

基于 ARM 的嵌入式系统程序开发要点（一）

—— 嵌入式程序开发过程

ARM®系列微处理器作为全球 16/32 位 RISC 处理器市场的领先者，在许多领域内得到了成功的应用。近年来，ARM 在国内的应用也得到了飞速的发展，越来越多的公司和工程师在基于 ARM 的平台上面开发自己的产品。

与传统的 4/8 位单片机相比，ARM 的性能和处理能力当然是遥遥领先的，但与之相应，ARM 的系统设计复杂度和难度，较之传统的设计方法也大大提升了。本文旨在通过讨论系统程序设计中的几个基本方面，来说明基于 ARM 的嵌入式系统程序开发的一些特点，并提出和解决了一些常见的问题。

文章分成几个相对独立的章节刊载。第一部分讨论基于 ARM 的嵌入式程序开发和移植过程中的一些基本概念。

1. 嵌入式程序开发过程

不同于通用计算机和工作站上的软件开发工程，一个嵌入式程序的开发过程具有很多特点和不确定性。其中最重要的一点是软件跟硬件的紧密耦合特性。

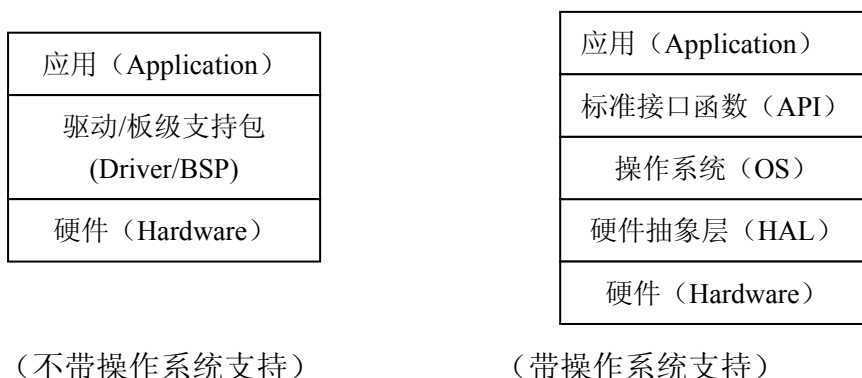


图-1：两类不同的嵌入式系统结构模型

这是两类简化的嵌入式系统层次结构图。由于嵌入式系统的灵活性和多样性，上面图中各个层次之间缺乏统一的标准，几乎每一个独立的系统都不一样。这样就给上层的软件设计人员带来了极大地困难。第一，在软件设计过程中过多地考虑硬件，给开发和调试都带来了很多不便；第二，如果所有的软件工作都需要在硬件平台就绪之后进行，自然就延长了整个的系统开发周期。这些都是应该从方法上加以改进和避免的问题。

为了解决这个问题，工程和设计人员提出了许多对策。首先在应用与驱动（或 API）这一层接口，可以设计成相对统一的一些接口函数，这对于具体的某一个开发平台或在某个公司内部，是完全做得到的。这样一来，就大大提高了应用层

软件设计的标准化程度，方便了应用程序在跨平台之间的复用和移植。

对于驱动/硬件抽象这一层，因为直接驱动硬件，其标准化变得非常困难甚至不太可能。但是为了简化程序的调试和缩短开发周期，我们可以在特定的 EDA 工具环境下面进行开发，通过后再进行移植到硬件平台的工作。这样既可以保证程序逻辑设计的正确性，同时使得软件开发可平行甚至超前于硬件开发进程。

我们把脱离于硬件的嵌入式软件开发阶段称之为“PC 软件”的开发，可以用下面的图来示意一个嵌入式系统程序的开发过程。

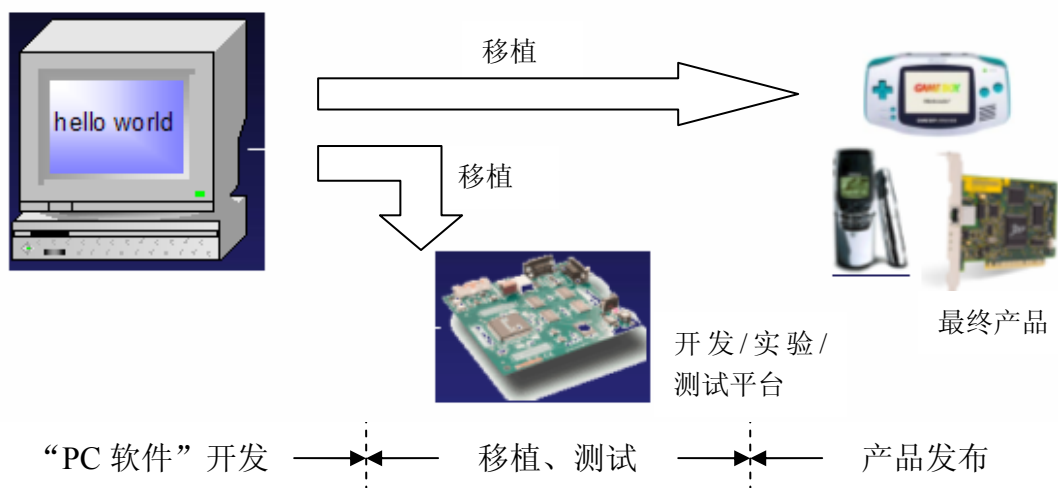


图-2: 嵌入式系统产品的开发过程

在“PC 软件”开发阶段，可以用软件仿真，即指令集模拟的方法，来对用户程序进行验证。在 ARM 公司的开发工具中，ADS®内嵌的 ARMulator 和 RealView® 开发工具中的 ISS，都提供了这项功能。在模拟环境下，用户可以设置 ARM 处理器的型号、时钟频率等，同时还可以配置存储器访问接口的时序参数。程序在模拟环境下运行，不但能够进行程序的运行流程和逻辑测试，还能够统计系统运行的时钟周期数、存储器访问周期数、处理器运行时的流水线状态(有效周期、等待周期、连续和非连续访问周期)等信息。这些宝贵的信息是在硬件调试阶段都无法取得的，对于程序的性能评估非常有价值。

为了更加完整和真实地模拟一个目标系统，ARMulator 和 ISS 还提供了一个开放的 API 编程环境。用户可以用标准 C 来描述各种各样的硬件模块，连同工具提供的内核模块一起，组成一个完整的“软”硬件环境。在这个环境下面开发的软件，可以更大程度地接近最终的目标。

利用这种先进的 EDA 工具环境，极大地方便了程序开发人员进行嵌入式开发的工作。当完成一个“PC 软件”的开发之后，只要进行正确的移植，一个真正的嵌入式软件就开发成功了。而移植过程是相对比较容易形成一套规范的流程的，其中三个最重要的方面是：

- 考虑硬件对库函数的支持

- 符合目标系统上的存储器资源分布
- 应用程序运行环境的初始化

2. 开发工具环境里面的库函数

如果用户程序里调用了跟目标相关的一些库函数，则在应用前需要裁剪这些函数以适合在目标上允许的要求。主要需要考虑以下三类函数：

- 访问静态数据的函数
- 访问目标存储器的函数
- 使用 semihosting（半主机）机制实现的函数

这里所指的 C 库函数，除了 ISO C 标准里面定义的函数以外，还包括由编译工具提供的另外一些扩展函数和编译辅助函数。

2. 1 裁剪访问静态数据的函数

库函数里面的静态数据，基本上都是在头文件里面加以定义的。比如 CTYPE 类库函数，其返回值都是通过预定义好的 CTYPE 属性表来获得的。比如，想要改变 `isalpha()` 函数的缺省判断，则需要修改对应 CTYPE 属性表里对字符属性的定义。

2. 2 裁减访问目标存储器的函数

有一类动态内存管理函数，如 `malloc()` 等，其本身是独立于目标系统而运行的；但是它所使用的存储器空间需要根据目标来确定。所以 `malloc()` 函数本身并不需要裁剪或移植，但那些设置动态内存区（地址和空间）的函数则是跟目标系统的存储器分布直接相关的，需要进行移植。例如堆栈的初始化函数 `__user_initial_stackheap()`，是用来设置堆（heap）和栈（stack）地址的函数，显然针对每一个具体的目标平台，该函数都需要根据具体的目标存储器资源进行正确移植。

下面是对示例函数 `__user_initial_stackheap()` 进行移植的一个例子：

```
__value_in_regs struct __initial_stackheap __user_initial_stackheap(  
    unsigned R0, unsigned SP, unsigned R2, unsigned SL)  
{  
    struct __initial_stackheap config;  
  
    config.heap_base = (unsigned int) 0x11110000;  
    // config.stack_base = SP;                // optional  
  
    return config;  
}
```

请注意上面的函数体并不完全遵循标准 C 的关键字和语法规则，使用了 ARM 公司编译器(ADS 或 RealView Compilation tool) 里的 C 语言扩展特性。关于编译器特定的 C 语言扩展，请参考相关的编译器说明，这里简单介绍函数 `__user_initial_stackheap()` 的功能，它主要是返回堆和栈的基地址。上面的程序中只对堆(heap) 的基地址进行了设置（设成了 `0x11110000`），也就是说用户把 `0x11110000` 开始的存储器地址用作了动态内存分配区（heap 区）。具体地址的确定是要由用户根据自己的目标系统和应用情况来确定的，至少要满足以下条件：

- `0x11110000` 开始的地址空间有效且可写（是 RAM）
- 该存储器空间不与其它功能区冲突（比如代码区、数据区、stack 区等）

因为 `__user_initial_stackheap()` 函数的全部执行效果就是返回一些数值，所以只要符合接口的调用标准，直接用汇编来实现看起来更加直观一些：

```
EXPORT __user_initial_stackheap

__user_initial_stackheap
    LDR    r0, 0x11110000
    MOV    pc,lr
```

如果不对这个函数进行移植，编译过程中将使用缺省的设置，这个设置适用于 ARM 公司的 Integrator 系列平台。

（注意：ARM 的编译/连接工具链也提供了绕过库函数来设置运行时存储器模型的方法，请参阅 ARM 公司其他的相关文档。）

2.3 裁剪使用 semihosting（半主机）机制实现的函数

库函数里有一大部分函数是涉及到输入/输出流设备的，比如文件操作函数需要访问磁盘 I/O，打印函数需要访问字符输出设备等。在嵌入式调试环境下，所有的标准 C 库函数都是有效且有其缺省行为的，很多目标系统硬件不能支持的操作，都通过调试工具来完成了。比如 `printf()` 函数，缺省的输出设备是调试器里面的信息输出窗口。

但是一个真实的系统是需要脱离调试工具而独立运行的，所以在程序的移植过程当中，需先对这些库函数的运行机制作一了解。

下图说明了在 ADS 下面这类 C 库函数的结构。

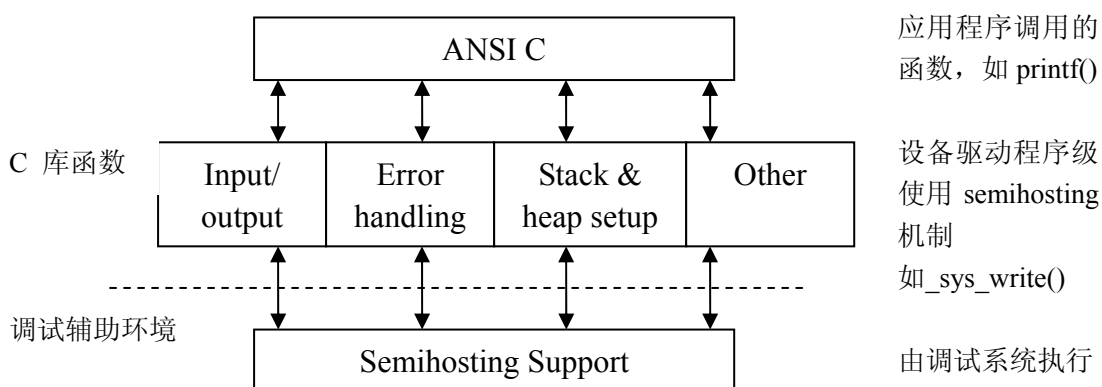


图-3: C 库函数实现过程中的层次调用

如图中例子所示，函数 `printf()` 最终是调用了底层的输入/输出函数 `_sys_write()` 来实现输出操作的，而 `_sys_write()` 使用了调试工具的内部机制来把信息输出到调试器。

显然这样的函数调用过程在一个真实的嵌入式系统里是无法实现的，因为独立运行的嵌入式系统将不会有调试器的参与。如果在最终系统中仍然要保留 `printf()` 函数，而且在系统硬件中具备正确的输出设备（如 LCD 等），则在移植过程中，需要把 `printf()` 调用的输出设备进行重新定向。

考察 `printf()` 函数的完整调用过程：

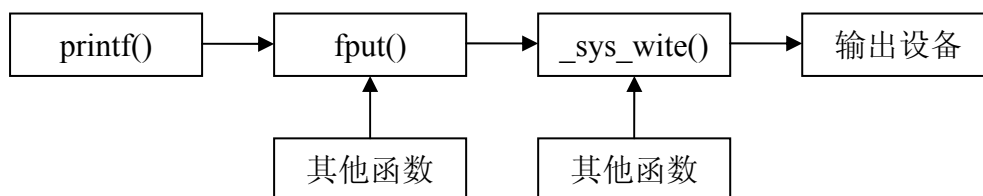


图-4: `printf()` 的调用过程

单纯考虑 `printf()` 的输出重新定向，可以有三种途径实现：

- ◇ 改写 `printf()` 本身
- ◇ 改写 `fputc()`
- ◇ 改写 `_sys_write()`

需要注意的是，越底层的函数，被其他上层函数调用的可能性越大，改变了一个底层函数的实现，则所有调用该函数的上层函数的行为都被改变了。

以 `fputc()` 的重新实现为例，下面是改变 `fputc()` 输出设备到系统串行通信端口的实例：

```
int fputc(int ch, FILE *f)
```

```

{ /* e.g. write a character to an UART */
  char tempch = ch;
  sendchar(&tempch);    // UART driver
  return ch;
}
    
```

代码中的函数 `sendchar()` 假定是系统的串口设备驱动函数。只要新建函数 `fput()` 的接口符合标准，经过编译连接后，该函数实现就覆盖了原来缺省的函数体，所有对该函数的调用，其行为都被新实现的函数所重新定向了。

具体哪些库函数是跟目标相关的，这些函数之间的相互调用关系等，请参考具体的编译器说明。

3. Semihosting (半主机) 机制

上面提到许多库函数在调试环境下的实现都调用了一种叫 `semihosting` 的机制。`Semihosting` 具体来讲是指一种让代码在 ARM 目标上运行，但使用运行了 ARM 调试器的主机上 I/O 设备的方法；也就是让 ARM 目标将输入/输出请求从应用程序代码传递到运行调试器的主机的一种机制。通常这些输入/输出设备包括键盘、屏幕和磁盘 I/O。

半主机由一组已定义的 SWI 操作来实现。库函数调用相应的 SWI（软件中断），然后调试代理程序处理 SWI 异常，并提供所需的与主机之间的通讯。

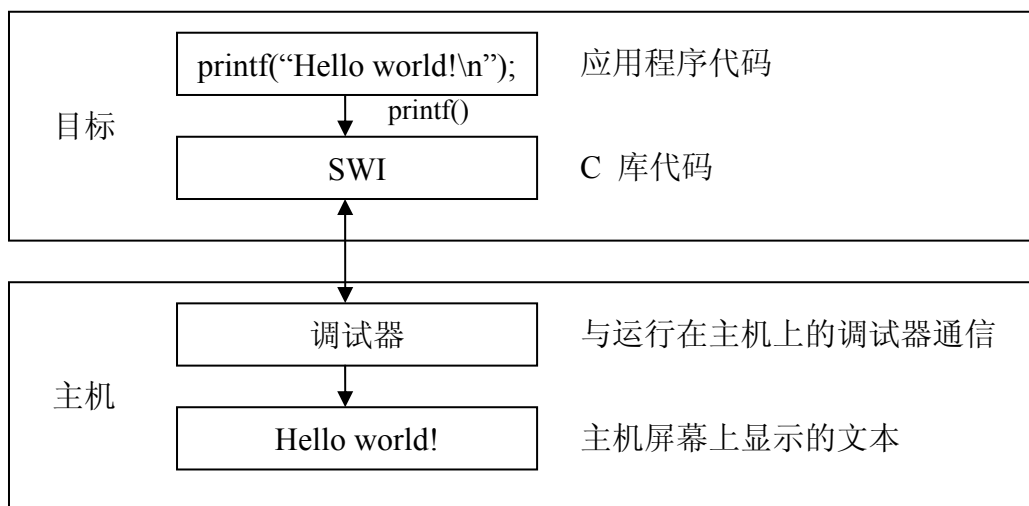


图-5: Semihosting 的实现过程

多数情况下，半主机 SWI 是由库函数内的代码调用的。但是应用程序也可以直接调用半主机 SWI。半主机 SWI 的接口函数是通用的。当半主机操作在硬件仿真器、指令集仿真器、`RealMonitor` 或 `Angel` 下执行时，不需要进行移植处理。

使用单个 SWI 编号请求半主机操作。其它的 SWI 编号可供应用程序或操

作系统使用。用于半主机的 SWI 号是：

在 ARM 状态下：0x123456

在 Thumb 状态下：0xAB

SWI 编号向调试代理程序指示该 SWI 请求是半主机请求。要辨别具体的操作类型，用寄存器 r0 作为参数传递。r0 传递的可用半主机操作编号分配如下：

- 0x00-0x31：这些编号由 ARM 公司使用，分别对应 32 个具体的执行函数。
- 0x32-0xFF：这些编号由 ARM 公司保留，以备将来用作函数扩展。
- 0x100-0x1FF：这些编号保留给用户应用程序。但是，如果编写自己的 SWI 操作，建议直接使用 SWI 指令和 SWI 编号，而不要使用半主机 SWI 编号加这些操作类型编号的方法。
- 0x200-0xFFFFFFFF：这些编号未定义。当前未使用并且不推荐使用这些编号。

半主机 SWI 使用的软件中断编号也可以由用户自定义，但若是改变了缺省的软中断编号，需要：

- 更改系统中所有代码（包括库代码）的半主机 SWI 调用
- 重新配置调试器对半主机请求的捕捉与相应

这样才能使用新的 SWI 编号。

有关半主机 SWI 处理函数实现的更详细信息，请参考 ARM 编译器的相关文档。

4. 应用环境的初始化和根据目标系统资源进行的移植

在下一期中介绍应用环境和目标系统的初始化。

基于 ARM 的嵌入式系统程序开发要点（二）

—— 系统的初始化过程

基于 ARM 的芯片多数为复杂的片上系统集成（SoC），这种复杂的系统里多数的硬件模块都是可配置的，需要由软件来设置其需要的工作状态。因此在用户的应用程序启动之前，需要有专门的一段启动代码来完成对系统的初始化。由于这类代码直接面对处理器内核和硬件控制器进行编程，一般都使用汇编语言。系统启动程序所执行的操作跟具体的目标系统和开发系统相关，一般通用的内容包括：

- 中断向量表
- 初始化存储器系统
- 初始化堆栈
- 初始化有特殊要求的端口、设备
- 初始化应用程序执行环境
- 改变处理器模式
- 呼叫主应用程序

1. 中断向量表

ARM 要求中断向量表必须放置在从 0 地址开始，连续 8×4 字节的空间内（ARM720T 和 ARM9/10 及以后的 ARM 处理器也支持从 0xFFFF0000 开始的高地址向量表，在本文的其他地方对此不再另加说明）。各个中断矢量在向量表中的位置分配如下图：

| | | |
|------|--------------------|---------|
| | ... | |
| 0x1C | FIQ | 外部快速中断 |
| 0x18 | IRQ | 普通外部中断 |
| 0x14 | (Reserved) | 保留 |
| 0x10 | Data Abort | 数据异常 |
| 0x0C | Prefetch Abort | 指令预取异常 |
| 0x08 | Software Interrupt | 软件中断 |
| 0x04 | Undef | 未定义指令中断 |
| 0x00 | Reset | 复位中断 |

图 1：中断向量表

每当一个中断发生以后，ARM 处理器便强制把 PC 指针置为向量表中对应中

断类型的地址值。因为每个中断只占据向量表中 1 个字的存储器空间，只能放置 1 条 ARM 指令，所以通常在向量表中放的是跳转指令，使程序能从向量表里跳转到存储器里的其他地方，再执行中断处理。

中断向量表的程序实现通常如下所示：

```
AREA Boot, CODE, READONLY
ENTRY
B Reset_Handler          ; Reset_Handler is a label
B Undef_Handler
B SWI_Handler
B PreAbort_Handler
B DataAbort_Handler
B .                      ; for reserved interrupt, stop here
B IRQ_Handler
B FIQ_Handler
```

其中的关键字 ENTRY 是指定编译器保留这段代码，因为编译器可能会认为这是一段冗余代码而加以优化。连接的时候要确保这段代码被链接在 0 地址处，并且作为整个程序的入口点（关键字 ENTRY 并非总是用来设置程序入口点，所以通常需要在连接选项里显式地指定程序入口点）。

2. 初始化存储器系统

初始化存储器系统的编程对象是系统的存储器控制器。存储器控制器并不是 ARM 内核的一部分，不同的系统其设计不尽相同，所以应该针对具体的要求来完成这部分的程序设计。一般来说，下面这两个方面是比较通用的。

2. 1. 存储器类型和时序配置

一个复杂的系统可能存在多种存储器类型的接口，需要根据实际的系统设计对此加以正确配置。对同一种存储器类型来说，也因为访问速度的差异，需要不同的时序设置。

通常 Flash 和 SRAM 同属于静态存储器类型，可以合用同一个存储器端口；而 DRAM 因为动态刷新和地址线复用等特性，通常配有专用的存储器端口。

存储器端口的接口时序优化是非常重要的，影响到整个系统的性能。因为一般系统运行的速度瓶颈都存在于存储器访问，所以存储器访问时序应尽可能地快；但同时又要考虑由此带来的稳定性问题。只有根据具体选定的芯片，进行多次的测试之后，才能确定最佳的时序配置。

2. 2. 存储器地址分布（memory map）

有些系统具有非常灵活的存储器地址分配特性，进行存储器初始化设计的时候一定要根据应用程序的具体要求来完成地址分配。

一种典型的情况是启动 ROM 的地址重映射 (remap)。如前面第 1 节所述，当一个系统上电后程序将自动从 0 地址处开始执行，因此在系统的初始状态，必须保证在 0 地址处存在正确的代码，即要求 0 地址开始处的存储器是非易性的 ROM 或 Flash 等。但是因为 ROM 或 Flash 的访问速度相对较慢，每次中断发生后都要从读取 ROM 或 Flash 上面的向量表开始，影响了中断响应速度。因此有的系统便提供一种灵活的地址重映射方法，可以把 0 地址重新指向到 RAM 中去。在这种地址映射的变化过程当中，程序员需要仔细考虑的是程序的执行流程不能被这种变化所打断。比如下面这种情况：

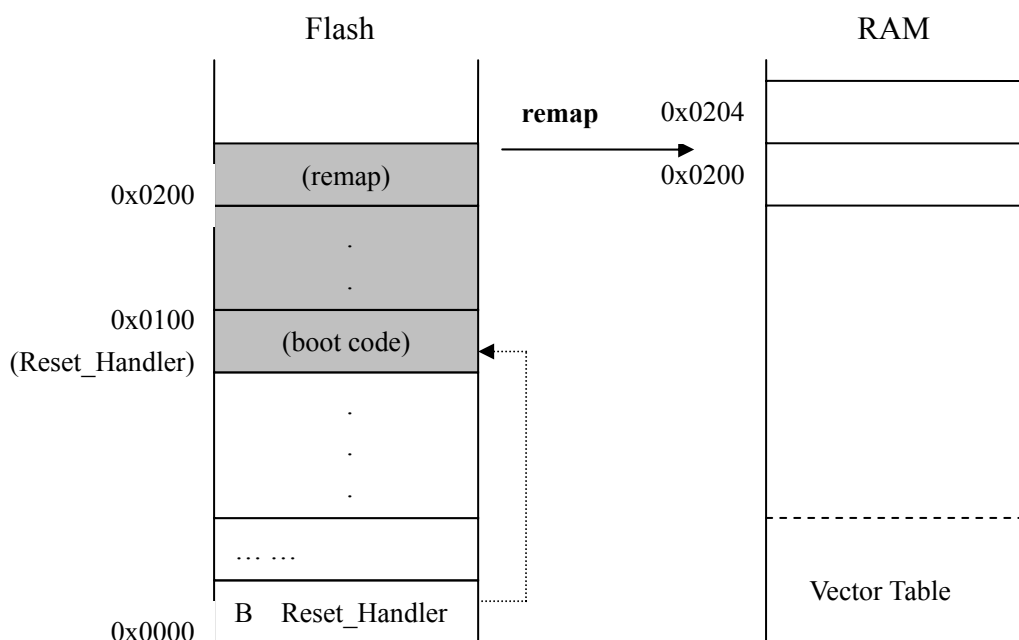


图 2：启动 ROM 的地址重映射对程序执行流程的影响

系统上电后从 Flash 内的 0 地址开始执行，启动代码位于地址 0x100 开始的空间，当执行到地址 0x200 时，完成了一次地址的重映射，把原来 0 开始的地址空间由 Flash 转给了 RAM。接下去执行的指令（这里为了简化起见，忽略流水线指令预取的模型）将来自从 0x204 开始的 RAM 空间。如果预先没有对 RAM 内容进行正确的设置，则里面的数据都是随机的，这样处理器在执行完 0x200 地址处的指令之后，再往下取指执行就会出错。解决的方法就是要使 RAM 在使用之前准备好正确的内容，包括开头的向量表部分。

有的系统不具备存储器地址重映射的功能，所有的空间地址就相对简单一些，不需要考虑这方面的问题。

3. 初始化堆栈

因为 ARM 处理器有 7 种执行状态，每一种状态的堆栈指针寄存器 (SP) 都

是独立的（System 和 User 模式使用相同的 SP 寄存器）。因此对程序中需要用到的每一种模式都要给 SP 寄存器定义一个堆栈地址。方法是改变状态寄存器 CPSR 内的状态位，使处理器切换到不同的状态，然后给 SP 赋值。注意不要切换到 User 模式进行 User 模式的堆栈设置，因为进入 User 模式后就不能再操作 CPSR 回到别的模式了。可能会对接下去的程序执行造成影响。

一般堆栈的大小要根据需要而定，但是要尽可能给堆栈分配快速和高带宽的存储器。堆栈性能的提高对系统整体性能的影响是非常明显的。

这是一段堆栈初始化的代码示例，其中只定义了三种模式的 SP 指针：

```

MRS      R0, CPSR           ; CPSR -> R0
BIC      R0, R0, #MODEMASK ; 安全起见，屏蔽模式位以外的其它位
ORR      R1, R0, #IRQMODE  ; 把设置模式位设置成需要的模式
MSR      CPSR_cxsf, R1     ; 转到 IRQ 模式
LDR      SP, =UndefStack   ; 设置 SP_irq

ORR      R1, R0, #FIQMODE
MSR      CPSR_cxsf, R1     ; FIQMode
LDR      SP, =FIQStack

ORR      R1, R0, #SVCMODE
MSR      CPSR_cxsf, R1     ; SVCMode
LDR      SP, =SVCStack
    
```

注意上面的程序中使用到的 3 个 SP 寄存器是不同的物理寄存器：SP_irq，SP_fiq 和 SP_svc。引用的几个标号假设已经正确定义。

4. 初始化有特殊要求的端口、设备

这要由具体的系统和用户需求而定。一般的外设初始化可以在系统初始化之后进行。

比较典型的应用是驱动一些简单的输出设备，如 LED 等，来指示系统启动的进程和状态。

5. 初始化应用程序执行环境

一个简单的可执行程序的映像结构通常如下：

| | |
|--------------------------------|--------------|
| ZI (Zero initialized R/W Data) | 只定义了变量名的全局变量 |
| RW (R/W Data) | 定义时带初始值的全局变量 |
| RO (Code + RO Data) | 编译结果 |

图 3：程序映像的结构

映像一开始总是存储在 ROM/Flash 里面的，其 RO 部分既可以在 ROM/Flash 里面执行，也可以转移到速度更快的 RAM 中去；而 RW 和 ZI 这两部分必须是需要转移到可写的 RAM 里去的。所谓应用程序执行环境的初始化，就是完成必要的从 ROM 到 RAM 的数据传输和内容清零。

不同的工具链会提供一些不同的机制和方法帮助用户完成这一步操作，主要是跟链接器（Linker）相关。下面是在 ARM 开发工具环境（ADS 或 RVCT）下，一种常用存储器模型的直接实现：

```

LDR    r0, =|Image$$RO$$Limit| ; Get pointer to ROM data
LDR    r1, =|Image$$RW$$Base| ; RAM copy address
LDR    r3, =|Image$$ZI$$Base| ; Zero init base => top of initialised data

CMP    r0, r1 ; Check that they are different
BEQ    %F1

0
CMP    r1, r3 ; Copy init data
LDRCC  r2, [r0], #4 ; ([r0] -> r2) and (r0+4)
STRCC  r2, [r1], #4 ; (r2 -> [r1]) and (r1+4)
BCC    %B0

1
LDR    r1, =|Image$$ZI$$Limit| ; Top of zero init segment
MOV    r2, #0

2
CMP    r3, r1
STRCC  r2, [r3], #4 ; (0 -> [r3]) and (r3+4)
BCC    %B2
    
```

程序实现了 RW 数据的拷贝和 ZI 区域的清零功能。其中引用到的 4 个符号是由连接器（linker）定义输出的：

|Image\$\$RO\$\$Limit|：表示 RO 区末地址后面的地址，即 RW 数据源的起始地址。
 |Image\$\$RW\$\$Base|：RW 区在 RAM 里的执行区起始地址，也就是编译选项 RW_Base 指定的地址；程序里是 RW 数据拷贝的目标地址。

|Image\$\$ZI\$\$Base|: ZI 区在 RAM 里面的起始地址。

|Image\$\$ZI\$\$Limit|: ZI 区在 RAM 里面的结束地址后面的一个地址。

程序先把 ROM 里 |Image\$\$RO\$\$Limit| 开始的 RW 初始数据拷贝到 RAM 里 |Image\$\$RW\$\$Base| 开始的地址，当 RAM 这边的目标地址到达 |Image\$\$ZI\$\$Base| 后就表示 RW 区的结束和 ZI 区的开始，接下去就对这片 ZI 区进行清零操作，直到遇到结束地址 |Image\$\$ZI\$\$Limit|。

6. 改变处理器模式

ARM 处理器（V4 架构以后的版本）一共有 7 种执行模式：

| | |
|-------------|----------|
| User: | 用户模式 |
| FIQ: | 快速中断响应模式 |
| IRQ: | 一般中断响应模式 |
| Supervisor: | 超级模式 |
| Abort: | 出错处理模式 |
| Undef: | 未定义模式 |
| System: | 系统模式 |

除用户模式以外，其他 6 种模式都是特权模式。因为在初始化过程中许多操作需要在特权模式下才能进行（比如 CPSR 的修改），所以要特别注意不能过早地进入用户模式。一般地，在初始化过程中会经历以下一些模式变化：

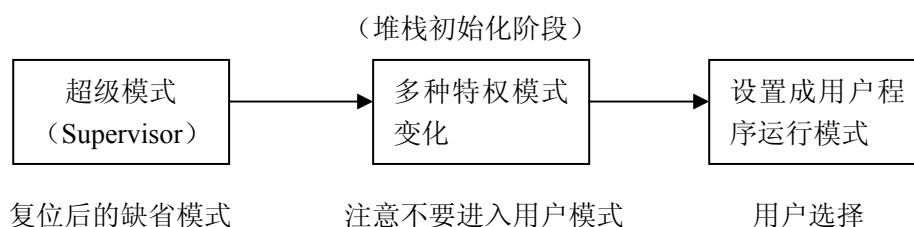


图 4：处理器模式变换过程

在最后阶段才把模式转换到最终应用程序运行所需的模式，一般是用户模式。

内核级的中断使能（CPSR 的 I、F 位状态）也可以考虑在这一步进行。如果系统中另外存在一个专门的中断控制器（多数情况下是这样的），这么做总是安全的，否则就需要考虑过早地打开中断可能带来的问题，比如在系统初始化完成之前就触发了有效中断，导致系统的死机。

7. 呼叫主应用程序

当所有的系统初始化工作完成之后，就需要把程序流程转入主应用程序。最简单的一种情况是：

```

IMPORT    main    ; get the label main if main() is defined in other files
B        man     ; jump to main()
    
```

直接从启动代码跳入应用程序主函数入口，主函数名字可由用户自己定义。

在 ARM ADS 环境中，还另外提供了一套系统级的呼叫机制。

```

IMPORT    __main
B        __main
    
```

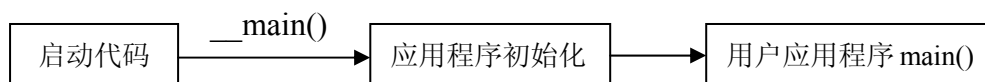


图 5：在应用程序主函数之前插入 __main

__main() 是编译系统提供的一个函数，负责完成库函数的初始化和第 5 节中所描述的功能，最后自动跳向 main() 函数。这种情况下用户程序的主函数名字必须得是 main。

用户可以根据需要选择是否使用 __main()。如果想让系统自动完成系统调用（如库函数）的初始化过程，可以直接使用 __main()；如果所有的初始化步骤都是由用户自己显式地完成，则可以跳过 __main()。

当然，使用 __main() 的时候，可能会涉及到一些库函数的移植和重定向问题。在 __main() 里面的程序执行流程如下图所示：

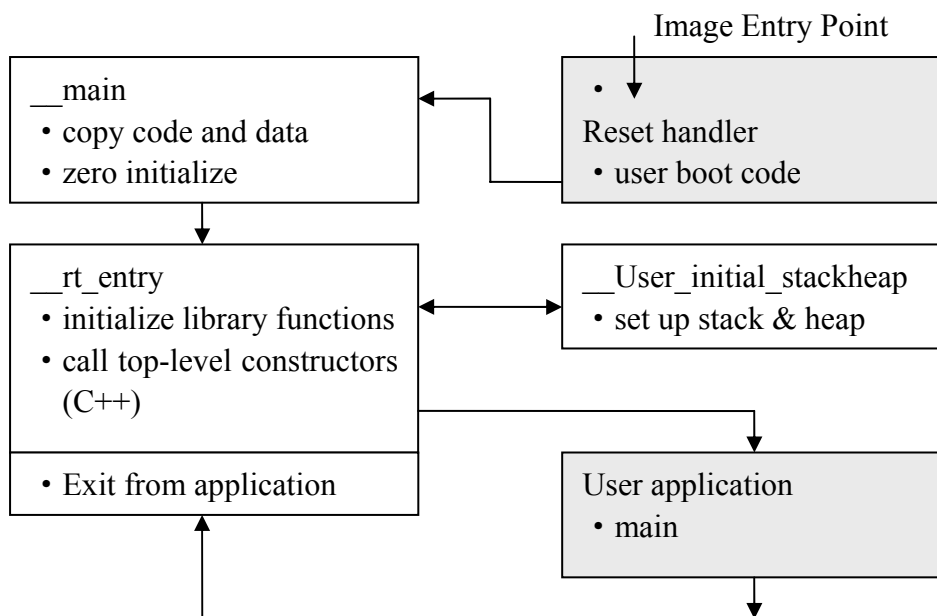


图 6：有系统调用参与的程序执行流程

关于在 __main() 里面调用到的库函数说明，可以参阅相关的编译器文档，库函数移植和重定向的方法，可以参考上一期文章里面的相关章节。

基于 ARM 的嵌入式系统程序开发要点（三）

—— 如何满足嵌入式系统的灵活需求？

因为嵌入式应用领域的多样性，每一个系统都具有各自的特点。在进行系统程序设计的时候，一定要进行具体分析，充分利用这些特点，扬长避短。

结合 ARM 架构本身的一些特点，在这里讨论几个常见的要点。

1. ARM 还是 Thumb？

在讨论 ARM 还是 Thumb 之前，先说明 ARM 内核型号和 ARM 结构体系之间的区别和联系。

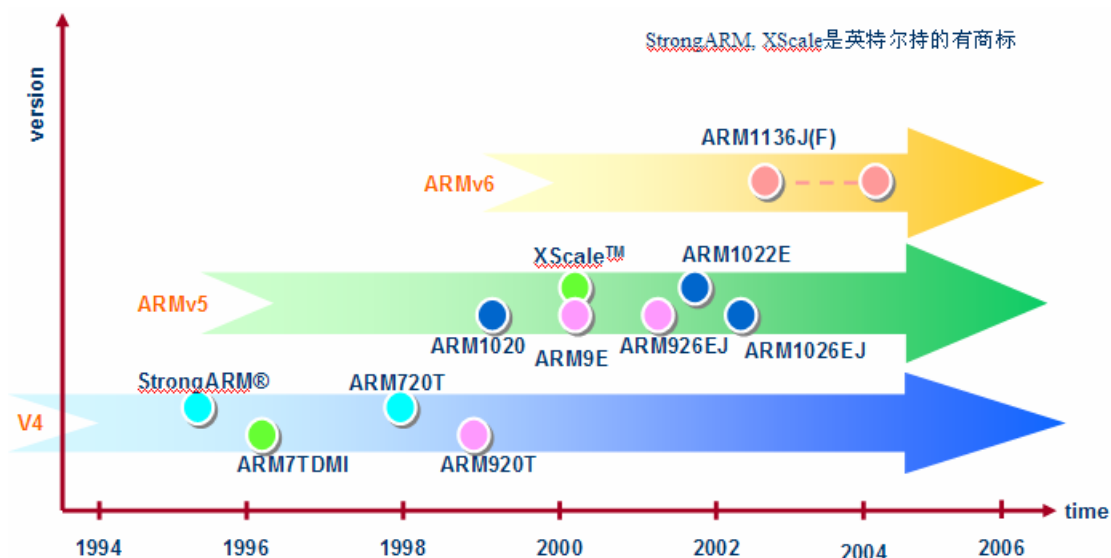


图-1 ARM 结构体系和处理器家族的演变发展

如图-1 所示，ARM 的结构体系主要从版本 4 开始，发展到了现在的版本 6，结构体系的变化，对程序员而言最直接的影响就是指令集的变化。结构体系的演变意味着指令集的不断扩展，值得庆幸的是 ARM 结构体系的发展一直保持了向上兼容，不会造成老版本程序在新结构体系上的不兼容。

在图中的横坐标上，显示了每一个体系结构上都含有众多的处理器型号，这是在同一体系结构下根据硬件配置和存储器系统的不同而作的进一步细分。需要注意的是通常我们用来区分 ARM 处理器家族的 ARM7、ARM9 或 ARM10，可能跨越不同的体系结构。

在 ARM 的体系结构版本 4 与 5 中，还可以再细分出几个小的扩展版本：V4T、V5TE 和 V5TEJ，其区别如图-2 中所示，这些后缀名也反映在各自拥有的处理器

型号上面，可以进行直观的分辨。V6 结构体系因为包含了以前版本的所有特性，所以不需要再进行分类。



图-2 结构体系特征

上面介绍了整个 ARM 处理器家族的分布，主要是说明在一个特定的平台上编写程序的时候，一定要先弄清楚目标的特性和一些细微的差别，特别是需要具体优化特征的时候。

从 ARM 体系结构 V4T 以后，最大的变化是增加了一套 16 位的指令集——Thumb。到底在一个具体应用中要否采用 Thumb 呢？首先我们来分析一下 ARM 和 Thumb 各自的特点和优势。先看下面一张性能分析图：

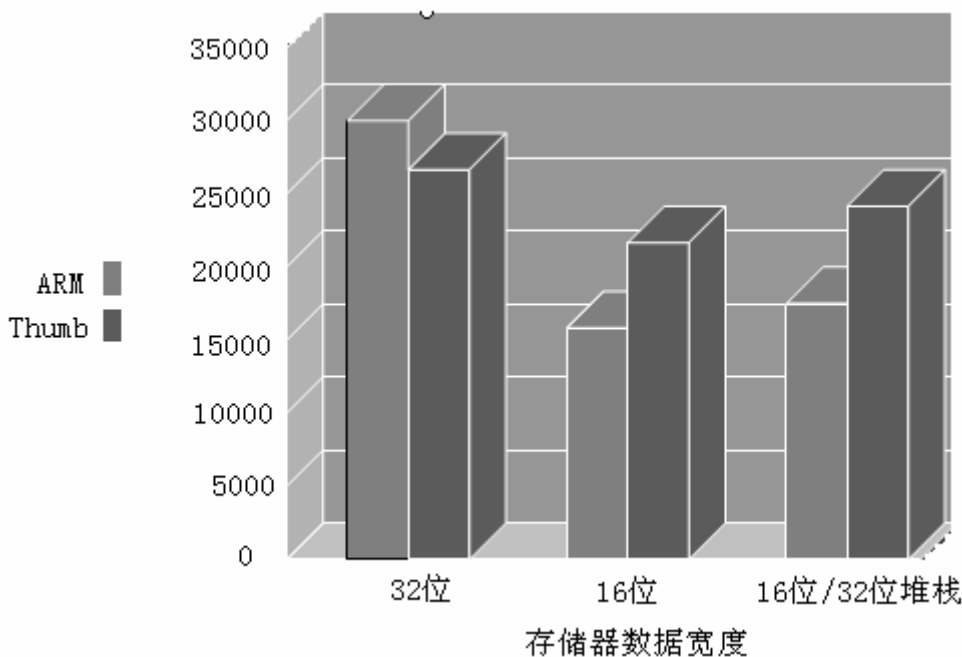


图-3 ARM 和 Thumb 指令集的比较

图中的纵坐标是测试向量 Dhrystone 在 20MHz 频率下运行 1 秒钟的结果，其值越大表明性能越好；横坐标是系统存储器系统的数据总线宽度。结果表明：

(a) 当系统具有 32 位的数据总线宽度时，ARM 比 Thumb 有更好的性能表现。

(b) 当系统的数据总线宽度小于 32 位时，Thumb 比 ARM 的性能更好。

由此可见，并不是 32 位的 ARM 指令集性能一定强于 16 位的 Thumb 指令集，要具体情况具体分析。考察个中的原因，其实不难发现，因为当在一个 16 位存储器系统里面取 1 条 32 位指令的时候，需要耗费 2 个存储器访问周期；比之 32 位的系统，其速度正好大概下降一半左右。而 16 位指令在 32 位存储器系统或 16 位存储器系统里的表现基本相同。正是存储器造成的系统瓶颈导致了这个有趣的差别。

除了在窄带宽系统里面的性能优势外，Thumb 指令的另外一个好处是代码尺寸。同样一段 C 代码，用 Thumb 指令编译的结果，其长度大约只占 ARM 编译结果的 65% 左右，可以明显地节省存储器空间。在大多数情况下，紧凑的代码和窄带宽的存储器系统，还会带来功耗上的优势。

当然，如果在 32 位的系统上面，并且对系统性能要求很高的情况下，ARM 是一个更好的选择。毕竟在这种情况下，只有 32 位的指令集才能完全发挥 32 位处理器的优势来。

因此，选择 ARM 还是 Thumb，需要从存储器开销和性能要求两方面加以权衡考虑。

2. 堆栈的分配

在图-3 中，横坐标上还有一种情况，就是 16 位的存储器宽度，但是堆栈空间是 32 位的。这种情况下无论 ARM 还是 Thumb，其性能表现都比单纯的 16 位存储器系统情况下要好。这是因为 ARM 和 Thumb 其指令集虽然分 32 位和 16 位，但是堆栈全部是采用 32 位的。因此在 16 位堆栈和 32 位堆栈的不同环境下，其性能当然都会相差很多。这种差别还跟具体的应用程序密切相关，如果一个程序堆栈的使用频率相当高，则这种性能差异很大；反之则要小一些。

在基于 ARM 的系统中，堆栈不仅仅被用来进行诸如函数调用、中断响应等时候的现场保护，还是程序局部变量和函数参数传递（如果大于 4 个）的存储空间。所以出于系统整体性能考虑，要给堆栈分配相对访问速度最快、数据宽度最大的存储器空间。

一个嵌入式系统通常存在多种多样的存储器类型。设计的时候一定要先清楚每一种存储器的访问速度，地址分配和数据线宽度。然后根据不同程序和目标模块对存储器的不同要求进行合理分配，以期达到最佳配置状态。

3. ROM 还是 RAM 在 0 地址处？

显然当系统刚启动的时候，0 地址处肯定是某种类型的 ROM，里面存储了系统的启动代码。但是很多灵活的系统设计中，0 地址处的存储器类型是可映射的。也就是说，可以通过软件的方法，把别的存储器（主要是快速的 RAM）分配以

0 起始的地址。

这种做法的最主要目的之一是提高系统对中断的反应速度。因为每一个中断发生的时候，ARM 都需要从 0 地址处的中断向量表开始其中断响应流程，显然把中断向量表放在 RAM 里，比放在 ROM 里有更快的访问速度。因此，如果系统提供了这一类的地址重映射功能，软件设计者一定要加以利用。

下面是一个典型的经过 0 地址重映射之后的存储空间分布图，注意尽可能把速度要求最高的部分放置在系统里面访问速度最快、带宽最宽的 RAM 里面。

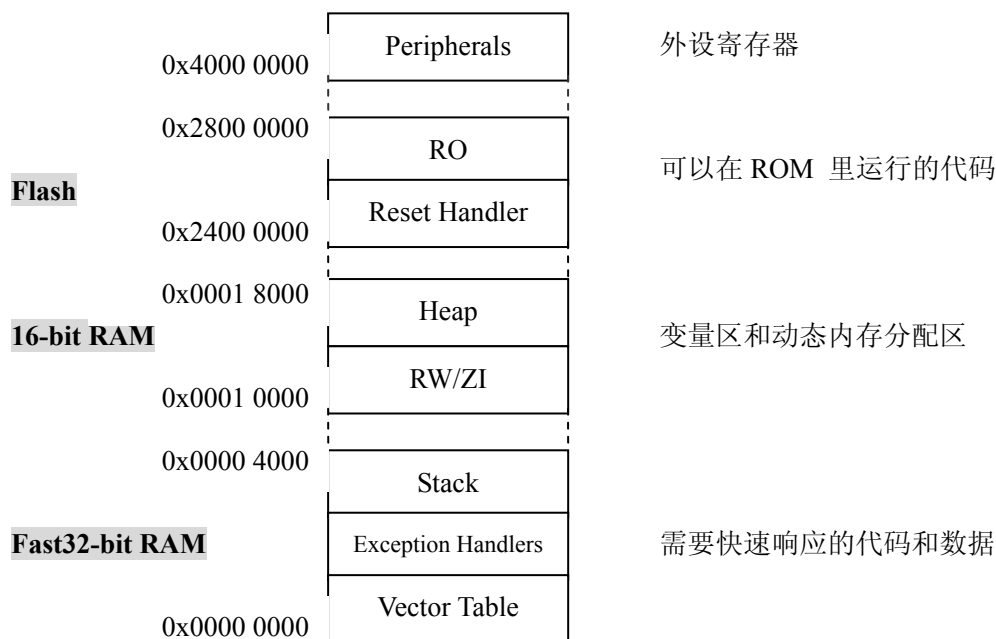


图-4 系统存储器分布的实例

4. 存储器地址重映射 (memory remap)

存储器地址重映射是当前很多先进控制器所具有的功能。在上一节中已经提到了 0 地址处存储器重映射的例子，简而言之，地址重映射就是可以通过软件配置来改变一块存储器物理地址的一种机制或方法。

当一段程序对运行自己的存储器进行重映射的时候，需要特别注意保证程序执行流程在重映射前后的承接关系。下面是一种典型的存储器地址重映射情况：

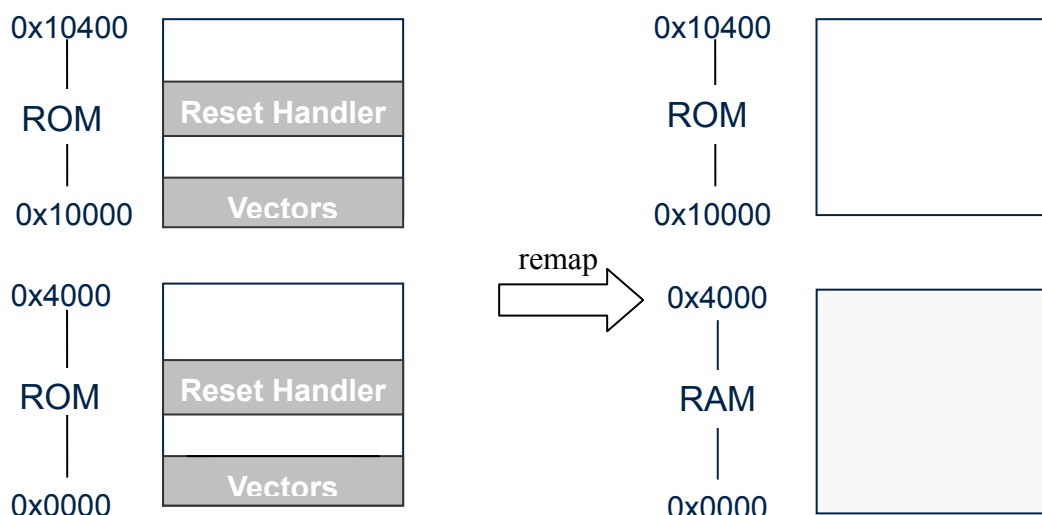


图-5 存储器重映射举例 1

系统上电后的缺省状态是 0 地址上放有 ROM，这块 ROM 有两个地址：从 0 起始和从 0x10000 起始，里面存储了初始化代码。当进行地址 remap 以后，从 0 起始的地址被定向到了 RAM 上，ROM 则只保留有唯一的从 0x10000 起始的地址了。

如果存储在 ROM 里的 Reset_Handler 一直在 0 - 0x4000 的地址上运行，则当执行完 remap 以后，下面的指令将从 RAM 里预取，必然会导致程序执行流程的中断。根据系统特点，可以用下面的办法来解决这个问题：

- (1) 上电后系统从 0 地址开始自动执行，设计跳转指令在 remap 发生前使 PC 指针指向 0x10000 开始的 ROM 地址中去，因为不同地址指向的是同一块 ROM，所以程序能够顺利执行。
- (2) 这时候 0 - 0x4000 的地址空间空闲，不被程序引用，执行 remap 后把 RAM 引进。因为程序一直在 0x10000 起始的 ROM 空间里运行，remap 对运行流程没有任何影响。
- (3) 通过在 ROM 里运行的程序，对 RAM 进行相应的代码和数据拷贝，完成应用程序运行的初始化。

下面是一段实现上述步骤的例程：

```

ENTRY
; 启动时，从 0 开始，设法跳转到“真”的 ROM 地址（0x10000 开始的空间里）
    LDR    pc, =start
; insert vector table here
    ...
Start    ; Begin of Reset_Handler
; 进行 remap 设置
    
```

```
LDR    r1, =Ctrl_reg    ; 假定控制 remap 的寄存器
LDR    r0, [r1]
ORR    r0, r0, #Remap_bit ; 假定对控制寄存器进行 remap 设置
STR    r0, [r1]
; 接下去可以进行从 ROM 到 RAM 的代码和数据拷贝
```

除此之外，还有另外一种常见的 remap 方式，如下图：

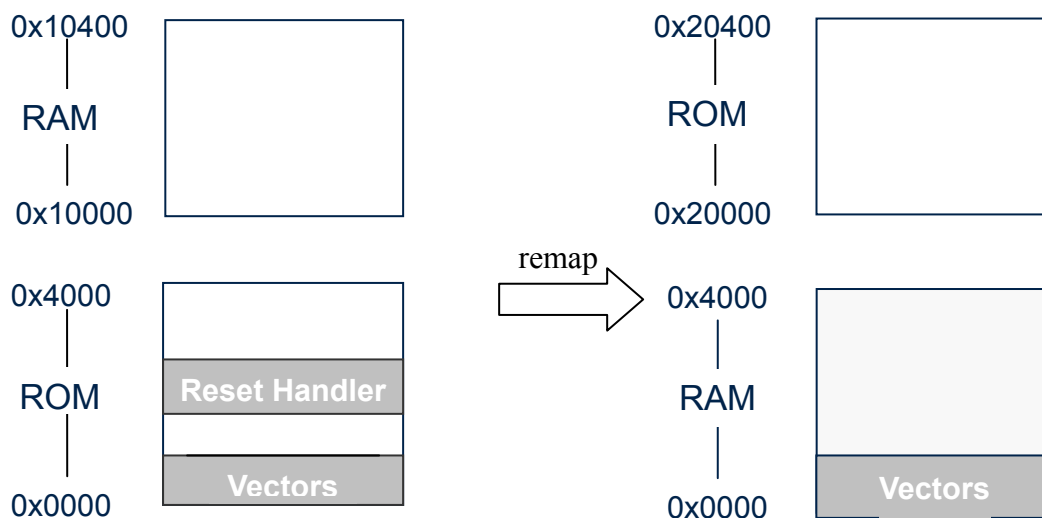


图-6 存储器重映射举例 2

原来 RAM 和 ROM 各有自己的地址，进行重映射以后 RAM 和 ROM 的地址都发生了变化，这种情况下，可以采用以下的方案：

- (1) 上电后，从 0 地址的 ROM 开始往下执行。
- (2) 根据映射前的地址，对 RAM 进行必要的代码和数据拷贝。
- (3) 拷贝完成后，进行 remap 操作。
- (4) 因为 RAM 在 remap 前准备好了内容，使得 PC 指针能继续在 RAM 里取到正确的指令。

不同的系统可能会有多种灵活的 remap 方案，根据上面提到的两个例子，可以总结出最根本的考虑是：要使程序指针在 remap 以后能继续往下得到正确的指令。

5. 根据目标存储器系统分散加载映像 (scatterloading)

Scatterloading 文件是 ARM 的工具链里面的一个特性，作为程序编译过程中给连接器使用的一个参数，用来指定最终生成的目标映像文件运行时的分布状态。如果用户程序映像只是如图 7 所示的最简状态，所有的可执行代码都集合放置在一起，那么可以不使用 Scatterloading 文件，直接用连接器的命令行选项就

能够完成设置：

RO = 0x00000: 表示映像的第一条指令开始地址；

RW = 0x10000: 表示变量区的起始地址，变量区一定要位于 RAM 区。

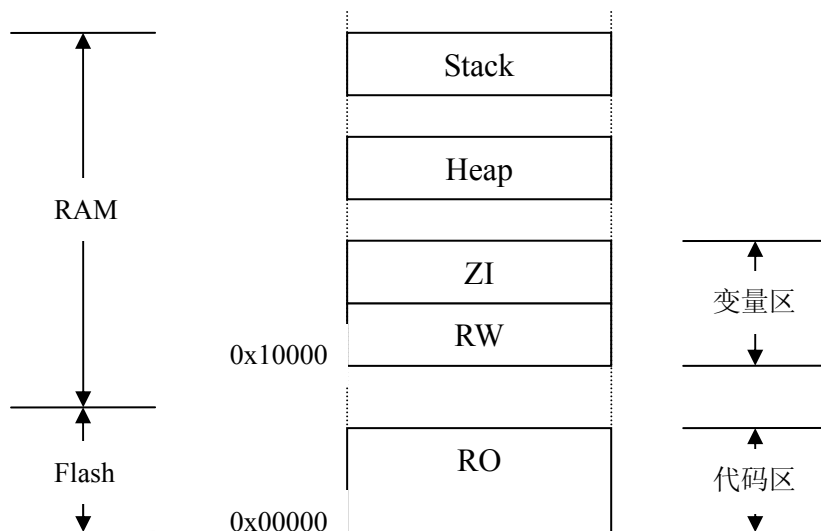


图-7 简单的映像分布举例

但是一个复杂的系统可能会把映像分割成几个部分。如图 8，系统中存在多种类型的存储器，不能的代码部分根据执行性能优化的考虑分布与不同的地方。

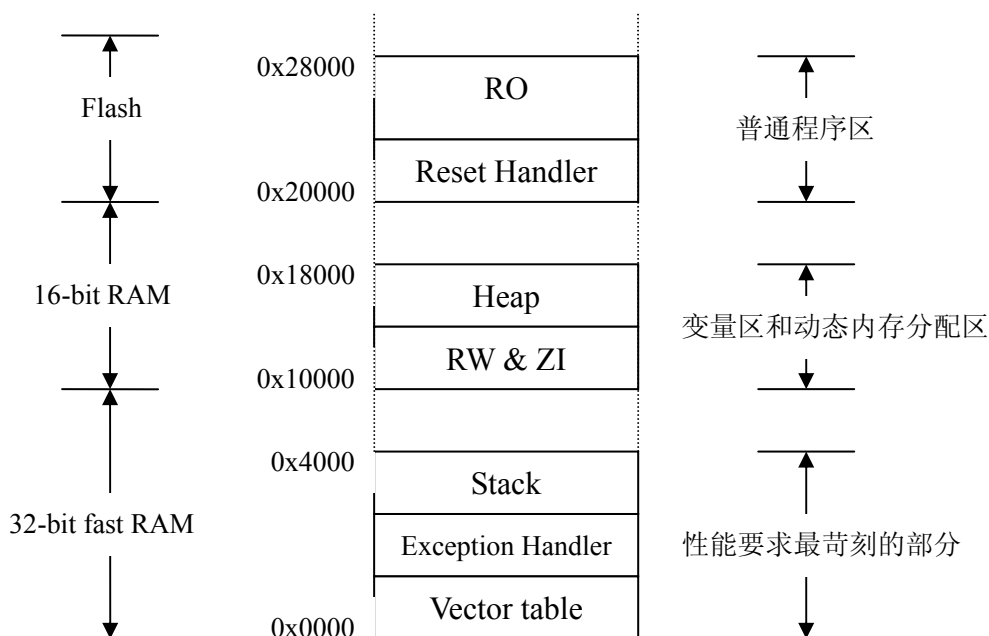


图-8 复杂的映像分布举例

这时候不能通过简单的 RO、RW 参数来完成实现上述配置，就要用到 scatterloading 文件了。在 scatterloading 文件里，可以给编译出来的各个目标模块

指定运行地址，下面的例子是针对图 8 的。

```
FLASH 0x20000 0x8000
{
    FLASH 0x20000 0x8000
    {
        init.o (Init, +First)
        * (+RO)
    }

    32bitRAM 0x0000
    {
        vectors.o (Vect, +First)
        handlers.o (+RO)
    }

    STACK 0x1000 UNINIT
    {
        stackheap.o (stack)
    }
:
:
16bitRAM 0x10000
{
    * (+RW,+ZI)
}

HEAP 0x15000 UNINIT
{
    stackheap.o (heap)
}
}
```

关于 scatterloading 文件的详细语法，请参阅 ARM 公司的相关手册。

基于 ARM 的嵌入式系统程序开发要点（四）

—— 异常处理机制的设计

异常或中断是用户程序中最基本的一种执行流程或形态，这部分对 ARM 架构下异常处理程序的编写作一个全面的介绍。

ARM 一共有 7 种类型的异常，按优先级从高到低排列如下：

Reset
Data Abort
FIQ
IRQ
Prefetch Abort
SWI
Undefined instruction

请注意在 ARM 的文档中，使用术语 **exception** 来描述异常。Exception 主要是从处理器被动接受异常的角度出发描述，而 **interrupt** 带有向处理器主动申请的色彩。在本文中，对“异常”和“中断”不作严格区分，都是指请求处理器打断正常的程序执行流程，进入特定程序循环的一种机制。

1. 异常响应流程

如以前介绍异常向量表时所提到过的，每一个异常发生时，总是从异常向量表开始起跳的，最简单的一种情况是：

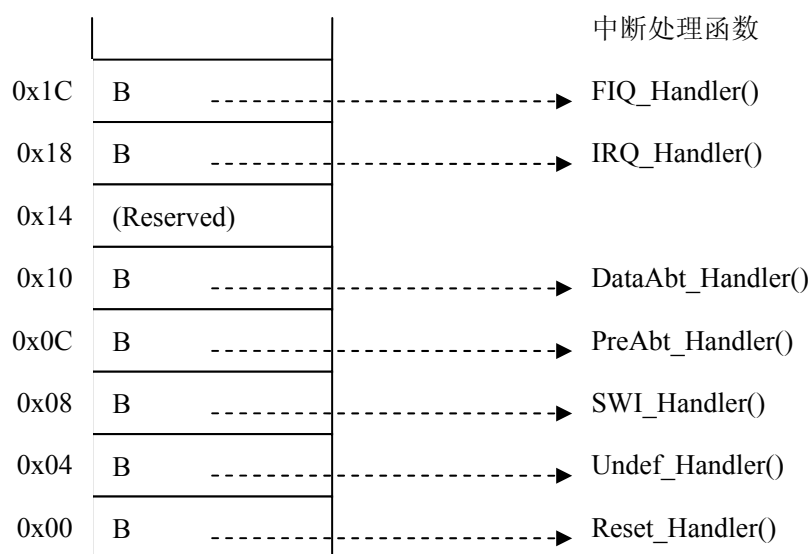


图-1 异常向量表

向量表里面的每一条指令直接跳向对应的异常处理函数。其中 FIQ_Handler() 可以直接从地址 0x1C 处开始，省下一条跳转指令。

但是当执行跳转的时候有 2 个问题需要讨论：跳转范围和异常分支。

1. 1 跳转范围

我们知道 ARM 的跳转指令 (B) 是有范围限制的 ($\pm 32\text{MB}$)，但很多情况下不能保证所有的异常处理函数都定位在向量表的 32MB 范围内，需要大于 32MB 的长跳转，而且因为向量表空间的限制只能由一条指令完成。这可以通过下面二种方法实现。

(a) MOV PC, #immed_value

把目标地址直接赋给 PC 寄存器。

但是这条指令受格式限制并不能处理任意立即数，只有当这个立即数能够表示为一个 8-bit 数值通过循环右移偶数位而得到，才是合法的。例如：

MOV PC, #0x30000000 是合法的，因为 0x30000000 可以通过 0x03 循环右移 4 位而得到。

而 MOV PC, #30003000 就是非法指令。

(b) LDR PC, [PC+offset]

把目标地址先存储在某一个合适的地址空间，然后把这个存储器单元上的 32 位数据传送给 PC 来实现跳转。

这种方法对目标地址值没有要求，可以是任意有效地址。但是存储目标地址的存储器单元必须在当前指令的 $\pm 4\text{KB}$ 空间范围内。

注意在计算指令中引用的 offset 数值的时候，要考虑处理器流水线中指令预取对 PC 值的影响，以图-2 的情况为例：

$$\begin{aligned}\text{offset} &= \text{address location} - \text{vector address} - \text{pipeline effect} \\ &= 0xFFC - 0x4 - 0x8 \\ &= 0xFF0\end{aligned}$$

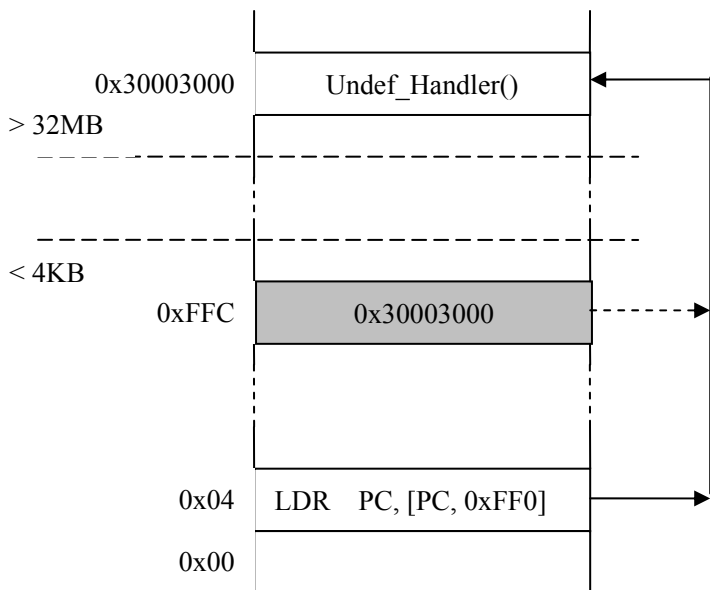


图-2 利 Literal pool 实现跳转

1. 2 异常分支

ARM 内核只有二个外部中断输入信号 nFIQ 和 nIRQ，但对于一个系统来说，中断源可能多达几十个。为此，在系统集成的时候，一般都会会有一个异常控制器来处理异常信号。

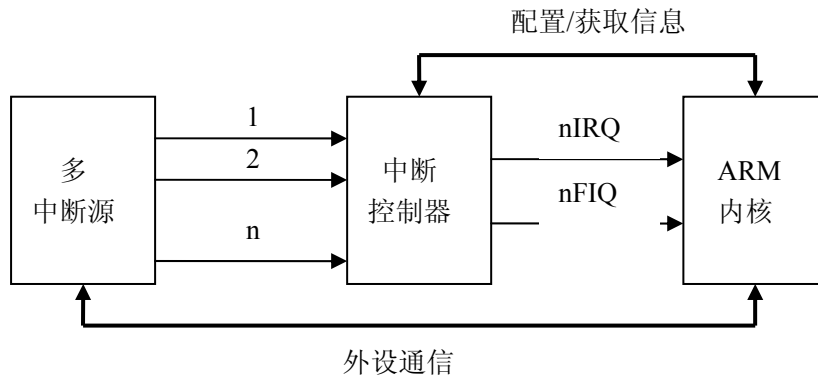


图-3 中断系统

这时候，用户程序可能存在多个 IRQ/FIQ 的中断处理函数，为了从向量表开始的跳转最终能找到正确的处理函数入口，需要设计一套处理机制和方法。

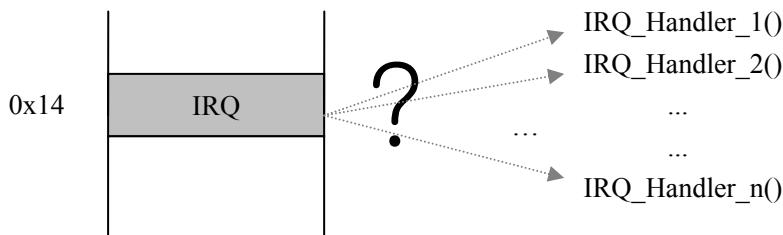


图-4 中断分支

(a) 硬件处理

有的系统在 ARM 的异常向量表之外，又增加了一张由中断控制器控制的特殊向量表。当由外设触发一个中断以后，PC 能够自动跳到这张特殊向量表中去，特殊向量表中的每个向量空间对应一个具体的中断源。

举例来说，下面的系统一共有 20 个外设中断源，特殊向量表被直接放置在普通向量表后面。

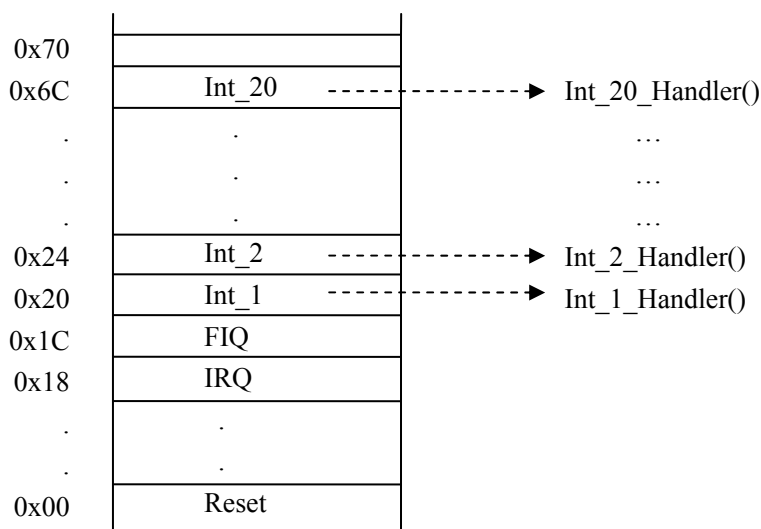


图-5 额外的硬件异常向量表

当某个外部中断触发之后，首先触发 ARM 的内核异常，中断控制器检测到 ARM 的这种状态变化，再通过识别具体的中断源，使 PC 自动跳转到特殊向量表中的对应地址，从而开始一次异常响应。需要检查具体的芯片说明，是否支持这类特性。

(b) 软件处理

多数情况下是用软件来处理异常分支。因为软件可以通过读取中断控制器来获得中断源的详细信息。

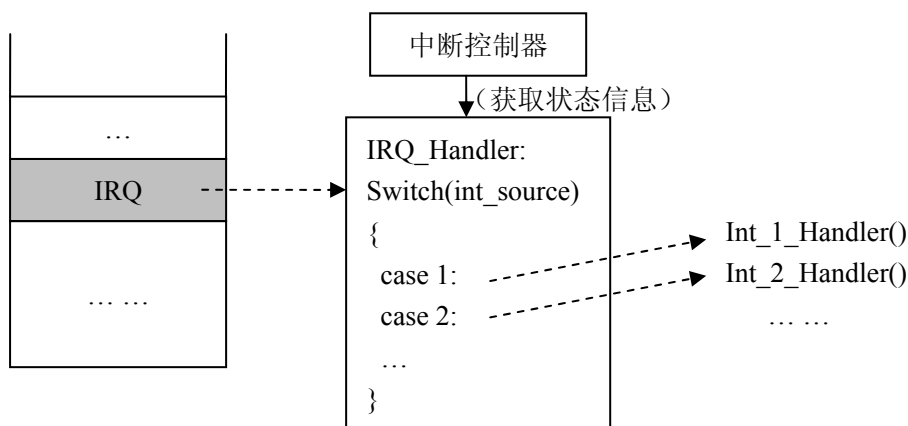


图-6 软件控制中断分支

因为软件设计的灵活性，用户可以设计出比上图更好的流程控制方法来。下面是一个例子：

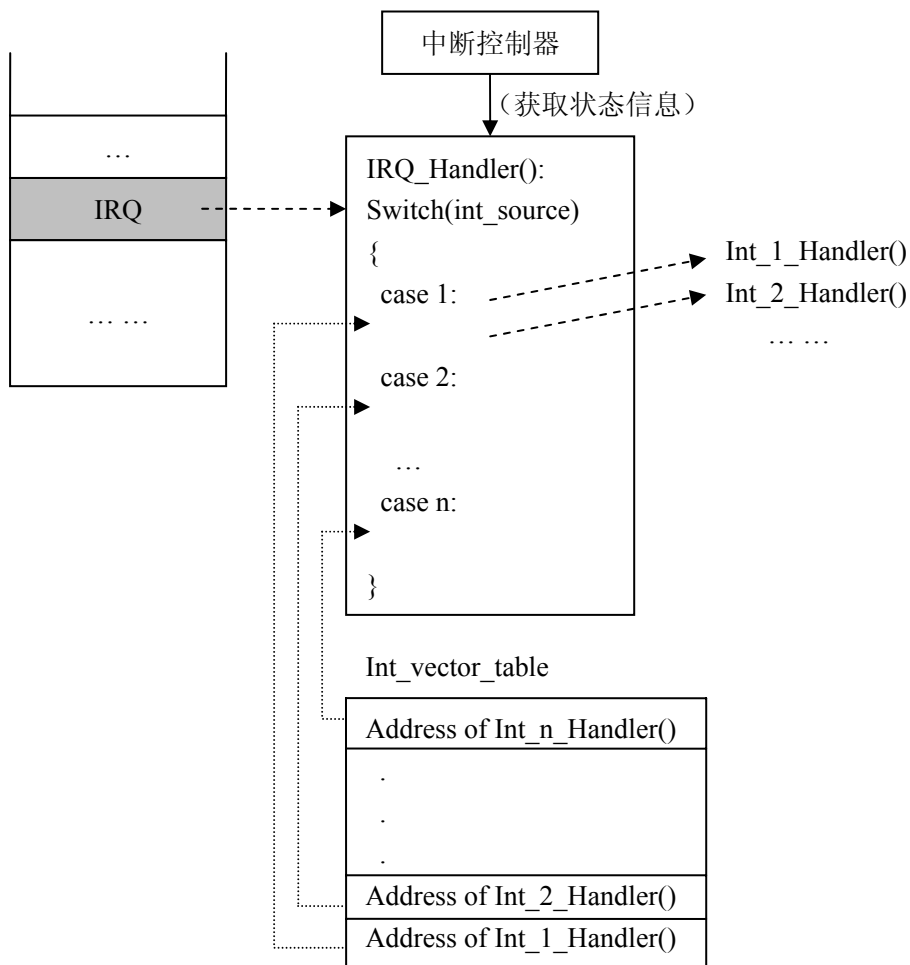


图-7 灵活的软件分支设计

`Int_vector_table` 是用户自己开辟的一块存储器空间，里面按次序存放异常处理函数的地址。`IRQ_Handler()` 从中断控制器获取中断源信息，然后再从 `Int_verctor_table` 中的对应地址单元得到异常处理函数的入口地址，完成一次异常响应的跳转。这种方法的好处是用户程序在运行过程中，能够很方便地动态改变异常服务内容。

2. 异常处理函数的设计

2.1 异常发生时处理器的动作

当任何一个异常发生并得到响应时，ARM 内核自动完成以下动作：

- 拷贝 CPSR 到 `SPSR_<mode>`

- 设置适当的 CPSR 位：
 - 改变处理器状态进入 ARM 状态
 - 改变处理器模式进入相应的异常模式
 - 设置中断禁止位禁止相应中断
- 更新 LR_<mode>
- 设置 PC 到相应的异常向量

注意当响应异常后，不管异常发生在 ARM 还是 Thumb 状态下，处理器都将自动进入 ARM 状态。另一个需要注意的地方是中断使能被自动关闭，也就是说缺省情况下中断是不可重入的。单纯的把中断使能位打开接受重入的中断会带来新的问题，在第 3 部分中对此会有详细介绍。

除这些自动完成的动作之外，如果在汇编级进行手动编程，还需要注意保存必要的通用寄存器。

2.2 进入异常处理循环后软件的任务

进入异常处理程序以后，用户可以完全按照自己的意愿来进行程序设计，包括调用 Thumb 状态的函数，等等。但是对于绝大多数的系统来说，有一个步骤必须处理，就是要把中断控制器中对应的中断状态标识清掉，表明该中断请求已经得到响应。否则等退出中断函数以后，又马上会被再一次触发，从而进入周而复始的死循环。

2.3 异常的返回

当一个异常处理返回时，一共有 3 件事情需要处理：通用寄存器的恢复、状态寄存器的恢复以及 PC 指针的恢复。

通用寄存器的恢复采用一般的堆栈操作指令，而 PC 和 CPSR 的恢复可以通过一条指令来实现，下面是 3 个例子：

```
MOVS pc, lr 或 SUBS pc, lr, #4 或 LDMFD sp!, {pc}^
```

这几条指令都是普通的数据处理指令，特殊之处就是把 PC 寄存器作为了目标寄存器，并且带了特殊的后缀“S”或“^”，在特权模式下，“S”或“^”的作用就是使指令在执行时，同时完成从 SPSR 到 CPSR 的拷贝，达到恢复状态寄存器的目的。

异常返回时另一个非常重要的问题是返回地址的确定。在 2.1 节中提到进入异常时处理器会有一个保存 LR 的动作，但是该保存值并不一定是正确中断的返回地址。下面以一个简单的指令执行流水状态图来对此加以说明。

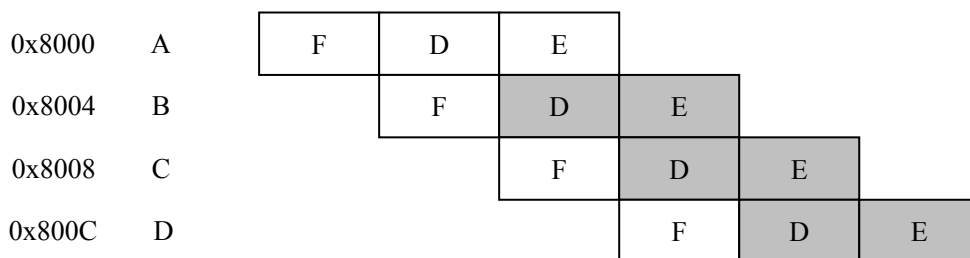


图-8 ARM 状态下 3 级指令流水线执行示例

我们知道在 ARM 架构里,PC 值指向当前执行指令的地址加 8 处。也就是说,当执行指令 A (地址 0x8000) 时,PC 等于指令 C 的地址 (0x8008)。假如指令 A 是“BL”指令,则当执行时,会把 PC (=0x8008) 保存到 LR 寄存器里面,但是接下去处理器会马上对 LR 进行一个自动的调整动作: LR=LR-0x4。这样,最终保存在 LR 里面的是 B 指令的地址,所以当从 BL 返回时,LR 里面正好是正确的返回地址。

同样的调整机制在所有 LR 自动保存操作中都存在,比如进入中断响应时处理器所做的 LR 保存中,也进行了一次自动调整,并且调整动作都是 LR=LR-0x4。由此我们来对不同异常类型的返回地址进行依次比较:

假设在指令 B 处 (地址 0x8004) 发生了中断响应,进入中断响应后 LR 上经过调整保存的地址值应该是 C 的地址 0x8008。

(a) 如果发生的是软件中断,即 B 是“SWI”指令

从 SWI 中断返回后下一条执行指令就是 C,正好是 LR 寄存器保存的地址,所以只要直接把 LR 恢复给 PC。

(b) 如果发生的是“IRQ”或“FIQ”等指令

因为外部中断请求中断了 B 指令的执行,当中断返回后,需要重新回到 B 指令的执行,也就是返回地址应该是 B (0x8004),需要把 LR 减 4。

(c) 如果发生的是“Data Abort”

在 B 上进入数据异常的响应,但导致数据异常的原因却应该是上一条指令 A。当中断处理程序修复数据异常以后,要回到 A 上重新执行导致数据异常的指令,因此返回地址应该是 LR 减 8。

如果原来的指令执行状态是 Thumb,异常返回地址的分析与此类似,对 LR 的调整正好与 ARM 状态完全一致。

2. 4 ARM 编译器对异常处理函数编写的扩展

考虑到异常处理函数在现场保护和返回地址的处理上与普通函数的不同之处，不能直接把普通函数体连接到异常向量表上，需要在上面加一层封装，下面是一个例子：

```

IRQ_Handler          ; 中断响应，从向量表直接跳来
    STMFD SP!, {R0-R12, LR}    ; 保护现场，一般只需保护 {r0-r3,lr} 即可
    BL    IrqHandler          ; 进入普通处理函数，C 或汇编均可
    LDMFD SP!, {R0-R12, LR}    ; 恢复现场
    SUBS  PC, LR, #4           ; 中断返回，注意返回地址
    
```

为了方便使用高级语言直接编写异常处理函数，ARM 编译器对此作了特定的扩展，可以使用函数声明关键字 `__irq`，这样编译出来的函数就满足异常响应现场保护和恢复的需要，并且自动加入对 LR 进行减 4 的处理，符合 IRQ 和 FIQ 中断处理的要求。

```

__irq void IRQ_Handler (void)
{...}
    
```

2.5 软件中断处理

软件中断由专门的软中断指令 SWI 触发，SWI 指令后面跟一个中断编号，以标识可能共存的多个软件中断程序。

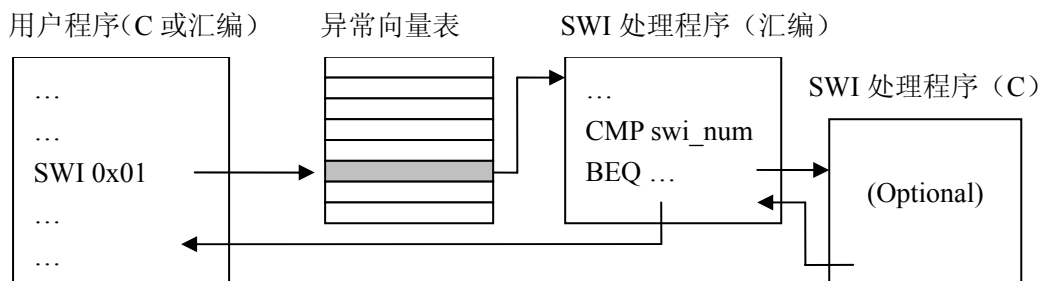


图-9 软件中断处理流程

在 C 程序中调用软件中断需要用到编译器的扩展功能，使用关键字 “`__swi`” 来声明中断函数。注意软中断号码同时在函数定义时指定。

```

__swi(0x24) void my_swi (void);
    
```

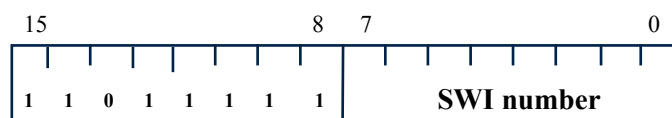
这样当调用函数 `my_swi` 的时候，就会用 “SWI 0x24” 来代替普通的函数调用 “BL `my_swi`”。

分析图 9 的流程，可以发现软件中断同样存在着中断分支的问题，即需要根据中断号码来决定调用不同的处理程序。软中断号码只存在于 SWI 指令码当中，因此需要在中断处理程序中读取触发中断的指令代码，然后提取中断号信息，再

进行进一步处理。下面是软中断指令的编码格式：



ARM 状态下的 SWI 指令编码格式，32 位长度，其中低 24 位是中断编号。



Thumb 状态下的 SWI 指令编码格式，16 位长度，其中低 8 位是中断编号。

图-10 SWI 指令编码格式

为了在中断处理程序里面得到 SWI 指令的地址，可以利用 LR 寄存器。每当响应一次 SWI 的时候，处理器都会自动保存并调整 LR 寄存器，使里面的内容指向 SWI 下一条指令的地址，所以把 LR 里面的地址内容上溯一条指令就是所需的 SWI 指令地址。需要注意的一点是当 SWI 指令的执行状态不同时，其指令地址间隔不一样，如果进入 SWI 执行前是在 ARM 状态下，需要通过 LR-4 来获得 SWI 指令地址，如果是在 Thumb 状态下进入，则只要 LR-2 就可以了。

下面是一段提取 SWI 中断号码的例程：

```

MRS      R0, SPSR          ; 检查进入 SWI 响应前的状态
TST      R0, #T_bit        ; 是 ARM 还是 Thumb? #T_bit=0x20
LDRNEH  R0, [LR, #-2]     ; 是 Thumb, 读回 SWI 指令码
BICNE   R0, R0, #0xff00   ; 提取低 8 位
LDREQ   R0, [LR, #-4]     ; 是 ARM, 读回 SWI 指令码
BICEQ   R0, R0, #0xff000000 ; 提取低 24 位
; 寄存器 R0 中的内容是正确的软中断编号了
    
```

3. 可重入中断设计

如 2.1 节所述，缺省情况下 ARM 中断是不可重入的，因为一旦进入异常响应状态，ARM 自动关闭中断使能。如果在异常处理过程中简单地打开中断使能而发生中断嵌套，显然新的异常处理将破坏原来的中断现场而导致出错。但有时候中断的可重入又是需要的，因此要能够通过程序设计来解决这个问题。其中有二点是这个问题的关键：

(a) 新中断使能之前必须要保护好前一个中断的现场信息，比如 LR_irq 和 SPSR_irq 等，这一点容易想到也容易做到。

(b) 中断处理过程中对 BL 的保护

在中断处理函数中发生函数调用（BL）是很常见的，假设有下面一种情况：

```

IRQ_Handler:
...
BL    Foo  ----->  Foo:
ADD   ...                STMFD SP!, {R0-R3, LR}
...                               ...
                                   LDMFD  SP!, {R0-R3, PC}
    
```

上述程序，在 IRQ 处理函数 IRQ_Handler() 中调用了函数 Foo()，这是一个普通的异常处理函数。但若是在 IRQ_Handler() 里面中断可重入的话，则可能发生问题，考察下面的情况：

当新的中断请求恰好在“BL Foo”指令执行完成后发生。

这时候 LR 寄存器（因在 IRQ 模式下，是 LR_irq）的值将调整为 BL 指令的下一条指令（ADD）地址，以期能从 Foo() 正确返回；但是因为这时候发生了中断请求，接下去要进行新中断的响应，处理器为了能使新中断处理完成后能正确返回，也将进行 LR_irq 保存。因为新中断是在指令流

```

BL  Foo    -->  STMFD  SP!, {R0-R3, LR}
    
```

执行过程中插入的，完成跳转操作后，进行流水线刷新，最后 LR_irq 保存的是 STMFD 后面一条指令的地址；这样当新中断利用（PC = LR - 4）操作返回时，正好可以继续原来的流程执行 STMFD 指令。这二次对 LR_irq 的操作发生了冲突，当新中断返回后往下执行 STMFD 指令，这时候压栈的 LR 已不是原来需要的 ADD 指令的地址，从而使子程序 Foo() 无法正确返回。

这个问题无法通过增加额外的现场保护指令来解决。一个巧妙的办法是在重新使能中断之前改变处理器的模式，也就是使上面程序中的“BL Foo”指令不要运行在 IRQ 模式下。这样当新中断发生时就不会造成 LR 寄存器的冲突了。考虑 ARM 的所有运行模式，采用 System 模式是最恰当的，因为它既是特权模式，又与中断响应无关。

所以一个完整的可重入中断应该遵循以下流程：

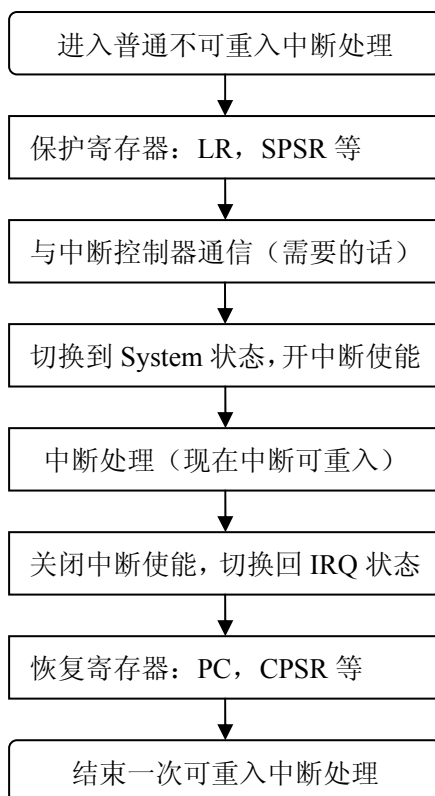
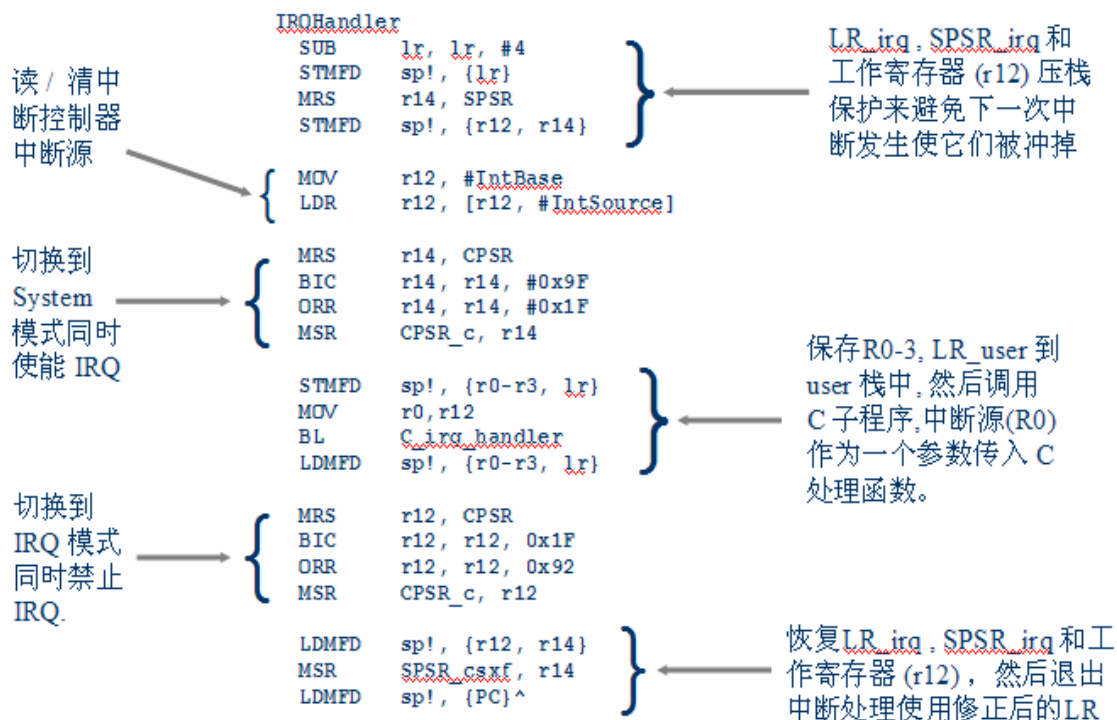


图-11 可重入中断处理流程

下面是一段实现的例程:



基于 ARM 的嵌入式系统程序开发要点（五）

—— ARM/Thumb 的交互工作

在前面的文章中提到过，很多情况下应用程序需要在 ARM 跟 Thumb 状态之间相互切换，这部分就讨论交互工作的实现方法和一些注意问题。

1. 需要交互的原因

前面提到过 Thumb 指令在某些特殊情况下具有比 ARM 指令更为出色的表现，主要是在代码长度和窄带宽存储器系统性能两方面。正因为 Thumb 指令在特定环境下面的优势，它在很多方面得到了广泛的应用。但是因为下面一些原因，Thumb 又不可能独立地组成一个应用系统，所以不可避免地会产生 ARM 与 Thumb 之间交互的问题。

- Thumb 指令集在功能上只是 ARM 指令集的一个子集，某些功能只能在 ARM 状态下执行，如 CPSR 和协处理器的访问。
- 进行异常响应时，处理器会自动进入 ARM 状态。
- 从系统优化考虑，在宽带存储器上不应该放置 Thumb 代码，很多窄带系统具有宽带的内部存储器。
- 即使是一个单纯的 Thumb 应用系统，也必须加一个汇编的交互头程序，因为系统总是自动从 ARM 开始启动。

2. 状态切换的实现

处理器在 ARM/Thumb 之间的状态切换是通过一条专用的跳转交换指令 BX 来实现的。BX 指令以通用寄存器（R0-R15）为操作数，通过拷贝 Rn 到 PC 来实现 4GB 空间范围内的一个绝对跳转。BX 利用 Rn 寄存器中存储的目标地址值的最后一位来判断跳转后的状态。

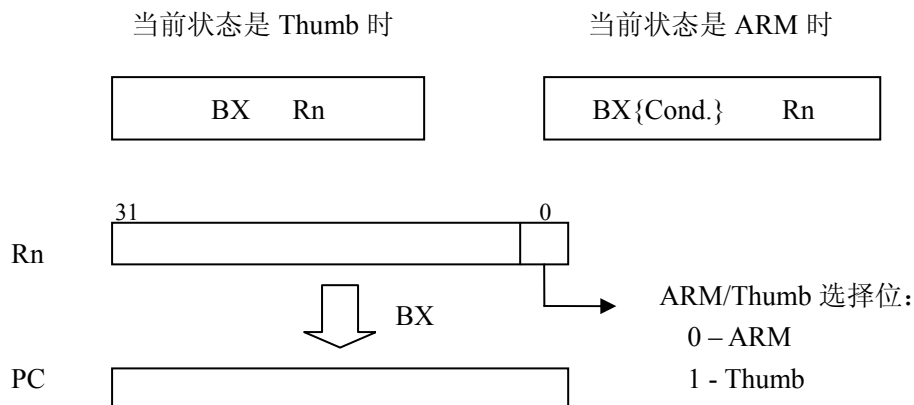


图-1 BX 指令实现状态切换

无论 ARM 还是 Thumb，其指令存储在存储器中都是边界对齐的（4-Byte 或 2-Byte 对齐），所以在执行跳转过程中，PC 寄存器中的最低位肯定被舍弃，不起作用。在 BX 指令的执行过程中，最低位正好被用作状态判断的标识，不会造成存储器访问不对齐的错误。

图 2 中是一段直接进行状态切换的例程：

```

; 从 ARM 状态开始
CODE32                ; 汇编关键字
ADR    R0, Into_Thumb+1 ; 得到目标地址，末位置 1，转向 Thumb
BX    R0                ; 执行
...                    ; 其他代码
CODE16                ; 汇编关键字
Into_Thumb            ; Thumb 代码段起始地址
...                    ; Thumb 代码
ADR    R5, Back_to_ARM ; 得到目标地址，末位缺省为 0，转向 ARM
BX    R5                ; 执行
...                    ; 其他代码
CODE32                ; 汇编关键字
Back_to_ARM           ; ARM 代码段起始地址
...
    
```

图-2 ARM/Thumb 交互工作的例子

我们知道 ARM 的状态寄存器 CPSR 中，bit-5 是状态控制位 T-bit，决定当前处理器的运行状态。如果直接修改 CPSR 的状态位，也能够达到改变处理器运行状态的目的，但是会带来一个问题。因为 ARM 采用了多级流水线的结构，所以在程序执行过程中指令流水线上会存在几条预取指令（具体数目视流水线级数而不同）。当修改 CPSR 的 T-bit 以后，状态的转变会造成流水线上预取指令执行的错误。而如果用 BX 指令，则执行后会进行流水线刷新动作，清除流水线上的残余指令，在新的状态下重新开始指令预取，从而保证状态转变时候指令流的正确衔接。

3. ARM/Thumb 之间的函数调用

在无交互的子程序调用中，其过程比较简单。实现调用通常只需要一条指令：

```
BL    function
```

实现返回也只需要从 LR 恢复 PC 即可：

```
MOV    PC, LR
```

如下图所示：

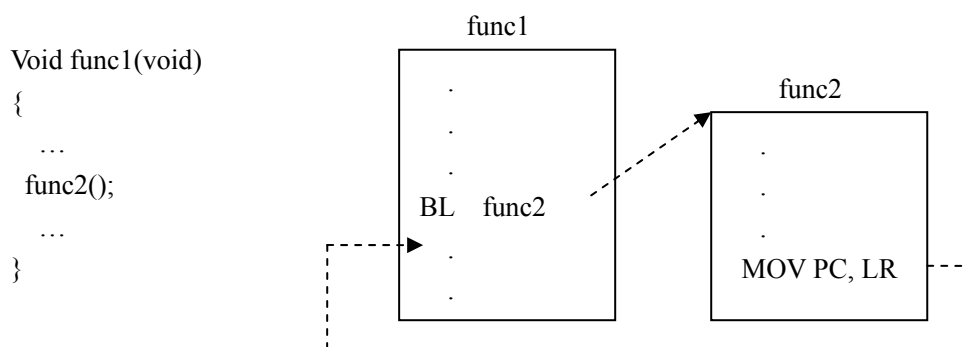


图-3 普通函数调用

如果子函数和父函数不是在同一种状态下执行的，因为状态切换，需要对函数调用作更多的考虑。

- (a) BL 不能完成状态切换，需要由 BX 来切换状态。
- (b) BX 不能自动保存返回地址到 LR，需要在 BX 之前先保存好 LR。
- (c) 用“BX LR”来返回，不能使用“MOV PC, LR”，因为这条指令同样不能实现状态切换。返回时要仔细考虑保存的 LR 中最低位内容是否正确。

假如用户直接使用汇编进行状态交互跳转，上述的几个问题都需要用手工编码加以处理。如果用户使用高级语言进行开发，不需要为 ARM/Thumb 之间的相互调用增加额外的编码，但是最好要对其调用过程加以了解。下面以 ARM ADS 中的编译工具为例进行说明（图 4）。

- (a) 两个函数 func1()和 func2()被编译成了不同的指令集(ARM 或 Thumb)。注意 func1 () 和 func2 () 在这里位于二个不同的源文件。
- (b) 编译时必须告诉编译器和连接器足够的信息，一方面让编译器能够使用正确的指令码进行编译，另一方面这样当在不同的状态之间发生函数调用时，连接器将插入一段连接代码（veneers）来实现状态转换。

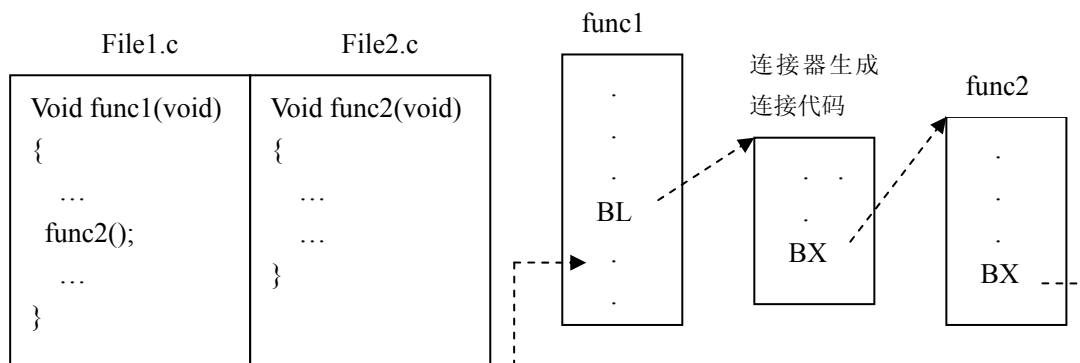


图-4 不同状态间函数调用的示例

上述过程中的一个特点是 func1 还是使用通常的 BL 指令来进行子程序调用，而 func2 返回时则直接使用“BX LR”，没有对 LR 进行判断和最低位的设置。这是因为当执行 BL 指令对 LR 进行保存时，其最低位会被自动设置，以满足返回时状态切换的需要，可直接使用“BX LR”。

在上面的例子中，为了让编译器在编译函数 func2 时使用 BX 而不是 BL 进行返回，必须告诉编译器要按照满足交互工作要求的方式进行编译。在 ARM 的编译器选项设置中，是“-apcs /interwork”。这样，函数的返回指令会被正确设置，并且当连接器进行目标代码的连接时，能够在需要的地方插入正确的连接代码实现状态切换。

当然，插入了连接代码会相应地增加代码长度，通常一段 veneer 包含 3 条指令，即 12B 字节长度。可以用“-info veneers”选项使连接器输出所有 veneers 的位置和长度信息。

4. 交互程序之间的兼容性

正因为指定交互选项后编译及连接后的输出代码跟在无交互情况下不同，所以当多个源文件如果使用了不同的设置进行编译，相互之间的调用可能产生兼容性问题。下面这张图说明了这些关系：

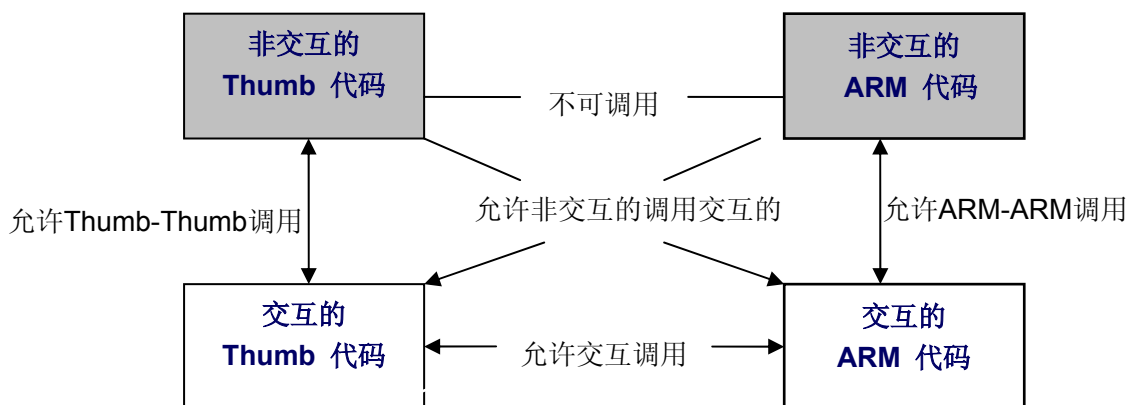


图-5 不同编译方式下函数调用的兼容性

在一个使用交互工作的项目工程管理中，对此要加以仔细考虑。

5. V5 架构的扩展

ARM 在 V5 版本的架构中，对 ARM/Thumb 的交互增加了新的支持。针对前面第 3 节中提到的函数调用和返回问题，V5 版本中专门对指令做了扩展。

(a) 增加了新指令 BLX，解决了原来 BX 和 BL 指令各自的欠缺。使交互的函数调用可以由一条指令实现，省略了跳转代码的开销。

(b) 扩展了以 PC 为目标地址的数据传输指令功能。PC 加载值的最低位将被

自动送到状态寄存器 CPSR 的 T 状态位。也就是说通过给 PC 赋值的方法也能实现状态的切换了，这样就使习惯的函数返回方法——从堆栈中恢复寄存器的方法，也能实现交互调用函数的正确返回了。

所以，V5 架构以后的代码，不再需要额外的连接代码（veneers）了，帮助缩小代码长度，提高状态切换时候的执行效率。当然，在 V5 及以后架构中，继续保持对以前代码的良好兼容性。

6. Thumb-2

ARM 和 Thumb 因为各自的优势都得到了极为广泛的应用，在一个应用程序中，用户要根据系统的具体情况灵活分配，使用不同的编译器，把不同的代码编译成 ARM 或 Thumb，以希望得到最优的代码长度和性能平衡。这样做能够达到系统优化的目的，但是也给设计人员带来了额外的交互处理工作。最近 ARM 公司公布了一项新的发明——Thumb-2 指令集，该指令集同时包含 32 位和 16 位指令，在代码长度和性能之间作了最佳的平衡；这样，以后用户就可以用一个统一的 Thumb-2 编译器来解决现在面临的很多问题了。

下图是 Thumb-2 指令集跟 ARM 和 Thumb 之间的比较。

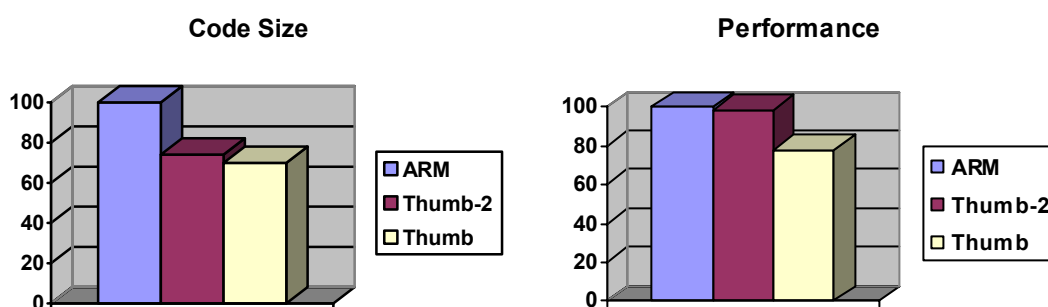


图-6 ARM、Thumb 和 Thumb-2 的比较

新的指令集将在最新的 ARM 授权中发布，更多信息请访问 www.arm.com。

基于 ARM 的嵌入式系统程序开发要点（六）

—— 开发高效程序的技巧

开发高效率的程序涉及很多方面，包括编程风格、算法实现、针对目标的特殊优化等等。这部分主要从 ARM 的体系结构特点出发，介绍几个程序开发中的注意点。如何根据目标硬件的存储器配置，对运行程序的映像文件进行优化布局的方法，在前面的专题中已经有过介绍。

1. 变量定义

变量定义虽然很简单，但是也有很多值得注意的地方。先看下面一个例子：

| | |
|-------|----|
| char | a; |
| short | b; |
| char | c; |
| int | d; |

| | |
|-------|----|
| char | a; |
| char | c; |
| short | b; |
| int | d; |

这里定义的 4 个变量形式都一样，只是次序不同，却导致了在最终的映像中不同的数据布局，见图 1 所示。显然，第二种方式节约了更多的存储器空间。

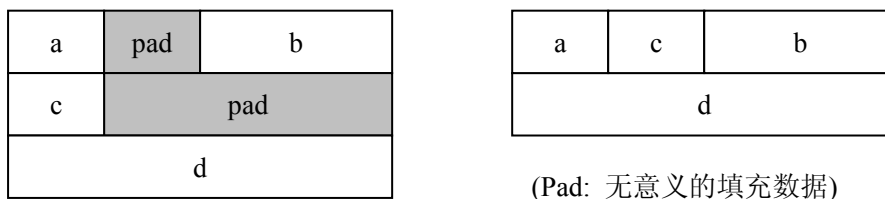


图 1：变量在数据区里的布局

由此可见在变量声明的时候，需要考虑怎样最佳地控制存储器布局。当然，编译器在一定程度上能够优化这类问题，但是最好的方法还是在编程的时候，把所有相同类型的变量放在一起定义。

第二个问题是局部变量的类型定义。一般情况下人们总是设法使用 short 或 char 来定义变量以节省存储器空间；但是，当一个函数的局部变量数目有限的情况下，编译器会把局部变量分配给内部寄存器，每个变量占用一个寄存器。这样，使用 short 和 char 型变量不但起不到节省空间的作用，还会带来其他的副作用，请看图 2。假定 a1 是任意可能的寄存器，存储函数的局部变量。同样完成加一的操作，32 位的 int 型变量最快，只用一条加法指令。而 8 位和 16 位变量，完成加法操作后，还需要在 32 位的寄存器中进行符号扩展，其中带符号的变量，要用逻辑左移（LSL）接算术右移（ASR）两条指令才能完成符号扩展；无符号的变量，要使用一条逻辑与（AND）指令对符号位进行清零。所以，使用 32 位的 int 或 unsigned int 局部变量最有效率。某些情况下，函数从外部存储器读入局部变量进行计算，这时候，往往值得先把不是 32 位的变量转换成 32 位（至于把 8 位或 16 位变量扩展成 32 位后，隐藏了原来可能的溢出异常这个问题，需要进

一步的仔细考虑)。

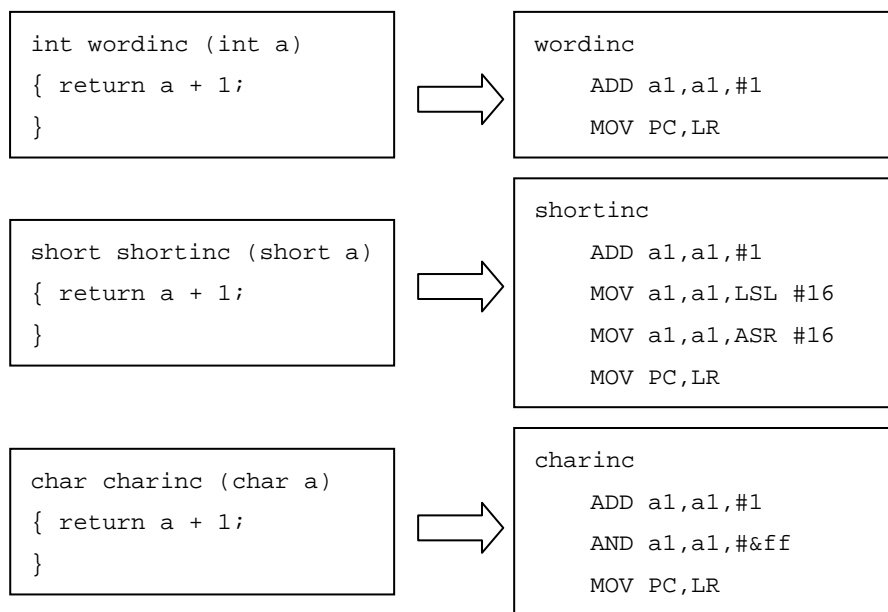


图 2: 不同类型局部变量的编译结果

变量定义中还有一个与习惯思维相悖的地方是冗于局部变量的使用。一般情况下程序员总是竭力避免使用冗余变量，以精简程序。通常情况下这是正确的，但是也有例外，请看下面一个例子：

```
int f(void);
int g(void);           // f()和 g()不访问全局变量errs
int errs;             // 全局变量
void test1(void)
{  errs += f();
   errs += g();
}
void test2(void)
{  int localerrs = errs; // 定义冗余的局部变量
   localerrs += f();
   localerrs += g();
   errs = localerrs;
}
```

在第一种情况 test1 () 里，每次访问全局变量 errs 时都要先从相应的存储器 load 到寄存器里，经 f () 或 g () 函数调用后再 store 回原来的存储器里面，在这个例子里一共要进行两次这样的 load/store 操作。而在第二种情况 test2 () 里，局部变量 localerrs 被分配以寄存器，这样一来，整个函数就只需要一次 load/store 全局变量存储器了；能够节省存储器访问的次数对于系统性能的提高是非常有好处的。

这个例子说明了增加局部变量可以减少存储器的访问。

2. 参数传递

在 ARM 的工具链里，定义了统一的函数过程调用标准 ATPCS (ARM-Thumb Procedure Call Standard)。ATPCS 定义了寄存器组中的 {R0 – R3} 作为参数传递和结果返回寄存器，如果参数数目超过四个，则使用堆栈进行传递。我们知道内部寄存器的访问速度是远远大于存储器的，所以要尽量使参数传递在寄存器里面进行，即应尽量控制函数的参数在四个以下。这是理解 ATPCS 后应该实现的一种编程风格。但是，利用 ATPCS 我们还可以得到更多，我们可以用它来实现 C 与汇编之间直接的函数调用。见图 3 中的例子：

从 C 中直接调用汇编函数

| | |
|--|---|
| <pre>extern void strcpy(char *d, const char *s); int main(void) { const char *src = "Source"; char dest[10]; ... strcpy(dest, src); ... }</pre> | <pre>AREA StrCopy, CODE, READONLY EXPORT strcpy strcpy LDRB R2, [R1], #1 STRB R2, [R0], #1 CMP R2, #0 BNE strcpy MOV PC, LR END</pre> |
|--|---|

图 3：利用 ATPCS 编写汇编函数

这个例子中的函数 strcpy (dest, src) 用汇编来实现，根据 ATPCS 的定义，函数参数从左到右由寄存器进行传递，所以在汇编中可以直接由 R0 和 R1 进行引用。有了这条途径，在 C 和汇编之间进行相互调用就容易实现了。

3. 循环条件

记数循环是程序中十分常用的流程控制结构。在 C 中，类似下面的 for 循环比比皆是：

```
for (loop = 1; loop <= limit; loop++)
```

这种累加计数的方法符合一般的自然思维习惯，所以比下面的递减计数方法使用更多：

```
for (loop = limit; loop != 0; loop--)
```

这两者在逻辑上并没有效率差异，但是映射到具体的体系结构中，就产生了

很大的不同，如图 4。

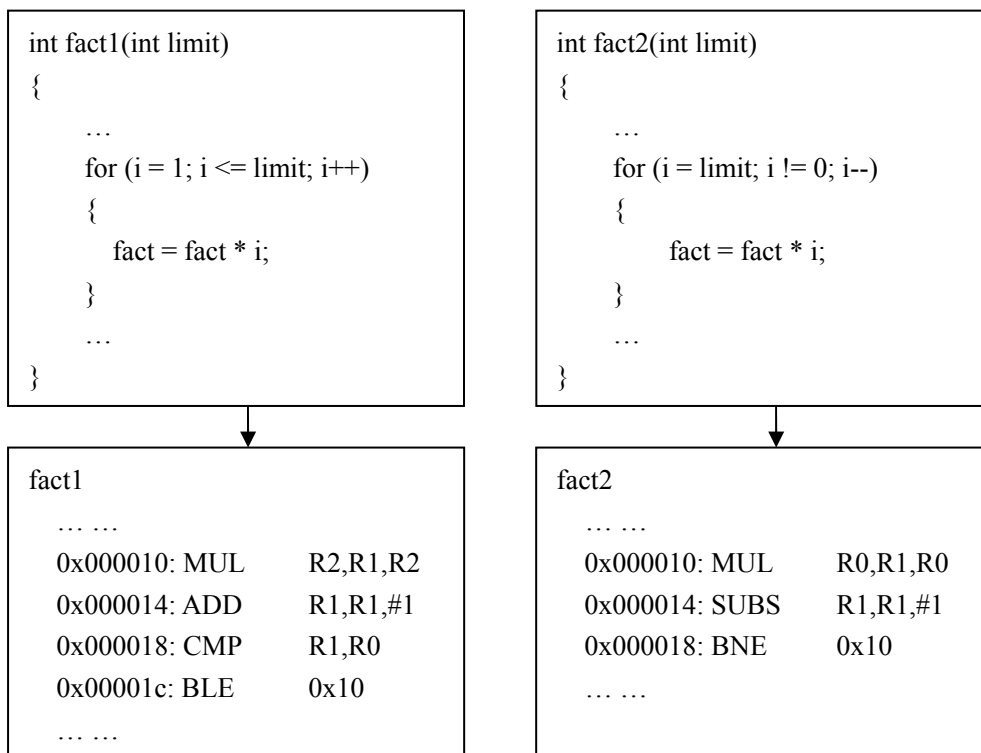


图 4：不同的循环条件设置比较

从图 4 中可以发现累加法比递减法多用了一条指令，当循环次数比较大的时候，这两段代码就会在性能上产生出明显的差异来。分析其中的本质原因，在于当进行一个非零常数比较时，必须用专门的 **CMP** 指令来执行；而当一个变量与零进行比较时，ARM 指令可以直接利用条件执行的特性（NE）来进行判别。

因此，在 ARM 的体系结构下编程，最好采用递减至零的方法来设置循环条件。

4. 条件执行

上面已经提及了 ARM 指令的条件执行，充分利用这个特性，可以有助于缩短代码长度，优化流程控制。请看下面一个例子（图 5）：

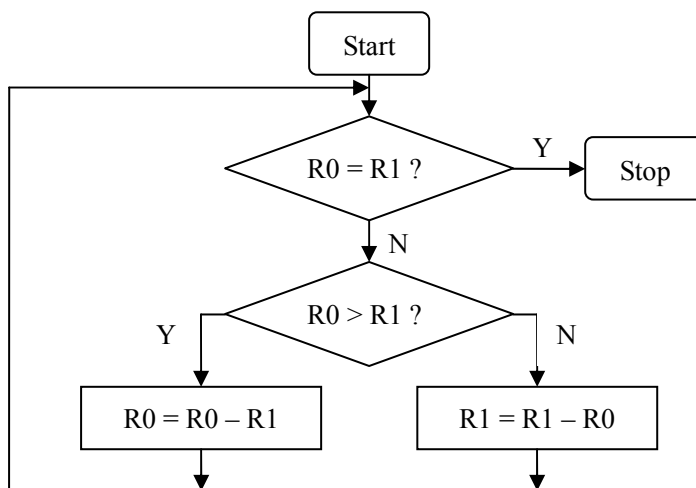


图 5: 求最大公约数的程序流程

图 5 中的流程是求 R0 和 R1 最大公约数的例子。流程图中有两次条件比较，怎样才能得到最优的结果呢？下面给出了解决方案：

```

Start      CMP      R0, R1
           SUBLT   R1, R1, R0
           SUBGT   R0, R0, R1
           BNE    Start
Stop
    
```

从这个例子得到启发，在充分利用条件执行情况下，可以从一次比较判断得到多个跳转分支。

不过成组的条件执行指令跟在一个比较指令后面，如果条件执行的语句太多，在性能上会有牺牲。一般一条跳转指令 B 最多耗费 3 个指令周期，如果一个条件执行指令组的数目超过 3 条，可以考虑用跳转指令来进行条件分支，有助于条件判决的速度，当然这么做就增加了代码长度，属于代码长度跟执行性能之间的一个矛盾平衡问题。如果性能是首要问题，那么在 C 中的条件描述语句中（如 if（条件描述）），应尽可能简化条件描述，因为复杂的条件描述容易产生较长的条件执行指令组来。

5. 混合编程

汇编和 C/C++ 语言的混合编程，在一个追求效率的程序中是比较常见的。前面已经讲到在汇编和 C/C++ 之间进行函数调用时，要遵循 ATPCS 的定义。这里介绍在 C/C++ 里加入汇编程序的两种方法：内联汇编（Inline Assemble）和嵌入式汇编（Embedded Assemble）。

内联汇编是指在 C/C++ 函数定义中插入汇编语句的方法，如下面的例子：

```
void enable_IRQ(void)
```

```

{
  int tmp;
  asm                                // 内联汇编定义
  {
    MRS tmp, CPSR                    // 可以引用外部的 C 变量定义
    BIC tmp, tmp, #0x80
    MSR CPSR_c, tmp
  }
}

```

内联汇编的用法跟真实汇编之间有很大的区别，并且不支持 Thumb，在内联汇编之中不能直接访问物理寄存器（CPSR 除外），即使使用寄存器名进行编程，也会被编译器进行重新分配。

与内联汇编不同，嵌入式汇编具有真实汇编的所有特性，同时支持 ARM 和 Thumb，但是不能直接引用 C/C++ 的变量定义，数据交换必须通过 ATPCS 进行。嵌入式汇编在形式上表现为独立定义的函数体，如下所示：

```

asm int add(int i, int j)           // 定义嵌入式汇编
{
  ADD R0, R0, R1                    // Value of i in R0 and j in R1, result in R0
  MOV PC, LR
}
void main()
{
  printf("12345 + 67890 = %d\n", add(12345, 67890));
}

```

灵活使用内联汇编和嵌入式汇编，可以帮助提高程序效率。

6. 性能分析

很多时候需要对程序的执行效率和性能进行分析，直接测试当然是最真实的途径，但是这种方法除了在运行时间上进行定量外，很难得到确切的数据信息。而指令集仿真这种方法（ARMulator 或 ISS），恰恰为程序执行过程中处理器的行为提供了一个参数统计方法。

图 6 是在 ADS 的 ARMulator 环境下，对某一段程序运行的统计情况。统计可以在执行代码流中任意选择一个参考点开始。

| Reference Points | Instructions | Core_Cycles | S_Cycles | N_Cycles | I_Cycles | C_Cycles | Total |
|------------------|--------------|-------------|----------|----------|----------|----------|-------|
| \$statistics | 10448 | 20008 | 12098 | 5898 | 2012 | 0 | 20008 |
| Referencel | 9307 | 17769 | 10765 | 5111 | 1893 | 0 | 17769 |

图 6: ARMulator 环境下的处理器运行状态分析

通过在感兴趣的代码段两端设置参考点，把执行过程中处理器的各种状态周期精确地统计出来，作为性能分析最直接的分析数据。图 6 中所示是一段例程在 ARM7TDMI 上面运行的状态统计，可以计算出平均每条指令花费的处理器时钟为 1.9 左右，进行代码优化时，目标就是减少非顺序访问周期和内部等待周期数。

7. 小结

代码优化是个很大的题目，这里只是抛砖引玉，索引几个要点进行讨论。更多这方面的知识，可以参阅这方面的众多论述和著作。

至此，关于在 ARM 体系结构下进行嵌入式系统编程的 6 个专题已经全部结束。对此感兴趣的读者可以访问 <http://www.arm.com> 进一步了解更多有关 ARM 的技术资料。