将8051应用程序迁移到ARM Cortex-M处理器上

作者：Joseph Yiu和Andrew Frame

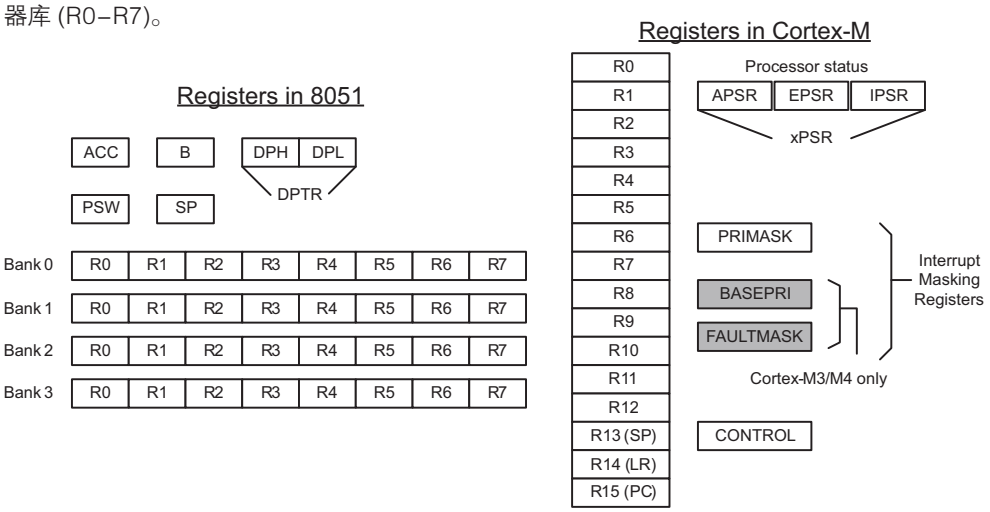
Cortex-M处理器系列包括广泛使用的Cortex-M3处理器、针对FPGA的Cortex-M1处理器、2009年初推出的Cortex-M0处理器（最小的 ARM 处理器）和2010年初推出的 Cortex-M4处理器（支持浮点和数字信号处理增强指令）。这些处理器具有先进的功能特点和简单易用的编程模型，对于想从8051微控制器迁移到ARM架构的开发人员来说，极具吸引力。本文是一篇入门指南，目的是帮助8051微控制器的开发人员了解8051和ARM Cortex-M处理器系列在架构、软件和硬件设计上的主要差异，从而加快迁移过程。

架构概述

对于一些嵌入式程序员（尤其是那些习惯使用汇编语言编程的程序员），首先要做的事情就是了解编程模型。

寄存器

ARM Cortex-M处理器具有一个32位寄存器库和一个xPSR（组合程序状态寄存器）。而8051具有ACC（累加器）、B、DPTR（数据指针）、PSW（处理器状态字）和四个各含八个寄存器的寄存器库 (R0–R7)。



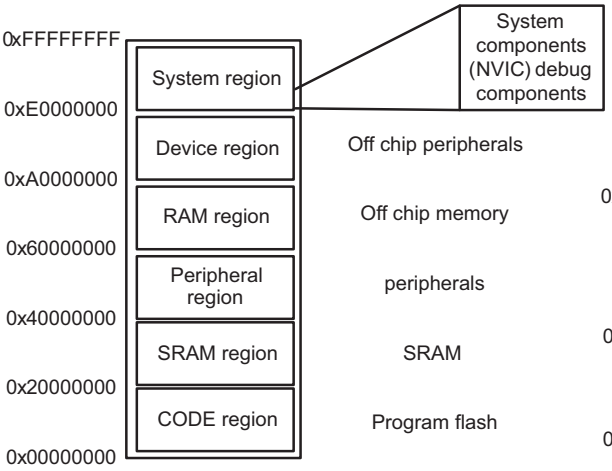
在8051中，一些指令会频繁使用某些寄存器，如ACC和 DPTR。这种相关性会极大降低系统的性能，而在ARM处理器中，指令可使用不同的寄存器来进行数据处理、内存存取和用作内存指针，因此不会有这个问题。

从根本上说，ARM 架构是一个基于加载（Load）和存储（Store）的RISC架构，处理器寄存器加载数据，然后将数据传给ALU进行单周期执行。而8051寄存器（ACC、B、PSW、SP和DPTR）可在SFR（特殊功能寄存器）的内存空间中访问。

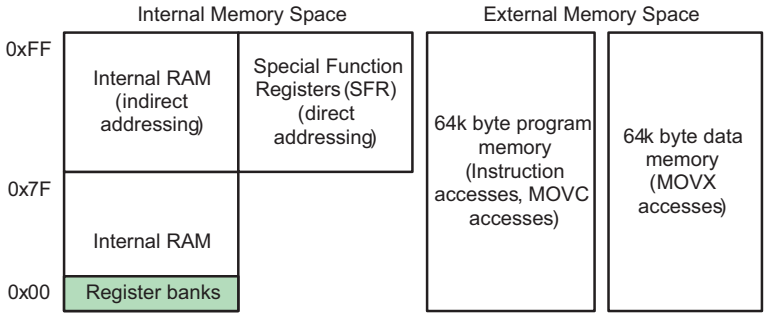
为了确保普通的C函数能够用作中断处理程序，在需要处理中断时，Cortex-M的寄存器（R0–R3、R12、LR、PC和xPSR）会被自动压入堆栈，而软件仅需在必要时将其他寄存器压入堆栈。虽然8051具有4个寄存器库，但是ACC、B、DPTR和PSW寄存器并不会自动压栈，因此通常需要通过中断处理程序对这些寄存器进行软件压栈。

内存空间

ARM处理器具有32位寻址，可实现一个4GB的线性内存空间。该内存空间在结构上分成多个区。每个区都有各自的推荐用法（虽



Memory Map in the ARM CortexM processors



Memory Map in the8051 microcontrollers

然并不是固定的)。统一内存架构不仅增加了内存使用的灵活性,而且降低了不同内存空间使用不同数据类型的复杂性。

相反地,8051微控制器具有多个内存空间。内存空间的分割使得有效地利用全部内存空间变得困难,而且需要借助C语言扩展来处理不同的内存类型。

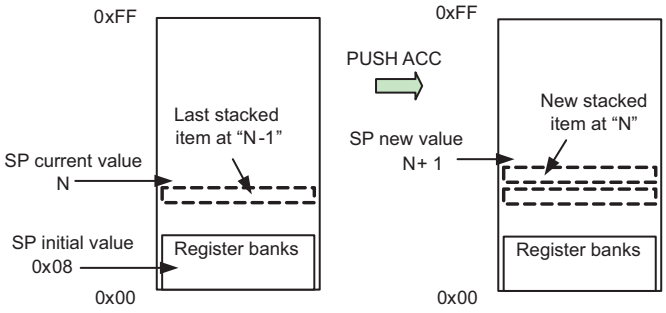
8051在外部RAM内存空间上最高支持64KB的程序内存和64KB的数据内存。理论上,可以利用内存分页来扩展程序内存大小。不过,内存分页解决方案并未标准化,换句话说,不同8051供应商的内存分页的实现并不相同。这不仅会增加软件开发的复杂性,而且由于处理页面切换所需的软件开销,还会显著降低软件性能。

在ARM Cortex-M3或M4上,SRAM区和外设区都提供了一个1MB的位段区(bit band region)。此位段区允许通过别名地址访问其内部的每个位。由于位段别名地址只需通过普通的内存存取指令即可访问,因此C语言完全可以支持,不需要任何特殊指令。而8051提供了少量的位寻址内存(内部RAM上16字节和SFR空间上16字节)。处理这些位数据需要特殊指令,而要支持此功能,C编译器中需要C语言扩展。

ARM Cortex-M处理器的内存映射包含多个内置外设块。例如,ARM Cortex-M处理器的一个特性是具有一个嵌套矢量中断控制器 (NVIC) 。此外,系统区中内存映射有数个指定控制寄存器和调试组件,以确保优异的中断处理并极大方便开发人员使用。

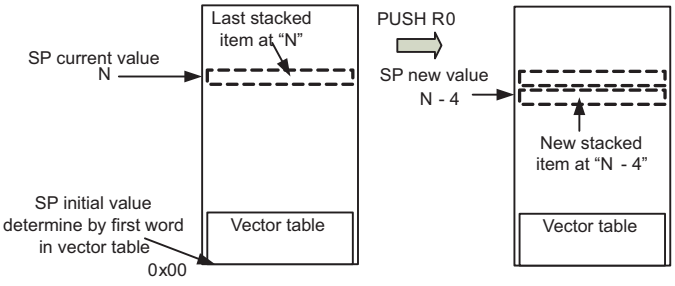
堆栈内存

堆栈内存操作是内存架构的重要组成部分。在8051中,堆栈指针只有8位,同时堆栈位于内部的内存空间(上限为256个字节,并由工作寄存器(四个各由R0至R7构成的寄存器库)和内部数据变量共享)。堆栈操作基于空递增模型。



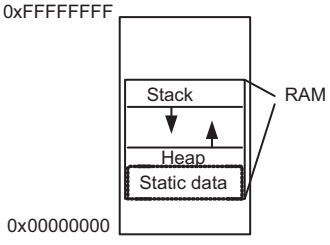
Stacking in 8051

与8051不同的是,ARM Cortex-M处理器使用系统内存作为堆栈,采用满递减模型。



Stacking in ARM Cortex -M processor

满递减堆栈内存模型更受C语言的支持。例如,微控制器中的SRAM的使用可组织为:



RAM usage in ARM Cortex -M

使用动态分配内存空间的C库和应用程序通常需要堆内存。

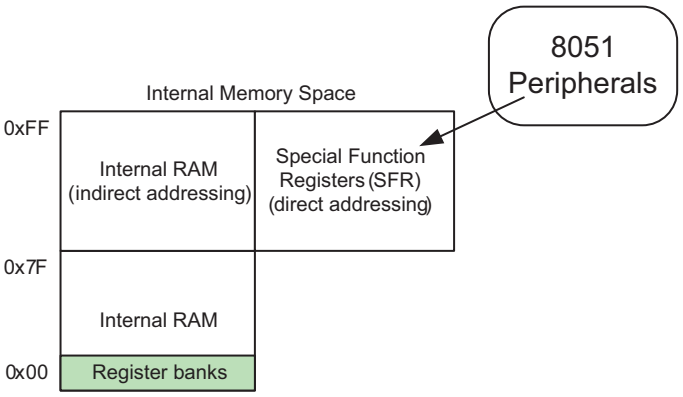
尽管Cortex-M处理器的每次压栈需要32位的堆栈内存,总

的RAM使用仍然要比8051小。8051的变量通常是静态地放在IDATA上，而ARM处理的局部变量是放在堆栈内存上的，因此，只有当函数执行的时候，局部变量才会占用RAM空间。

此外，ARM Cortex–M 处理器提供有第二个堆栈指针，以允许操作系统内核和进程堆栈使用不同的堆栈内存。这使得操作更可靠，也使操作系统设计更高效。（堆栈指针切换是自动处理的）

外设

8051中的很多外设是通过特殊功能寄存器 (SFR) 来控制的。由于SFR空间只有128个字节，而且其中一些已经为处理器寄存器和标准外设所占用，剩余的SFR地址空间通常非常有限，因此也就限制了可通过SFR控制的外设数量。虽然可以通过外部内存空间来控制外设，但是与SFR存取相比，外部存取通常需要更多的开销（需要将地址复制到DPTR，数据必须通过ACC传输）。



在ARM Cortex–M处理器中，所有外设都是内存映射的。由于所有寄存器都可用作指针或数据访问中的数据值，因此效率非常高。在C语言中，访问外设地址的一个简单方法就是使用指针，如：

```
(*((volatile unsigned long *)(LED_ADDRESS))) = 0xFF; // Output to LED
ReceviedData = (*((volatile unsigned long *)(IO_INPUT_ADDRESS))); // Read from IO
```

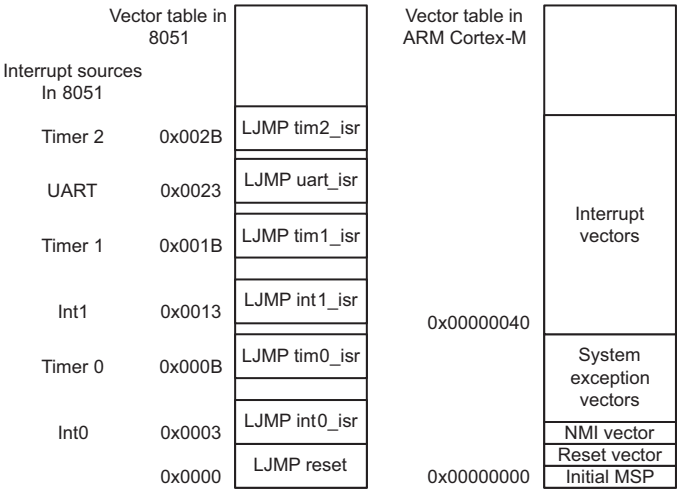
此外，您可以声明外设块的数据结构。使用数据结构，程序代码只需要存储外设的基址，而且每个寄存器访问可以利用带有立即数偏移量的加载或存储指令来执行，因此效率会得到提高。例如，具有四个寄存器的外设可以定义为：

```
typedef struct
{
volatile unsigned long register0;
volatile unsigned long register1;
volatile unsigned long register2;
volatile unsigned long register3;
} SomePeripheral_Type;
#define SomePeripheral ((SomePeripheral_Type *) 0x40003000) /* define base address */
SomePeripheral–>register2 = 0x3; /* Set register  #2 to 0x3 */
```

由于ARM处理器中外设总线协议的特性，外设寄存器通常定义为32位，即使只会用到其中几位。此外，外设寄存器的地址是字对齐的。例如，如果外设位于地址0x40000000处，那么对应外设寄存器的地址就是0x40000000、0x40000004和0x40000008等。某些运行在主系统总线上的外设没有这个限制。

异常

8051支持具有两个可编程优先级的矢量中断。一些较新的8051支持四个级别和稍多的中断源。它们也支持嵌套中断。当中断发生时，程序会保存返回地址，然后跳转到矢量表中的固定地址。矢量表通常包含有另一个分支指令，以便跳转至中断服务程序的实际开始位置。示例：



进入中断服务程序时，需要通过软件代码将PSW（也可能包括ACC和DPTR 等）压入堆栈并切换寄存器库。

ARM Cortex–M处理器的中断处理由嵌套矢量中断控制器 (NVIC) 提供。NVIC紧密地耦合到处理器内核，支持矢量中断和嵌套中断。此外，它还支持更多中断源：Cortex–M0/M1支持最多32个IRQ，Cortex–M3支持最多240个IRQ。Cortex–M0/M1支持4个可编程优先级，而 Cortex–M3则支持8至256个级别，具体数目视实现情况而定（通常为8或16个级别）。

与8051不同的是，ARM Cortex–M处理器的矢量表存储的是异常处理程序的开始地址。此外，Cortex–M处理器支持NMI（非屏蔽中断）和一些系统异常。系统异常包括特别针对操作系统的异常类型和用于检测非法操作的故障处理异常。这些功能都是8051上所没有的。

8051中的中断服务程序需要通过RETI指令来终止，该指令与用于标准函数的RET指令不同。在ARM Cortex–M中，中断服务程

序与普通的C函数完全相同。异常机制使用异常进入期间LR中生成的特殊返回地址代码来检测异常返回。

软件

简单了解架构情况后，我们现在来讨论软件代码的移植。在很多情况下，针对8051编写的C代码需要进行大量的修改。很显然，内存映射和外设驱动代码是不同的。除此之外，我们还需要特别注意其他一些地方：

数据类型

8051和ARM处理器的数据类型有一些差异。由于数据大小不同，如果程序代码依赖于数据大小或溢出行为，不做修改可能会无法工作。下表所示为常见数据类型的大小，具体视编译器而定。这里是指8051的KEIL C编译器和ARM RealView编译器（也适用于KEIL RealView微控制器开发套件）。

Type	Number of bits in 8051	Number of bits in ARM
“char”, “unsigned char”	8	8
“enum”	16	8/16/32 (smallest is chosen)
“short”, “unsigned short”	16	16
“int”, “unsigned int”	16	32
“long”, “unsigned long”	32	32

数据类型大小不同的另一个影响是在ROM中保留常数数据所需的大小。例如，如果8051程序中包含一个整数型常数数组，那么你需要修改代码，将该数组定义为短整型常数。否则，代码长度可能会因为该数组从16位变成32位而增加。

由于8051架构的特性，8051的C编译器还支持一些数据类型和内存类型扩展。这些数据类型在 ARM 处理器上是不受支持的。示例：

Type	Description	Number of bits
“bit”	Bit addressable memory in 0x20 to 0x3C	1
“sbit”	Bit addressable memory in SFR space	1
“sfr”	Special Function Register	8
“sfr16”	16-bit Special Function Register	16
“idata”	Specify data is in internal data memory	
“xdata”	Specify data is in external data memory	
“bdata”	Specify data is in bit addressable memory	

Cortex–M3处理器的用户可以使用位段区来管理位数据。由于位段允许利用位段别名地址通过普通的内存存取指令来访问位数据，因此可以将位数据声明为指向位段别名地址的内存指针。

或者，如果你正在使用ARM RealView编译器或KEIL MDK–ARM，那么可以使用编译器特有的位段功能。

- __attribute__((bitband))
- --bitband command line option

有关此功能的更多详细信息，请参阅《ARM RealView编译器用户指南》或Keil在线文档。

对于外设地址，您可以按照前文所述将SFR数据类型替换为易失性内存指针。由于8051指令集的特性，SFR地址是硬编码在指令中的。在ARM微控制器中，您可以将外设的寄存器定义为内存指针，并将寄存器作为数据结构或数组来访问。这要比8051灵活很多。

浮点

由于8051的处理能力限制，大多数8051 C编译器会将“双精度”数据类型（64位）作为单精度（32位）来处理。而在ARM处理器中使用相同代码时，C编译器将使用双精度，因此程序行为可能会发生变化。例如，如果您只需要单精度，就需要对以下从Whetstone中提取的代码进行修改：

```
X=T*atan(T2*sin(X)*cos(X)/(cos(X+Y)+cos(X–Y)–1.0)); /* double precision on ARM*/
Y=T*atan(T2*sin(Y)*cos(Y)/(cos(X+Y)+cos(X–Y)–1.0));
```

对于单精度运算，代码需要更改为：

```
X=T*atanf(T2*sinf(X)*cosf(X)/(cosf(X+Y)+cosf(X–Y)–1.0F)); /* single precision on ARM*/
Y=T*atanf(T2*sinf(Y)*cosf(Y)/(cosf(X+Y)+cosf(X–Y)–1.0F));
```

对于不需要双精度精确度的应用程序，将代码更改为单精度能够提高性能以及缩短代码长度。

中断处理程序

为了使8051 C编译器为中断处理程序产生正确的代码，需要用到一些函数扩展。这可确保函数使用RETI（而非RET）来返回并确保将所有用到的寄存器保存到堆栈中。在8051的KEIL C编译器中，这是通过“interrupt”扩展来实现的。示例：

```
void timer0_isr (void) interrupt 1
{ /* 8051 timer ISR */
  …
  return;
}
```

在ARM Cortex–M处理器中，中断服务程序被作为普通的C函数那样来编译。因此，可以去掉“interrupt”扩展。在ARM RealView编译器中，你也可以添加 __irq关键词来加以说明。示例：

```
__irq void timer0_isr (void)
{ /* ARM timer ISR */
  …
  return;
}
```

8051编译器的另一C扩展用于指定所使用的寄存器库。示例：

```
void timer0_isr (void) interrupt 1 using 2
{ /* use register bank #2 */
  …
  return;
}
```

同样，ARM处理器不需要此扩展，可以将其删除。

可重入函数

对于8051，普通的C函数无法用作可重入函数。这是因为局部变量是静态的，如果重入函数，局部变量可能会遭到损坏。为了解决此问题，一些8051 C编译器支持“reentrant”扩展。例如，使用KEIL 8051 C编译器时，可以将函数声明为：

```
void foo (void) reentrant
{
  return;
}
```

ARM处理器的局部变量存储在堆栈中，重入普通函数并不会出现问题，因此可以删除“reentrant”扩展。

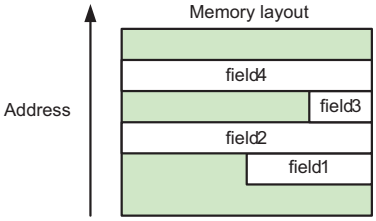
非对齐数据

在ARM微控制器编程中，数据变量的地址通常必须是对齐地址。换句话说，变量“X”的地址应该是sizeof(X)的倍数。例如，字变量的地址最低两位应该是零。

ARM Cortex–M0/M1要求数据对齐。Cortex–M3处理器支持非对齐数据访问，然而C编译器通常不会生成非对齐数据。如果数据不对齐，那么访问数据将需要更多的总线周期，因为AMBA AHB LITE总线标准（在Cortex–M处理器中使用）不支持非对齐数据。访问非对齐数据时，总线接口必须将其拆分成数个对齐传输。

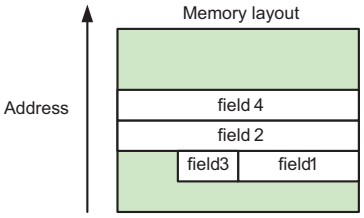
在使用不同大小的元素来创建数据结构时，你可能会尝试各元素的不同排列方式使该数据结构所需的内存最少。例如，像下面这样的结构：

```
struct sample {
  short field1; // 2 bytes
  Int field2;  // 4 bytes
  char field3; // 1 byte
  Int field4;  // 4 bytes
};
```



通过重新排列结构中的元素，可以使该结构所需的内存减小：

```
struct sample {
  short field1; // 2 bytes
  char field3; // 1 byte
  Int field2;  // 4 bytes
  Int field4;  // 4 bytes
};
```



由于Cortex–M0/M1不支持非对齐数据处理，如果应用程序尝试使用非对齐传输，会触发故障异常。C程序通常不会产生非对齐传输，但如果你手动安排C指针的位置，就可以生成非对齐数据，并导致Cortex–M0/M1的故障异常。Cortex–M3可以配置异常陷阱来检测非对齐传输，从而强制非对齐传输生成故障异常。

故障异常

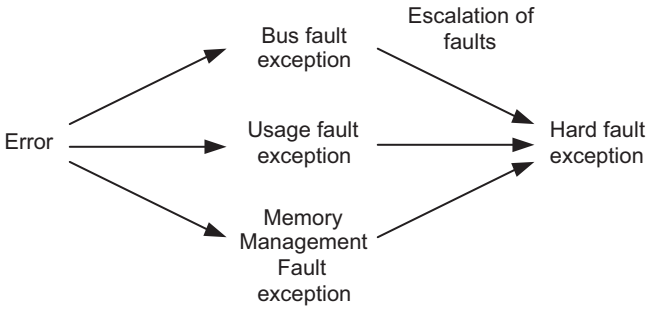
ARM处理器和8051之间的一个主要差别在于，ARM处理器通过故障异常来处理错误事件。内存或外设可能会发生错误（总线错误响应），检测到异常操作时，处理器内部也可能会发生错误（如无效指令）。错误检测功能有助于构建可靠的系统。

常见故障包括：

- 内存（数据或指令）访问无效内存空间
- 无效指令（例如指令内存损坏）
- 不允许的操作（例如尝试切换到ARM指令集，而非Thumb指令集）
- 违反MPU内存访问权限（非特权程序尝试访问特权地址）

在Cortex–M0/M1处理器中，检测到任何错误时，都将使用称为硬故障的异常类型。硬故障处理程序的优先级要高于除NMI之外的其他异常。您可以使用此异常来报告错误，或者在必要时复位系统。

Cortex–M3处理器中有两个级别的错误处理程序。当错误发生时，如果已启用第一级错误处理程序，并且这些处理程序的优先级高于当前的执行级别，就执行这些处理程序。如果未启用第一级错误处理程序，或者这些处理程序的优先级并不高于当前的执行级别，就调用第二级错误异常，即硬故障异常。



此外，Cortex–M3处理器包含有数个故障状态寄存器，用于对故障进行诊断。对于Cortex–M0/M1，由于进入硬故障异常时会将数个核心寄存器（如PC和PSR）压入堆栈，因此可通过堆栈跟踪获取基本调试信息。

设备驱动程序和CMSIS

微控制器厂商会以设备驱动程序库的形式提供很多外设控制程序代码。这类代码可显著缩短软件开发时间。即使你不直接使用该设备驱动程序代码，它也可为设置和控制各种外设提供颇具价值的参考。

在一些ARM微控制器厂商提供的设备驱动程序中包含CMSIS（Cortex微控制器软件接口标准）。CMSIS是用于Cortex–M处理器的一套函数和定义。这些函数和定义是多个厂商共同采用的标准，它使得在不同Cortex–M微控制器之间移植软件变得更容易。CMSIS由以下内容组成：

- 寄存器定义，包括NVIC中断控制、系统控制块(用于处理器控制)、SysTick定时器(用于嵌入式操作系统的24位减法计数器)。
- 一些用于 NVIC 中断控制的函数
- 一些实现处理器核心功能的函数
- 标准化的系统初始化函数

例如，如果希望禁用或启用所用中断，你可以使用CMSIS函数“__disable_irq”和“__enable_irq”。借助CMSIS，此代码可以在不同的Cortex–M微控制器上使用，并且得到了ARM开发工具（ARM RealView开发套件和KEIL MDK–ARM）、GCC（如CodeSourcey G++）和IAR C编译器的支持。

此外，CMSIS包含一些隐含函数，让你可以产生一些特殊指令，这些指令无法用普通C代码由C编译器来产生。例如，您可以使用隐含函数来访问特殊寄存器和创建独占访问（对于Cortex–M3的多处理器编程）等。同样，CMSIS使得所开发的软件可以在多个C编译器产品之间进行移植。

CMSIS对所有Cortex–M开发人员都非常重要，尤其是那些为多个项目开发嵌入式操作系统、中间件和可重用嵌入式软件的人员。CMSIS包含在微控制器厂商提供的设备驱动程序中，也可以从www.onarm.com网站下载。

混用C语言和汇编程序

大多数情况下，您可以完全用C语言来编写Cortex–M应用程序。即使你需要访问一些C编译器无法通过普通C代码生成的特殊指令，也可以使用CMSIS提供的隐含函数，或者根据需要在应用程序中使用汇编语言。您可以在单独的汇编程序文件中编写汇编代码，也可以使用C编译器的特定方法将汇编代码混合在C程序文件中。

使用ARM（和KEIL）开发工具时，将汇编代码插入C编程文件的方法称为“嵌入式汇编程序”。汇编代码声明为函数，并可以被C代码调用。示例：

```
int main (void)
{
  int  status;
  status = get_primask();
  while(1);
}
__asm int get_primask (void)
{
  MRS  R0,  PRIMASK ; Put interrupt masking register in R0
  BX  LR ; Return
}
```

有关嵌入式汇编程序的更多详细信息，请参阅《RealView编译器用户指南》。使用GCC和IAR编译器时，您可以使用内嵌汇编程序将汇编代码插入到C程序代码。请注意，虽然包括RVDS和KEIL MDK–ARM在内的ARM开发工具中也包含内嵌汇编程序

Common tasks	8051	Cortex-M0/M3 with CMSIS
Disable all interrupt	EA = 0;	__disable_irq();
Enable all interrupt	EA = 1;	__enable_irq();
IDLE mode / Sleep	PCON = PCON | 1;	__WFI(); /* Wait for Interrupt */ Vendor specific sleep modes access available in device driver libraries.

功能，但是ARM工具中的内嵌汇编程序仅支持ARM指令，并不支持Thumb指令，因此不能用于Cortex-M处理器。

在汇编程序和C语言混合环境中，您可以通过汇编程序代码调用C函数，也可以通过C函数调用汇编程序代码。数据传输的寄存器使用可参见“ARM 架构程序调用标准 (AAPCS)”的文档。此文档可以从 ARM 网站获取。在简单的情况下，您可以使用R0-R3作为函数的输入（R0作为第一个输入变量，以此类推），并使用R0来返回结果。函数应该保留R4至R11的值，而如果你调用C函数，那么返回时该C函数可能会更改R0-R3和R12的值。

调试

调试和跟踪功能概述

ARM Cortex微控制器可以使用JTAG或串行线协议来调试嵌入式应用程序。串行线协议是在Cortex-M3中引入的。该协议可以完成相同的调试任务，却仅需要2根信号线（JTAG需要5根信号线）。大多数Cortex-M0微控制器允许使用JTAG或串行线协议，但并不允许使用两者。

Cortex-M处理器包含有一些断点和观察点比较器，你还可以使用断点指令向程序代码中插入其他断点。调试器允许你暂停、重新启动和单步调试程序执行，并允许你检查核心寄存器和内存中的数据。你甚至可以在处理器运行时访问系统内存。

此外，Cortex-M3处理器支持可选的嵌入式跟踪宏单元 (ETM)。借助ETM，RealView Trace或Keil ULINK Pro等跟踪端口分析器可以跟踪程序指令执行序列并收集相关信息。在程序编译期间，跟踪信息可用来对程序代码中的问题进行调试，或者由RealView Profiler用来提高优化性能。

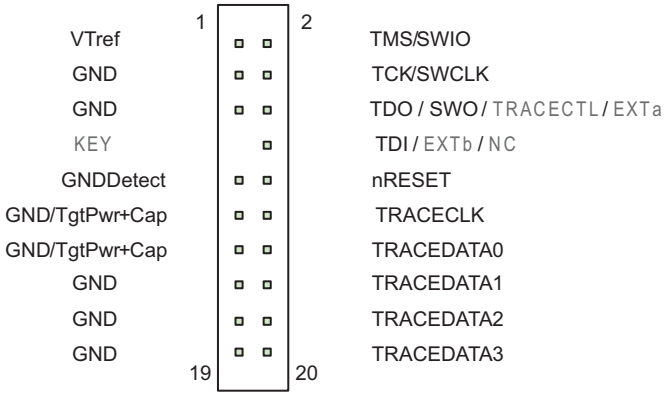
对于一些不具备指令跟踪(ETM)的Cortex-M3微控制器，仍然可实现基本的跟踪功能。Cortex-M3处理器支持单线浏览器(Single-wire Viewer)输出，允许通过单个引脚输出少量信息。您可以利用ULINK2等低成本的调试硬件来收集跟踪信息。利用单线浏览器提供的跟踪信息，您可以实现数据跟踪、事件跟踪（如中断）、PC采样和仪器跟踪，其中仪器跟踪是一种新的调试功能，允许调试器收集软件生成的消息（例如，可以实现printf，而不会对软件执行速度造成太大影响）。

调试连接

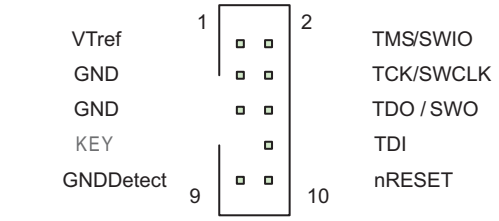
在8051微控制器中，调试连接通常非常少，而ARM微控制器具有更多的调试连接。

- JTAG/串行线
- 串行线输出（通常与TDO共享，仅限Cortex-M3/M4）
- 跟踪输出（仅限Cortex-M3/M4，通常在使用ETM跟踪时使用。它包含5个信号）

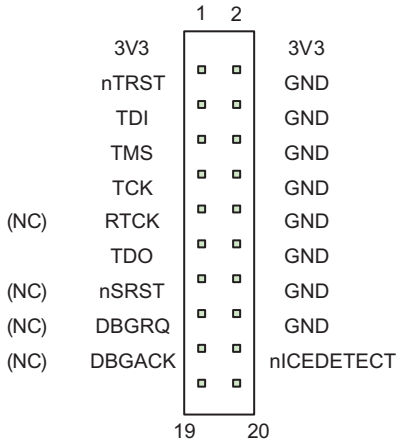
请注意，ARM规定了调试连接的物理连接器标准。进行PCB设计时，使用标准连接器会简单很多。对于新设计，建议使用新的Cortex调试和ETM连接器（0.05” 20引脚头 – Samtec FTSH-120）。有关信号协议的详细信息，请参阅“ETM架构规范”和“CoreSight 架构规范”。



对于没有ETM的设备，您可以使用更小的0.05i±10引脚连接器。



或者，您还可以使用旧的ARM JTAG/串行线20路IDC连接器。



此外，还有一个旧的基于38位Mictor连接器的跟踪连接。对于新设计，不建议采用。跟踪信号引脚分配情况如下页左图所示。

ARM网站上有一篇有关调试连接器的技术文章，网址为：http://infocenter.arm.com/help/topic/com.arm.doc.faqs/attached/13634/cortex_debug_connectors.pdf

Signal	Pin	Pin	Signal
NC	1	2	NC
NC	3	4	NC
GND	5	6	TRACECLK
Pulldown	7	8	Pulldown
Pulldown	9	10	Pulldown
Pulldown	11	12	Pullup (VRef)
Pulldown	13	14	Pullup (VSupply)
Pulldown	15	16	0
Pulldown	17	18	0
Pulldown	19	20	0
Pulldown	21	22	0
0	23	24	TRACEDATA[3]
0	25	26	TRACEDATA[2]
0	27	28	TRACEDATA[1]
0	29	30	0
0	31	32	0
0	33	34	1
0	35	36	0 (TRACECTL)
0	37	38	TRACEDATA[0]

结论

由于架构上的差异，在从8051迁移到ARM Cortex微控制器时，需要对应用程序代码进行一些修改。修改的原因主要在于数据类型大小上的差异，以及8051的一些C语言扩展并不适用于 ARM Cortex微控制器。此外，ARM微控制器上具有一些新的异常类型和架构特性，需要修改软件代码才能使用这些新功能。

结束