Technical Article

C2000 程序跑飞的原因及解决方法

Strong Zhang

程序跑飞是软件开发常见的问题，无论是在产品开发过程中，还是在批量量产后都可能会出现，本文探讨如何快速查找导致程序跑飞的代码，分析程序跑飞几种原因以及提出相关的解决方法。

如何查找非法中断的返回地址？

程序跑飞通常代码会进入非法中断，首先需要定位到运行哪行代码会导致程序跑飞，通过 RPC 寄存器的值可以查看非法中断的返回地址，即在代码跑飞之后，在 CCS 寄存器窗口查看 Core Register 的 RPC 的值。

2.2.8 Return Program Counter (RPC)

When a call operation is performed using the LCR instruction, the return address is saved in the RPC register and the old value in the RPC is saved on the stack (in two 16-bit operations). When a return operation is performed using the LRETR instruction, the return address is read from the RPC register and the value on the stack is written into the RPC register (in two 16-bit operations). Other call instructions do not use the RPC register. For more information, see the instructions in Chapter 6.

需要注意 RPC 只保存了上次由 LCR 指令跳转过来的返回地址，对于其他跳转指令如"LB #22bit address"等，RPC 则提供不了有效信息。

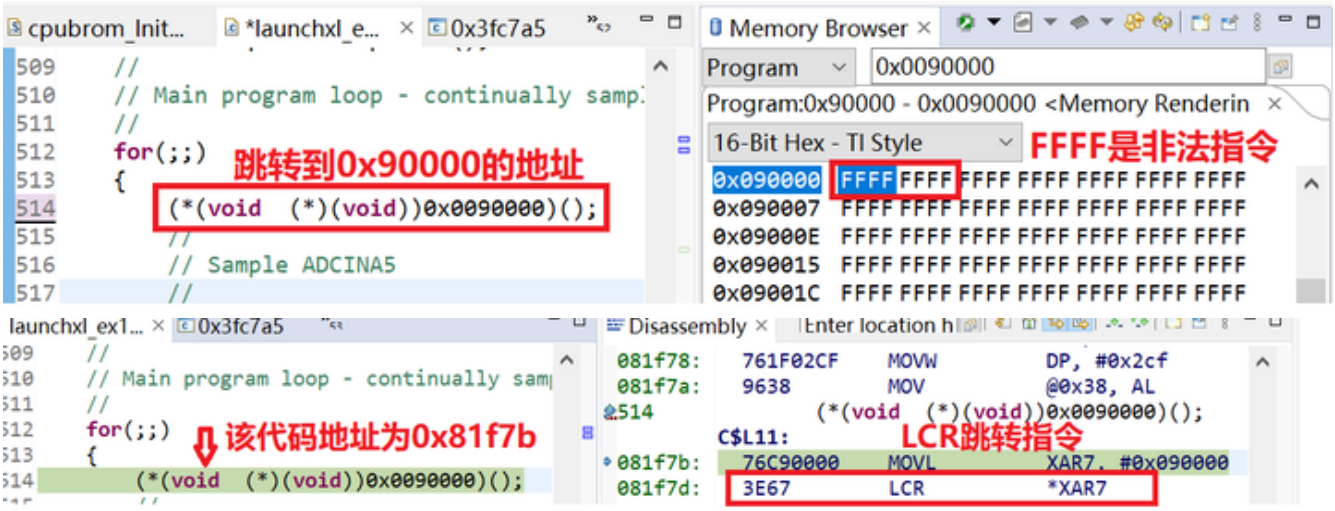
如果是 LB 指令，则需要从堆栈里查找，如 TMS320C28x CPU and Instruction Set (Rev. F)手册所说，堆栈指针 SP 往回偏移 7 和 8 个字节的地址保存了上次跳到非法中断的返回地址。

Table 3-5. Register Pairs Saved and SP Positions for Context Saves

Save Operation[1]	Register Pairs	Bit 0 of Storage Address SP Starts at Odd Address	SP Starts at Even Address
1st	ST0	0	0 ← SP position before step 5
	T	1	1
2nd	AL	0	0
	AH	1	1
3rd	PL [2]	0	0
	PH	1	1
4th	AR0	0	0
	AR1	1	1
5th	ST1	0	0
	DP	1	1
6th	IER	0	0
	DBGSTAT [3]	1	1
7th	Return address (low half)	0	0
	Return address (high half)	1	1
		0 SP position after save	0
		1	1 SP position after save

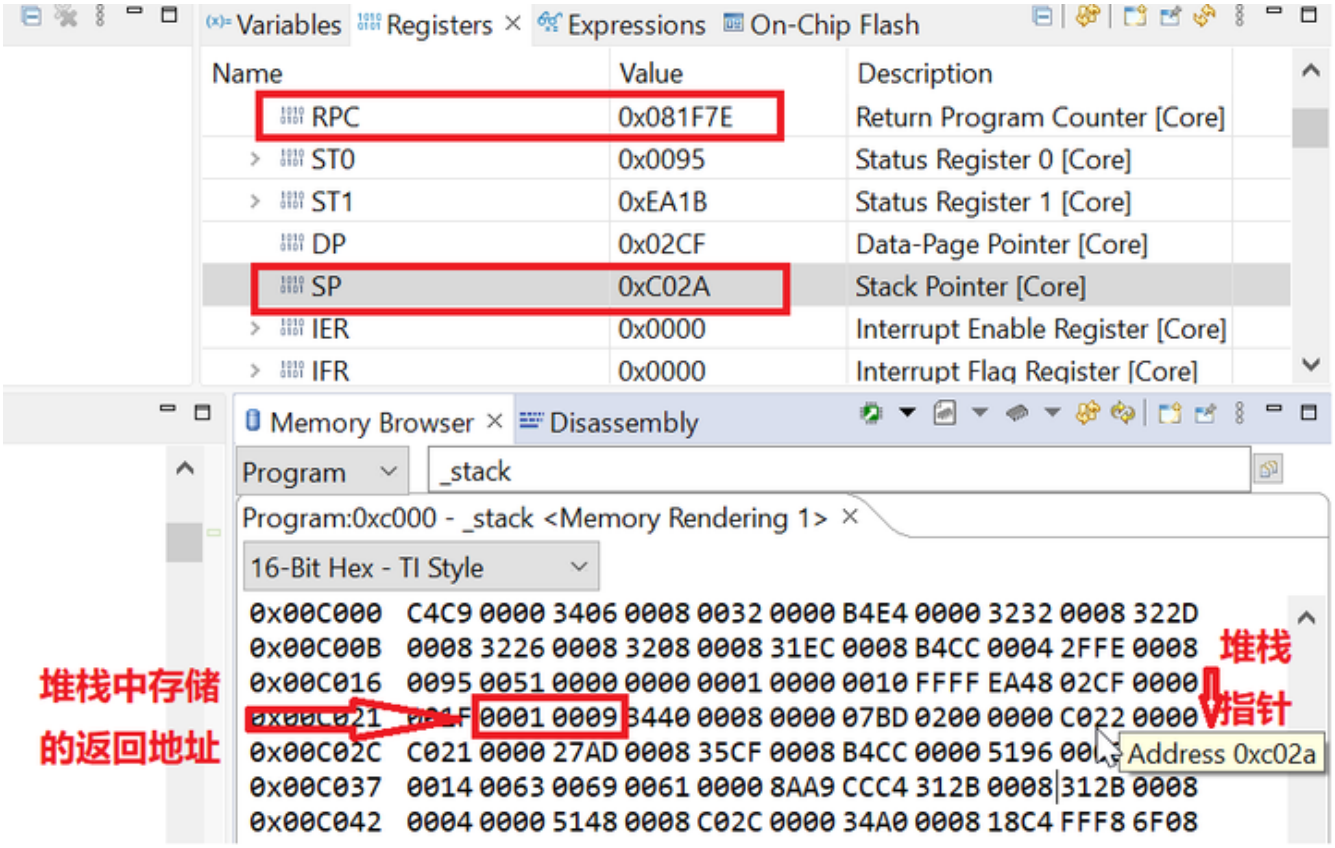
堆栈保存的返回地址，位置在堆栈指针SP-7和SP-8的地址

例如下图的 C 代码 “(*(void (*)(void))0x0090000)()” 的汇编代码包含了 LCR 指令，当该代码从地址 0x81f7b 执行 LCR 跳转指令时，会跳到 0x90000 继续运行，而 0x90000 是只有 0xFF 内容的无效指令的 Flash 地址，因此必然会跳到非法中断。



这时可以从 RPC 寄存器得到地址为 0x81F7E，也就是指向"(*(void (*)(void))0x0090000)()"代码的地址。

而堆栈 0xC022~0xC023（当前堆栈指针 0xC02A 偏移 7~8 个值）保存了 0x0090000 的地址信息，也就是执行地址为 0x0090000 的非法指令 FFFF 导致了异常。



因此，RPC 寄存器只保存了 LCR 跳转指令的返回地址，而堆栈则是始终保存了上一个跳到非法中断的返回地址，但如果堆栈被破坏了或者返回地址存的指令本身是非法的，则 RPC 指针有时候可以提供更为有效的信息，在实际调试中，两者可以结合起来，帮助找到程序跑飞的原因。

不过有时候即使找到了具体哪行代码导致程序跑飞，也未必知道根本的原因，根据 C28x Interrupt FAQ (ti.com)提供的原因，有下面几种情况下会导致非法中断。

Q: What causes an illegal (ITRAP) interrupt?

- An invalid instruction is decoded (this includes invalid addressing modes).
- The opcode value 0x0000 is decoded. This opcode corresponds to the ITRAP0 instruction.
- The opcode value 0xFFFF is decoded. This opcode corresponds to the ITRAP1 instruction.
- A 32-bit operation attempts to use the @SP register addressing mode.
- Address mode setting AMODE=1 and PAGE0=1which is an illegal combination.

下面从方便调试的角度对可能导致非法中断的原因进行分类，以提供进一步解决该类问题的思路。

程序跑飞的原因：

1. 堆栈溢出导致程序跑飞

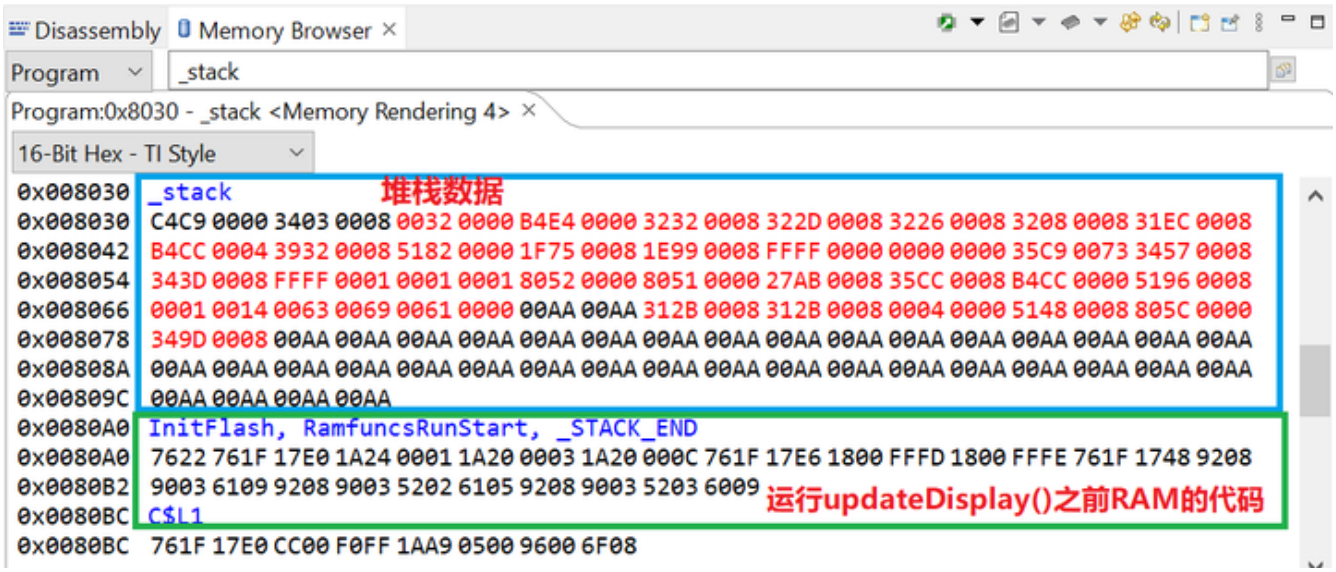
cmd 文件里面会定义分配给堆栈的 RAM 空间大小，但实际堆栈所占的空间是动态的，只有程序跑起来后才知道堆栈的实际占用的空间有多大，因此如果 cmd 文件里面定义堆栈太小，编译器是不会报错提醒的，这个时候运行代码，堆栈数据会溢出，侵占了其他 RAM 的空间，刷掉 RAM 程序或者全局变量的内容。

```
   Stack_Data          : origin = 0x008030, length = 0x000070
            RAM_Code             : origin = 0x0080A0, length = 0x000600
            .stack               : > Stack_Data,     fill=0xAA
```

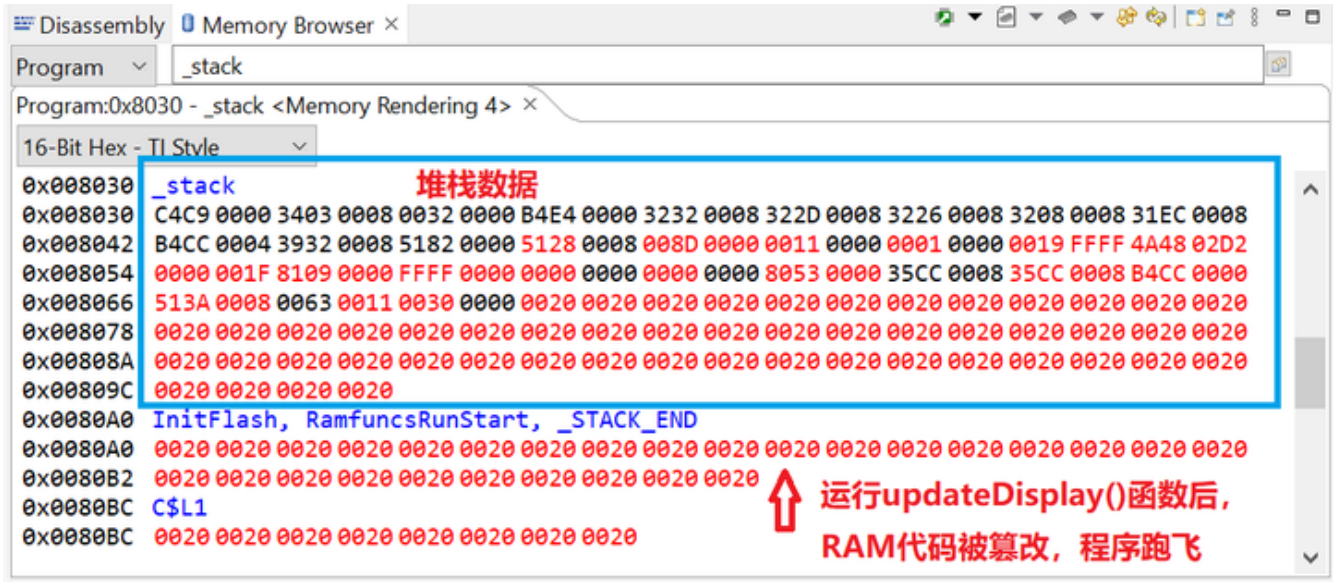
例如下面的函数 updateDisplay()定义了很多临时变量，运行的时候需要动态分配大量的内存空间。

```
502     for(;;)
503     {
504       //  (*JumpToApp)();
505         //
506         // Sample ADCINA5
507         //
508         currentSample = sampleADC();
509
510         // Update the serial terminal output
511         updateDisplay();
512
513         //
514         // If the sample is above midscale light one LED
515         //
516         if(currentSample > 2048)
517         {
518             GpioDataRegs.GPBSET.bit.GPIO34 = 1;
```

在运行 updateDisplay()之前，如下面 0x0080A0 地址所示存放的是 RAM 的代码，这是初始化之后就应该固定不变的。



但跑完这个函数后，堆栈空间数据溢出，RAM 代码被堆栈数据所刷新，导致程序跑飞。



解决方法：

通过在 cmd 文件中初始化 stack 堆栈空间，如初始化填充了 0xAA，在代码运行起来的时候，观察有多少 0xAA 的值被动态更新了，这样大致能够确定代码所需要的堆栈空间，一般设置需要留一定的裕量，如果不够，可以把下面分配给堆栈的空间 Stack_Data 设大一些。

```
   Stack_Data          : origin = 0x008030, length = 0x000070
           .stack           : > Stack_Data,     fill=0xAA
```

另外还需要注意的是堆栈的指针是 16 位的，因此存储堆栈区域的地址是不能超过 0xFFFF 的，否则也可能会出现程序跑飞的情况。

2. 在 RAM 运行的函数没有从 Flash 中拷贝成功

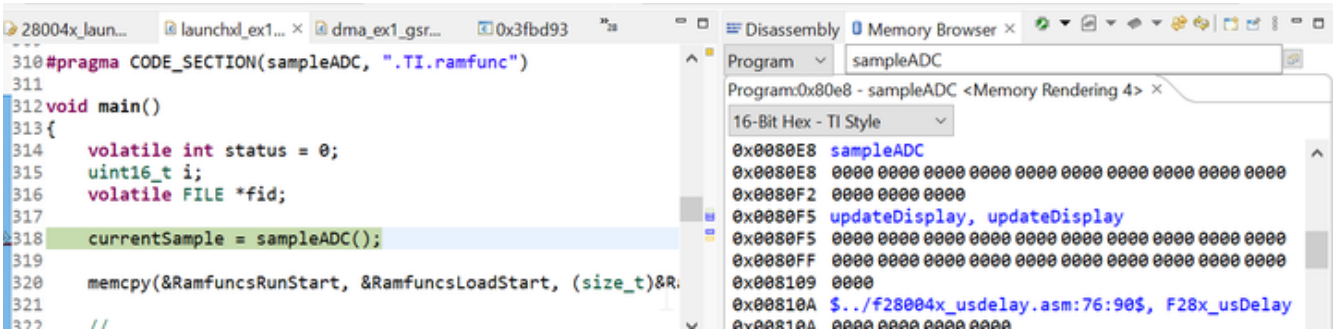
如果函数声明在 RAM 里面运行，但在应用程序中并没有用 memcpy 函数从 Flash 拷贝到 RAM，这时候程序运行就会跑飞。

例如下面 sampleADC()函数声明放在段.TI.ramfunc 里面，而.TI.ramfunc 在 cmd 文件中的定义是从 Flash 装载，然后在 RAM 里面运行，这个时候编译器会自动分配 RAM 空间给 sampleADC 函数，但不会自动把代码从 Flash 拷贝到 RAM，

```
#pragma CODE_SECTION(sampleADC, ".TI.ramfunc")
 //将函数存在段.TI.ramfunc 所在的 RAM 空间
```

```
89    .TI.ramfunc      : LOAD = FLASH_BANK0_SEC1,
90                           RUN = RAM_Code,
91                           LOAD_START(_RamfuncsLoadStart),
92                           LOAD_SIZE(_RamfuncsLoadSize),
93                           LOAD_END(_RamfuncsLoadEnd),
94                           RUN_START(_RamfuncsRunStart),
95                           RUN_SIZE(_RamfuncsRunSize),
96                           RUN_END(_RamfuncsRunEnd),
97                           PAGE = 0, ALIGN(4)
```

如下图所示函数 sampleADC()分配的起始 RAM 地址为 0x0080E8，但这部分 RAM 空间的值都是 0，即还没有从对应 Flash 程序中拷贝内容，运行该函数将会导致程序跑飞。



解决方法：

确保所有在 RAM 运行的函数调用之前，必须先用 memcpy 函数把代码从 Flash 拷贝到 RAM。

```
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
            DELAY_US(1000);
            InitFlash();
```

3. 代码所占 Flash 空间靠近无效内存边界

对于代码空间，需要注意存储的代码在 flash 或者 RAM 的空间不能太靠近边界，如 F28003x 的 Errata 所说，例如下面所说的 0x0AFFF0~0x0AFFFF 是不能用的，否则有可能出现异常。

Advisory	***Memory: Prefetching Beyond Valid Memory***
Revision Affected	0
Details	The C28x CPU prefetches instructions beyond those currently active in its pipeline. If the prefetch occurs past the end of valid memory, then the CPU may receive an invalid opcode.
Workaround	**M1, GS3 –** The prefetch queue is 8 x16 words in depth. Therefore, code should not come within 8 words of the end of valid memory. Prefetching across the boundary between two valid memory blocks is all right. Example 1: M1 ends at address 0x7FF and is not followed by another memory block. Code in M1 should be stored no farther than address 0x7F7. Addresses 0x7F8–0x7FF should not be used for code. Example 2: M0 ends at address 0x3FF and valid memory (M1) follows it. Code in M0 can be stored up to and including address 0x3FF. Code can also cross into M1, up to and including address 0x7F7. **Flash –** The prefetch queue is 16 x16 words in depth. Therefore, code should not come within 16 words of the end of valid memory; otherwise, it generates a Flash ECC uncorrectable error.

Table 3-2. Memories Impacted by Advisory

MEMORY TYPE	ADDRESSES IMPACTED
M1	0x0000 07F8–0x0000 07FF
GS3	0x0000 FFF8–0x0000 FFFF
Flash	0x000A FFF0–0x000A FFFF

解决方法：

在 cmd 配置中，不使用地址为 0x0AFFF0~0x0AFFFF 的 Flash 空间。

```
81    FLASH_BANK2_SEC13 : origin = 0x0AD000, length = 0x001000
82    FLASH_BANK2_SEC14 : origin = 0x0AE000, length = 0x001000
83    FLASH_BANK2_SEC15 : origin = 0x0AF000, length = 0x000FF0
84
85/   FLASH_BANK0_SEC15_RSVD     : origin = 0x0AFFF0, length = 0x000010 /* Reserve and do not use
86
```

不使用该flash空间

4. 非加密区程序访问加密区的数据

如果对代码使用了 DCSM/CSM 模块进行加密，需要注意的是非加密区的程序不能访问加密区的数据。

There are three types of accesses: data/program reads, JTAG access, and instruction fetches (calls, jumps, code executions, ISRs). Instruction fetches are never blocked. JTAG accesses are always blocked when a memory is secure. Data reads to a secure memory are always blocked unless the program is executing from a memory which belongs to the same zone. Data reads to unsecure memory are always allowed. Table 3-17 shows the levels of security.

另外以下几种情况容易被忽视，需要特别注意：

1. ROM 的代码一般是没有加密的，如果调用了 ROM 的代码，例如 Flash API，IQmath 库的函数，需要注意该函数访问的数据是不能加密；
2. 对于同一工程里面划分了加密区和非加密区的 Flash 或者 RAM，堆栈的 RAM 空间往往不能设置为加密，因为加密区和非加密区都会共用了一个堆栈空间。
3. 如果在 RAM 跑的程序空间为非加密区，则注意不能访问加密区的数据。

解决方法：

确保非加密的程序不能访问加密区数据，特别注意 ROM 和部分 RAM 空间是不加密的，调用了这里的函数不能访问加密区的数据。

总结：

通过 RPC 的值或者堆栈保存下来的返回地址，可以找到具体哪行代码导致程序跑飞，以帮助进一步分析原因。

而程序跑飞的原因有很多，比如堆栈大小不够导致溢出，运行了没有被成功初始化的 RAM 程序，代码跑到 Errata 所说的禁用内存区间，或者用了 DCSM 加密模块而有非加密程序访问了加密的数据等等。

当然还有其他因为软件 bug 导致的原因，如果 CCS 编译之后有 warning 的提示，一般都要尽可能消掉，否则会存在难以排查的潜在风险。

参考文献

1. C28x Interrupt FAQ (ti.com)
2. TMS320C28x CPU and Instruction Set (Rev. F)

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265