第15章

# 调试系统架构

* 调试特性概览
* CoreSight技术概览
* 调试模式
* 调试事件
* Cortex-M3中的断点
* 调试时访问寄存器
* 内核的其它调试特性

## 15.1 调试特性概览

一直以来，单片机的调试不是很突出的主题，很多山寨点的程序在开发中，甚至都没有调试的概念，而只是把生成的映像直接烧入片子，再根据错误症状来判断问题，然后修改程序重新烧，周而复始，直到问题解决或放弃为止。能够格算得上调试的活动，至少也是设置断点、观察寄存器和内存、监视变量等。使用仿真头和JTAG（如AVR），可以方便地实现这些基本的调试要求。在开发比较大的应用程序时，强劲的调试手段是非常重要的。当bug复杂到无法分析时，只能用调试来追踪它。如果没有调试手段，简直就束手无策。

正因为此，在CM3中，调试机能突然在一夜之间，就从丑小鸭变成了白天鹅，得到了登峰造极般的，令人非常惊艳的强化。CM3提供了多种多样的调试模型和调试组件，很多想到的和没想到的调试方式这里都有，让人惊叹“原来调试还可以做到这种程度”。为了方便进一步学习，我们把CM3丰满的调试功能分为两类，每类中都有更具体的调试项目，如下所列：

#### 侵入式调试（这也是基本的调试机能）

* 1. 停机以及单步执行程序
  2. 硬件断点
  3. 断点指令（BKPT）
  4. 数据观察点，作用于单一地址、一个范围的地址，以及数据的值。
  5. 访问寄存器的值（既包括读，也包括写）
  6. 调试监视器异常
  7. 基于ROM的调试（闪存地址重载(flash patching) ）

#### 非侵入式调试（大多数人更少接触到的，高级的调试机能）

* 1. 在内核运行的时候访问存储器
  2. 指令跟踪，需要通过可选的嵌入式跟踪宏单元（ETM）
  3. 数据跟踪
  4. 软件跟踪（通过ITM（指令跟踪单元））
  5. 性能速写（profiling）（通过数据观察点以及跟踪模块）

可见，我们以前最常用的调试都属于侵入式调试。所谓“侵入式”，主要是强调这种调试会打破程序的全速运行。非侵入式调试则是锦上添花的一类，当调试大型软件和多任务环境下的软件系统时，非侵入式调试有不可替代之强大功效。

在CM3处理器的内部，包含了一系列的调试组件。CM3的调试系统基于ARM亲手打造且吐血推荐的“CoreSight（内核景象）”调试架构。该架构是一个专业设计的体系，它允许使用标准的方案来访问调试组件，收集跟踪信息，以及检测调试系统的配置。

## 15.2 CoreSight技术概览

CoreSight调试架构的定义简直包罗万象，包括调试接口协议、调试总线协议、对调试组件的控制、安全特性、跟踪接口等。在《CoreSight Technology System Design Guide(Ref3)》中，对CoreSight有详细的讲述，此外，在Cortex-M3 TRM中也开出了若干章，专门叙述CM3中调试组件的设计。但是这些内容通常只是给设计调试软件的人看的，我们软硬件开发者不要陷得太深。不过，懂一点调试系统的组成结构和基本工作原理，还是很有助于让我们善加利用这强大无比的调试系统，大幅加速程序的开发的。

### 15.2.1 处理器的调试接口

CM3的调试系统已经与ARM7/ARM9的大相径庭了，基于新好CoreSight架构，它从头到脚都是新的。以前的ARM处理器都提供JTAG接口，通过它来控制对寄存器和存储器的访问。在CM3中全变了——对处理器上总线逻辑的控制使用另外的总线接口，即通过所谓的“调试访问端口(DAP)”。DAP与AMBA中的APB很相似。在CM3中，把JTAG或串行线协议都转换成DAP总线接口协议，再控制DAP来执行调试动作。

CM3内部的调试总线DAP是APB的近亲，所以很容易在它上面挂上很多调试组件，从而使得调试系统可大可小，伸缩性极强。此外，把调试接口和调试硬件分开，也是颇具匠心的：芯片中实际使用的调试接口类型变得透明化。从而不管使用了什么样的调试接口，相同的调试任务都可以按照同一个方式执行。

在CM3处理器内核中，实际的调试功能由NVIC和若干调试组件来协作完成。调试组件包括FPB, DWT, ITM等。NVIC中有一些寄存器，用于控制内核的调试动作，如停机、单步；其它的一些功能块则控制观察点、断点，以及调试消息的输出等。

就目前来看，CM3支持两种调试主机接口（debug host interface ）:第一个是广为使用的JTAG接口，另一个则是新的“串行线(Serial Wire, SW)调试接口”。新出的SW接口对信号线的需求只有两条。ARM公司还提供了若干种调试主机接口模块（称为“调试接口”（DP））。DP充当处理器与调试器的中介：它的一端连接到调试器上，另一端则连接到CM3的DAP接口上。

选择串行线的理由

CM3主要针对低成本的单片机市场。单片机往往没有很富裕的管脚资源。而JTAG协议需要使用4根脚，而SW则只需要两根。

### 15.2.2 DP模块，AP模块和DAP

从外部调试器到CM3调试接口的连接，需要多级互联才能完成，如图15.1所示。

第一步，是通过DP接口模块（通常是SWJ-DP或SW-DP），先把外部信号转换成一个通用的32位调试总线信号（图表中的DAP总线）。SWJ-DP支持SW与JTAG两种协议，而SW-DP则只支持SW。另外，在CoreSight产品中还可以使用一种JTAG-DP，它只支持JTAG协议。DAP总线上的地址是32位的，其中高8位用于选择访问哪一个设备，由此可见，最多可以在DAP总线上面挂256个设备。在CM3处理器的内部，只用掉了一个设备的地址，还剩下的255个都可以用于连接访问端口（AP）到DAP总线上。

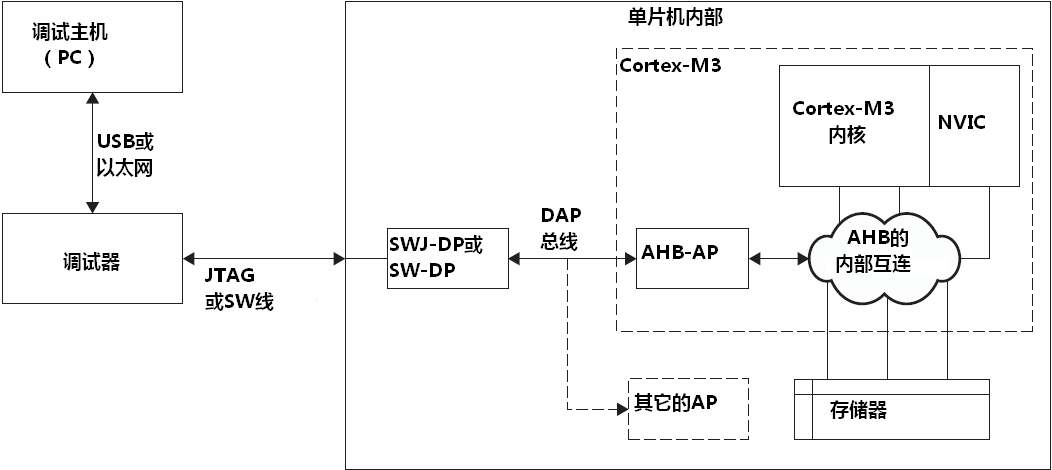


图15.1 高度主机到Cortex-M3的连接

在把数据从DAP接口传递给CM3处理器后，下一步就连接到了一个称为“AHB-AP”的AP设备上，它相当于一个总线桥，用于把DAP总线的命令转换为AHB总线上的数据传送，再插入到CM3内部的总线网络中。这么一来，CM3的整个寻址空间就都在覆盖范围之内了，连NVIC中的调试控制寄存器组也包括在内。在CoreSight系列产品中，AP设备可以有好几种类型，包括APB-AP和JTAG-AP。APB-AP顾名思义，是用于产生APB总线数据传送动作的，而JTAG-AP则用于控制传统的、基于JTAG的测试接口，例如ARM7上的调试接口。

### 15.2.3 跟踪接口

CoreSight架构的另一个部分用于跟踪。在CM3中有3种跟踪源：

* 1. 指令跟踪：由ETM（嵌入式跟踪宏单元）产生
  2. 数据跟踪：由DWT产生
  3. 调试消息：由ITM产生，提供形如printf的消息输入，送到调试器的GUI中

在跟踪过程中，由先把跟踪源产生的数据裹成数据包，然后把数据包送到“高级跟踪总线（ATB）”上进行传送。在CoreSight的架构中，如果某SoC含有多个跟踪源（例如，多核系统），则需要一种硬件水平的ATB归并器（merger），把各ATB数据流归并成一条（在CoreSight架构中，这种硬件被名为ATB funnel）。归并后的数据流都送往TPIU（跟踪端口接口单元），TPIU再把数据导出到片外的跟踪硬件设备。在数据送到了调试主机（PC）后，再由PC端的调试软件还原为先前的多条数据流。

尽管在CM3中拥有多个跟踪源，但CM3内建了一个归并硬件，因此不需要再添加ATB funnel模块了。跟踪输出接口可以直接连接到专为CM3设计的TPIU上，然后就可以供PC控制的外部硬件捕捉仪来跟踪数据。

### 15.2.4 CoreSight的性质

基于CoreSight的调试设计有很多优势：

* + 即使在处理器运行时，也可以查看存储器和外设的寄存器的内容
  + 使用单一调试器，就可以控制多核系统的调试接口。例如，如果使用JTAG，则只需要一个TAP控制器，不管芯片中有几个处理机都一样。
  + 内部的调试接口是基于单总线的方式设计的，因此非常有弹性，也简化了为芯片的其它部分设计附加的测试逻辑。
  + 它使得多条“跟踪数据流”可以由单一的“跟踪捕获设备”来收集，送到PC机上之后再还原出先前的各条数据流。

CM3中的调试系统是基于CoreSight的，但是又有一些“变异”：

* + CM3的跟踪组件是重新设计的，有些在CM3中的ATB接口是8位的，而纯种的CoreSight的都是32位的。
  + CM3的调试系统没有实现TrustZone——ARM提供的一种技术，用于在嵌入式产品中提供安全特性。
  + 调试组件所需的空间挤到了系统的存储器映射中。而在标准的CoreSight系统中，是为调试总线另开了一个地址空间的。例如，在CoreSight系统中，系统连接的概念图如图15.2所示：

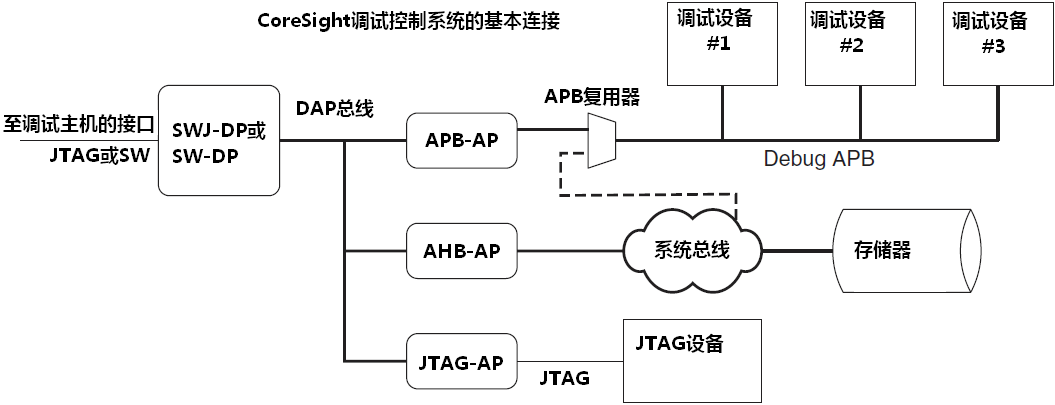


图15.2 CoreSight系统设计概念图

而在CM3中，调试设备共享同一个同一个存储器映射，如图15.3所示

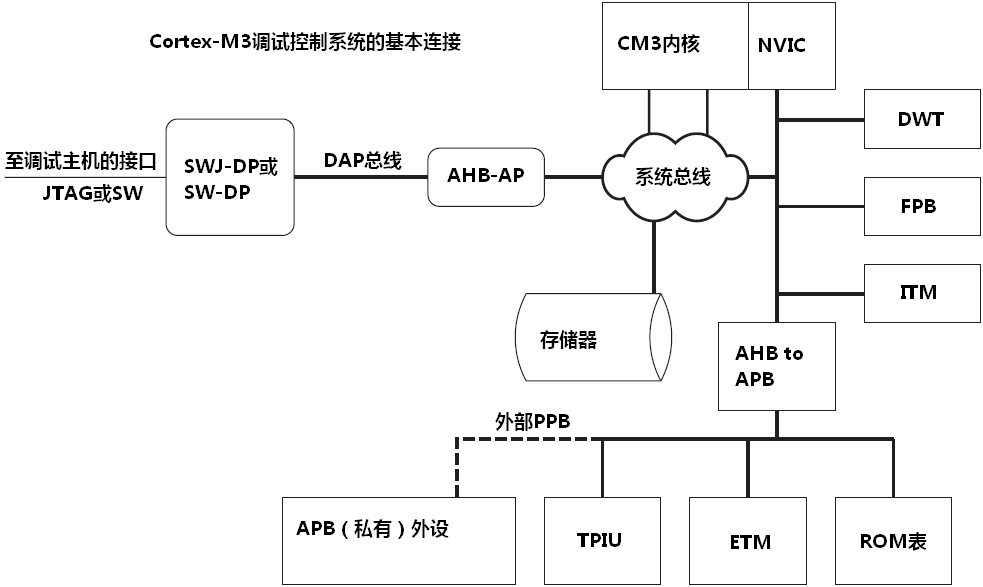


图15.3 Cortex-M3的调试系统

尽管CM3的调试组件在实现上与标准CoreSight系统的有些出入，但是通信接口与协议是与CoreSight架构兼容的，并且可以直接挂接到CoreSight系统上，标准CoreSight的调试组件也可以挂接到CM3上。例如，（标准）CoreSight调试组件，诸如TPIU，调试端口以及跟踪基础设施等，可以供CM3使用，并且以此来把调试能力延伸到多核调试系统中。

关于CoreSight架构的更多内容，请参阅《CoreSight Technology System Design Guide(Ref3)》。

## 15.3 调试模式

在CM3中的调试操作模式分为两种。第一种称为“halt”（停机模式），在进入此模式时，处理器完全停止程序的执行。第二种则称为“debug monitor exception”（调试监视器模式），此时处理器执行相应的调试监视器异常服务例程，由它来执行调试任务，并且依然允许更高优先级的异常抢占它。调试监视器的异常号为12，优先级可编程。除了调试事件可以触发异常外，手工设置其悬起位也可以触发本异常。

1. 停机模式
   * 指令执行被停止
   * SysTick定时器停止
   * 支持单步操作
   * 中断可以在这期间悬起，并且可以在单步执行时响应。也可以掩蔽它们，使得单步时不受干扰
2. 调试监视器模式
   * 处理器执行调试监视器异常的服务例程（异常号：12）
   * SysTick定时器继续运行
   * 新来的中断按普通执行时的原则来抢占
   * 执行单步操作
   * 存储器的内容（如堆栈内存）会在调试监视器的响应始末得到更新，因为有自动入栈和出栈的动作

之所以加入调试监视器模式，是考虑到了在某些电子系统运行的过程中，是不可以停机的。例如，对于汽车引擎控制器以及电机控制器，就必须在处理调试动作的同时让处理器继续运行下去，这样才能保证被测试的设备不会意外损坏（例如，不需要在调试过程中让电机停转——译者注）。有了调试监视器，就可以停止并调试线程级的应用程序，也可以调试低优先级的中断服务例程。在这同时，高优先级的中断和异常能够响应。

如果要进入停机模式，需要把NVIC调试停机控制及状态寄存器（DHCSR）的C\_DEBUGEN位置位。这个位只能由调试器来设置，没有调试器是不能把CM3停机的。在C\_DEBUGEN置位后，就可以设置DHCSR.C\_HALT位来喊停处理器。此C\_HALT位可以由软件置位。

DHCSR的位段定义比较特殊：读时是一种定义，写时又是另外一种定义。对于写操作，必须先往[31:16]中写入一个“访问钥匙”值。而对于读操作，则无此钥匙，并且读回来的高半字包含了状态位，如表15.1所示。

表15.1 调试停机控制及状态寄存器DHCSR （地址：0xE０00\_EDF0）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 31:15 | KEY | W | - | 调试钥匙。必须在任何写操作中把该位段写入A05F，否则忽略写操作 |
| 25 | S\_RESET\_ST | R | - | 内核已经或即将复位，读后清零 |
| 24 | S\_RETIRE\_ST | R | - | 在上次读取以后指令已执行完成，读后清零 |
| 19 | S\_LOCKUP | R | - | 1=内核进入了锁定状态 |
| 18 | S\_SLEEP | R | - | 1=内核睡眠中 |
| 17 | S\_HALT | R | - | 1=内核已停机 |
| 16 | S\_REGRDY | R | - | 1=寄存器的访问已经完成 |
| 15:6 | 保留 | - | - |  |
| 5 | C\_SNAPSTALL | RW | 0\* | 打断一个stalled存储器访问 |
| 4 | 保留 | - | - |  |
| 3 | C\_MASKINTS | RW | 0\* | 调试期间关中断，只有在停机后方可设置 |
| 2 | C\_STEP | RW | 0\* | 让处理器单步执行，在C\_DEBUGEN=1时有效 |
| 1 | C\_HALT | RW | 0\* | 喊停处理器，在C\_DEBUGEN=1时有效 |
| 0 | C\_DEBUGEN | RW | 0\* | 使能停机模式的调试 |

\*：DHCSR中的控制位是在上电复位时得到复位的。系统复位（例如，往NVIC应用程序中断及复位寄存器中写命令）不会影响到它们

在正常情况下，只有调试器会操作DHCSR，应用程序不要乱动它，以免使调试工具出现问题。

当使用调试监视器模式时，由另一个NVIC中的寄存器来负责控制调试活动，它是NVIC调试异常及监视器控制寄存器（DEMCR），其定义如表15.2所示。

表15.2 调试及监视器控制寄存器DEMCR （地址：0xE０00\_EDFC）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 24 | TRCENA | RW | 0\* | 跟踪系统使能位。在使用DWT,ETM,ITM和TPIU前，必须先设置此位 |
| 23:20 | 保留 |  |  |  |
| 19 | MON\_REQ | RW | 0 | 1=调试监视器异常不是由硬件调试事件触发，而是由软件手工悬起的 |
| 18 | MON\_STEP | RW | 0 | 让处理器单步执行，在MON\_EN=1时有效 |
| 17 | MON\_PEND | RW | 0 | 悬起监视器异常请求，内核将在优先级允许时响应 |
| 16 | MON\_EN | RW | 0 | 使能调试监视器异常 |
| 15:11 | 保留 |  |  |  |
| 10 | VC\_HARDERR | RW | 0\* | 发生硬fault时停机调试 |
| 9 | VC\_INTERR | RW | 0\* | 指令/异常服务错误时停机调试 |
| 8 | VC\_BUSERR | RW | 0\* | 发生总线fault时停机调试 |
| 7 | VC\_STATERR | RW | 0\* | 发生用法fault时停机调试 |
| 6 | VC\_CHKERR | RW | 0\* | 发生用法fault使能的检查错误时停机调试（如未对齐，除数为零） |
| 5 | VC\_NOCPERR | RW | 0\* | 发生用法fault之无处理器错误时停机调试 |
| 4 | VC\_MMERR | RW | 0\* | 发生存储器管理fault时停机调试 |
| 3:1 | 保留 |  |  |  |
| 0 | VC\_CORERESET | RW | 0\* | 发生内核复位时停机调试 |

\*：DEMCR中的控制位是在上电复位时得到复位的。系统复位（例如，往NVIC应用程序中断及复位寄存器中写命令）不会影响到它们

该寄存器不仅包含了调试监视器的控制位，还包含了跟踪系统的使能位（TRCENA）以及若干向量抓捕（Vector Catch, VC）控制位。VC功能只有在停机模式下才能使用。如果某个异常（或者内核复位）发生了，并且对应的VC位置位，则将自行产生一个停机请求，并且在执行完当前指令后立即把处理器喊停。

虽然TRCENA和VC控制相关的位只有上电时才复位，但是其它用于控制监视器模式的位，则也会因系统复位而复位。

## 15.4 调试事件

CM3可以由很多种理由进入调试模式（both停机模式和调试监视器模式）。对于停机模式，满足图15.4所示的条件可以喊停处理器。但即使是停机后，也可由上电复位和系统复位来复位处理器。

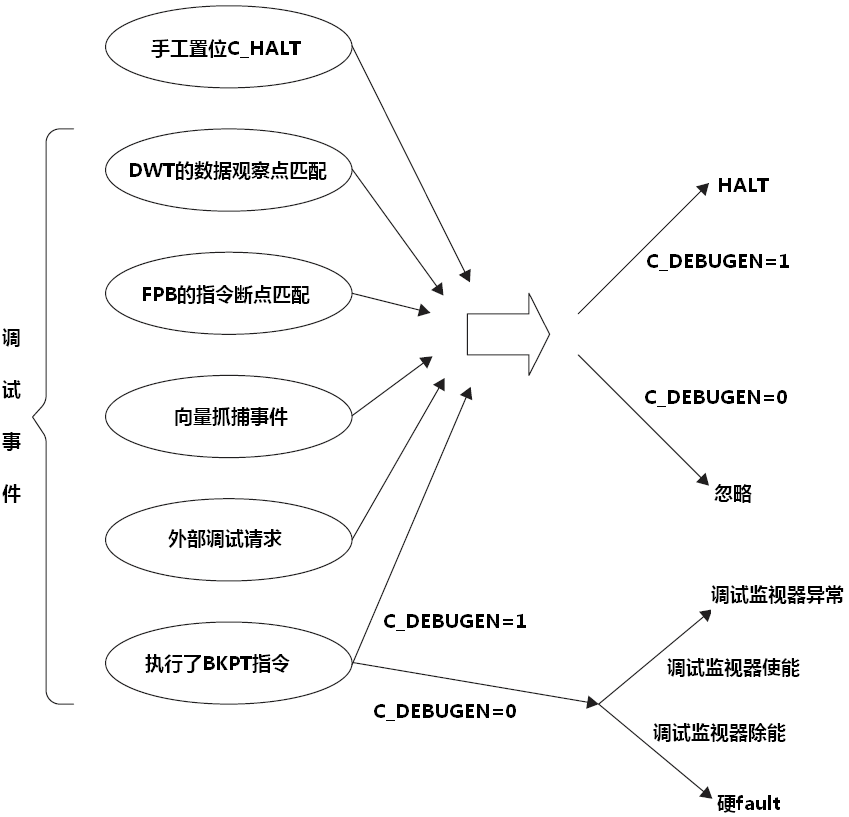


图15.4 停机模式下对调试事件的响应

图中，外部调试请求信号是通过CM3上的一个称为“EDBGREQ”的信号线传来的，该信号线的实际连接方式取决于单片机/SoC的设计。在有些场合下可以把该信号硬线连至低电平，从而使外部调试请求永远无法送达；也可以把它连接到附加的调试组件上（芯片厂商可以添加额外的调试组件）；或者在多核系统中，可以用来连接其它处理机的调试事件。

在调试活动完成后，通过清除C\_HALT位，可以让程序继续执行。

类似地，在调试监视器模式下，也可以由一系列的调试事件来进入调试模式，如图15.5所示。

从图中可见，在调试监视器模式下，与在停机模式下的动作方式还是有一点区别的。这是因为调试监视器异常仅仅是异常的一种，它可以影响当前的优先级，但是不能使处理器停下来。

在调试活动完成后，通过该异常的返回，即可回到正常的程序执行中。

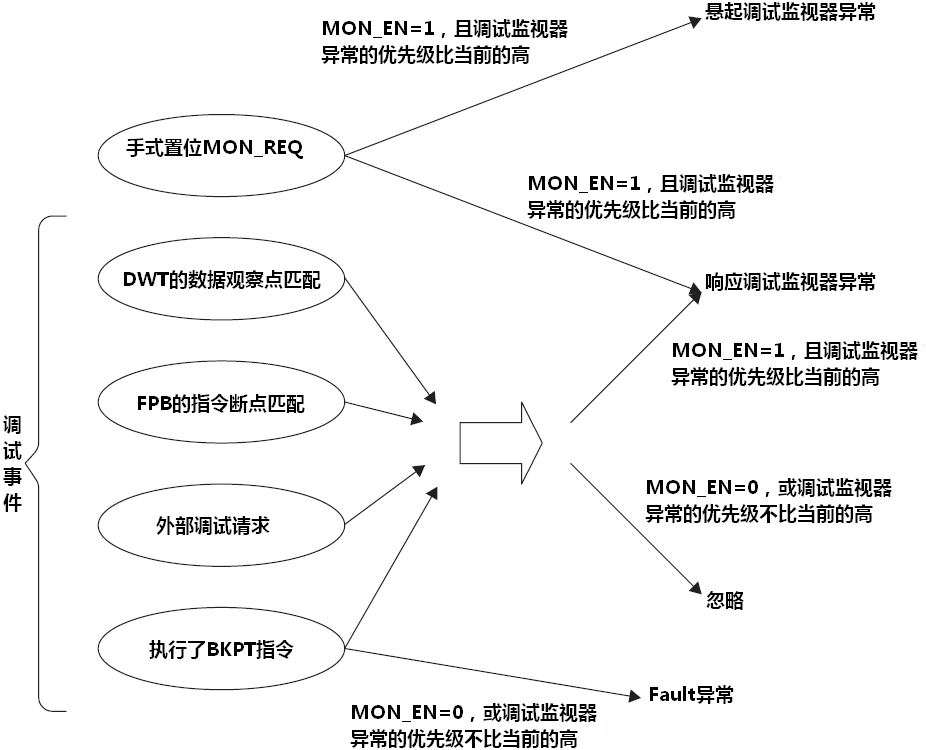


图15.5 调试监视器模式下对调试事件的响应

## 15.5 Cortex-M3中的断点

在大多数单片机中，用得最多的可能就是断点了。在CM3中，有两种断点机制：

* + 断点指令
  + 基于FPB地址比较器的断点

断点指令的格式为BKPT #im8，它是一个16位的Thumb指令，编码为0xBExx——其低8位就是指令中#im8的值。当该指令执行时，会产生一个调试事件。当C\_DBGEN置位时可以用于喊停处理器内核；或者当调试监视器使能时，触发调试监视器异常。对于后者，因为调试监视器异常也是一种优先级可编程的普通异常，所以也可以因为其优先级不够高而不能立即响应。可见，因为NMI和硬fault的优先级总是比它的高，所以不能在它们的服务例程中使用BKPT指令来启动调试——只有在它们返回时才能响应调试监视器异常。

使用BKPT时另一个要注意的是，当调试监视器异常返回后，它返回到的是BKPT指令的地址，而不是返回BKPT后面一条指令的地址。这与常规的异常返回是不同的，原因在于，在正常情况下使用BKPT指令时，BKPT用于取代一条正常的指令，并且当命中了该断点而执行了调试动作后，把该BKPT指令所占用的内存恢复为先前被BKPT取代的指令，并且让该指令是下一条即将执行的指令，而其它的部分不受影响(这其实也是软件断点的实现方式)。

如果在BKPT指令执行时却发现C\_DEBUGEN和MON\_EN都为0，则会因为无法进入调试而上访成硬fault，并且把硬fault状态寄存器(HFSR)的DEBUGEVT位给置1，同时在调试fault状态寄存器（DFSR）中的BKPT位也置1。

如果程序存储器的值不能更改，则可以通过编程FPB来产生硬件断点。但是，只支持6个指令地址和两个文字地址。下一章将展开叙述FPB。

使用BKPT指令取代正常指令，以及对FPB的编程，通常都是在我们设置断点时，由调试器负责做的事。

## 15.6 调试时访问寄存器

在NVIC中，还有两个寄存器与与调试功能有关。它们分别是：调试内核寄存器选择者寄存器（DCRSR），以及调试内核寄存器数据寄存器（DCRDR），如表15.3和表15.4所示。调试器需要通过这两个寄存器来访问处理器的寄存器，并且只有在处理器停机时，才能使用这里的寄存器传送功能。

表15.3 调试内核寄存器选择者寄存器DCRSR（地址：0xE０00\_EDF4）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 16 | REGWnR | W | - | 1=写寄存器  0=读寄存器 |
| 15:5 | 保留 | - | - | - |
| 4:0 | REGSEL | W | - | 00000= R0  00001=R1  …  01111=R15  10000=xPSR  10001=MSP  10010=PSP  10100=特殊功能寄存器组  [31:24]: CONTROL  [23:16]: FAULTMASK  [15:8]: BASEPRI  [7:0]: PRIMASK |

表15.4 调试内核寄存器数据寄存器DCRDR（地址：0xE０00\_EDF8）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 31:0 | DATA | R/W | - | 读回来的寄存器的值，或欲写入寄存器的值，寄存器由DCRSR选择 |

欲使用这两个寄存器来读取内核的寄存器的内容，则必须按如下的顺序做：

1. 确定处理器已停机
2. 往DCRSR写数据，其中位16要为0，表示这是要读数据
3. 查询，直到DHCSR.S\_REGRDY=1
4. 读取DCRDR以获取寄存器的内容

寄存器写操作的顺序与上面的类似：

1. 确定处理器已停机
2. 往DCRDR中写数据
3. 往DCRSR写数据，其中位16要为1，表示这是要写数据
4. 查询，直到DHCSR.S\_REGRDY=1

使用DCRSR和DCRDR来访问寄存器，只适用于停机模式。如果选择了调试监视器模式，则对于自动入栈的寄存器，可以从堆栈中读写它们；对于其它寄存器，就可以直接在服务例程中访问。

如果有合适的函数库和调试器的支持，还可以使用DCRDR来做半主机（semihosting）。比如说，当应用程序执行了printf语句时，文字的输出可以通过一系列的putc()调用来完成。在实现putc()时，可以让它把输出的字符和状态写到DCRDR中，然后触发调试模式。接下来，调试器可以检测到内核停机状态，并且读取被输出的字符。然而，这种形式的半主机需要喊停内核。更正点的半主机是使用ITM，它则没有此限制。

## 15.7 内核的其它调试特性

在NVIC中，还有其它一些与调试有关的特性，它们包括：

* + 外部调试请求信号：NVIC提供了一个外部调试请求信号，通过它可以让CM3处理器由外部调试事件触发而进入调试模式。举一个外部调试事件的例子：在多核系统中，可以是其它处理机的调试状态，这对于调试多核系统的意义决非等闲。如果是单核的单片机，则基本上是把该信号拉低。
  + 调试fault状态寄存器：因为在CM3上有多种调试事件，故而设置了一个DFSR，以资调试器来判断是发生了哪种调试事件。
  + 复位控制：在调试期间，可以使用VECTRESET控制位来重启处理器内核（位于NVIC应用程序中断及复位控制寄存器中（地址：0xE000\_ED0C））。通过使用这种方式，可以不让处理器的复位波及到调试系统。
  + 中断掩蔽：在单步时这个功能是非常体贴的。因为在单步时，往往是为了集中精力分析某段代码的逻辑，此时不希望受到任何骚扰，哪怕是响应中断也是很讨人厌的事。通过置位C\_MASKINTS位（在调试停机控制及状态寄存器中，（地址：0xE000\_EDF0）），就可以在单步期间掩蔽中断。
  + 终止Stalled总线传送：如果一个总线传送被stall了一个很长的时间，就可以强制终结它。在调试停机及状态寄存器中有一个C\_SNAPSTALL位，把它置位即可。但是这个功能只有在停机模式下才能由调试器使用。