

## Quickly Trace HardFaultHardler

## 前言

这篇应用笔记描述了怎么使用CmBacktrace库，快速追踪和定位产生HardFault原因的方法。

注：本应用笔记对应的代码是基于雅特力提供的V2.x.x 板级支持包（BSP）而开发，对于其他版本BSP，需要注意使用上的区别。

支持型号列表：

支持型号	AT32F 系列
------	----------

## 目录

1	概述.....	5
2	HardFault 产生原因.....	6
3	HardFault 分析方法.....	7
3.1	基于 CmBacktrace 库分析方法 .....	7
3.2	基于 MDK 的 CmBacktrace 库使用流程 .....	7
3.3	案例展示 .....	12
4	版本历史 .....	13

## 表目录

表 1. 文档版本历史 .....	13
-------------------	----

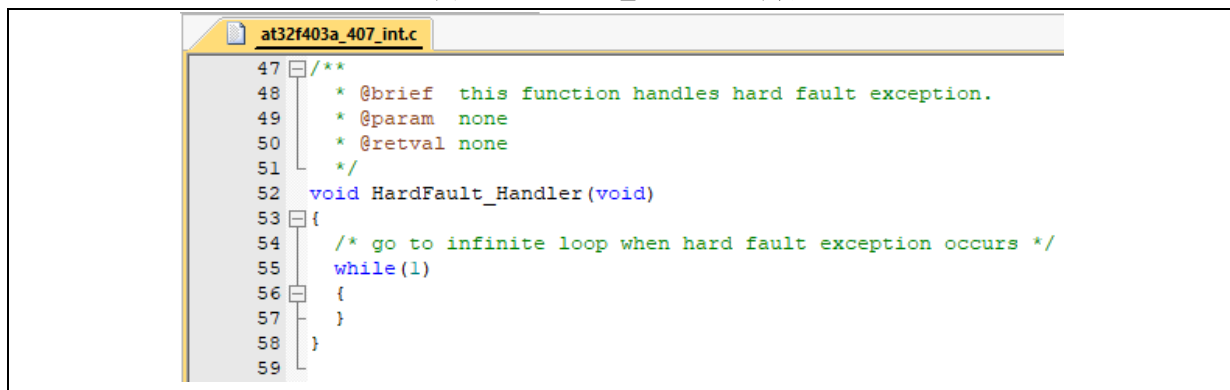
## 图目录

图 1. HardFault_Handler 函数.....	5
图 2. cm_backtrace 库文件夹.....	7
图 3. 添加 cm_backtrace 后 keil 工程目录.....	8
图 4. Keil 中配置 C99 和头文件.....	8
图 5. cmb_cfg.h 文件配置 .....	9
图 6. at32f4xx_it.c 编译报错 .....	9
图 7. HardFault_Handler 函数屏蔽 .....	9
图 8. 编写除零错误函数 .....	10
图 9. main 函数调用除零错误函数 .....	10
图 10. 串口助手输出错误信息 .....	10
图 11. 定位 addr2line.exe 位置 .....	11
图 12. 拷贝 addr2line.exe 工具 .....	11
图 13. 调用 CMD 运行 addr2line.exe 工具.....	11
图 14. 确认错误代码区域 .....	12

## 1 概述

在使用ARM Cortex-M 系列 MCU时（如AT32 MCU），有时会出现程序运行异常。当通过编译器在 debug模式查原因时，会发现程序跑到 HardFault\_Handler函数中，产生 HardFault，即硬件错误。

图 1. HardFault\_Handler 函数



```
47  /**
48   * @brief  this function handles hard fault exception.
49   * @param  none
50   * @retval none
51   */
52  void HardFault_Handler(void)
53  {
54      /* go to infinite loop when hard fault exception occurs */
55      while(1)
56      {
57      }
58  }
```

本文档主要介绍一种基于CmBacktrace库，快速追踪和定位产生HardFault原因的方法。

## 2 HardFault 产生原因

常见产生HardFault产生的原因大概有如下几类：

- 数组越界操作；
- 内存溢出，访问越界；
- 堆栈溢出，程序跑飞；
- 中断处理错误。

### 数组越界

程序中使用了静态数组，而在动态传参时数组赋值溢出。或者动态分配内存太小，导致程序异常。

### 内存溢出

重点检查RAM区域，程序编译后执行的RAM数据量大小为多少是否可能越界。一般不要设置到极致的情况，程序中的一些动态数组传参时会导致异常。

### 堆栈溢出

这在使用操作系统的代码中尤其容易发生，在操作系统中，任务的变量均分配放置在任务所申请的堆栈空间中。

例如FreeRTOS中调用xTaskCreate来创建任务，该函数以参数usStackDepth指定任务堆栈的大小，如果指定的堆栈太小，则会堆栈申请不足，进入HardFault。

### 中断处理异常

程序中开启了某些中断，例如USART,TIMER,RTC等。

但在程序执行中，满足中断条件，但并未能查找到该部分对应的中断服务函数，则可能会出现该异常。

### 3 HardFault 分析方法

常见的分析方法是：发生异常之后可首先查看LR寄存器中的值，确定当前使用堆栈为MSP或PSP，然后找到相应堆栈的指针，并在内存中查看相应堆栈里的内容。由于异常发生时，内核将R0~R3、R12 Returnaddress、PSR、LR寄存器依次入栈，其中Return address即为发生异常前PC将要执行的下一条指令地址。

但以上方法要求对ARM内核比较熟悉，且操作较为繁琐。

以下重点介绍采用开源库CmBacktrace作为快速分析的方法。

#### 3.1 基于 CmBacktrace 库分析方法

CmBacktrace（Cortex Microcontroller Backtrace）是一款针对 ARM Cortex-M 系列 MCU 的错误代码自动追踪、定位，错误原因自动分析的开源库。主要特性如下：

- 支持的错误包括：
  - 1) 断言（Assert）
  - 2) 故障（Hard Fault, Memory Management Fault, Bus Fault, Usage Fault, Debug Fault）
- 故障原因自动诊断：可在故障发生时，自动分析出故障的原因，定位发生故障的代码位置，而无需再手动分析繁杂的故障寄存器；
- 适配 Cortex-M0/M3/M4/M7 MCU；
- 支持 IAR、KEIL、GCC 编译器；
- 支持 FreeRTOS、UCOSII、UCOSIII、RT-Thread等OS；

#### 3.2 基于 MDK 的 CmBacktrace 库使用流程

基于MDK的移植方法按如下步骤进行：

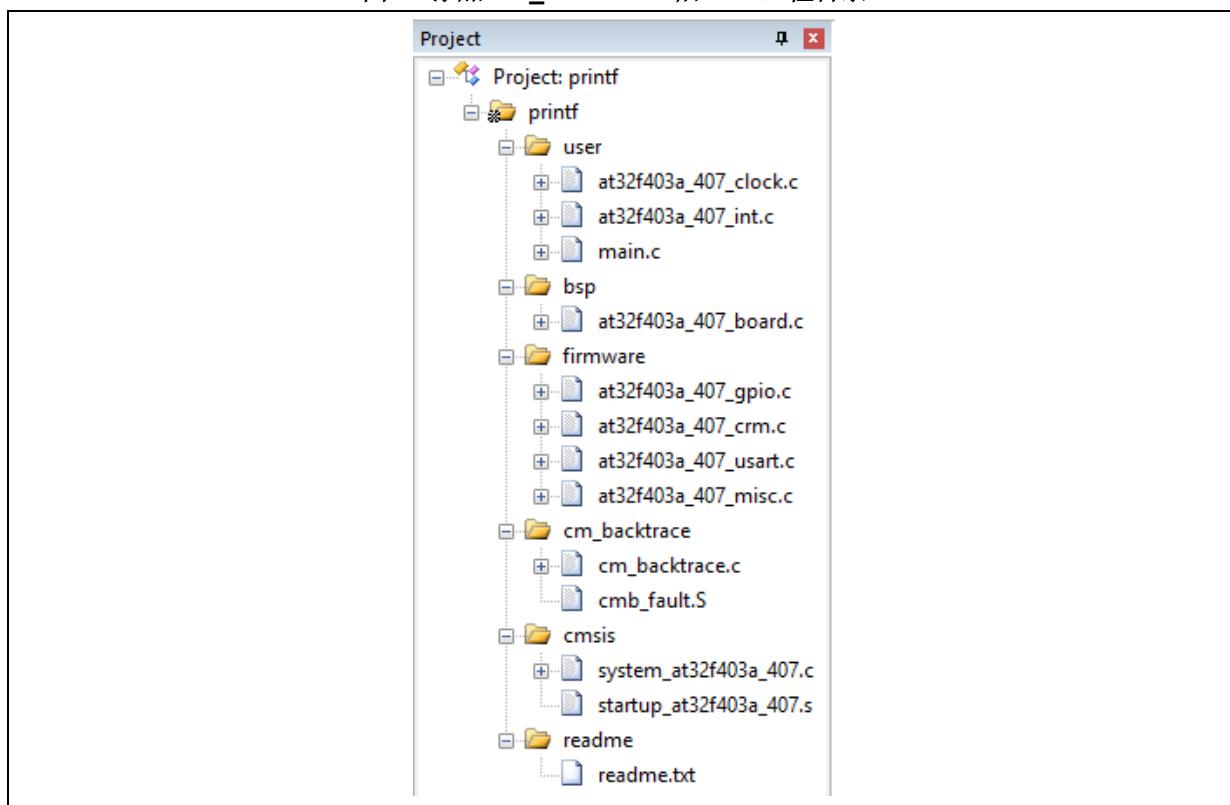
步骤一 添加cm\_backtrace库文件到MDK中

图 2. cm\_backtrace 库文件夹

> AN0028_SourceCode_V2.0.0 > utilities > AN0028_demo > non_os > src > cm_backtrace			
名称	修改日期	类型	大小
fault_handler	2022/1/27 15:40	文件夹	
cm_backtrace.c	2019/7/15 18:42	C 文件	29 KB
cm_backtrace	2019/7/15 18:42	H 文件	2 KB
cmb_cfg	2022/1/28 13:54	H 文件	3 KB
cmb_def	2019/7/15 18:42	H 文件	15 KB

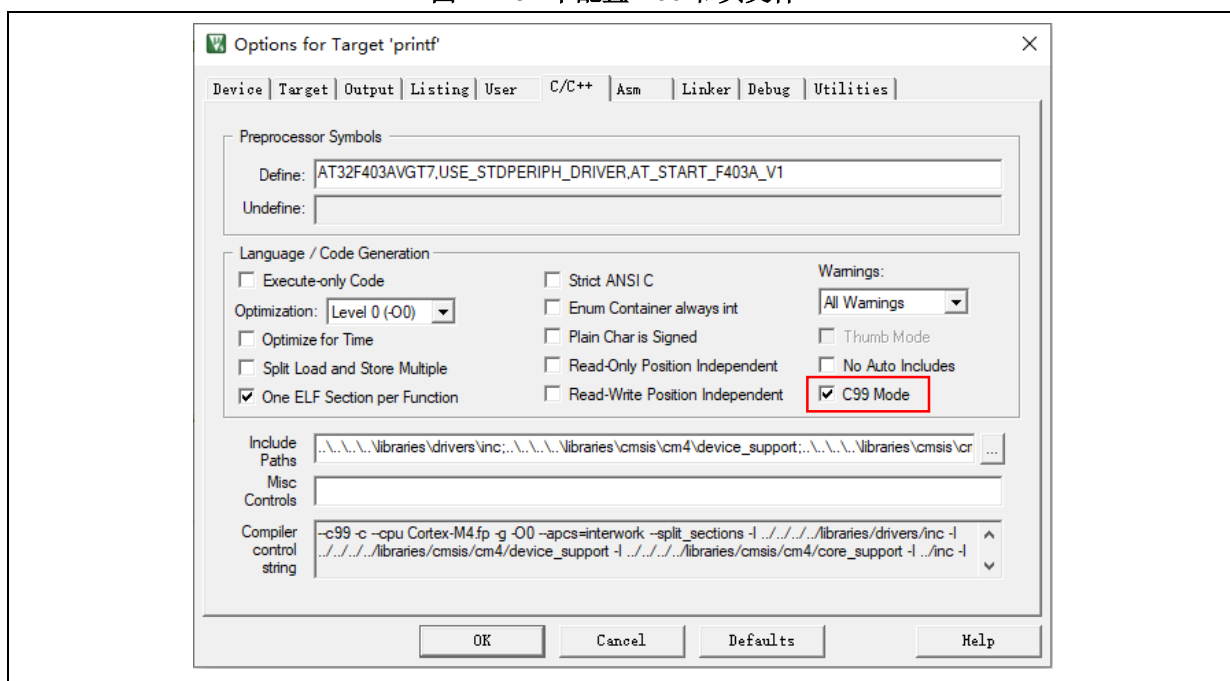
把cm\_backtrace文件夹复制到我们的工程目录下，并添加至keil工程中。

图 3. 添加 cm\_backtrace 后 keil 工程目录



步骤二 添加头文件、勾选C99模式

图 4. Keil 中配置 C99 和头文件

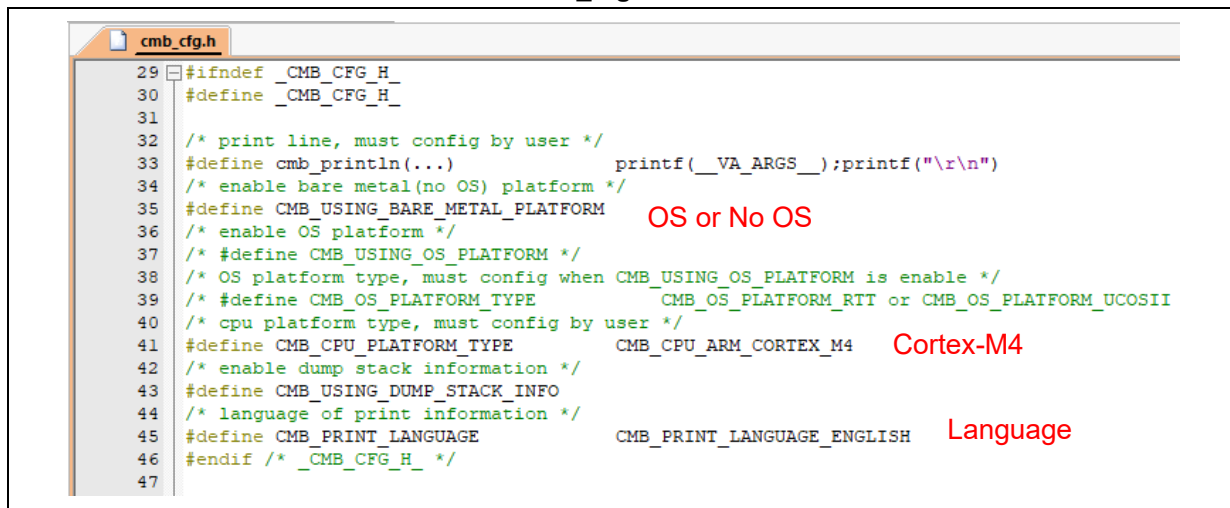


步骤三 编译和调试

首先，cmb\_cfg.h文件按以下提示配置修改。

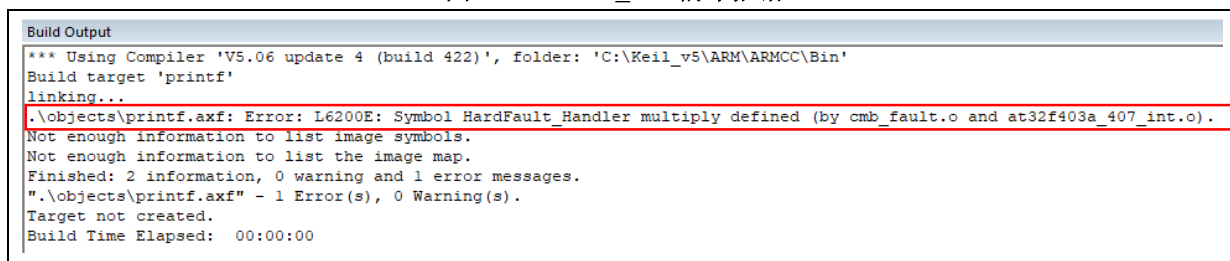


图 5. cmb\_cfg.h 文件配置



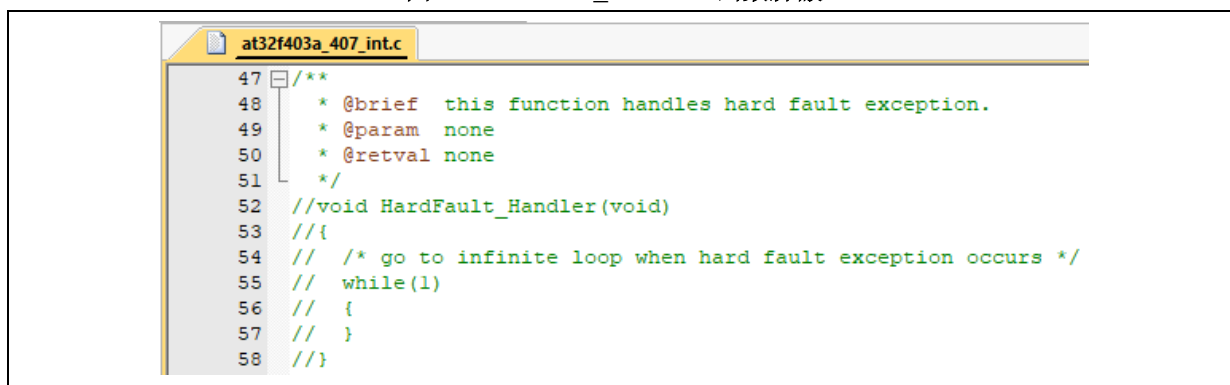
这时候编译有一个错误，这是因为cmb\_fault.c与at32f4xx\_int.c中的HardFault\_Handler函数重复定义：

图 6. at32f4xx\_it.c 编译报错



需要把at32f4xx\_int.c中的HardFault\_Handler函数屏蔽掉。

图 7. HardFault\_Handler 函数屏蔽



#### 步骤四 测试与查看

这时候就可以编译通过了。下面测试这个库的功能。

测试函数如下：

图 8. 编写除零错误函数

```
fault_test.c
30 void fault_test_by_div0(void) {
31     volatile int * SCB_CCR = (volatile int *) 0xE000ED14; // SCB->CCR
32     int x, y, z;
33
34     *SCB_CCR |= (1 << 4); /* bit4: DIV_0_TRP. */
35
36     x = 10;
37     y = 0;
38     z = x / y;
39     printf("z:%d\n", z);
40 }
```

然后在主函数中调用cm\_backtrace\_init();来初始化cm\_backtrace，并调用该测试函数：

图 9. main 函数调用除零错误函数

```
main.c
50 int main(void)
51 {
52     system_clock_config();
53     at32_board_init();
54     uart_print_init(115200);
55
56     /* output a message on hyperterminal using printf function */
57     printf("usart printf example: retarget the c library printf function to
58
59     cm_backtrace_init("CmBacktrace", HARDWARE_VERSION, SOFTWARE_VERSION);
60     fault_test_by_div0();
61
62     while(1)
63     {
64         printf("usart printf counter: %u\r\n", time_cnt++);
65         delay_sec(1);
66     }
67 }
68
```

下载运行程序，PC端接收串口信息：

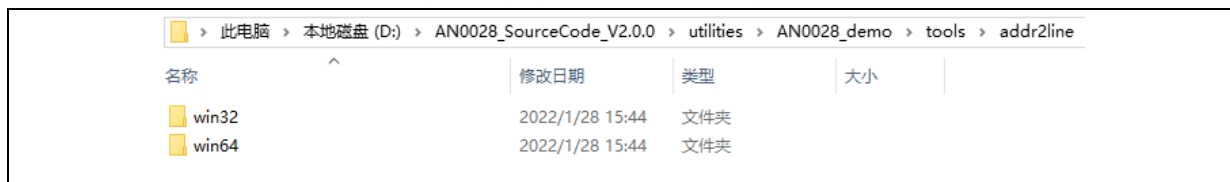
图 10. 串口助手输出错误信息

```
ATK
XCOM V2.0

usart printf example: retarget the c library printf function to the usart
Firmware name: CmBacktrace, hardware version: V1.0.0, software version: V0.1.0
Fault on interrupt or bare metal(no OS) environment
===== Thread stack information =====
addr: 200007d0    data: 00000000
addr: 200007d4    data: 20000184
addr: 200007d8    data: 00000000
addr: 200007dc    data: 00000000
addr: 200007e0    data: 00000000
addr: 200007e4    data: 08001aed
===== Registers information =====
R0 : 00000210 R1 : 20000044 R2 : 00000000 R3 : 00000000
R12: 00000000 LR : 08001aed PC : 080019c6 PSR: 61000000
Usage fault is caused by Indicates a divide by zero has taken place (can be set only
if DIV_0_TRP is set)
Show more call stack info by run: addr2line -e CmBacktrace.axf -a -f 080019c6 08001aed
```

可以看到，列出了出错原因（除0）和一条命令。运行这个命令需要用到addr2line.exe工具，该工具在tools文件夹中：

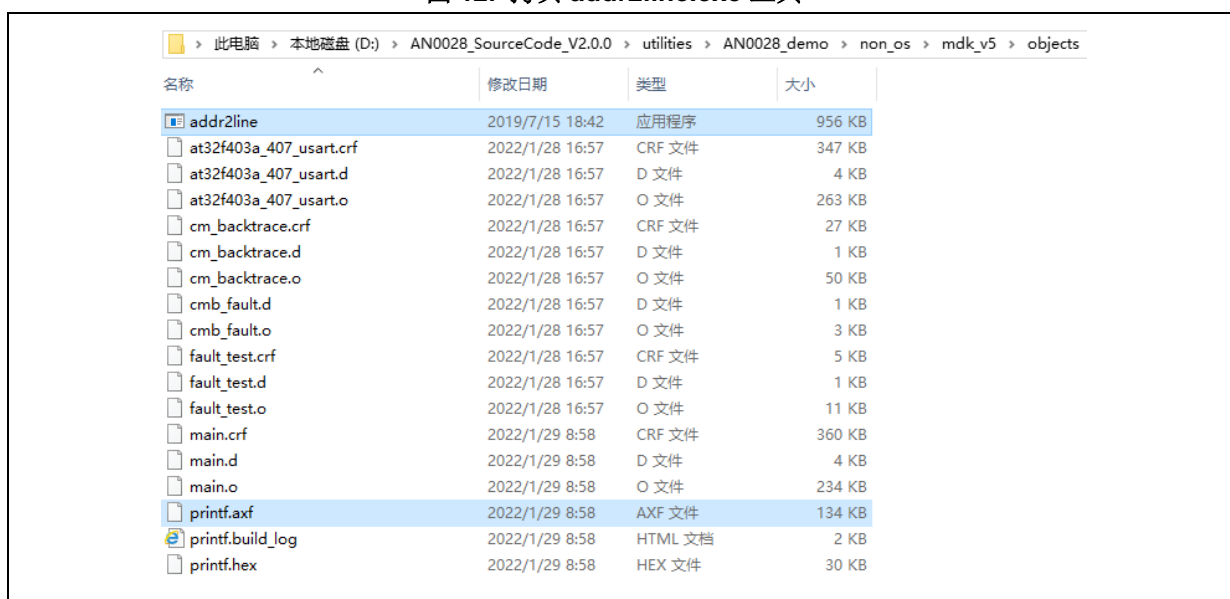
图 11. 定位 addr2line.exe 位置



有32bit和64bit两个版本，根据环境选择，并拷贝到keil工程目录下的.axf文件所在的文件夹中，如demo中所附工程，则拷贝到如下目录：

AN0028\_SourceCode\_V2.0.0\utilities\AN0028\_demo\non\_os\mdk\_v5\objects

图 12. 拷贝 addr2line.exe 工具

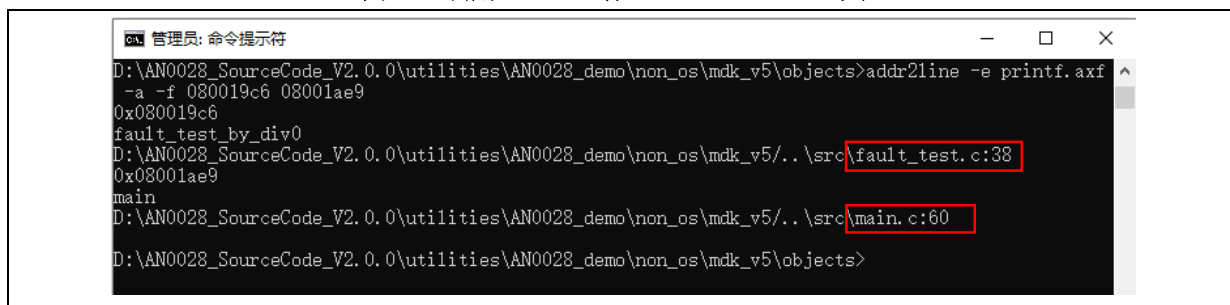


进入到cmd窗口，转到上述文件夹位置，运行串口助手中的那条命令：

addr2line -e CmBacktrace(此处要依据用户的工程名修改).axf -a -f 080019c6 08001ae9

如demo中工程名为printf，命令则应修改为addr2line -e printf.axf -a -f 080019c6 08001ae9

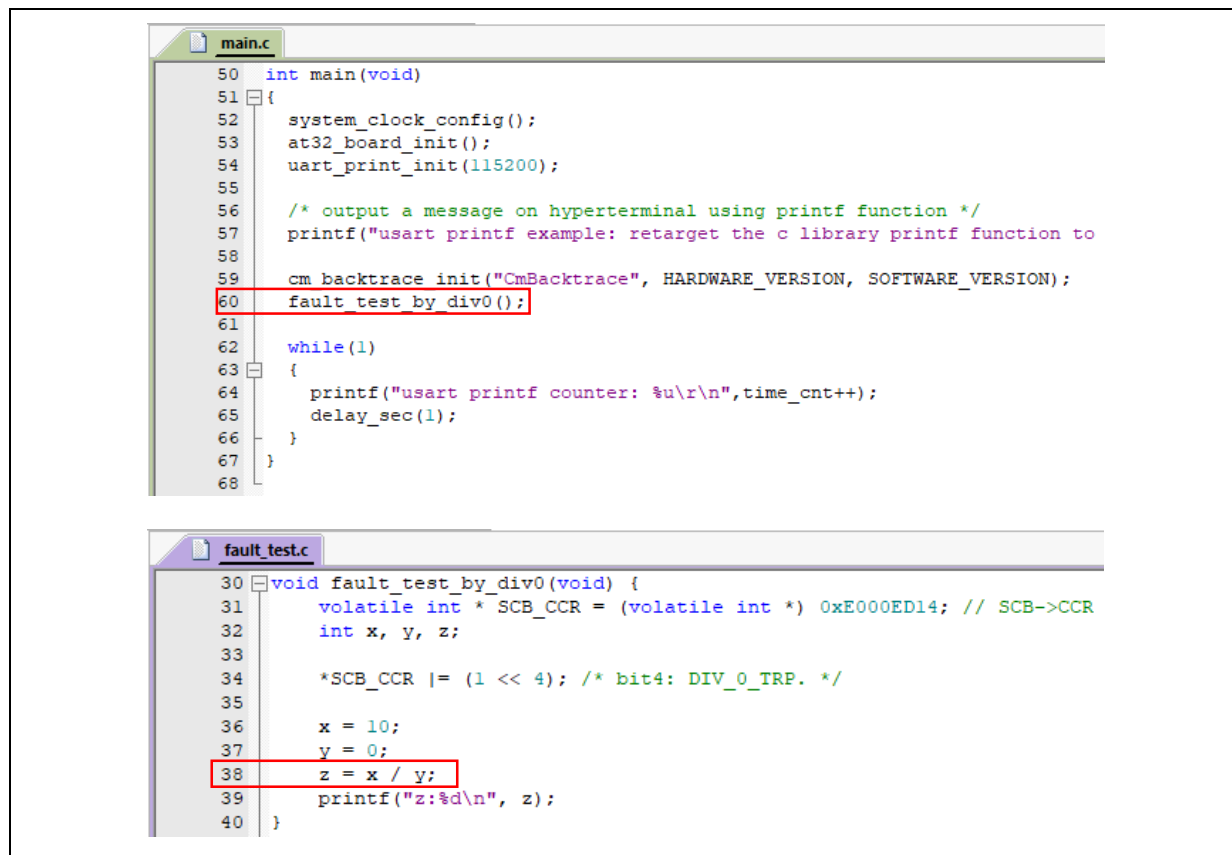
图 13. 调用 CMD 运行 addr2line.exe 工具



可以看到addr2line.exe工具定位出了错误相关的代码行号，查看对应的代码：

图 14. 确认错误代码区域

可以看到addr2line.exe工具定位出了错误相关的代码行号，main.c的第60行，fault\_test.c的第38行，查看对应行的代码：



```
main.c
50 int main(void)
51 {
52     system_clock_config();
53     at32_board_init();
54     uart_print_init(115200);
55
56     /* output a message on hyperterminal using printf function */
57     printf("usart printf example: retarget the c library printf function to
58
59     cm backtrace init("CmBacktrace", HARDWARE_VERSION, SOFTWARE_VERSION);
60     fault_test_by_div0();
61
62     while(1)
63     {
64         printf("usart printf counter: %u\r\n", time_cnt++);
65         delay_sec(1);
66     }
67 }
68

fault_test.c
30 void fault_test_by_div0(void) {
31     volatile int * SCB_CCR = (volatile int *) 0xE000ED14; // SCB->CCR
32     int x, y, z;
33
34     *SCB_CCR |= (1 << 4); /* bit4: DIV_0_TRP. */
35
36     x = 10;
37     y = 0;
38     z = x / y;
39     printf("z:%d\n", z);
40 }
```

可见，对应的行号正是出错的地方，使用这个CmBacktrace 库能帮助用户有效、快速地定位到HardFault之类的错误。

## 3.3 案例展示

### 案例一 无OS除零错误

工程位置：AN0028\_SourceCode\_V2.0.0\utilities\AN0028\_demo\non\_os

测试内容：在裸机上除零错误

### 案例二 FreeRTOS上除零错误

工程位置：AN0028\_SourceCode\_V2.0.0\utilities\AN0028\_demo\os\freertos

测试内容：在FreeRTOS上除零错误，需注意tasks.c中有注释/\*< Support For CmBacktrace >\*/的三处为针对CmBacktrace做出的修改

### 案例三 USOCIII上非对齐访问错误

工程位置：AN0028\_SourceCode\_V2.0.0\utilities\AN0028\_demo\os\ucosiii

测试内容：在UCOSIII上非对齐访问错误，需注意os\_cfg.h 中#define OS\_CFG\_DBG\_EN 为 1u

## 4 版本历史

表 1. 文档版本历史

日期	版本	变更
2022.2.7	2.0.0	最初版本

**重要通知 - 请仔细阅读**

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途(及其依据任何司法管辖区的法律的对应情况)，或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力的产品不得应用于武器。此外，雅特力产品也不是为下列用途而设计并不得应用于下列用途：(A) 对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；(B) 航空应用；(C) 汽车应用或汽车环境，且/或(D) 航天应用或航天环境。如果雅特力产品不是为前述应用设计的，而采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，采购商仍将独自承担因此而导致的任何风险，雅特力的产品设计规格明确指定的汽车、汽车安全或医疗工业领域专用产品除外。根据相关政府主管部门的规定，ESCC、QML 或 JAN 正式认证产品适用于航天应用。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。

© 2022 雅特力科技 (重庆) 有限公司 保留所有权利