀資料 */
    }
    . = 0x20000000;   /* 從 0x20000000 */
    .data : {
        *(.data)       /* 資料記憶體 */
    }
    .bss : {
        *(.bss)       /* 填上 0 的執行期間配置的資料記憶體 */
    }
}
=====
end of file =====
```

接著在編譯階段，傳送記憶體映射資訊到編譯器：

```
$> arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb example5.c -nostartfiles -T simple.
ld -o example5.o
```

接著使用連接器腳本，再度連結輸出的目的檔：

```
$> arm-none-eabi-ld -T simple.ld -o example5.out example5.o
```

在此情形下我們僅有一個來源檔，故可省略連結的階段。最後，可以產生二進位檔與反組譯檔：

```
$> arm-none-eabi-objcopy -O binary example5.out example5.bin
$> arm-none-eabi-objdump -S example5.out > example5.list
```

在這個例子裡我們使用了一個稱作-nostartfiles 的編譯器選項。這會避免 C 編譯器將啟動程式庫函數插入可執行的映像。這樣做的一個理由是可減少程式映像的大小。然而，主要使用此選項的理由是 GNU 工具鏈的啟動程式庫程式，會隨發佈的供應商而定。其中有一些可能不適合 Cortex-M3 使用，而是為例如 ARM7 的傳統 ARM 處理器所編譯(使用了 ARM 程式碼而非 Thumb 程式碼)。

但是在許多例子下，隨著所使用的應用程式與程式庫而定，可能需要使用啟動程式庫以執行初始化的程序，例如資料區的初始化(例如，在執行應用程式前，資料需要初始化為 0 的區域)。接著即是進行此設定的一個簡單範例。

## 範例 6：僅用 C，以及標準的 C 啟動程式

在正常的情形下，編譯 C 程式時，會將標準 C 程式庫啟動程式自動地包含於輸出裡。這樣可以確保執行時期程式庫會正確地初始化。C 程式庫啟動程式由 GNU 工具鏈來提供；然而，在不同的工具鏈提供者之間其設定可能會有變化。下面的範例根據了 CodeSourcery GNU ARM Tool Chain version 2006q3-26。就此版本，你需要聯絡 CodeSourcery 支援以得到正確的啟動程式目的檔：armv7m-crt0.o，因為此版本提供了一個以 ARM 程式碼而非 Thumb 程式碼編譯的不正確的啟動程式。2006q3-27 及其

後的版本已經修正了此問題。不同的供應商的 GNU 工具鏈版本可能會有不同的啟動程式實作與不同的檔名。請檢查你的工具鏈的文件以決定定義啟動程式的最佳安排。

在我們編譯 C 原始程式之前，範例 5 的 C 程式需要幾個小修正。藉由預設，啟動程式 armv7m-crt0 已經包含一個向量表，並且它的 NMI 處理程式與硬錯誤處理程式的名稱分別定義為\_nmi\_isr 與\_fault\_isr。因此，我們需要從 C 程式移除我們的向量表，並且重新命名 NMI 與硬錯誤處理程式：

```
===== example6.c =====
//宣告函數
void myputs (char *string1);
void myputc (char mychar);
int main (void)
void _nmi_isr (void);
void _fault_isr (void);

//主程式開始
int main (void)
{
    const char *helloworld[] = {"Hello world\n"};
    myputs (*helloworld);
    while (1);
    return (0);
}

//函數
void myputs (char *string1)
{
    char mychar;
    int j = 0;
    do {
        mychar = string1[j];
        if (mychar != 0)
            myputc(mychar);
        j++;
    }
} while (mychar != 0);
return;
}

void myputc (char mychar)
{
#define     UART0_DATA      ((volatile unsigned long *) (0x4000C000))
#define     UART0_FLAG      ((volatile unsigned long *) (0x4000C018))
```

```
// 等待直到忙碌旗標清除為止
while ((*UART0_FLAG & 0x20) != 0);
// 輸出字元到 UART
*UART0_DATA = mychar;
return;
}

//假的處理程式
void _nmi_isr (void)
{
    return;
}
void _fault_isr (void)
{
    return;
}

===== end of file =====
```

一些連結器腳本已經包括在 CodeSourcery 的安裝中，可以在目錄 codesourcery/sourcery g++/arm-none-eabi/lib 裡找到。在下面的例子裡，使用了 lm3s8xx-rom.ld 的檔案。這個連結器腳本支援了 Luminary Micro LM3S8XX 系列的設備。

除了現在的目錄之外，當找到 C 程式時，一個稱為 lib 的程式庫次目錄也在現有目錄裡產生；這樣使得程式庫搜尋途徑的設定更為簡單。啟動程式目的檔 armv7m-crt0.o 與需要的連結器腳本被複製到這個 lib 目錄，並且在下面的例子裡，選項-L lib 定義 lib 目錄為程式庫的搜尋途徑。

現在我們可以編譯 C 程式：

```
$> arm-none-eabi-gcc -mcpu = cortex-m3 -mthumb example6.c -L lib -T lm3s8xx-rom.
ld -o example6.out
```

這樣產生並連結輸出目的檔為 example6.out。因為僅有一個目的檔，故二進位檔案可以直接產生：

```
$> arm-none-eabi-objcopy -Obinary example6.out example6.bin
```

產生反組譯程式的方式與先前的例子相同：

```
$> arm-none-eabi-objdump -S example6.out > example6.list
```

## 存取特殊暫存器

CodeSourcery GNU ARM 工具鏈支援了特殊暫存器的存取。特殊暫存器的名字需要小寫。例如：

```
msr      control, r1
msr      r1, control
msr      apsr, R1
mrs      r0, psr
```

## 使用未支援的指令

如果你使用另外的 GNU ARM 工具鏈，可能會出現你所使用的 GNU 組譯器並不支援你想要的組合語言指令的情形。在這樣的情形下，你依然可以使用.word 插入二進位資料型態的指令。例如：

```
.equ DW_MSRR0, 0x8814F380
...
MOV    R0, #0x1
.word DW_MSRR0      /* 此於用戶模式裡設定處理器 */
...
```

## GNU C 編譯器裡的行內組譯器

與 ARM C Compiler 一樣，GNU C Compiler 支援行內組譯器。它的語法稍有不同：

```
_asm (
    "inst1 op1, op2..."      \n"
    "inst2 op1, op2..."      \n"
    ...
    "inst  op1, op2..."      \n"
    : output_operands
    : input_operands
    : clobbered_register_list
);
```

/\* 選擇性的 \*/  
/\* 選擇性的 \*/  
/\* 選擇性的 \*/

例如，一個進入睡眠模式的簡單程式如下所示：

```
void Sleep (void)
{ // 使用 Wait-For-Interrupt 以進入睡眠模式
    _asm (
        "WFI\n"
    );
}
```

如果組譯器程式需要有一個輸入變數與一個輸出變數，例如：在下面程式裡把一個變數除以 5，則可以如此撰寫：

```
unsigned int DataIn, DataOut; /* 輸入與輸出變數 */
...
__asm ("mov    r0, %0\n"
       "mov    r3, #5\n"
       "udiv   r0, r0, r3\n"
       "mov    %1, r0\n"
       : "=r" (DataOut) : "r" (DataIn) : "cc", "r3" );
```

就此程式而言，輸入參數為一個稱作 DataIn 的 C 變數(%0 第一個參數)，並且程式回傳結果到另一個稱作 DataOut 的 C 變數(%1 第二個參數)。行內組譯器程式主動地修改暫存器 r3 並且改變條件旗標 cc，故會被列在竄改(clobbered)暫存器清單裡。

想要得到更多的行內組譯器的例子，可參考網際網路上的 GNU 工具鏈文件：GCC-Inline-Assembly-HOWTO。

Chapter

20

## KEIL RealView 微控制器 開發套件入門

本章內容包括：

- |                             |             |
|-----------------------------|-------------|
| ✓ 概觀                        | ✓ 使用除錯器     |
| ✓ $\mu$ Vision 入門           | ✓ 指令集模擬器    |
| ✓ 藉由 UART 輸出"Hello World"訊息 | ✓ 修改向量表     |
| ✓ 測試軟體                      | ✓ 使用中斷的碼表範例 |

### 概觀

Cortex-M3 有各種可用的商業開發平台。流行的選擇其中之一是 KEIL RealView Microcontroller Development Kit (RealView MDK)。RealView MDK 包括各類的元件：

- ◆  $\mu$ Vision
- ◆ 整合開發環境(IDE)
- ◆ 除錯器
- ◆ 模擬器

MEMO.

### ◆ ARM 的 RealView 編譯工具

1. C/C++編譯器
2. 組譯器
3. 連結器

### ◆ RTX 即時的 Kernel

### ◆ 微控制器詳細的啟動程式

### ◆ 快閃程式設計演算法

### ◆ 程式範例

使用 RealView MDK 學習 Cortex-M3, 並不需要有 Cortex-M3 硬體。  
 $\mu$  Vision 環境包含了一個指令集模擬器, 允許測設並不需要開發板的簡單程式。

RealView MDK 亦可以與其他工具鏈一起使用, 例如:

### ◆ GNU ARM Compiler

### ◆ ARM Development Suite (ADS)

Keil 工具免費的測試版 CD-ROM 可以從 KEIL 網址([www.keil.com](http://www.keil.com))取得。  
 此版本也包括在 Luminary Micro Stellaris Evaluation Kit<sup>1</sup> ([www.luminarmicro.com](http://www.luminarmicro.com))裡面。

## **$\mu$ Vision 入門**

RealView MDK 提供了一些範例, 其中並包含一些 Luminary Micro Stellaris 微控制器產品的例子。這些範例提供了可馬上使用的設備驅動程式庫, 是一套強而有力的集合。要修改這些範例來開始開發你的應用是非常容易的, 或者你可以從零開始開發專案。下面顯示了如何開發的例子。本章的例子根據了 v3.03 beta 版並在 Luminary Micro LM32811 的設備上執行。

<sup>1</sup> Stellaris 為屬於 Luminary Micro 公司的註冊商標。

安裝了 RealView MDK 之後, 你可從程式功能表啟動  $\mu$  Vision。在安裝之後,  $\mu$  Vision 可能會以傳統 ARM 處理器的預設專案啟動。我們可以關閉現有的專案, 並藉著選擇下拉功能表中 New  $\mu$  Vision Project 以開始新專案(參見圖 20-1)。

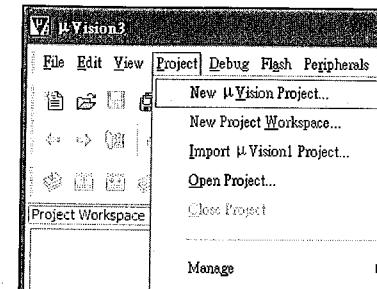


圖 20-1 從程式功能表選擇一個新的專案

此處建立了一個 CortexM3 新的專案目錄(參見圖 20-2)。

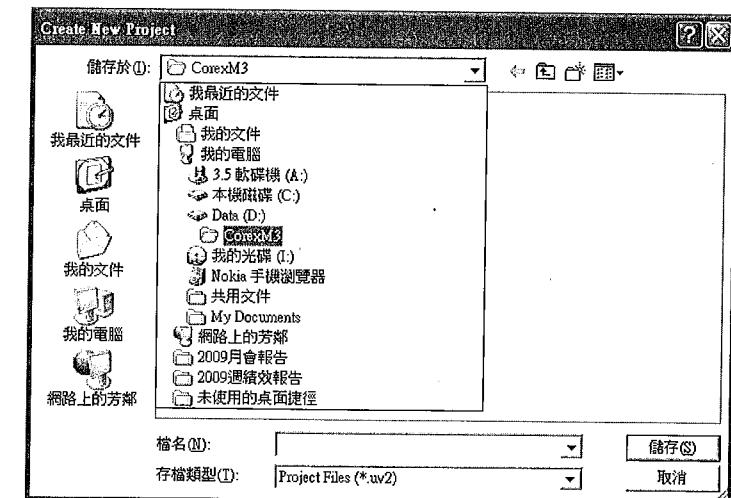


圖 20-2 選擇 CortexM3 專案目錄

現在我們需要為這個專案選擇目標的設備。在此例子裡, 選擇了 LM3S811(參見圖 20-3)。

# Jason 嘴書—EETOP 世界唯一貼

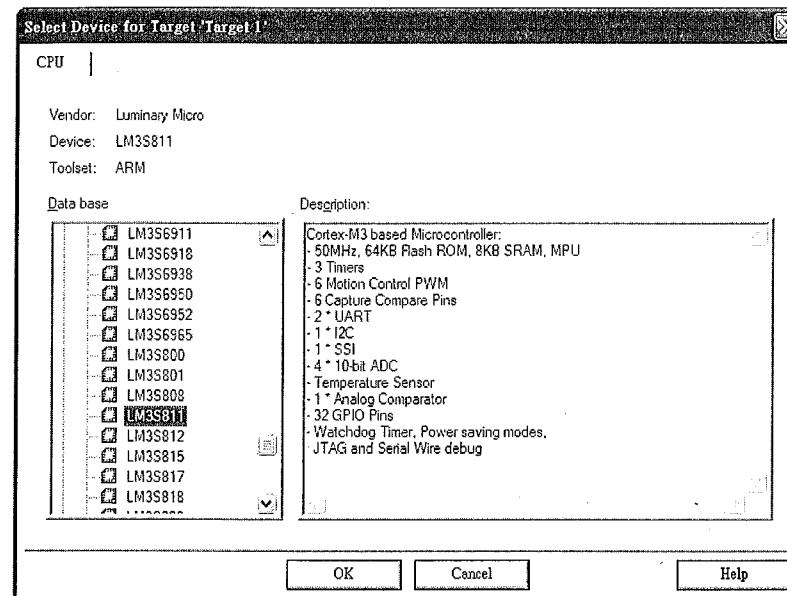


圖 20-3 選擇 LM3S811 設備

接著軟體會問你是否要使用預設的啟動程式。在這例子我們選擇是（參見圖 20-4）。

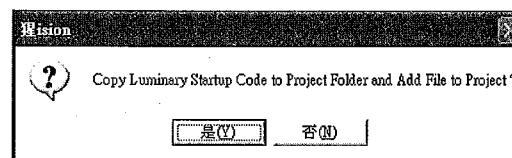


圖 20-4 選擇使用預設的啟動程式

現在我們有一個稱作 Hello 的專案，裡面有一個名為 Startup.s 的檔案（參見圖 20-5）。

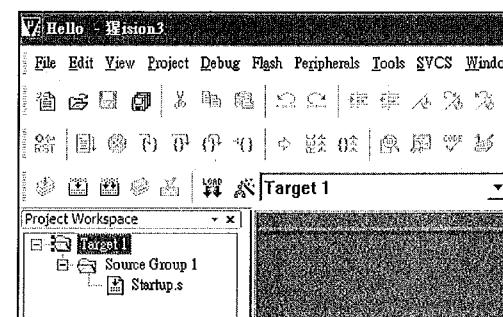


圖 20-5 預設的啟動程式產生的專案

我們可以建立包含主程式的新 C 程式檔案（參見圖 20-6）。

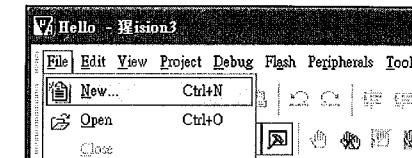


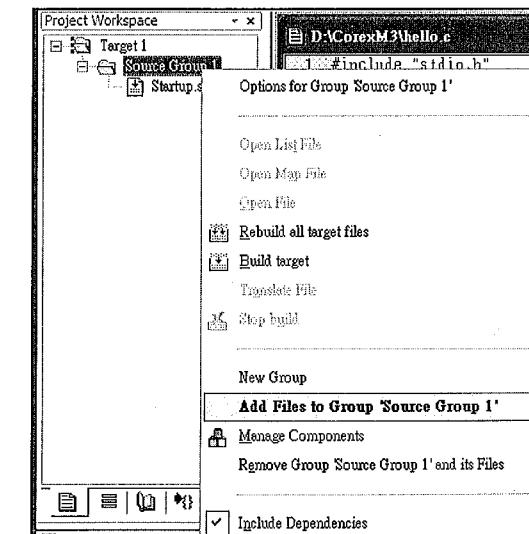
圖 20-6 製造一個新的 C 程式檔案

請建立一個文字檔案，並且儲存為 hello.c（參見圖 20-7）。

```
自 D:\CorexM3\hello.c
1 #include "stdio.h"
2 int main (void)
3 {
4     printf("Hello world!\n");
5     while(1);
6 }
7
```

圖 20-7 一個 Hello World C 範例

現在我們需要將此檔案增加到專案裡：可在 Source Group 1 上按一下滑鼠右鍵（參見圖 20-8）。



# Jason 嘴書—EETOP 世界唯一貼

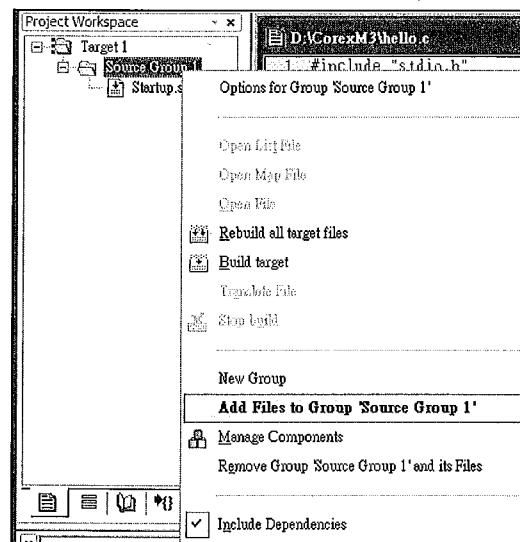


圖 20-8 將 Hello World C 程式加進專案

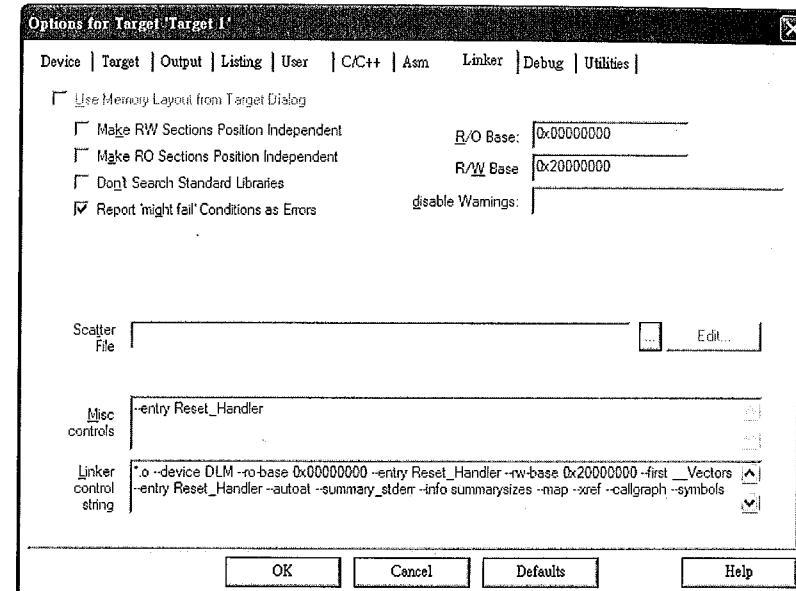
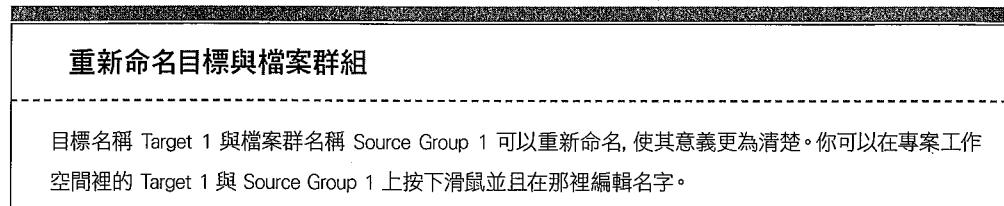


圖 20-10 定義專案裡的進入點



選擇我們建立的 hello.c，接著關閉 Add File 視窗。專案現在包含了兩個檔案(參見圖 20-9)。

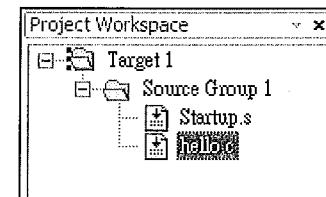


圖 20-9 在增加了 Hello World C 例子之後的專案視窗

我們亦需要設定好連結器設定，以定義程式的進入點。為此我們可以在 Misc Control 欄位增加 --entry Reset\_Handler (參見圖 20-10)。此選項定義了程式的起始點。其中 Reset\_Handler 為一個可以在 Startup.s 找到的指令位址。

我們現在可以編譯程式。在 Target 1 上按滑鼠右鍵，並且選擇 Build target(參見圖 20-11)。

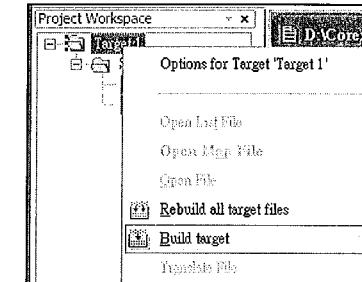


圖 20-11 開始編譯

你應該在輸出視窗看到編譯成功的訊息(參見圖 20-12)。

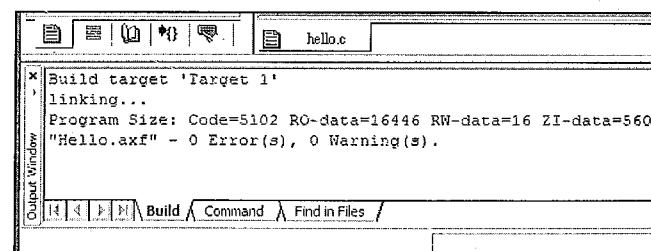


圖 20-12 輸出視窗裡的編譯結果

## 藉由 UART 輸出 "Hello World" 訊息

在我們建立的程式碼裡，使用了標準 C 程式庫裡的 printf 函數。因為 C 程式庫並不知道我們實際使用的硬體，如果需要使用真實的硬體(例如晶片上的 UART)輸出文字訊息，就需要額外的程式。

如本書先前所言，輸出到實際硬體的實作通常稱為重定目標(retargeting)。除了製造文字輸出之外，重定目標的程式可能也包括錯誤處理與程式終止的函數。在此例子裡，僅討論文字輸出的重定目標。

在下面的程式裡，輸出了"Hello world"的訊息到 LM3S811 設備的 UART0。使用的目標系統為 Luminous Micro LM3S811 開發板。此板有一個 6MHz 石英作為時脈來源，並且利用內部的鎖相迴路(Phase Locked Loop, PLL)模組做了簡單的設定程序之後，可將時脈提升到 50MHz。傳輸速度 baud rate 設定為 115200，並且輸出到在 Windows PC 上執行的 HyperTerminal。

為了對 printf 訊息作重定目標，我們需要實作 fputc 函數。在下面的程式裡，我們建立一個 fputc 函數來呼叫執行 UART 控制的 sendchar 函數：

```
===== hello.c =====
#include "stdio.h"

#define CR 0x0D      // 返回游標
#define LF 0x0A      // 换行

void Uart0Init (void) ;
void SetClockFreq (void) ;
int sendchar (int ch) ;

// 下一行定義使用 6MHz 時脈
#define CLOCK50MHZ

// 暫存器位址
#define SYSCTRL_RCC      ((volatile unsigned long *) (0x400FE060))
#define SYSCTRL_RIS       ((volatile unsigned long *) (0x400FE050))
#define SYSCTRL_RCGC1    ((volatile unsigned long *) (0x400FE104))
#define SYSCTRL_RCGC2    ((volatile unsigned long *) (0x400FE108))
#define GPIOA_AFSEL      ((volatile unsigned long *) (0x40004420))

#define UART0_DATA       ((volatile unsigned long *) (0x4000C000))
```

```
#define UART0_FLAG          ((volatile unsigned long *) (0x4000C018))
#define UART0_IBRD           ((volatile unsigned long *) (0x4000C024))
#define UART0_FBRD           ((volatile unsigned long *) (0x4000C028))
#define UART0_LCRH           ((volatile unsigned long *) (0x4000C02C))
#define UART0_CTRL            ((volatile unsigned long *) (0x4000C030))
#define UART0_RIS             ((volatile unsigned long *) (0x4000C03C))

int main (void)
{
    SetClockFreq () ; // 設定時脈 (50MHz / 6MHz)
    Uart0Init () ; // 初始化 Uart0

    printf ("Hello world!\n");
    while (1);
}

void SetClockFreq (void)
{
#ifdef CLOCK50MHZ
    // 設定 BYPASS，清除 USRSYSDIV 與 SYSDIV
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFFFF) | 0x800;
    // 清除 OSCSRC, PWRDN 與 OEN
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFFCFCF);
    // 變更 SYSDIV，設定 USRSYSDIV 與石英的值
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF87FFFC3F) | 0x01C002C0;
    // 等待至 PLLRIS 被設定
    while ((*SYSCTRL_RIS & 0x40) == 0);
    // 清除 bypass
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFF7FF);
#else
    // 設定 BYPASS，清除 USRSYSDIV 與 SYSDIV
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFFFF) | 0x800;
#endif
    return;
}

void Uart0Init (void)
{
    *SYSCTRL_RCGC1 = *SYSCTRL_RCGC1 | 0x0003; // 啟動 UART0
    // 與 UART1 時脈
    *SYSCTRL_RCGC2 = *SYSCTRL_RCGC2 | 0x0001; // 啟動 PORTA 時脈

    *UART0_CTRL = 0; // 除能 UART
#ifdef CLOCK50MHZ
    *UART0_IBRD = 27; // 50MHz 時脈的程式 baud rate
    *UART0_FBRD = 9;
#else
    *UART0_IBRD = 3; // 6MHz 時脈的程式 baud rate
    *UART0_FBRD = 9;
#endif
}
```

```

*UART0_FBRD = 17;
#endif
*UART0_LCRH = 0x60;           // 8 bit, no parity
*UART0_CTRL = 0x301;          // 致能 TX 與 RX, 並作 UART 致能
*GPIOPA_AFSEL = *GPIOPA_AFSEL | 0x3; // 使用 GPIO 接腳為 UART0

return;
}

/* 輸出字元到 UART0 (被 printf 函數使用以輸出資料) */
int sendchar (int ch) {
    if (ch == '\n') {
        while ((*UART0_FLAG & 0x8)); // 若忙碌則等待
        *UART0_DATA = CR;          // 輸出額外的 CR, 在 HyperTermial
    }                                // 得到正確的顯示
    while ((*UART0_FLAG & 0x8)); // 若忙碌則等待
    return (*UART0_DATA = ch);      // 輸出資料
}
/* 將程式作重定目標以輸出本文 */
int fputc (int ch, FILE *f) {
    return (sendchar (ch));
}
===== end of file =====

```

SetupClockFreq 副程式設定系統時脈為 50MHz。其設定的程序隨設備而定。如果沒有設定 CLOCK50MHZ 編譯假指令，則此副程式也可以用來設定時脈頻率到 6MHz。

UART 初始化在 Uart0Init 副程式裡執行。設定的過程包括設定 baud rate 產生器以提供 115200 的 baud rate；將 UART 組態為 8-bit, 無 parity, 以及 1 個停止位元；並且切換 GPIO 埠到替代的功能，因為 UART 接腳與 GPIO 埠 A 共享。在存取 UART 與 GPIO 之前，應該啟動這些區塊的時脈，這可藉由寫進 SYSCTRL\_RCGC1 與 SYSCTRL\_RCGC2 來完成。

作為字元輸出預先定義的函數名稱 fputc 執行了重定目標程式。此程式呼叫 sendchar 函數以輸出字元到 UART。當偵測到新行時 sendchar 函數會輸出一個額外的返回游標字元。為了在 HyperTermial 上得到正確的本文輸出，這是需要的；否則新的一行文字會覆蓋先前一行的文字。

在修改了 hello.c 程式以包含重定目標程式之後，再一次編譯程式。

## 測試軟體

如果你有 Luminous Micro LM3S811 開發板，為了驗證範例，你可將編譯後的程式下載到快閃，並從 HyperTermial 上得到 "Hello world" 的訊息顯示的輸出。假設你已經設定好開發板附帶的軟體驅動程式，則可以藉由下面的步驟載入並測試程式。

首先，設定快閃下載選項。此可由下拉功能表來進行，如圖 20-13 所示。

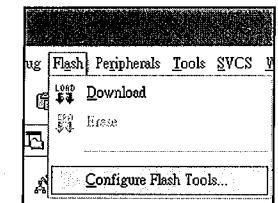


圖 20-13 設定快閃程式組態

在此功能表裡，我們選擇 **Luminous Evaluation Board** 為下載目標(參見圖 20-14)。

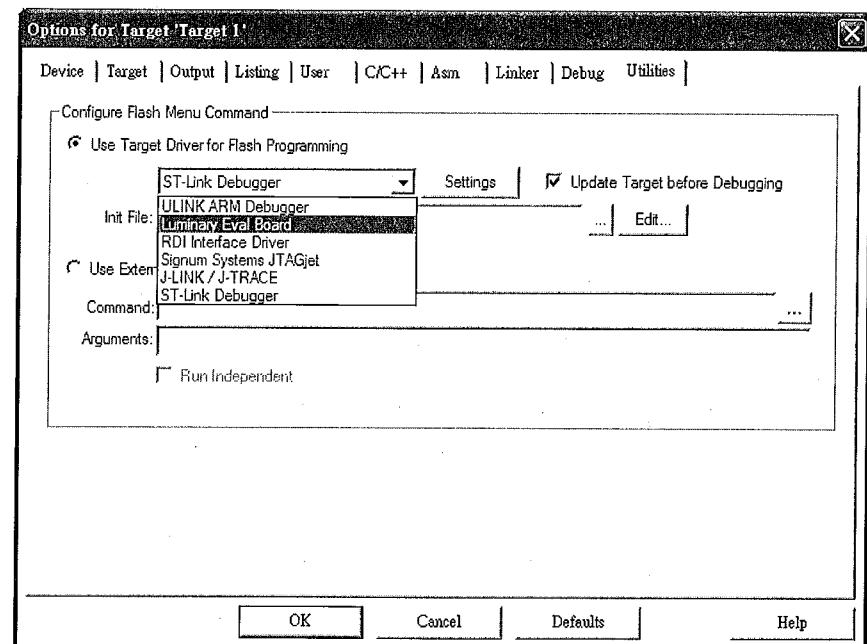


圖 20-14 選擇快閃設定驅動

接著我們可以選擇下拉功能表裡的 **Download** 以載入程式到晶片上的快閃(參見圖 20-15)。

這時將會出現圖 20-16 所示的訊息，顯示完成載入。注意：如果你的開發板正在執行 HyperTermial，你可能需要關閉 HyperTermial，從 PC 上去除 USB cable 線的連接，並且在設定快閃之前重作連接。

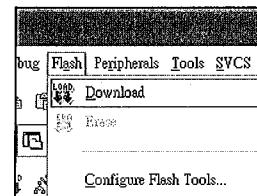


圖 20-15 開始進行載入程序

```

assembling Startup.s...
compiling hello.c...
linking...
"hello.axf" - 0 Error(s), 0 Warning(s).
Load "C:\Keil\ARM\RV30\Examples\CortexM3\hello.AXF"
Found Luminary Micro LM3S811-C0
Flash=65536, RAM=8192
Flash Erase Done.
Flash Write Done: 2064 bytes programmed.
Flash Verify Done: 2064 bytes verified.

```

圖 20-16 在輸出視窗裡載入程序的報告

在完成設定之後，你可以啟動 HyperTermial，使用 Virtual COM Port driver 連接到開發版(經由 USB 連結)，並從在微控制器執行的程式得到本文的顯示(參見圖 20-17)。

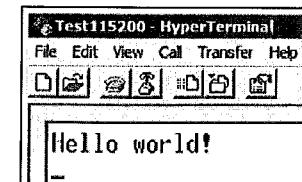
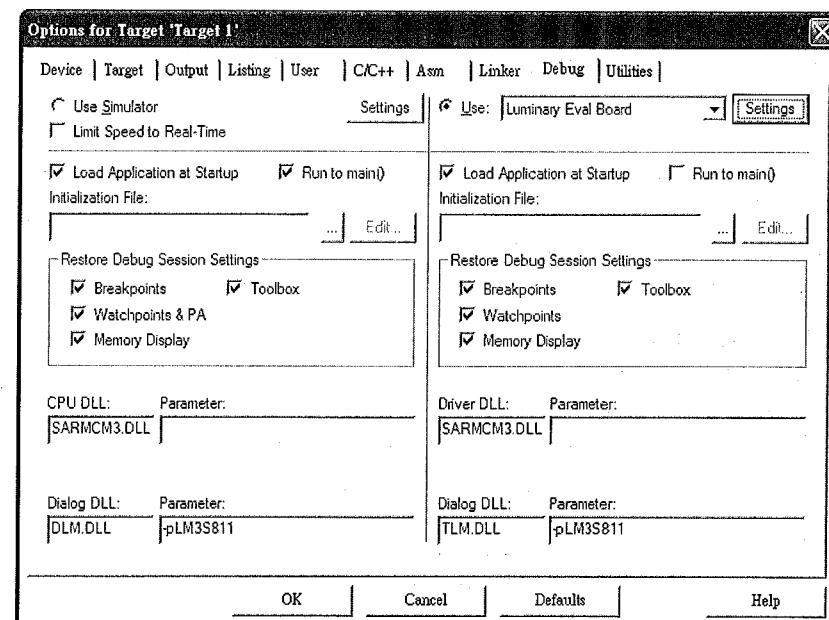


圖 20-17 從 HyperTermial 控制台輸出 Hello World 範例

## 使用除錯器

你可以使用  $\mu$  Vision 裡的除錯器以連接到 Luminary 開發板為應用程式除錯。在專案 Target 1 按一次滑鼠右鍵，並選取 **Options** 以設定除錯選項。在這個例子裡我們選擇使用 **Luminary Eval Board** 以進行除錯(參見圖 20-18)。

圖 20-18 組態成透過  $\mu$  Vision 除錯器去使用 LuminaryMicro 開發板

接著我們可以從下拉的功能表開始除錯階段(參見圖 20-19)。注意：如果你的開發板正在執行 HyperTerminal，你可能需要先關閉 HyperTerminal，從 PC 上拔除 USB cable 線連接，並且在開始除錯階段之前重作連接。

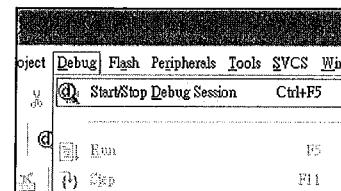


圖 20-19 在 μVision 裡開始進行除錯階段

當除錯器開始，IDE 提供了暫存器視窗以顯示暫存器的內容。你也可以取得反組譯程式視窗並看到現行的指令位址。在圖 20-20 裡我們可以看到核心暫停在 Reset\_Handler 處。

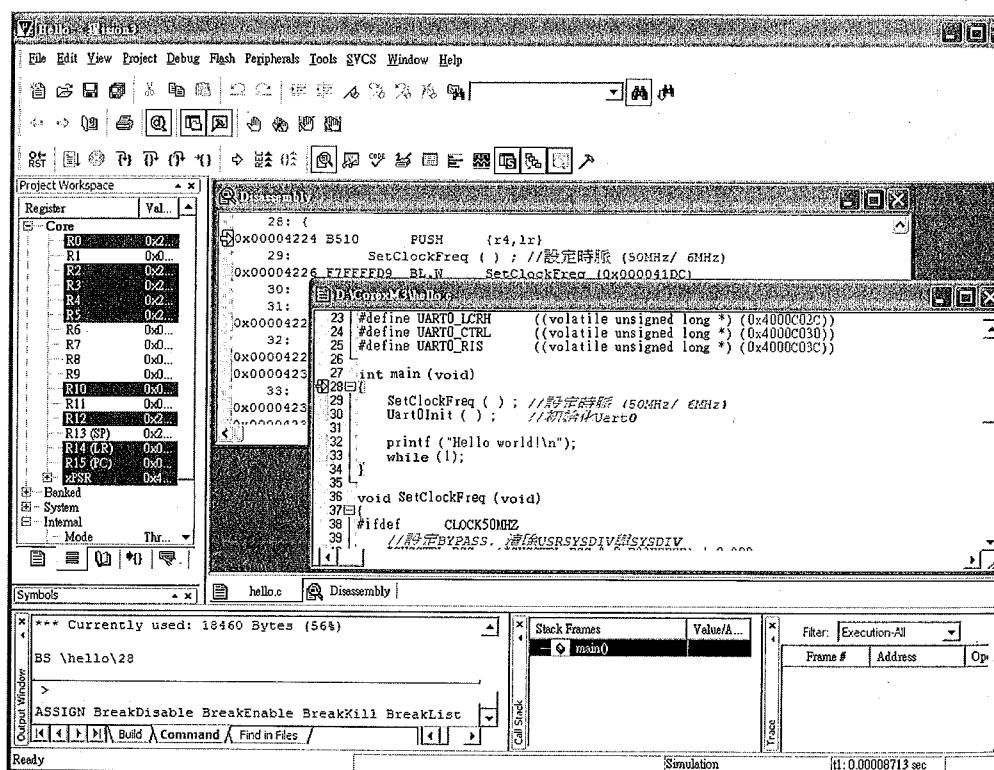


圖 20-20 μVision 除錯環境

為了測試用途，設定中斷點將程式停在主程式開始處。在程式視窗按一下滑鼠右鍵並且選擇 Insert/Remove Breakpoint 以設定中斷點(參見圖 20-21)。注意：我們也可以使用除錯選項裡的 Run to main() 以將程式執行停止在主程式開始處。

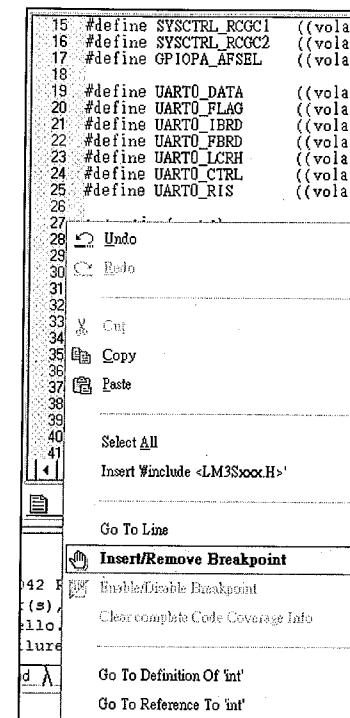


圖 20-21 插入或移除中斷點

接著可以使用 tool bar 的 Run 按鈕開始去執行程式(參見圖 20-22)。

接著程式開始執行，並且執行到主程式開始處會停止(參見圖 20-23)。

接下來我們可以使用 tool bar 的單步控制以測試我們的應用，並使用暫存器視窗檢查結果。

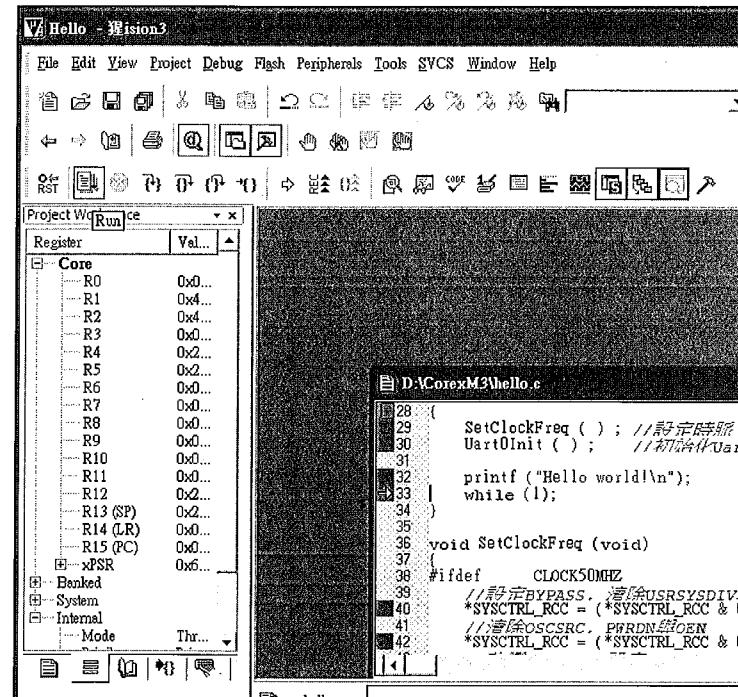
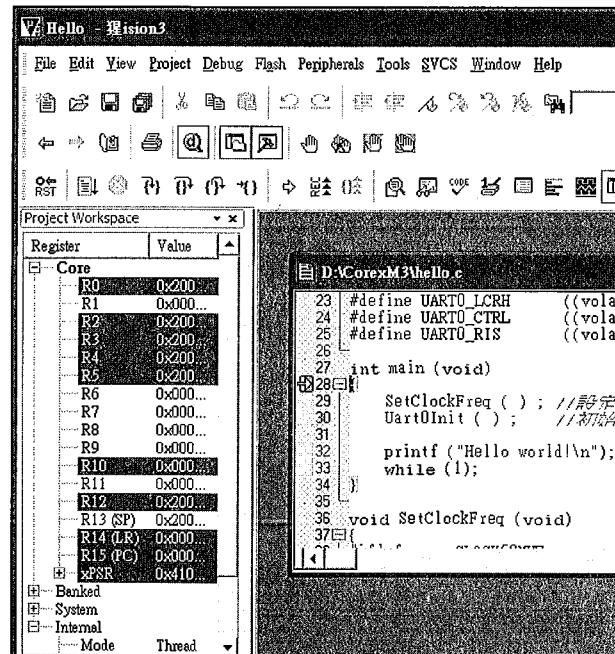


圖 20-22 使用 RUN 按鈕以開始執行程式

圖 20-23 當碰到中  
斷點，程式的執行停  
止在主程式開始處

## 指令集模擬器

$\mu$  Vision IDE 也有一個可做除錯應用的指令集模擬器。其操作類似於在硬體使用除錯器，並且是學習 Cortex-M3 有用的工具。要使用指令集模擬器，需改變專案的除錯選項為 Use Simulator(參見圖 20-24)。注意：模擬器不能模擬所有的硬體週邊的行為，故可能不會正確地模擬 UART 介面程式。

當使用模擬器作除錯時，你也可能需要調整模擬的記憶體設定。這可在開始除錯階段之後設定記憶體映射選項來達成(參見圖 20-25)。

例如，你可能需要將 UART 記憶體位址範圍加到記憶體映射(參見圖 20-26)。否則當你試圖存取 UART 時，可能會在模擬中得到中止(abort)例外。

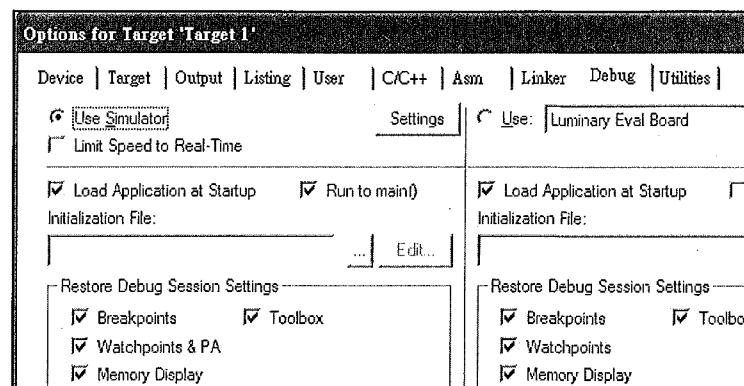


圖 20-24 選擇模擬器作為除錯的目標

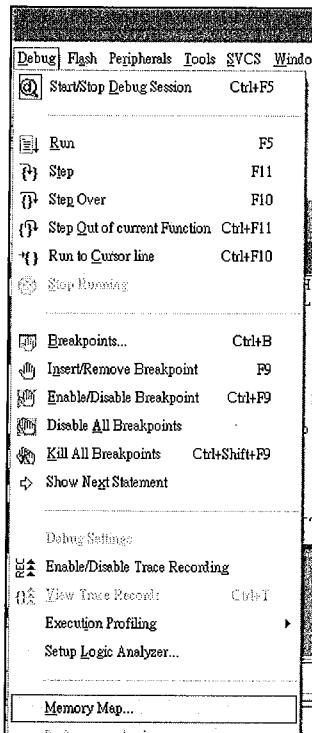


圖 20-25 進行記憶體映射的選項

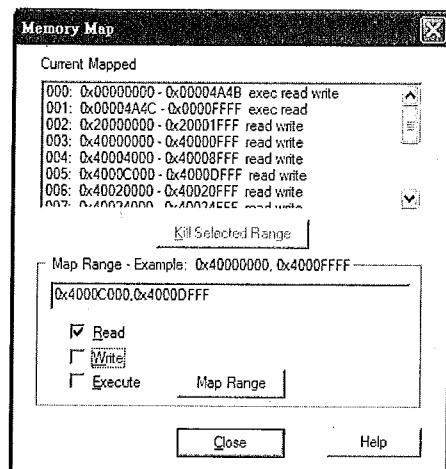


圖 20-26 加上 UART 記憶體到模擬器記憶體設定

## 修改向量表

在先前的範例裡，向量表定義在檔案 Startup.s 裡，此檔案是工具自動準備的標準啟動程式。檔案中包含了向量表、預設的重置處理程式、預設的 NMI 處理程式、預設的硬錯誤處理程式、以及預設的中斷處理程式。隨著你的應用而定，這些例外處理程式可能需要客製化或修改。例如，如果你的應用需要週邊的中斷，則你需要改變向量表，故當中斷被觸發時，將會執行你建立的中斷服務程式(ISR)。

預設的例外處理程式，以組合語言程式的型態存在於 Startup.s 裡。然而，例外處理程式可以使用 C 或者不同的組合語言程式檔案來實現。在這些情形下，將需要組譯器裡的 IMPORT 命令以指出中斷處理程式的位址標籤是定義在其他的檔案裡。在一節裡包含了一個範例，說明了如何使用 IMPORT 命令，並說明了使用 C 的一個簡單的例外處理。

## 使用中斷的碼表範例

此範例包括了 SYSTICK 與中斷(UART0)等例外的使用。如圖 20-27 所示，欲開發的碼表有三種狀態。

根據先前的例子，碼表被 PC 藉著 UART 介面所控制。為了簡化範例的程式，我們把運算速度固定在 50MHz。

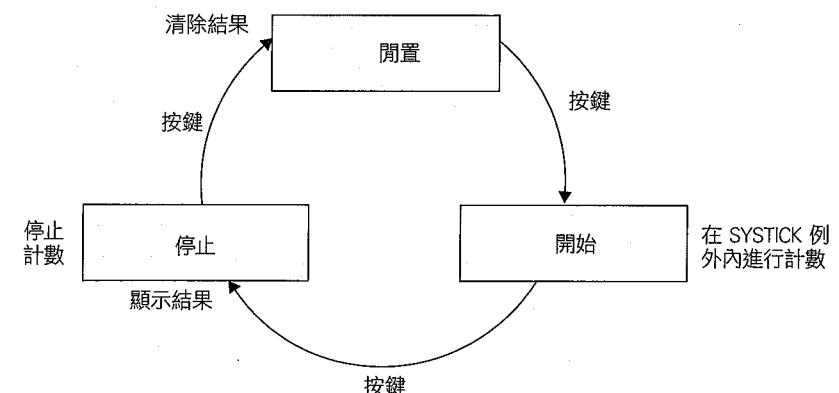


圖 20-27 碼表的狀態機設計

SYSTICK 執行了定時測量，它以 100Hz 的頻率來中斷處理器。SYSTICK 以 50MHz 的核心時脈頻率執行。每一次執行 SYSTICK 例外處理程式時，若正在跑動碼表，則會遞增計數器變數 TickCounter 的值。

因為經由 UART 作本文顯示會相對地慢，所以碼表的控制是在例外處理程式之內處理，而文字與碼表數值的顯示是在主程式(執行緒層級)內執行。一個簡單的軟體狀態機用來控制碼表的開始、停止、與清除。狀態機經由 UART 處理程式來控制，每當接收到一字元時會觸發 UART 一次。

透過與"Hello world"範例相同的程序，讓我們開始一個稱作 stopwatch 的新專案。增加一個稱作 stopwatch.c 的 C 程式檔案以取代 hello.c：

```
===== stopwatch.c =====
#include "stdio.h"
#define CR      0x0D          //返回游標
#define LF      0x0A          //換行

void Uart0Init (void);
void SysTickInit (void);
void SetClockFreq (void);
void DisplayTime (void);
void PrintValue (int value);
int sendchar (int ch);
int getkey (void);
void Uart0Handler (void);
void SysTickHandler (void);

// 暫存器位址
#define SYSCTRL_RCC      ((volatile unsigned long *) (0x400FE060))
#define SYSCTRL_RIS       ((volatile unsigned long *) (0x400FE050))
#define SYSCTRL_RCGC1    ((volatile unsigned long *) (0x400FE104))
#define SYSCTRL_RCGC2    ((volatile unsigned long *) (0x400FE108))
#define GPIOA_AFSEL      ((volatile unsigned long *) (0x40004420))
#define UART0_DATA       ((volatile unsigned long *) (0x4000C000))
#define UART0_FLAG        ((volatile unsigned long *) (0x4000C018))
#define UART0_IBRD      ((volatile unsigned long *) (0x4000C024))
#define UART0_FBRD      ((volatile unsigned long *) (0x4000C028))
#define UART0_LCRH      ((volatile unsigned long *) (0x4000C02C))
#define UART0_CTRL      ((volatile unsigned long *) (0x4000C030))
#define UART0_IM        ((volatile unsigned long *) (0x4000C038))
#define UART0_RIS       ((volatile unsigned long *) (0x4000C03C))
#define UART0_ICR        ((volatile unsigned long *) (0x4000C044))
#define NVIC IRQ_EN0     ((volatile unsigned long *) (0xE000E100))
```

```
//全域變數
volatile int CurrState;           //狀態機
volatile unsigned long TickCounter; //碼表值
volatile int KeyReceived;         //顯示用戶按下鍵
volatile int userInput;          //用戶所按之鍵

#define IDLE_STATE 0                //狀態機的定義
#define RUN_STATE 1
#define STOP_STATE 2

int main (void)
{
    int CurrStateLocal; //區域變數
    //初始化全域變數
    CurrState = 0;
    KeyReceived = 0;
    //初始化硬體
    SetClockFreq( );               //設定時脈的設定 (50MHz)
    Uart0Init( );                 //初始化 Uart0
    SysTickInit( );               //初始化 Systick

    printf ("Stop Watch\n");
    while (1) {
        CurrStateLocal = CurrState; //建立區域副本，因為它的值可能
                                    //隨時會被 UART 處理程式改變
        switch (CurrStateLocal) {
            case (IDLE_STATE):
                printf ("\nPress any key to start\n");
                break;
            case (RUN_STATE):
                printf ("\nPress any key to stop\n");
                break;
            case (STOP_STATE):
                printf ("\nPress any key to clear\n");
                break;
            default:
                CurrState = IDLE_STATE;
                break;
        } // end of switch
        while (KeyReceived ==0) {
            if (CurrState==RUN_STATE) {
                DisplayTime( );
            }
        }; //等待用戶輸入
        if (CurrStateLocal==STOP_STATE) {
            TickCounter=0;
            DisplayTime( );           //結果已清除的顯示
        }
    }
}
```

```

else if (CurrStateLocal==RUN_STATE) {
    DisplayTime ( );           //顯示結果
}
if (KeyReceived!=0) KeyReceived = 0;
}; //end of while loop
} // end of main

void SetClockFreq (void)
{
    //設定 BYPASS, 清除 USRSYSDIV 與 SYSDIV
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFFF) | 0x800;
    //清除 OSCSRC, PWRDN 與 OEN
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFFCFC);
    //改變 SYSDIV, 設定 USRSYSDIV 與石英的值
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF87FFC3F) | 0x01C002C0;
    //等待至 PLLRIS 被設定
    while ((*SYSCTRL_RIS & 0x40) ==0);
    //清除 bypass
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFF7FF);
    return;
}

//初始化 UART0
void Uart0Init (void)
{
    *SYSCTRL_RCGC1 = *SYSCTRL_RCGC1 | 0x0003;      //致能 UART0
    //與 UART1 時脈
    *SYSCTRL_RCGC2 = *SYSCTRL_RCGC2 | 0x0001;      //致能 PORTA 時脈

    *UART0_CTRL = 0;                                //除能 UART
    #ifdef CLOCK50MHZ
    *UART0_IBRD = 27;                               //50MHz 時脈的程式 baud rate
    *UART0_FBRD = 9;
    *UART0_LCRH = 0x60;                             //8 bit, no parity
    *UART0_CTRL = 0x301;                            //致能 TX 與 RX, 並作 UART 致能
    *UART0_IM = 0x10;                                //致能UART中斷以接收資料
    *GPIOPA_AFSEL = *GPIOPA_AFSEL | 0x3;            //使用 GPIO 接腳為 UART0
    *NVIC_IRQ_EN0 = (0x1<<5);                     //在 NVIC 致能 UART 中斷

    return;
}

//初始化 SYSTICK
void SysTickInit (void)
{

#define NVIC_STCSR ((volatile unsigned long *) (0xE000E010))
#define NVIC_RELOAD ((volatile unsigned long *) (0xE000E014))
#define NVIC_CURRVAL ((volatile unsigned long *) (0xE000E018))
#define NVIC_CALVAL ((volatile unsigned long *) (0xE000E01C))
}

```

```

*NVIC_STCSR = 0;                                //除能 SYSTICK
*NVIC_RELOAD = 499999;                           //50MHz 時脈下 100Hz 的重載值
*NVIC_CURRVAL = 0;                             //清除現有值
*NVIC_STCSR = 0x7;                            //以中斷與核心時脈來致能 SYSTICK
return;
}

//SYSTICK 例外處理程式
void SysTickHandler (void)
{
    if (CurrState==RUN_STATE) {
        TickCounter++;
    }
    return;
}
// UART0 RX 中斷處理程式
void Uart0Handler (void)
{
    userinput = getkey ( );
    //顯示接收到一個 key
    KeyReceived++;
    //去除 UART 中斷的宣稱
    *UART0_ICR = 0x10;
    //切換狀態
    switch (CurrState) {
        case (IDLE_STATE):
            CurrState = RUN_STATE;
            break;
        case (RUN_STATE):
            CurrState = STOP_STATE;
            break;
        case (STOP_STATE):
            CurrState = IDLE_STATE;
            break;
        default:
            CurrState = IDLE_STATE;
            break;
    } // end of switch
    return;
}

//顯示時間值
void DisplayTime (void)
{
    unsigned long TickCounterCopy;
    unsigned long TmpValue;

    sendchar (CR);
    TickCounterCopy = TickCounter; //建立區域副本，因為它的值
                                  //可能隨時會被 SYSTICK 處理
}

```

```

//程式改變
TmpValue      = TickCounterCopy / 6000; //分
PrintValue (TmpValue);
TickCounterCopy = TickCounterCopy - (TmpValue * 6000);
TmpValue      = TickCounterCopy / 100; //秒
sendchar (':');
PrintValue(TmpValue);
TmpValue      = TickCounterCopy - (TmpValue * 100);
sendchar (':');
PrintValue(TmpValue);           //毫秒
sendchar (' ');
sendchar (' ');
return;
}

//顯示十進位值
void PrintValue (int value)
{
    printf ("%d", value);
    return;
}

// 輸出字元到 UART0 (被 printf 函數使用以輸出資料)
int sendchar (int ch) {
    if (ch == '\n') {
        //while ((*UART0_FLAG & 0x8)); //若忙碌則等待
        while ((*UART0_FLAG & 0x20)); //若 TXFIFO 已滿則等待
        *UART0_DATA = CR;           //輸出額外的 CR, 在 HyperTerminal
    }
    //得到正確的顯示
    //while ((*UART0_FLAG & 0x8)); //若忙碌則等待
    while ((*UART0_FLAG & 0x20)); //若 TXFIFO 已滿則等待
    return (*UART0_DATA = ch);   //輸出資料
}

//取得用戶輸入
int getkey (void) {           //從序列埠讀值
    while (*UART_FLAG & 0x10); //若 RX FIFO 空了則等待
    return (*UART0_DATA);
}

//重定目標以輸出本文

int fputc (int ch, FILE *f) {
    return (sendchar (ch));
}
===== end of file =====

```

UART 初始化已經稍微改變，使得 UART 介面接收到字元時可以致能中斷。為了致能 UART 中斷要求，中斷需要在 UART 中斷遮罩暫存器以及 NVIC 被致能。就 SYSTICK 而言，僅需要去設定 SYSTICK 控制與狀態暫存器的例外控制。

除此之外，增加了一些額外的函數，包括 UART 與 SYSTICK 處理程式、顯示函數、以及 SYSTICK 初始化。隨著週邊設計而定，一個例外/中斷處理程式可能需要清除例外/中斷要求。在此例子裡，UART 處理程式使用中斷清除暫存器(UART0\_ICR)去清除 UART 中斷要求。同時也修改了啟動程式 Startup.s 以設定例外處理程式(參見圖 20-28)。

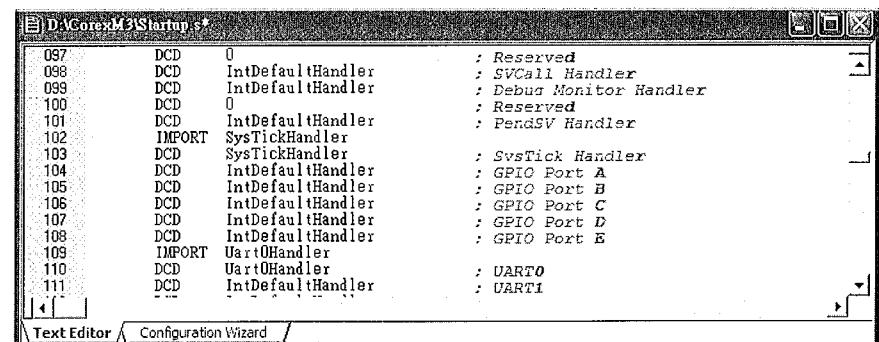


圖 20-28 藉著 IMPORT 與 DCD 命令增加 SysTickHandler 與 Uart0Handler 到向量表

因為處理程式在 C 程式檔案裡，故需要 IMPORT 命令以讓組譯器知道位址標籤來自不同的檔案。

在程式編譯並載入開發板之後，可將它連結到執行 HyperTerminal 的 PC 以作測試。圖 20-29 顯示其結果。

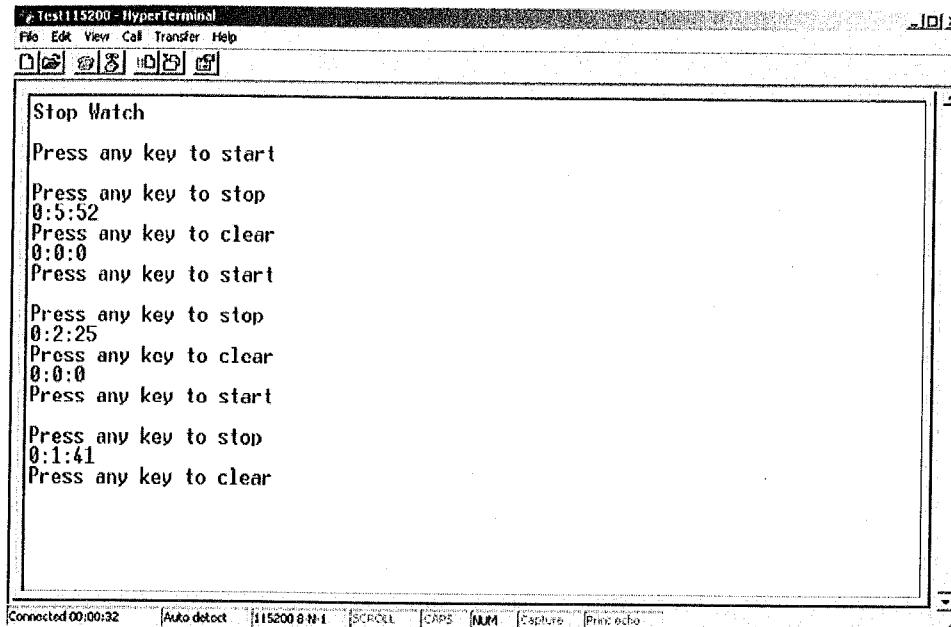


圖 20-29 在 HyperTerminal 控制台輸出碼表範例

注意：如果測試板為 Luminary Micro 開發板，並且使用 Virtual COM Port 作為 UART 通信，因為 Virtual COM port 設備驅動程式的問題，此範例可能不會正確執行(HyperTerminal 裡的按鍵信號 keystroke 無法傳到開發板)。在此情形下，你可能需要在僅有設備驅動程式，但未安裝 RealView MDK 的另一台 PC 上測試此程式。

## Appendix

## Cortex-M3 指令摘要

這些資料得到 ARM Limited 許可，轉載自 Cortex-M3 Technical Reference Manual (技術參考手冊)。上標加號(+)的指令表示旗標(APSRR)會更新。

## 支援的 16-bit Thumb 指令

表 A-1 支援的 16-bit Thumb 指令

組合語言	運算
ADC <Rd>, <Rm>+	將暫存器值與 C 旗標加到暫存器： $Rd = Rd + Rm + C$
ADD <Rd>, <Rn>, #<immed_3>+	將 3-bit 立即值加到暫存器： $Rd = Rn + immed_3$
ADD <Rd>, #<immed_8>+	將 8-bit 立即值加到暫存器： $Rd = Rd + immed_8$
ADD <Rd>, <Rn>, <Rm>+	將低暫存器值加到低暫存器值： $Rd = Rn + Rm$
ADD <Rd>, <Rm>	將高暫存器值加到低或高暫存器值
ADD <Rd>, PC, #<immed_8>*4	將 4x(8-bit 立即值)+(word 對齊的 PC 值)加到暫存器： $Rd = PC + 4*immed_8$
ADD <Rd>, SP, #<immed_8>*4	將 4x(8-bit 立即值)+(word 對齊的 SP 值)加到暫存器： $Rd = SP + 4*immed_8$
ADD SP, #<immed_7>*4	將 4x(7-bit 立即值)加到 SP： $SP = SP + 4*immed_7$
AND <Rd>, <Rm>+	以位元依次做 AND 暫存器運算： $Rd = Rd \text{ AND } Rm$
ASR <Rd>, <Rm>, #<immed_5>+	以立即值指定位數作算術右移： $Rd = Rm \gg immed_5$
ASR <Rd>, <Rs>+	以暫存器值指定位數作算術右移： $Rd = Rm \gg Rs$
B <cond> <target_address_8>	條件式跳躍： if <cond> then $PC = (PC+4) + (\text{SignExtend}(\text{target\_address\_8}) * 2)$
B <target_address_11>	無條件跳躍： $PC = (PC+4) + (\text{SignExtend}(\text{target\_address\_11}) * 2)$
BIC <Rd>, <Rm>+	位元清除： $Rd = Rd \text{ AND } (\text{NOT } Rm)$
BKPT <immed_8>	軟體中斷點
BL <target_address_11>	跳躍並作連結
BLX <Rm>	跳躍並作連結與交換(Rm[bit 0]必須為 1)

接下頁

組合語言	運算
BX <Rm>	跳躍並作交換(Rm[bit 0]必須為 1)
CBNZ <Rn>, <label>	比較並且若非零則跳躍(僅作向前跳躍)
CBZ <Rn>, <label>	比較並且若零則跳躍(僅作向前跳躍)
CMN <Rn>, <Rm>+	將暫存器之負值與其他暫存器作比較： 計算 $(Rn - (-Rm))$ 並且更新旗標
CMP <Rn>, #<immed_8>+	將暫存器與 8-bit 立即值作比較
CMP <Rn>, <Rm>+	比較暫存器值
CMP <Rn>, <Rm>+	將高暫存器與低或高暫存器作比較
CPSIE <l 或 f>	改變處理器狀態
CPSID <l 或 f>	CPSIE 藉著清除 PRIMASK(l)或 FAULTMASK(f)致能中斷 CPSID 藉著清除 PRIMASK(l)或 FAULTMASK(f)除能中斷
CPY <Rd>, <Rm>	複製一個高或低暫存器值至另一個高或低的暫存器
EOR <Rd>, <Rm>+	以位元依次將暫存器值依次做 XOR 運算
IT <x> <cond>	IF-THEN 條件式區塊；根據條件<cond>執行接下來的兩到四個指令
IT <x> <y> <cond>	
IT <x> <y> <z> <cond>	
LDMIA <Rn>!, <registers>	多重載入之後遞增；由 Rn 所指定之位址的記憶體開始載入多重的 words
LDR <Rd>, [<Rn>, #<immed_5>*4]	從基底暫存器位址+5-bit 立即值位移載入記憶體 word
LDR <Rd>, [<Rn>, <Rm>]	從基底暫存器位址+暫存器位移值載入記憶體 word
LDR <Rd>, [PC, #<immed_8>*4]	從 PC 位址+8-bit 立即值位移載入記憶體 word
LDR <Rd>, [SP, #<immed_8>*4]	從 SP 位址+8-bit 立即值位移載入記憶體 word
LDRB <Rd>, [<Rn>, #<immed_5>]	從基底暫存器位址+5-bit 立即值位移載入記憶體 byte[7:0]
LDRB <Rd>, [<Rn>, <Rm>]	從基底暫存器位址+暫存器位移載入記憶體 byte[7:0]
LDRH <Rd>, [<Rn>, #<immed_5>*2]	從基底暫存器位址+5-bit 立即值位移載入記憶體 half word[15:0]
LDRH <Rd>, [<Rn>, <Rm>]	從基底暫存器位址+暫存器位移載入記憶體 half word[15:0]
LDRSB <Rd>, [<Rn>, <Rm>]	從基底暫存器位址+暫存器位移值載入有號的記憶體 byte[7:0]
LDRSH <Rd>, [<Rn>, <Rm>]	從基底暫存器位址+暫存器位移載入有號的記憶體 half word[15:0]
LSL <Rd>, <Rm> #<immed_5>+	以立即值指定位數做邏輯左移： $Rd = Rd \ll immed_5$
LSL <Rd>, <Rs>+	以暫存器值指定位數做邏輯左移： $Rd = Rd \ll Rs$
LSR <Rd>, <Rm> #<immed_5>+	以立即值指定位數做邏輯右移： $Rd = Rd \gg immed_5$
LSR <Rd>, <Rs>+	以暫存器內的數值做邏輯右移： $Rd = Rd \gg Rs$
MOV <Rd>, #<immed_8>+	移動立即的 8-bit 值至暫存器： $Rd = immed_8$

接下頁

# Jason 嘴書—EETOP 世界唯一貼

組合語言	運算
MOV <Rd>, <Rn>+	移動低暫存器值至低暫存器
MOV <Rd>, <Rm>	移動高或低暫存器值至高或低暫存器
MUL <Rd>, <Rm>+	乘以暫存器值： $Rd = Rd * Rm$
MVN <Rd>, <Rm>+	移動暫存器補數值至暫存器： $Rd = \text{NOT}(Rm)$
NEG <Rd>, <Rm>+	將暫存器值取負數並且存放至暫存器裡： $Rd = 0 - Rm$
NOP	無運算
ORR <Rd>, <Rm>+	以位元依次 OR 暫存器值 $Rd = Rd \text{ OR } Rm$
POP <registers>	從堆疊 POP 暫存器值
POP <registers, PC>	從堆疊 POP 暫存器與 PC 值
PUSH <registers>	將暫存器值推進堆疊
PUSH <registers, LR>	將暫存器與 LR 值推進堆疊
REV <Rd>, <Rn>	將 word 值 bytes 顛倒並且複製至暫存器： $Rd = \{\text{Rn}[7:0], \text{Rn}[15:8], \text{n}[23:16], \text{Rn}[31:24]\}$
REV16 <Rd>, <Rn>	將兩個 half word 值 bytes 顛倒並且複製至暫存器： $Rd = \{\text{Rn}[23:16], \text{Rn}[31:24], \text{Rn}[7:0], \text{Rn}[15:8]\}$
REVSH <Rd>, <Rn>	將低 half word 值 bytes 顛倒並且作正負號擴充再複製至暫存器： $Rd = \text{SignExtend}(\{\text{Rn}[7:0], \text{Rn}[15:8]\})$
ROR <Rd>, <Rs>+	依暫存器值指定位數作向右旋轉
SBC <Rd>, <Rm>+	減去暫存器值並且從暫存器值借位(-C)： $Rd = Rd - Rm - \text{NOT}(C)$
SEV	送出事件
STMIA <Rn>l, <registers>	儲存多重暫存器 word 至循序的記憶體位置
STR <Rd>, [<Rn>, #<immed_5>*4]	儲存暫存器 word 至暫存器位址+5-bit 立即值位移處
STR <Rd>, [<Rn>, <Rm>]	儲存暫存器 word 至基底暫存器位址+暫存器位移處
STR <Rd>, [PC, #<immed_8>*4]	儲存暫存器 word 至 PC 位址+8-bit 立即值位移處
STR <Rd>, [SP, #<immed_8>*4]	儲存暫存器 word 至 SP 位址+8-bit 立即值位移處
STRB <Rd>, [<Rn>, #<immed_5>]	儲存暫存器 byte[7:0]至暫存器位址+5-bit 立即值位移處
STRB <Rd>, [<Rn>, <Rm>]	儲存暫存器 byte[7:0]至暫存器位址+暫存器位移處
STRH <Rd>, [<Rn>, #<immed_5>*2]	儲存暫存器 half word[15:0]至暫存器位址+5-bit 立即值位移處
STRH <Rd>, [<Rn>, <Rm>]	儲存暫存器 half word[15:0]至暫存器位址+暫存器位移處
SUB <Rd>, <Rn>, #<immed_3>+	從暫存器減去 3-bit 立即值： $Rd = Rn - \text{immed}_3$
SUB <Rd>, #<immed_8>+	從暫存器減去 8-bit 立即值： $Rd = Rd - \text{immed}_8$

接下頁

組合語言	運算
SUB <Rd>, <Rn>, <Rm>+	減去低暫存器值： $Rd = Rn - Rm$
SUB XSP, #<immed_7>*4	從 SP 減去 4(7-bit 立即值)： $SP = SP - \text{immed}_7 * 4$
SVC <immed_8>	以 8-bit 立即值為呼叫碼做作業系統服務呼叫
SXTB <Rd>, <Rm>	從暫存器減去 byte[7:0], 移到暫存器，並且正負號擴充至 32-bit
SXTH <Rd>, <Rm>	從暫存器減去 half word[15:0], 移到暫存器，並且正負號擴充至 32-bit
TST <Rn>, <Rm>+	藉 ANDing 其他的暫存器值以對設定位元作暫存器值之測試
UXTB <Rd>, <Rm>	從暫存器粹取 byte[7:0], 移至暫存器，並做 0 的擴充至 32-bit
UXTH <Rd>, <Rm>	從暫存器粹取 half word[15:0], 移至暫存器，並做 0 的擴充至 32-bit
WFE	等待事件
WFI	等待中斷

## 支援的 32-bit Thumb-2 指令

有後綴字{S}的指令僅在使用了 S 後綴字，才會更新旗標(APSR)。以正號(+)標示之指令表示旗標(APSR)會更新。

注意：為了支援經常需要數值範圍的立即值資料，許多的 Thumb-2 指令使用立即值資料編碼方式，在表 A-2 以 modify\_constant 標籤表示。其編碼方式之詳細文獻紀錄在 ARM Architecture Application Level Reference Manual, Section A5.2, "Immediate Constants"裡面。

A

# Jason 嘴書—EETOP 世界唯一貼

表 A-2 支援的 32-bit Thumb 指令

組合語言	運算
ADC{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	將暫存器值、立即值與 C 旗標加到暫存器： $Rd = Rd + \text{modify\_constant (immed\_12)} + C$
ADC{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	加上暫存器值、移位的暫存器值與 C 旗標： $Rd = Rn + (Rm \ll \text{shift}) + C$
ADD{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	將暫存器值與立即值加到暫存器值： $Rd = Rd + \text{modify\_constant (immed\_12)}$
ADD{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	暫存器值與移位的暫存器值相加： $Rd = Rn + (Rm \ll \text{shift})$
ADDW.W <Rd>, <Rn>, #<immed_12>	暫存器值與立即的 12-bit 值相加
AND{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	以位元依次做 AND 暫存器值與立即值之運算
AND{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	以位元依次做 AND 暫存器值與移位的暫存器值之運算
ASR{S}.W <Rd>, <Rn>, <Rm>	以暫存器值指定位數作算術右移
B.W	無條件跳躍
B{cond}.W <label>	條件式跳躍
BFC.W <Rd>, #<lsb>, #<width>	清除位元欄位
BFI.W <Rd>, <Rn>, #<lsb>, #<width>	將位元欄位從一個暫存器值插入另一個暫存器
BIC{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	暫存器值與立即值之補數以位元依次做 AND 運算
BIC{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	暫存器值與移位的暫存器值之補數以位元依次做 AND 運算
BL <label>	跳躍並連結
CLZ.W <Rd>, <Rn>	計數暫存器值開頭 0 的個數
CLREX.W	清除獨占存取監視器狀態
CMN.W <Rn>, #<modify_constant (immed_12)>+	暫存器值與立即值之 2 的補數作比較
CMN.W <Rn>, <Rm> {, <shift>}+	暫存器值與暫存器值之 2 的補數作比較
CMP.W <Rn>, #<modify_constant (immed_12)>+	暫存器值與立即值作比較
CMP.W <Rn>, <Rm> {, <shift>}+	暫存器值與暫存器值作比較
DMB	資料記憶體障礙
DSB	資料同步化障礙
EOR{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	暫存器值與立即值作 XOR 運算
EOR{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	暫存器值與移位的暫存器值作 XOR 運算
LDM{[A DB]}.W <R>{}, <registers>	載入多重的記憶體暫存器，並在之後遞增或之前遞減
LDR.W <Rxf>, <Rn>, #<offset_12>	從基底暫存器位址+12-bit 立即值位移讀取記憶體 word

接下頁

組合語言	運算
LDR.W <Rxf>, <Rn>, #+/-<offset_8>	讀取記憶體 word, 從基底暫存器位址+8-bit 立即值位移, 後索引
LDR.W <Rxf>, <Rn>, #+/-<offset_8>!	讀取記憶體 word, 從基底暫存器位址+8-bit 立即值位移, 前索引
LDR.W <Rxf>, <Rn>, <Rm> {, LSL #<shift>})	讀取記憶體 word, 從基底暫存器位址+移位的暫存器值位移(移位範圍從 0 至 3)
LDR.W <Rxf>, [PC, #+/-<offset_12>]	讀取記憶體 word, 從 PC+ 12-bit 立即值位移
LDR.W PC, <Rn>, #<offset_12>	讀取跳躍目的, 從基底暫存器位址+ 12-bit 立即值位移並作跳躍
LDR.W PC, <Rn>, #+/-<offset_8>	讀取跳躍目的, 從基底暫存器位址+ 8-bit 立即值位移, 後索引, 並作跳躍
LDR.W PC, <Rn>, #+/-<offset_8>!	讀取跳躍目的, 從基底暫存器位址+ 8-bit 立即值位移, 前索引, 並作跳躍
LDR.W PC, <Rn>, <Rm> {, LSL #<shift>})	讀取跳躍目的, 從基底暫存器位址+移位的暫存器值位移(移位範圍從 0 至 3), 並作跳躍
LDR.W PC, [PC, #+/-<offset_12>]	讀取跳躍目的, 從 PC+ 12-bit 立即值位移並作跳躍
LDRB.W <Rxf>, <Rn>, #<offset_12>	讀取記憶體 byte, 從基底暫存器位址+ 12-bit 立即值位移
LDRB.W <Rxf>, <Rn>, #+/-<offset_8>	讀取記憶體 byte, 從基底暫存器位址+ 8-bit 立即值位移, 後索引
LDRB.W <Rxf>, <Rn>, #+/-<offset_8>!	讀取記憶體 byte, 從基底暫存器位址+ 8-bit 立即值位移, 前索引
LDRB.W <Rxf>, <Rn>, <Rm> {, LSL #<shift>})	讀取記憶體 byte, 從基底暫存器位址+移位的暫存器值位移(移位範圍從 0 至 3)
LDRB.W <Rxf>, [PC, #+/-<offset_12>]	讀取記憶體 byte, 從 PC+ 12-bit 立即值位移
LDRD.W <Rxf1>, <Rxf2>, <Rn>, #+/-<offset_8>*4 {}	由記憶體讀取 double word, 從基底暫存器位址 +/- 立即值位移, 前索引
LDRD.W <Rxf1>, <Rxf2>, <Rn>, #+/-<offset_8>*4	由記憶體讀取 double word, 從基底暫存器位址 +/- 立即值位移, 後索引
LDREX.W <Rxf>, <Rn> {, #<offset_8>*4})	獨占載入 word, 從基底暫存器位址+ 立即值位移
LDREXB.W <Rxf>, <Rn>	從暫存器位址獨占載入 byte
LDREXH.W <Rxf>, <Rn>	從暫存器位址獨占載入 half word
LDRH.W <Rxf>, <Rn>, #<offset_12>)	讀取記憶體 half word, 從基底暫存器位址+12-bit 立即值位移
LDRH.W <Rxf>, <Rn>, #+/-<offset_8>)	讀取記憶體 half word, 從基底暫存器位址+8-bit 立即值位移, 後索引
LDRH.W <Rxf>, <Rn>, #+/-<offset_8>!	讀取記憶體 half word, 從基底暫存器位址+8-bit 立即值位移, 前索引
LDRH.W <Rxf>, <Rn>, <Rm> {, LSL #<shift>})	讀取記憶體 half word, 從基底暫存器位址+移位的暫存器值位移(移位範圍從 0 至 3)

接下頁

A

## Jason 嘴書—EE TOP 世界唯一貼

組合語言	運算
LDRH.W <Rxf>, [PC, #+/-<offset_12>]	讀取記憶體 half word, 從 PC + 12-bit 立即值位移
LDRSB.W <Rxf>, [<Rn>, #<offset_12>]	讀取記憶體 byte, 從基底暫存器位址 + 12-bit 立即值位移, 作正負號擴充, 並且複製到暫存器
LDRSB.W <Rxf>, [<Rn>, #+/-<offset_8>]	讀取記憶體 byte, 從基底暫存器位址 + 8-bit 立即值位移, 作正負號擴充, 並且複製到暫存器, 後索引
LDRSB.W <Rxf>, [<Rn>, #+/-<offset_8>!]	讀取記憶體 byte, 從基底暫存器位址 + 8-bit 立即值位移, 作正負號擴充, 並且複製到暫存器, 前索引
LDRSB.W <Rxf>, [<Rn>, <Rm> {, LSL #<shift>}]	讀取記憶體 byte, 從基底暫存器位址 + 移位的暫存器位移(移位範圍從 0 至 3), 作正負號擴充, 並且複製到暫存器
LDRSB.W <Rxf>, [PC, #+/-<offset_12>]	讀取記憶體 byte, 從 PC + 12-bit 立即值位移, 作正負號擴充, 並且複製到暫存器
LDRSH.W <Rxf>, [<Rn>, #<offset_12>]	讀取記憶體 half word, 從基底暫存器位址 + 12-bit 立即值位移, 作正負號擴充, 並且複製到暫存器
LDRSH.W <Rxf>, [<Rn>, #+/-<offset_8>]	讀取記憶體 half word, 從基底暫存器位址 + 8-bit 立即值位移, 作正負號擴充, 並且複製到暫存器, 後索引
LDRSH.W <Rxf>, [<Rn>, #+/-<offset_8>!]	讀取記憶體 half word, 從基底暫存器位址 + 8-bit 立即值位移, 作正負號擴充, 並且複製到暫存器, 前索引
LDRSH.W <Rxf>, [<Rn>, <Rm> {, LSL #<shift>}]	讀取記憶體 half word, 從基底暫存器位址 + 移位的暫存器位移(移位範圍從 0 至 3), 作正負號擴充, 並且複製到暫存器
LDRH.W <Rxf>, [PC, #+/-<offset_12>]	讀取記憶體 half word, 從 PC + 12-bit 立即值位移, 作正負號擴充, 並且複製到暫存器
LDRT.W <Rxf>, [<Rn>, #<offset_8>]	Word 讀取及轉換；在特權的模式裡, 讀取基底暫存器位址加上立即值位移並且以用戶存取等級
LDRBT.W <Rxf>, [<Rn>, #<offset_8>]	Byte 讀取及轉換；在特權的模式裡, 讀取基底暫存器位址加上立即值位移並且以用戶存取等級
LDRHT.W <Rxf>, [<Rn>, #<offset_8>]	Half word 讀取及轉換；在特權的模式裡, 讀取基底暫存器位址加上立即值位移並且以用戶存取等級
LSL{S}.W <Rd>, <Rn>, <Rm>	根據暫存器裡的數值以邏輯左移暫存器值
LSR{S}.W <Rd>, <Rn>, <Rm>	根據暫存器裡的數值以邏輯右移暫存器值
相乘並累加：	
將兩個有號或無號的暫存器值相乘, 並將低的 32 位元加到暫存器值：	
MLA.W <Rd>, <Rn>, <Rm>, <Racc>	$Rd = (Rn * Rm) + Racc$

接下頁

組合語言	運算
MLS.W <Rd>, <Rn>, <Rm>, <Racc>	相乘並累減： 將兩個有號或無號的暫存器值相乘, 並以低的 32 位元將暫存器值減去： $Rd = Racc - (Rn * Rm)$
MOV{S}.W <Rd>, #<modify_constant (immed_12)>	移動立即值至暫存器： $Rd = modify\_constant (immed_12)$
MOV{S}.W <Rd>, <Rm> {, <shift>}	移動被移位的暫存器值至暫存器
MOVT.W <Rd>, #<immed_16>	移動立即的 16-bit 值至暫存器頂端 half word[31:16]; 低的 half word 不受影響
MOVW.W <Rd>, #<immed_16>	移動立即的 16-bit 值至暫存器底部 half word[15:0]並且清除上半的 half word
MRS <Rd>, <sreg>	讀取特殊暫存器並且複製至暫存器
MSR <sreg>, <Rd>	將暫存器值寫至特殊暫存器
MUL.W <Rd>, <Rn>, <Rm>	將兩個有號或無號的數值相乘： $Rd = Rm * Rn$
NOP.W	無運算
ORN{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	暫存器值與立即值以位元依次作 OR NOT 運算
ORN{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	暫存器值與被移位的暫存器值以位元依次作 OR NOT 運算
ORR{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	暫存器值與立即值以位元依次作 OR 運算
ORR{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	暫存器值與被移位的暫存器值以位元依次作 OR 運算
POP.W <registers>	從堆疊 POP 暫存器
POP.W <registers, PC>	從堆疊 POP 暫存器與 PC
PUSH.W <registers>	PUSH 暫存器進堆疊
PUSH.W <registers, PC>	PUSH 暫存器與 PC 進堆疊
RBIT.W <Rd>, <Rm>	反轉位元次序
REV.W <Rd>, <Rm>	反轉 word 內之 bytes 次序
REV16.W <Rd>, <Rn>	反轉每一個 half word 內之 bytes 次序
REVSH.W <Rd>, <Rn>	反轉底部 half word 內之 bytes 次序並作正負號擴充
ROR{S}.W <Rd>, <Rn>, <Rm>	依暫存器值指定位數向右旋轉
RSB{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	從立即值反轉減去暫存器值
RSB{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	從被移位的暫存器值反轉減去暫存器值
RRX{S}.W <Rd>, <Rm>	向右旋轉並作 1 bit 擴充
SBC{S}.W <Rd>, <Rn>, #<modify_constant (immed_12)>	從暫存器值減去立即值與 C 位元
SBC{S}.W <Rd>, <Rn>, <Rm> {, <shift>}	從暫存器值減去被移位的暫存器值與 C 位元

接下頁

A

# Jason 嘴書—EETOP 世界唯一貼

組合語言	運算
SBNX.W <Rd>, <Rn>, #<lsb>, #<width>	從暫存器複製位元欄位並作正負號擴充至 32 位元
SDIV.W <Rd>, <Rn>, <Rm>	有號數除法： $Rd = Rn / Rm$
SEV	送出事件
SMLAL.W <RdLo>, <RdHi>, <Rn>, <Rm>	有號數 word 相乘並加上有號數擴充值至雙暫存器值： $\{RdHi, RdLo\} = (Rn * Rm) + \{RdHi, RdLo\}$
SMULL.W <RdLo>, <RdHi>, <Rn>, <Rm>	有號數暫存器值相乘： $\{RdHi, RdLo\} = (Rn * Rm)$
SSAT.W <Rd>, #<imm>, <Rn>, {<shift>}	將被移位的暫存器值作有號飽和至立即值內的 bit 位置；若發生飽和則更新 Q 旗標
STM{A DB}W <Rn>{}, <registers>	將多個暫存器 words 寫至連續的記憶體位置；之後遞增或之前遞減
STR.W <Rxf>, {<Rn>, #<offset_12>}	寫 word 至基底暫存器位址+12-bit 立即值位移
STR.W <Rxf>, {<Rn>}, #+/-<offset_8>	寫 word 至基底暫存器位址+8-bit 立即值位移，後索引
STR.W <Rxf>, {<Rn>}, #+/-<offset_8>!	寫 word 至基底暫存器位址+8-bit 立即值位移，前索引
STR.W <Rxf>, {<Rn>, <Rm> {, LSL #<shift>}}	寫 word 至基底暫存器位址+被移位的暫存器值位移(移位範圍從 0 至 3)
STRB.W <Rxf>, {<Rn>, #<offset_12>}	寫 byte 至基底暫存器位址+12-bit 立即值位移
STRB.W <Rxf>, {<Rn>}, #+/-<offset_8>	寫 byte 至基底暫存器位址+8-bit 立即值位移，後索引
STRB.W <Rxf>, {<Rn>}, #+/-<offset_8>!	寫 byte 至基底暫存器位址+8-bit 立即值位移，前索引
STRB.W <Rxf>, {<Rn>, <Rm> {, LSL #<shift>}}	寫 byte 至基底暫存器位址+被移位的暫存器值位移(移位範圍從 0 至 3)
STRD.W <Rxf1>, <Rxf2>, {<Rn>}, #+/-<offset_8>*4 {}	從基底暫存器位址+/-立即值位移處寫 double word 至記憶體，前索引
STRD.W <Rxf1>, <Rxf2>, {<Rn>}, #+/-<offset_8>*4	從基底暫存器位址+/-立即值位移處寫 double word 至記憶體，後索引
STREX.W <Rxf>, {<Rn> {, #<offset_8>*4}}	從基底暫存器位址+立即值位移處獨占儲存 word
STREXB.W <Rxf>, {<Rn>}	從暫存器位址處獨占儲存 byte
STREXH.W <Rxf>, {<Rn>}	從暫存器位址處獨占儲存 half word
STRH.W <Rxf>, {<Rn>}, #<offset_12>	寫 half word 至基底暫存器位址+12-bit 立即值位移
STRH.W <Rxf>, {<Rn>}, #+/-<offset_8>	寫 half word 至基底暫存器位址+8-bit 立即值位移，後索引

接下頁

組合語言	運算
STRH.W <Rxf>, {<Rn>, #+/-<offset_8>}!	寫 half word 至基底暫存器位址+8-bit 立即值位移，前索引
STRH.W <Rxf>, {<Rn>, <Rm> {, LSL #<shift>}}	寫 half word 至基底暫存器位址+被移位的暫存器值位移(移位範圍從 0 至 3)
STRT.W <Rxf>, {<Rn>, #<offset_8>}	word 儲存及轉換；在特權的模式裡，寫入基底暫存器位址加上立即值位移並且以用戶存取等級
STRBT.W <Rxf>, {<Rn>, #<offset_8>}	byte 儲存及轉換；在特權的模式裡，寫入基底暫存器位址加上立即值位移並且以用戶存取等級
STRHT.W <Rxf>, {<Rn>, #<offset_8>}	Half word 儲存及轉換；在特權的模式裡，寫入基底暫存器位址加上立即值位移並且以用戶存取等級
SUB{S}W <Rd>, <Rn>, #<modify_constant (immed_12)>	從暫存器減去立即值位移： $Rd = Rd - \text{modify\_constant (immed\_12)}$
SUB{S}W <Rd>, <Rn>, <Rm> {, <shift>}	從暫存器減去被移位的暫存器值： $Rd = Rn + (Rm << \text{shift})$
SUBW.W <Rd>, <Rn>, #<immed_12>	從暫存器減去 12-bit 立即值： $Rd = Rd - \text{immed\_12}$
SXTB.W <Rd>, <Rm> {, <rotation>}	正負號擴充 byte 至 32 位元： $Rd = \text{sign\_extend}(\text{byte}(\text{rotate\_right}(Rm)))$ ，旋轉可能是 0-3 bytes
SXTH.W <Rd>, <Rm> {, <rotation>}	正負號擴充 half word 至 32 位元： $Rd = \text{sign\_extend}(\text{hword}(\text{rotate\_right}(Rm)))$ ，旋轉可能是 0-3 bytes
TBB.W {<Rn>, <Rm>}	表格跳躍 byte
TBH.W {<Rn>, <Rm>, LSL #1}	表格跳躍 half word
TEQ.W <Rn>, #<modify_constant (immed_12)>+	測試暫存器與立即值之間是否相等
TEQ.W <Rn>, <Rm> {, <shift>}+	測試暫存器與被移位的暫存器之間是否相等
TST.W <Rn>, #<modify_constant (immed_12)>+	測試暫存器被設定的位元值，藉著與立即值之 AND 運算以設定位元
TST.W <Rn>, <Rm> {, <shift>}+	測試暫存器被設定的位元值，藉著與被移位的暫存器之 AND 運算設定位元
UBFX.W <Rd>, <Rn>, #<lsb>, #<width>	從暫存器複製位元欄位並且以 0 擴充至 32 位元
UDIV.W <Rd>, <Rn>, <Rm>	無號數除法： $Rd = Rn / Rm$
UMLAL.W <RdLo>, <RdHi>, <Rn>, <Rm>	無號數 word 相乘並加上無號數擴充值至雙暫存器值： $\{RdHi, RdLo\} = (Rn * Rm) + \{RdHi, RdLo\}$
UMULL.W <RdLo>, <RdHi>, <Rn>, <Rm>	無號數暫存器值相乘： $\{RdHi, RdLo\} = (Rn * Rm)$

接下頁

組合語言		運算
USAT.W	<code>&lt;Rd&gt;, #&lt;imm&gt;, &lt;Rn&gt;, {&lt;shift&gt;}</code>	將被移位的暫存器值作無號飽和至立即值內的 bit 位置
UXTB.W	<code>&lt;Rd&gt;, &lt;Rm&gt;, {, &lt;rotation&gt;}</code>	無號擴充 byte 至 32 位元： Rd =
UXTH.W	<code>&lt;Rd&gt;, &lt;Rm&gt;, {, &lt;rotation&gt;}</code>	無號擴充 half word 至 32 位元： Rd = <code>unsigned_extend(hword(rotate_right(Rm)))</code> , 旋轉可以是 0-3 bytes
WFE.W		等待事件
WFI.W		等待中斷

## Appendix

**B**

## 16-Bit Thumb 指令與各種架構版本

- ✓ 支援的 16-bit Thumb 指令
- ✓ 支援的 32-bit Thumb-2 指令

大部分的 16-bit Thumb 指令在 v4T (ARM7TDMI) 架構裡是可用的。然而，其中一部分指令是到 v5、v6 與 v7 架構才加入。表 B-1 列上了這些指令。

表 B-1 在各種最近 ARM 架構版本 16-bit 指令支援的改變

指令	v4T	v5	v6	Cortex-M3 (v7-M)
BKPT	N	Y	Y	Y
BLX	N	Y	Y	僅有 BLX <reg>
CBZ, CBNZ	N	N	N	Y
CPS	N	N	Y	CPSIE</f>, CPSID</f>
CPY	N	N	Y	Y
NOP	N	N	N	Y
IT	N	N	N	Y
REV(各種形式)	N	N	Y	REV, REV16, REVSH
SEV	N	N	N	Y
SETEND	N	N	Y	N
SWI	Y	Y	Y	更改為 SVC
SXTB, SXTH	N	N	Y	Y
UXTB, UXTH	N	N	Y	Y
WFE, WFI	N	N	N	Y

## Appendix

## C

## Cortex-M3 例外快速參考

## 例外類型與致能

表 C-1 Cortex-M3 例外類型與其優先權組態的快速摘要

例外類型	名稱	優先權(等級位址)	致能
1	重置	-3	永遠
2	NMI	-2	永遠
3	硬錯誤	-1	永遠
4	MemManage	可程式的(0xE000ED18)	NVIC SHCSR (0xE000ED24) bit[16]
5	BusFault	可程式的(0xE000ED19)	NVIC SHCSR (0xE000ED24) bit[17]
6	用法錯誤	可程式的(0xE000ED1A)	NVIC SHCSR (0xE000ED24) bit[18]
7-10	-	-	-
11	SVC	可程式的(0xE000ED1F)	永遠
12	除錯監視器	可程式的(0xE000ED20)	NVIC DEMCR (0xE000EDFC) bit[16]
13	-	-	-
14	PendSV	可程式的(0xE000ED22)	永遠
15	SysTick	可程式的(0xE000ED23)	SYSTICK_CTRL_STAT (0xE000E010) bit[1]
16-255	IRQ	可程式的(0xE000E400)	NVIC SETEN (0xE000E100)

## 例外進堆疊之後的堆疊內容

表 C-2 例外堆疊框

位址	資料	推進次序
舊的 SP (N)->	(先前推進的資料)	-
(N-4)	PSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
新的 SP (N-32)->	R0	3

注意：如果使用了 double word 堆疊對齊特性，並且當例外發生時 SP 並未以 double word 對齊時，則堆疊框頂部可能會由((OLD\_SP-4) AND 0xFFFFFFF8)開始，並且表格其餘部分會往下移一個 word。

## Appendix D

# NVIC 暫存器快速參考

表 D-1 中斷控制器類型暫存器 (0xE000E004)

位元	名稱	類型	重置值	描述
4:0	INTLINESNUM	R	-	中斷輸入號碼以 32 為一階級 0 = 1 至 32 1 = 33 至 64 ...

表 D-2 SYSTICK 控制與狀態暫存器 (0xE000E010)

位元	名稱	類型	重置值	描述
16	COUNTFLAG	R	0	若計時器自從上次此暫存器被讀取之後計數至 0 則讀值為 1；在讀取時或現行計數器之值被清除時，將自動清除為 0
2	CLKSOURCE	R/W	0	0 = 外部參考時脈 (STCLK) 1 = 使用核心時脈
1	TICKINT	R/W	0	當 SYSTICK 計時器計數至 0 1 = 致能 SYSTICK 中斷的產生 0 = 並不產生中斷
0	ENABLE	R/W	0	致能 SYSTICK 計時器

表 D-3 SYSTICK 重載值暫存器(0xE000E014)

位元	名稱	類型	重置值	描述
23:0	RELOAD	R/W	0	計時器計數至 0 時重新載入數值

表 D-4 SYSTICK 現行值暫存器(0xE000E018)

位元	名稱	類型	重置值	描述
23:0	CURRENT	R/W	0	讀取以回傳計時器現有的值。 寫入以清除計數器為 0。 清除現有值亦清除了 SYSTICK 控制與狀態暫存器裡的 COUNTFLAG。

表 D-5 SYSTICK 校正值暫存器(0xE000E01C)

位元	名稱	類型	重置值	描述
31	NOREF	R	-	1 = 無外部參考時脈 (無 STCLK 可用) 0 = 有外部參考時脈可用
30	SKEW	R	-	1 = 校正值並非正好為 10 ms 0 = 校正值為精確的
23:0	TENMS	R/W	0	10 ms 的校正值；晶片設計師需要經由 Cortex-M3 輸入信號提供此值。若此值被讀取為 0 則無校正值可用

表 D-6 外部中斷 SETEN 暫存器 (0xE000E100-0xE000E11C)

位元	名稱	類型	重置值	描述
0xE000E100	SETENA0	R/W	0	致能外部中斷 #0-31 bit[0] for 中斷 #0 bit[1] for 中斷 #1 ... bit[31] for 中斷 #31
0xE000E104	SETENA1	R/W	0	致能外部中斷 #32-63
...	-	-	-	-

表 D-7 外部中斷 CLREN 暫存器 (0xE000E180-0xE000E19C)

位元	名稱	類型	重置值	描述
0xE000E180	CLRENA0	R/W	0	清除致能外部中斷 #0-31 bit[0] for 中斷 #0 bit[1] for 中斷 #1 ... bit[31] for 中斷 #31
0xE000E184	CLRENA1	R/W	0	清除致能外部中斷 #32-63
...	-	-	-	-

表 D-8 外部中斷 SETPEND 暫存器 (0xE000E200-0xE000E21C)

位元	名稱	類型	重置值	描述
0xE000E200	SETPEND0	R/W	0	置於等待外部中斷 #0-31 bit[0] for 中斷 #0 bit[1] for 中斷 #1 ... bit[31] for 中斷 #31
0xE000E204	SETPEND1	R/W	0	置於等待外部中斷 #32-63
...	-	-	-	-

表 D-9 外部中斷 CLRPEND 暫存器 (0xE000E280-0xE000E29C)

位元	名稱	類型	重置值	描述
0xE000E280	CLRPEND0	R/W	0	清除等待外部中斷 #0-31 bit[0] for 中斷 #0 bit[1] for 中斷 #1 ... bit[31] for 中斷 #31
0xE000E284	CLRPEND1	R/W	0	清除等待外部中斷 #32-63
...	-	-	-	-

# Jason 嘴書—EETOP 世界唯一貼

表 D-10 外部中斷 ACTIVE 暫存器 (位址 0xE000E300-0xE000E31C)

位元	名稱	類型	重置值	描述
0xE000E300	ACTIVE0	R	0	外部中斷 #0-31 的活動狀態 bit(0) for 中斷 #0 bit(1) for 中斷 #1 ... bit(31) for 中斷 #31
0xE000E304	ACTIVE1	R	0	外部中斷 #32-63 的活動狀態
...	-	-	-	-

表 D-11 外部中斷優先權等級暫存器 (0xE000E400-0xE000E4EF; 以 byte 位址列出)

位元	名稱	類型	重置值	描述
0xE000E400	PRI_0	R/W	0	優先權等級外部中斷 #0
0xE000E401	PRI_1	R/W	0	優先權等級外部中斷 #1
...	-	-	-	-
0xE000E41F	PRI_31	R/W	0	優先權等級外部中斷 #31
...	-	-	-	-

表 D-12 CPU ID 基底暫存器 (位址 0xE000ED00)

位元	名稱	類型	重置值	描述
31:24	IMPLEMENTER	R	0x41	實作碼；ARM 為 0x41
23:20	VARIANT	R	0x0 / 0x1 / 0x2	實作定義的變異號碼
19:16	Constant	R	0xF	常數
15:4	PARTNO	R	0xC23	零件號碼
3:0	REVISION	R	0x0 / 0x1	改版碼

表 D-13 中斷控制與狀態暫存器 (位址 0xE000ED04)

位元	名稱	類型	重置值	描述
31	NMIPENDSET	R/W	0	NMI 被置於等待
28	PENDSVSET	R/W	0	寫入 1 以將系統呼叫置於等待 讀取值顯示等待狀態
27	PENDSVCLR	W	0	寫入 1 以清除 PendSV 之等待狀態
26	PENDSTSET	R/W	0	寫入 1 以將 SYSTICK 例外置於等待 讀取值顯示等待狀態
25	PENDSTCLR	W	0	寫入 1 以清除 SYSTICK 之等待狀態
23	ISRPREEMPT	R	0	顯示等待中的中斷在下一步(為了除錯)將會活動
22	ISR PENDING	R	0	外部中斷置於等待(但不包括系統例外, 例如 NMI 錯誤)
21:12	VECTPENDING	R	0	被置於等待的 ISR 號碼
11	RETTTOBASE	R	0	當處理器執行例外處理程式時被設定為 1; 如果無其他等待中之例外, 則中斷返回時會進入執行緒層級
9:0	VECTACTIVE	R	0	現在正執行的中斷服務程式

表 D-14 向量表位移暫存器 (位址 0xE000ED08)

位元	名稱	類型	重置值	描述
29	TBLBASE	R/W	0	表格基底是位於 Code(0)或 RAM(1)
28:7	TBLOFF	R/W	0	程式區或 RAM 區域的表格位移值

表 D-15 應用中斷與重置控制暫存器 (位址 0xE000ED0C)

位元	名稱	類型	重置值	描述
31:16	VECTKEY	R/W	-	存取鑰：欲寫進此暫存器則需將 0x05FA 寫入此欄位，否則寫入將被忽略，upper half word 的讀回值為 0xFA05
15	ENDIANNESS	R	-	顯示了資料的 endianness；1 表 big endian(BE8), 0 表 little endian；此僅可於重置後改變
10:8	PRIGROUP	R/W	0	優先權群組
2	SYSRESETREQ	W	-	要求晶片控制邏輯產生重置
1	VECTCLRACTIVE	W	-	清除所有例外的活動狀態資訊；通常用於除錯或 OS 以允許系統從系統錯誤恢復(使用重置較為安全)
0	VECTRESET	W	-	重置 Cortex-M3 處理器(不包含除錯邏輯)，但此不會重置處理器之外的電路

表 D-16 系統控制暫存器(0xE000ED10)

位元	名稱	類型	重置值	描述
4	SEVONPEND	R/W	0	等待期間送出事件；如果新的中斷被置於等待，則會從 WFE 喚醒，而不管此中斷擁有的優先權是否高於現行的等級。
3	保留的	-	-	-
2	SLEEPDEEP	R/W	0	當進入睡眠模式時致能 SLEEPDEEP 輸出信號
1	SLEEPONEXIT	R/W	0	致能 SleeponExit 特性
0	保留的	-	-	-

表 D-17 組態控制暫存器(0xE000ED14)

位元	名稱	類型	重置值	描述
9	STKALIGN	R/W	0 或 1	強迫例外堆疊開始於 double word 對齊的位址！ 在 Cortex-M3 revision 1 裡預設值為 0，在 Cortex-M3 revision 2 裡預設值為 1
8	BFHFNIGN	R/W	0	在硬錯誤與 NMI 處理程式期間忽略資料匯流排錯誤

接下頁

1 自從 Cortex-M3 revision 1 以後方可使用。Revision 0 並無此特性

# Jason 嘴書—EETOP 世界唯一貼

位元	名稱	類型	重置值	描述
7:5	保留的	-	-	-
4	DIV_0_TRP	R/W	0	除以 0 時的陷阱
3	UNALIGN_TRP	R/W	0	未對齊存取時的陷阱
2	保留的	-	-	-
1	USERSETMPEND	R/W	0	若被設定為 1，則允許用戶程式寫入軟體觸發中斷暫存器
0	NONBASETHRDENA	R/W	0	致能 Nonbase 執行緒。如果被設定為 1，則允許例外處理程式在任意層級藉著控制返回值以返回到執行緒狀態

表 D-18 系統例外優先權等級暫存器(0xE000ED18-0xE000ED23；以 Byte 位址列出)

位元	名稱	類型	重置值	描述
0xE000ED18	PRI_4	R/W	0	記憶體管理錯誤的優先權等級
0xE000ED19	PRI_5	R/W	0	匯流排錯誤的優先權等級
0xE000ED1A	PRI_6	R/W	0	用法錯誤的優先權等級
0xE000ED1B	-	-	-	-
0xE000ED1C	-	-	-	-
0xE000ED1D	-	-	-	-
0xE000ED1E	-	-	-	-
0xE000ED1F	PRI_11	R/W	0	SVC 的優先權等級
0xE000ED20	PRI_12	R/W	0	除錯監視器的優先權等級
0xE000ED21	-	-	-	-
0xE000ED22	PRI_14	R/W	0	PendSV 的優先權等級
0xE000ED23	PRI_15	R/W	0	SYSTICK 的優先權等級

表 D-19 系統處理程式控制與狀態暫存器(0xE000ED24)

位元	名稱	類型	重置值	描述
18	USGFAULTENA	R/W	0	致能用法錯誤處理程式
17	BUSFAULTENA	R/W	0	致能匯流排錯誤處理程式
16	MEMFAULTENA	R/W	0	致能記憶體管理錯誤
15	SVCALLPENDED	R/W	0	SVC 被置於等待；SVCall 被啟動但又被更高的優先權之例外所取代
14	BUSFAULTPENDED	R/W	0	匯流排錯誤被置於等待；匯流排錯誤處理程式被啟動但又被更高的優先權之例外所取代
13	MEMFAULTPENDED	R/W	0	記憶體管理錯誤被置於等待；記憶體管理錯誤被啟動但又被更高的優先權之例外所取代
12	USGFAULTPENDED	R/W	0	用法錯誤被置於等待；用法錯誤被啟動但又被更高的優先權之例外所取代
11	SYSTICKACT	R/W	0	讀值為 1，如果 SYSTICK 例外處於活動
10	PENDSVACT	R/W	0	讀值為 1，如果 PendSV 例外處於活動

接下頁

位元	名稱	類型	重置值	描述
8	MONITORACT	R/W	0	讀值為 1，如果除錯監視例外處於活動
7	SVCALLACT	R/W	0	讀值為 1，如果 SVCall 例外處於活動
3	USGFAULTACT	R/W	0	讀值為 1，如果用法錯誤例外處於活動
1	BUSFAULTACT	R/W	0	讀值為 1，如果匯流排錯誤例外處於活動
0	MEMFAULTACT	R/W	0	讀值為 1，如果記憶體管理錯誤處於活動

注意：在 Cortex-M3 的 revision 0 中，並無 bit 12 (USGFAULTPENDED) 可用。

表 D-20 記憶體管理錯誤狀態暫存器(0xE000ED28；Byte 的大小)

位元	名稱	類型	重置值	描述
7	MMARVALID	-	0	顯示 MMAR 合法
6:5	-	-	-	-
4	MSTKERR	R/Wc	0	進堆疊錯誤
3	MUNSTKERR	R/Wc	0	去堆疊錯誤
2	-	-	-	-
1	DACCVIOL	R/Wc	0	資料存取違法
0	IACCVIOL	R/Wc	0	指令存取違法

表 D-21 瀝流排錯誤狀態暫存器(0xE000ED29 Byte 的大小)

位元	名稱	類型	重置值	描述
7	BFARVALID	-	0	顯示 BFAR 合法
6:5	-	-	-	-
4	STKERR	R/Wc	0	進堆疊錯誤
3	UNSTKERR	R/Wc	0	去堆疊錯誤
2	IMPRECISERR	R/Wc	0	不確切的資料存取違法
1	PRECISERR	R/Wc	0	確切的資料存取違法
0	IBUSERR	R/Wc	0	指令存取違法

表 D-22 用法錯誤狀態暫存器(0xE000ED2A；Half Word 大小)

位元	名稱	類型	重置值	描述
9	DIVBYZERO	R/Wc	0	顯示發生了除以 0 (僅可在 DIV_0_TRP 被設定時設定)
8	UNALIGNED	R/Wc	0	顯示發生了未對齊的存取錯誤
7:4	-	-	-	-
3	NOCP	R/Wc	0	試圖執行輔助處理器指令
2	INVPC	R/Wc	0	試圖在 EXC_RETURN 號碼裡以不正確的數值去執行例外
1	INVSTATE	R/Wc	0	試圖切換到不合法的狀態(例如，ARM 狀態)
0	UNDEFINSTR	R/Wc	0	試圖執行未定義的指令

# Jason 嘴書—EETOP 世界唯一貼

表 D-23 硬錯誤狀態暫存器(0xE000ED2C)

位元	名稱	類型	重置值	描述
31	DEBUGEV	R/Wc	0	顯示硬錯誤被除錯事件所觸發
30	FORCED	R/Wc	0	顯示因為匯流排錯誤、記憶體管理錯誤、或用法錯誤而採取了硬錯誤
29:2	-	-	-	-
1	VECTBL	R/Wc	0	顯示因為向量擷取失敗而造成硬錯誤
0	-	-	-	-

表 D-24 除錯錯誤狀態暫存器(0xE000ED30)

位元	名稱	類型	重置值	描述
4	EXTERNAL	R/Wc	0	EDBGRO 信號被宣稱
3	VCATCH	R/Wc	0	發生了向量擷取
2	DWTTRAP	R/Wc	0	發生了 DWT 呼合
1	BKPT	R/Wc	0	執行了 BKPT 指令
0	HALTED	R/Wc	0	在 NVIC 裡作了暫停請求

表 D-25 記憶體管理位址暫存器 MMAR (0xE000ED34)

位元	名稱	類型	重置值	描述
31:0	MMAR	R	-	造成記憶體管理錯誤的位址

表 D-26 汇流排錯誤管理位址暫存器 BFAR (0xE000ED38)

位元	名稱	類型	重置值	描述
31:0	BFAR	R	-	造成匯流排錯誤的位址

表 D-27 輔助的錯誤狀態暫存器(0xE000ED3C)

位元	名稱	類型	重置值	描述
31:0	Vender controlled	R/Wc	-	供應商控制的(選擇性的)

表 D-28 MPU Type 暫存器(0xE000ED90)

位元	名稱	類型	重置值	描述
23:16	IREGION	R	0	受到此 MPU 支援的指令區域之數量；因為 ARM v7-M 架構使用了統合的 MPU，此值永遠為 0
15:8	DREGION	R	0 或 8	受到此 MPU 支援的區域之數量；在 Cortex-M3 中此值不是 0(沒有 MPU)就是 8(有 MPU)
0	SEPARATE	R	0	因為 MPU 為統合的故此值永遠為 0

表 D-29 MPU 控制暫存器(0xE000ED94)

位元	名稱	類型	重置值	描述
2	PRIV/DEFENA	R/W	0	致能特權的預設記憶體映射。
1	HFNMIENA	R/W	0	如果設定為 1，則在硬錯誤處理程式與 NMI 處理程式期間會致能 MPU；否則 MPU 不會因為硬錯誤處理程式與 NMI 而致能。
0	ENABLE	R/W	0	如果設定為 1，則致能 MPU。

表 D-30 MPU 區域號碼暫存器(0xE000ED98)

位元	名稱	類型	重置值	描述
7:0	REGION	R/W	-	選擇正被編程的區域

表 D-31 MPU 區域基底位址暫存器(0xE000ED9C)

位元	名稱	類型	重置值	描述
31:N	ADDR	R/W	-	區域的基底位址；N 依照區域大小而定。
4	VALID	R/W	-	若此為 1，則 bit[3:0]定義的 REGION 將會在此程式步驟裡使用；否則，會使用由 MPU 區域號碼暫存器選擇的區域。
3:0	REGION	R/W	-	若 VALID 為 1，此欄位會蓋過 MPU 區域號碼暫存器的選擇，否則會被忽略。

表 D-32 MPU 區域基底屬性與大小暫存器(0xE000EDA0)

位元	名稱	類型	重置值	描述
31:29	保留的	-	-	-
28	XN	R/W	-	指令存取除能 (1 = 除能)
27	保留的	-	-	-
26:24	AP	R/W	-	資料存取許可欄位
23:22	保留的	-	-	-
21:19	TEX	R/W	-	類型擴展欄位
8	S	R/W	-	可共享的
17	C	R/W	-	可快取的
16	B	R/W	-	可緩衝的
15:8	SRD	R/W	-	次區域除能
7:6	保留的	-	-	-
5:1	REGION SIZE	R/W	-	MPU 保護區域大小
0	SZENABLE	R/W	-	區域致能

# Jason 嘴書—EETOP 世界唯一貼

表 D-33 MPU Alias 暫存器(0xE000EDA4~0xE000EDB8)

位址	名稱	描述
0xE000EDA4	D9C 的 Alias	MPU Alias 1 區域基底位址暫存器
0xE000EDA8	DA0 的 Alias	MPU Alias 1 區域屬性與大小暫存器
0xE000EDAC	D9C 的 Alias	MPU Alias 2 區域基底位址暫存器
0xE000EDB0	DA0 的 Alias	MPU Alias 2 區域屬性與大小暫存器
0xE000EDB4	D9C 的 Alias	MPU Alias 3 區域基底位址暫存器
0xE000EDB8	DA0 的 Alias	MPU Alias 3 區域屬性與大小暫存器

表 D-34 除錯暫停控制與狀態暫存器(0xE000EDF0)

位元	名稱	類型	重置值	描述
31:16	KEY	W	-	除錯鑰；欲寫值進此暫存器則需將 0xA05F 之值寫入此欄位，否則寫入將會被忽略
25	S_RESET_ST	R	-	核心已被重置或正被重置；讀取將清除此位元
24	S_RETIRE_ST	R	-	自先前讀取之後的指令已完成；讀取將清除此位元
19	S_LOCKUP	R	-	當此位元為 1，核心會處於鎖住狀態
18	S_SLEEP	R	-	當此位元為 1，核心會處於睡眠狀態
17	S_HALT	R	-	當此位元為 1，核心會被暫停
16	S_REGRDY	R	-	暫存器讀取/寫入運算已完成
15:6	保留的	-	-	保留的
5	C_SNAPSTALL	R/W	-	用來中斷暫停的記憶體存取
4	保留的	-	-	保留的
3	C_MASKINTS	R/W	-	當步進時遮罩中斷；僅可在處理器被暫停時作修改
2	C_STEP	R/W	-	處理器單步運算；僅在 C_DEBUGEN 被設定時有效
1	C_HALT	R/W	-	暫停處理器核心；僅在 C_DEBUGEN 被設定時有效
0	C_DEBUGEN	R/W	-	致能暫停模式除錯

表 D-35 除錯核心暫存器選擇器暫存器(0xE000EDF4)

位元	名稱	類型	重置值	描述
16	REGWrR	W	-	資料傳輸的方向：寫入 = 1，讀取 = 0
15:5	保留的	-	-	-
4:0	REGSEL	W	-	被存取的暫存器：00000 = R0 00001 = R1 ... 01111 = R15 10000 = xPSR/旗標 10001 = MSP (主要堆疊指位器) 10010 = PSP (程序堆疊指位器) [31:24]控制 [23:16]FAULTMASK [15:8]BASEPRI [7:0]PRIMASK 其他被保留

表 D-36 除錯核心暫存器資料暫存器(0xE000EDF8)

位元	名稱	類型	重置值	描述
31:0	Data	R/W	-	用來保留暫存器讀取結果或者寫資料到選定暫存器的暫存器

表 D-37 除錯例外與監視器控制暫存器(0xE000EDFC)

位元	名稱	類型	重置值	描述
24	TRCENA	R/W	0	追蹤系統效能；欲使用 DWT、ETM、ITM、與 TPIU 則此位元必須被設定為 1
23:20	保留的	-	-	保留的
19	MON_REQ	R/W	0	顯示除錯監視器是因手動等待要求而造成，而非硬體除錯事件
18	MON_STEP	R/W	0	處理器單步運算；僅在 MON_EN 被設定時有效
17	MON_PEND	R/W	0	將監視器例外要求置於等待；在優先權允許之下，核心將進入監視器例外
16	MON_EN	R/W	0	致能除錯監視器例外
15:11	保留的	-	-	保留的
10	VC_HARDERR	R/W	0	在硬錯誤上的除錯陷阱
9	VC_INTERR	R/W	0	在中斷/例外服務錯誤的除錯陷阱
8	VC_BUSERR	R/W	0	在匯流排錯誤的除錯陷阱
7	VC_STATERR	R/W	0	在用法錯誤之狀態錯誤的除錯陷阱
6	VC_CHKERR	R/W	0	在用法錯誤被致能的檢查錯誤的除錯陷阱(例如，未對齊的，除以 0 等)

接下頁

位元	名稱	類型	重置值	描述
5	VC_NOCPERR	R/W	0	在用法錯誤的除錯陷阱，並無輔助處理器的錯誤
4	VC_MMERR	R/W	0	在記憶體管理錯誤的除錯陷阱
3:1	保留的	-	-	保留的
0	VC_CORERESET	R/W	0	在核心重置的除錯陷阱

表 D-38 軟體觸發中斷暫存器(0xE000EF00)

位元	名稱	類型	重置值	描述
8:0	INTID	W	-	寫入中斷號碼以設定中斷的等待位元

表 D-39 NVIC 週邊 ID 暫存器 (0xE000EF00-0xE000EFFC)

位元	名稱	類型	重置值	描述
0xE000EF00	PERIPHID4	R	0x04	週邊 ID 暫存器
0xE000EF04	PERIPHID5	R	0x00	週邊 ID 暫存器
0xE000EF08	PERIPHID6	R	0x00	週邊 ID 暫存器
0xE000EF0C	PERIPHID7	R	0x00	週邊 ID 暫存器
0xE000EFE0	PERIPHID0	R	0x00	週邊 ID 暫存器
0xE000EFE4	PERIPHID1	R	0xB0	週邊 ID 暫存器
0xE000EFE8	PERIPHID2	R	0x0B/0x1B	週邊 ID 暫存器
0xE000EFEC	PERIPHID3	R	0x00	週邊 ID 暫存器
0xE000EFF0	PCELLID0	R	0xD	元件 ID 暫存器
0xE000EFF4	PCELLID1	R	0xE0	元件 D 暫存器
0xE000EFF8	PCELLID2	R	0x05	元件 ID 暫存器
0xE000EFFC	PCELLID3	R	0xB1	元件 ID 暫存器

注意：Cortex-M3 revision 0 的 PERIPHID2 值為 0xB；revision 1 為 0x1B

## Appendix



## Cortex-M3 問題解決指南

## 綜觀

使用 Cortex-M3 的一項挑戰，在於程式發生錯誤時找出問題所在。Cortex-M3 處理器提供了一些錯誤狀態暫存器以協助解決問題(參見表 E-1)。

表 E-1 在 Cortex-M3 上的錯誤狀態暫存器

位址	暫存器	全名	大小
0xE000ED28	MMSR	記憶體管理錯誤狀態暫存器	Byte
0xE000ED29	BFSR	匯流排錯誤狀態暫存器	Byte
0xE000ED2A	UFSR	用法錯誤狀態暫存器	Half word
0xE000ED2C	HFSR	硬錯誤狀態暫存器	Word
0xE000ED30	DFSR	除錯錯誤狀態暫存器	Word
0xE000ED3C	AFSR	輔助錯誤狀態暫存器	Word

藉著使用一個 word 傳輸指令，可以一次存取 MMSR、BFSR 與 UFSR 暫存器。在此情形下，複合的錯誤狀態暫存器被稱之為組態錯誤狀態暫存器(CFSR)。

另一項重要的資訊是被堆疊的程式計數器(PC)，它被置於記憶體位址[SP + 0x24]。因為在 Cortex-M3 裡有兩個堆疊指位器，所以在取得被堆疊的 PC 之前，錯誤處理程式可能需要先判斷使用的堆疊指位器是哪一個。

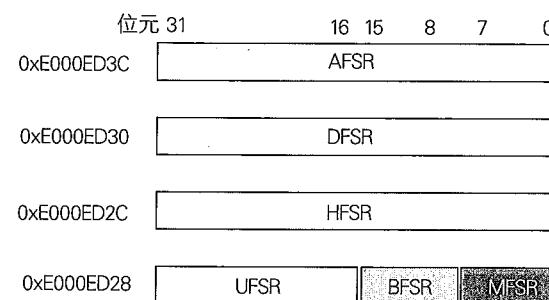


圖 E-1 存取錯誤狀態暫存器

除此之外，就匯流排錯誤與記憶體管理錯誤而言，你也可以判斷出造成錯誤的位址，這可藉著存取 MemManage(記憶體管理)錯誤位址暫存器(MMAR)與匯流排錯誤位址暫存器(BFAR)來達成。只有當 MMAVALID 位元(在 MMSR 內)或 BFARVALID

位元(在 BFAR 內)被設定時，此兩個暫存器的內容方為合法的。實體上 MMAR 與 BFAR 為相同的暫存器，故在同一時間其中僅有一者是合法的(參見表 E-2)。

表 E-2 在 Cortex-M3 上的錯誤位址暫存器

位址	暫存器	全名	大小
0xE000ED34	MMAR	記憶體管理錯誤位址暫存器	Word
0xE000ED38	BFAR	匯流排錯誤位址暫存器	Word

最後，當進入錯誤處理程式時，連結暫存器(LR)也提供了有關造成錯誤的線索。以不合法的 EXC\_RETURN 值造成的錯誤為例，在進入錯誤處理程式後的 LR 值顯示了當錯誤發生時先前的 LR 值。錯誤處理程式可以報告錯誤的 LR 值，讓軟體程式師可以利用此資訊以檢查為何 LR 會得到非法的返回值。

## 開發錯誤處理程式

在大多數的情形下，用於開發與用於真實執行系統的錯誤處理程式有所不同。對軟體開發來說，錯誤處理程式應該專注於回報錯誤的類型；而作為執行系統的錯誤處理程式將可能專注於回復系統的動作。在此我們僅探討錯誤報告部分，因為系統回復動作高度地依賴設計類型與要求。

在複雜的軟體裡，與其在錯誤處理程式內輸出結果，可以改為將這些暫存器的內容複製到記憶體區塊，並在稍後使用 PendSV 報告錯誤細節。這樣可以避免顯示或輸出程式的潛在可能錯誤而造成鎖住。對簡單的應用，這可能無關緊要，而還是可以在錯誤處理程式內直接輸出錯誤細節。

## 報告錯誤狀態暫存器

錯誤處理程式最基本的步驟在於回報錯誤狀態暫存器值，包括：

- ◆ UFSR
- ◆ BFSR
- ◆ MMSR
- ◆ HFSR
- ◆ DFSR
- ◆ AFSR(選擇性的)

## 報告被堆疊的 PC

取得被堆疊 PC 的步驟類似於本書中 SVC 的例子。

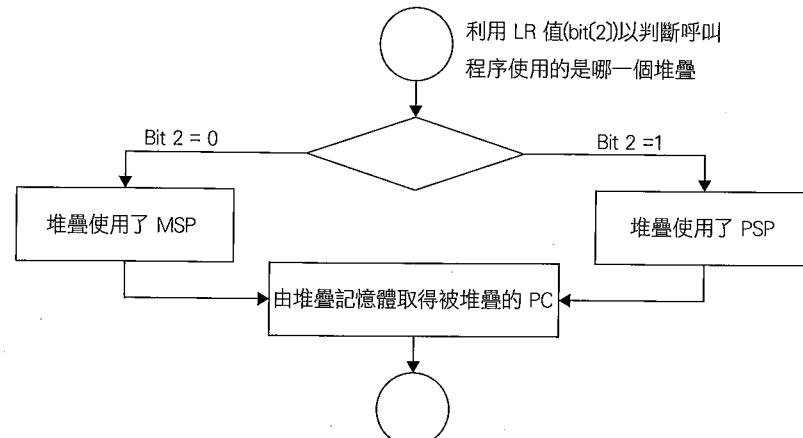


圖 E-2 從堆疊記憶體取得被堆疊的 PC 值

此程序可以以組合語言執行如下：

TST	LR, #0x4	; 測試在 LR bit 2 的 EXC_RETURN 值
ITTEE	EQ	; 如果 0 (相等) 則
MRSEQ	R0, MSP	; 使用了主要堆疊，將 MSP 放進 R0
LDREQ	R0, [R0, #24]	; 由堆疊取得被堆疊的 PC
MRSNE	R0, PSP	; 否則，使用了程序堆疊，將 PSP 放進 R0
LDRNE	R0, [R0, #24]	; 由堆疊取得被堆疊的 PC

為了有助於除錯，我們也應該產生一個反組譯的程式列表檔以利尋找問題來源。

## 讀取錯誤位址暫存器

錯誤位址暫存器在 MMARVALID 或 BFARVALID 被清除之後會被消除。為了正確地存取錯誤位址暫存器，應該採取下面的程序：

1. 讀取 BFAR/MMAR。
2. 讀取 BFARVALID/MMARVALID；若其值為 0 則需要丟棄 BFAR/MMAR 的讀取值。

3. 清除 BFARVALID/MMARVALID。

之所以用此程序取代先讀取合法位元的理由，在於避免讀取合法位元之後，錯誤處理程序被另一個更高優先權錯誤處理程式所強佔，這樣可能會導致如下出錯的錯誤報告序列：

1. 讀取 BFARVALID/MMARVALID。
2. 合法位元已經設定，準備讀取 BFAR/MMAR。
3. 更高的優先權例外強佔了現存的錯誤處裡程式，產生了另外的錯誤，造成另外的錯誤處理程式會被執行。
4. 更高優先權錯誤處理程式清除 BFARVALID/MMARVALID 位元，而造成 BFAR/MMAR 被消除。
5. 在返回原先錯誤處理程式之後，讀取 BFAR/MMAR，但現在的內容並不合法，導致回報不正確的錯誤位址。

因此，在讀取了錯誤位址暫存器之後，讀取 BFARVALID/MMARVALID 以確定位址暫存器內容的合法性是相當重要的。

## 清除錯誤狀態位元

錯誤報告完成之後，應該清除 FSR 裡的錯誤狀態位元，這樣下次錯誤處理程式執行時，先前的錯誤才不會混淆錯誤處理程式。除此之外，如果沒有清除錯誤位址合法位元，下次發生錯誤時就不會更新錯誤位址暫存器。

## 其他

在錯誤處理程式的一開始，經常需要保留 LR 的內容。然而，如果錯誤是因堆疊錯誤所造成，則把 LR 推進堆疊只會使情況變得更糟糕。如我們所知，R0-R3 與 R12 應該已經被保留了，故我們在做任何函數呼叫之前，可以把 LR 複製到這些暫存器其中之一。

## 在 C 裡報告被堆疊的暫存器值與錯誤狀態暫存器

大多數 Cortex-M3 開發者使用 C 開發專案。然而，在 C 中難以找出並直接存取堆疊框(被堆疊的值)，因為在 C 裡無法取得堆疊指標值。為了以 C 在錯誤處理程式內報告堆疊框內容，你需要使用一段簡短的組合語言程式碼以取得堆疊指標值，然後將它當作參數傳給 C 裡面的錯誤報告函數。此機制與第十二章裡的 SVC 範例("Using SVC with C")相同。在下面的範例使用嵌入式組合語言，它可以跟 RealView Development Suite (RVDS) 以及 KEIL RealView Microcontroller Development Kit (ReadView-MDK)一起使用。

此程式的第一部分是一個組合語言包裹程式。向量表中應該在硬錯誤進入點記錄此包裹程式的起始位址。此包裹程式複製正確的堆疊指位器值到 R0 裡面，並且將它當作參數傳給 C 函數。

```
// 以組合語言寫的硬錯誤處理程式
// 它取出堆疊框的位置並且將它當作指標傳給 C 裡的處理程式
__asm void hard_fault_handler_asm(void)
{
    IMPORT      hard_fault_handler_c
    TST         LR, #4
    ITE         EQ
    MRSEQ      R0, MSP
    MRSNE      R0, PSP
    B          hard_fault_handler_c
}
```

處理程式的第二部分是 C 程式。在此我們展示如何存取被堆疊的暫存器內容與錯誤狀態暫存器。

```
// 以 C 寫的硬錯誤處理程式
// 以堆疊框位置當作輸入參數
void hard_fault_handler_c(unsigned int * hardfault_args)
{
    unsigned int stacked_r0;
    unsigned int stacked_r1;
    unsigned int stacked_r2;
```

```
unsigned int stacked_r3;
unsigned int stacked_r12;
unsigned int stacked_lr;
unsigned int stacked_pc;
unsigned int stacked_psr;

stacked_r0 = ((unsigned long) hardfault_args[0]);
stacked_r1 = ((unsigned long) hardfault_args[1]);
stacked_r2 = ((unsigned long) hardfault_args[2]);
stacked_r3 = ((unsigned long) hardfault_args[3]);

stacked_r12 = ((unsigned long) hardfault_args[4]);
stacked_lr = ((unsigned long) hardfault_args[5]);
stacked_pc = ((unsigned long) hardfault_args[6]);
stacked_psr = ((unsigned long) hardfault_args[7]);

printf ("[Hard fault handler]\n");
printf ("R0 = %x\n", stacked_r0);
printf ("R1 = %x\n", stacked_r1);
printf ("R2 = %x\n", stacked_r2);
printf ("R3 = %x\n", stacked_r3);
printf ("R12 = %x\n", stacked_r12);
printf ("LR = %x\n", stacked_lr);
printf ("PC = %x\n", stacked_pc);
printf ("PSR = %x\n", stacked_psr);
printf ("BFAR = %x\n",
       (*((volatile unsigned long *) (0xE000ED38))));

printf ("CFSR = %x\n",
       (*((volatile unsigned long *) (0xE000ED28))));

printf ("HFSR = %x\n",
       (*((volatile unsigned long *) (0xE000ED2C))));

printf ("DFSR = %x\n",
       (*((volatile unsigned long *) (0xE000ED30))));

printf ("AFSR = %x\n",
       (*((volatile unsigned long *) (0xE000ED3C))));

exit(0); // 終止

return;
}
```

請注意如果堆疊指位器指向不合法的記憶體區域(例如因為堆疊溢位)，則此處理程式將不會正確地運作。此將影響所有 C 程式，因為在大多數的情形下 C 函數需要堆疊。

## 了解錯誤造成的原因

在獲得我們需要的資訊之後，我們可以確立問題的肇因。表 E-3~E-7 列出一些錯誤產生的共同原因。

表 E-3 記憶體管理錯誤狀態暫存器

位元	可能的造成原因
MSTKERR	在進堆疊期間發生錯誤(例外開始): 1. 堆疊指位器遭受破壞。 2. 堆疊大小成長過大, 到達未被 MPU 定義或不為 MPU 組態允許的區域。
MUNSTKERR	在去堆疊期間發生錯誤(例外結束), 如果進堆疊並無錯誤但在去堆疊期間發生錯誤, 則可能是: 1. 堆疊指位器在例外期間遭受破壞。 2. MPU 組態被例外處理程式所改變。
DACCVIOL	違反 MPU 設定所定義的記憶體存取保護。例如, 用戶應用程式試圖存取僅可特權存取的區域。
IACCVIOL	1. 違反 MPU 設定所定義的記憶體存取保護。例如, 用戶應用程式試圖存取僅可特權存取的區域。被堆疊的 PC 可能可以找出造成問題的程式碼。 2. 跳躍到不可執行的區域。 3. 不合法的例外返回程式碼。 4. 例外向量表中有不合法的項目。例如, 載入一個傳統 ARM 程式的可執行映像到記憶體, 或在向量表設定之前發生了例外。 5. 在例外處理期間, 置於堆疊的 PC 遭受破壞。

表 E-4 決流排錯誤狀態暫存器

位元	可能的造成原因
STKERR	在進堆疊期間發生錯誤(例外開始): 1. 堆疊指位器遭受破壞。 2. 堆疊大小成長過大, 到達未定義的記憶體區域。 3. 使用了未初始化的 PSP。
UNSTKERR	在去堆疊期間發生錯誤(例外結束), 如果進堆疊並無錯誤但在去堆疊期間錯誤發生了, 則可能是堆疊指位器在例外期間遭受破壞。
IMPRECISERR	在資料存取期間的匯流排錯誤。可能造成原因: 設備未被初始化; 在用戶模式裡存取僅可特權存取的設備; 或者傳輸量對於特定的設備並不正確。

接下頁

位元	可能的造成原因
PRECISERR	在資料存取期間的匯流排錯誤。錯誤的位址可能由 BFAR 指出。匯流排錯誤可能造成的原因: 設備未被初始化、在用戶模式裡存取僅可特權存取的設備、或者傳輸量對於特定的設備並不正確。
IBUSERR	1. 違反 MPU 設定所定義的記憶體存取保護。例如, 用戶應用程式試圖跳躍至僅可特權存取的區域。 2. 跳躍到不可執行的區域。 3. 不合法的例外返回程式碼。 4. 例外向量表中有不合法的項目。例如, 載入一個傳統 ARM 程式的可執行映像到記憶體, 或在向量表被設定之前發生了例外。 5. 在例外處理期間, 置於堆疊的 PC 遭受破壞。

表 E-5 用法錯誤狀態暫存器

位元	可能的造成原因
DIVBYZERO	發生除以零並且 DIV_0_TRP 被設定。可利用被堆疊的 PC 以找出造成錯誤的程式碼。
UNALIGNED	在 UNALIGN_TRP 被設定下, 試圖作未對齊的存取。可利用被堆疊的 PC 以找出造成錯誤的程式碼。
NOCP	嘗試執行輔助處理器指令。可利用被堆疊的 PC 以找出造成錯誤的程式碼。
INVPC	在例外返回期間 EXC_RETURN 裡有不合法的值。例如: ◦ 以 EXC_RETURN = 0xFFFFFFFF1 返回至執行緒層級 ◦ 以 EXC_RETURN = 0xFFFFFFFF9 返回至處理程式 為了探究問題所在, LR 的目前值提供了例外返回失敗時的 LR 值。 2. 不合法的例外活動狀態。例如: ◦ 例外返回但現行例外的例外活動位元已經被清除。可能的原因是使用了 VECTCLRACTIVE 或者清除了 NVIC SHCSR 裡的例外活動狀態。 ◦ 例外返回至執行緒層級, 但還有例外活動位元依然設為活動中。 3. 堆疊遭受破壞因而造成被堆疊的 IPSR 不正確。對 INVPC 錯誤而言, 堆疊中的 PC 顯示了主程式/被強佔的程式被錯誤例外中斷之處。為了探究造成問題所在, 最好使用 ITM 裡的例外追蹤特性。 4. 對現行的指令 ICI/T 位元不合法。可能發生此情形的原因: 當多重的載入/儲存指令被中斷, 並且在中斷處理程式期間被堆疊的 PC 遭到修改。當中斷返回發生時, 非零的 ICI 位元被應用到並無使用 ICI 位元的指令。相同的問題可能也會因為破壞了被堆疊的 PSR 而發生。
INVSTATE	1. 載入跳躍目標位址至 PC 而 LSB 為零。堆疊中的 PC 應該可以指出跳躍目標。 2. 向量表裡向量位址的 LSB 為零。被堆疊的 PC 應該可以指出例外處理程式的開始處。 3. 在例外處理期間被堆疊的 PSR 遭到破壞, 所以在例外之後核心試圖以 ARM 狀態返至被中斷的程式。
UNDEFINSTR	1. 使用 Cortex-M3 未支援的指令。 2. 壞的/遭受破壞的記憶體內容。 3. 在連結階段載入 ARM 目的程式。請檢查編譯步驟。 4. 指令對齊問題。例如, 如果使用了 GNU 工具鏈, .ascii 之後忽略了.align 可能會造成下一個指令不會對齊(以奇數記憶體位址開始而非 half word 位址)。

表 E-6 硬錯誤狀態暫存器

位元	可能的造成原因
DEBUGEVF	錯誤會因為下列除錯事件而造成： 1. 中斷點/觀察點事件。 2. 如果正在執行錯誤處理程式，則可能造成錯誤的原因，是因為執行了 BKPT 而未啟能監視器處理程式(MON_EN=0)且暫停除錯未被啟能(C_DEBUGEN=0)。在預設下，一些 C 編譯器可能包含使用 BKPT 的 semihosting(半主機)程式。
FORCED	1. 在 SVC/監視器或者其他具相同或更高優先權的處理程式之內，試圖執行 SVC/BKPT。 2. 錯誤發生了，但與其相關的處理程式被除能，或者正執行具相同或更高優先權的例外或因為設定了例外遮罩，使得相關的處理程式不能被啟動。
VECTBL	向量擷取失敗，可能造成原因如下： 向量擷取期間匯流排錯誤。 2. 不正確的向量表位移設定。

表 E-7 除錯錯誤狀態暫存器

位元	可能的造成原因
EXTERNAL	EDBGRQ 信號已經被宣稱。
VCATCH	向量捕捉事件已經發生。
DWTTRAP	DWT 觀察點事件已經發生。
BKPT	1. 中斷點指令被執行 2. FPB 單元產生了中斷點事件。 在某些情形下，作為 semihosting 除錯設定的一部份，BKPT 指令會被 C 啟動程式插入。在實際的應用程式中應該將此移除。細節請參考你的編譯器文件。
HALTED	暫停 NVIC 裡的請求。

## 其他可能的問題

一些其他常見的問題列於表 E.8 裡。

表 E-8 其他可能的問題

位元	可能的造成原因
程式未能執行	向量表可能有設定不正確： 被置於不正確的記憶體位置。 向量(包括硬錯誤處理程式的 LSB 未設定為 1。 在向量表裡(有如在傳統 ARM 處理器裡的向量表中)使用跳躍指令。 產生反組譯程式列表以檢查是否正確地設定了向量表。
程式執行一些指令之後當機	可能造成原因如下：不正確的 endian 設定或不正確的堆疊指位器設定(檢查向量表)，或者使用傳統 ARM 處理器的 C 目的程式庫(ARM 程式而非 Thumb 程式)。造成錯誤的 C 目的程式庫可能是 C 啟動程式的一部份，請檢查編譯器與連結器選項以確定使用了 Thumb 或 Thumb-2 程式庫檔案。



好書能增進知識 提高學習效率 卓越的品質是旗標的信念與堅持