CmBacktrace简介

- 支持断言（assert）和故障（Hard Fault）
- 故障原因自动诊断
- 输出错误现场的 函数调用栈
- 适配 Cortex-M0/M3/M4/M7等MCU；
- 支持 IAR、KEIL、GCC 编译器；

原理及移植方法

1 基本原理 其原理主要基于 Cortex-M 架构的压栈特性和指令分析，以下是其工作原理的详细介绍：

1.1 压栈特性

- Cortex-M 架构在发生异常或函数调用时，会自动将相关寄存器（如 R0-R3、R12、LR、PC、PSR 等）压入栈中。CmBacktrace 通过分析栈中的数据来获取函数调用栈的信息。

1.2 指令分析与函数调用栈还原

- 当程序出现异常时，CmBacktrace 会获取当前的栈顶指针（SP）和栈的起始地址及大小。然后从栈顶开始遍历，每次读取一个地址值。如果该地址值减去一个字的大小后是奇数（即符合 Thumb 指令模式的地址），并且该地址对应的指令是 BL 或 BLX（函数调用指令），则认为这是一个有效的函数调用地址。

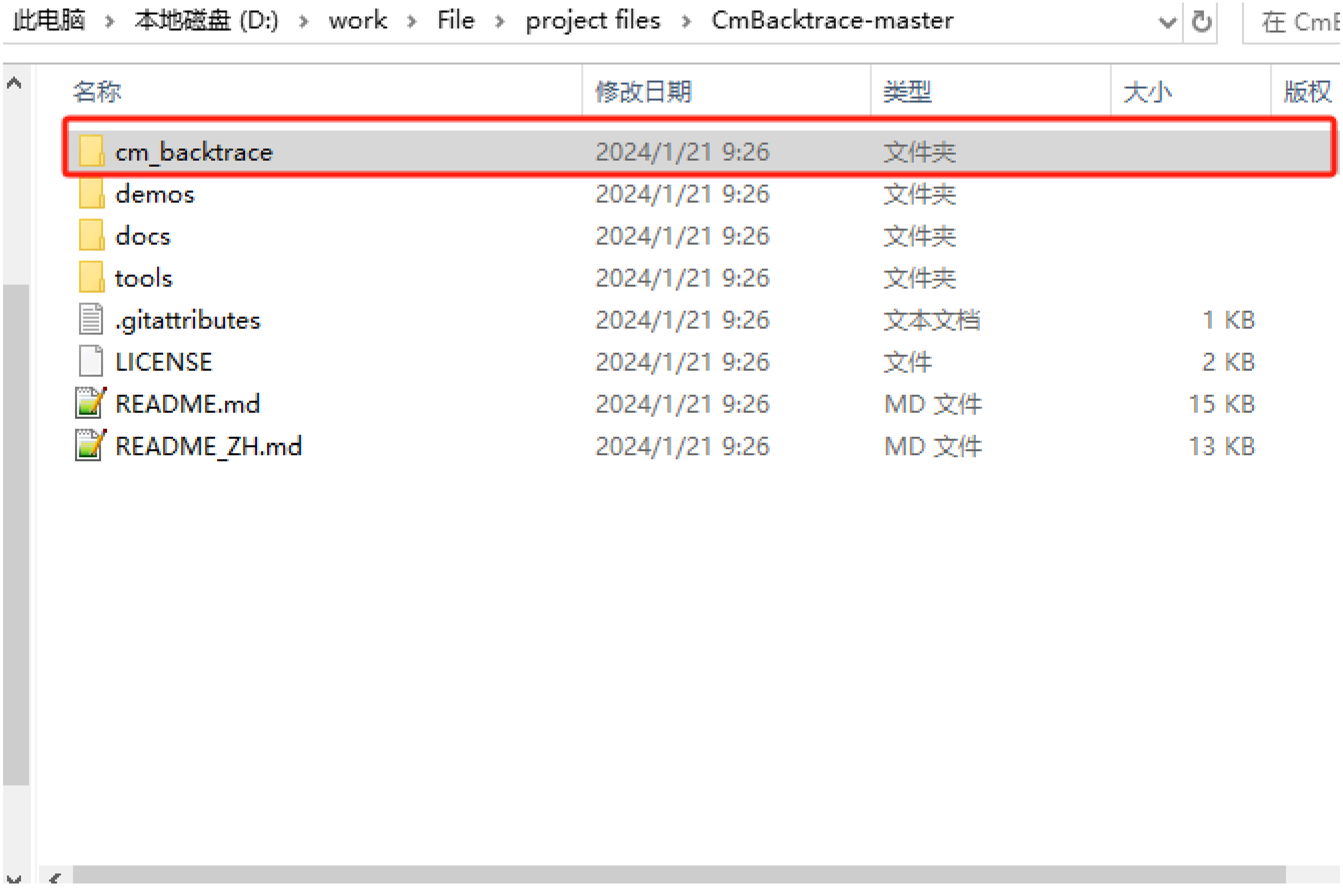
1.3 错误现场信息保存

- CmBacktrace 会在异常发生时保存 CPU 的寄存器状态，包括 R0-R12、LR、PC、PSR 等。这些寄存器的值可以帮助开发者了解异常发生时程序的执行状态。
- CmBacktrace 还会根据异常类型（如 Hard Fault、Bus Fault 等）保存相关的故障状态寄存器（如 HFSR、BFSR、MMFSR 等），这些寄存器的值可以用来分析异常的具体原因。

2. 源码及例程

官方源码链接：https://github.com/armink/CmBacktrace **示例项目链接**：https://github.com/XUAN9527/cmbacktrace-demo **说明文档链接**：https://xuan9527.github.io/2024/04/19/CmBacktrace%E7%A7%BB%E6%A4%8D/

源码目录：



此电脑 › 本地磁盘 (D:) › work › File › project files › CmBacktrace-master

名称	修改日期	类型	大小	版权
cm_backtrace	2024/1/21 9:26	文件夹		
demos	2024/1/21 9:26	文件夹		
docs	2024/1/21 9:26	文件夹		
tools	2024/1/21 9:26	文件夹		
.gitattributes	2024/1/21 9:26	文本文档	1 KB	
LICENSE	2024/1/21 9:26	文件	2 KB	
README.md	2024/1/21 9:26	MD 文件	15 KB	
README_ZH.md	2024/1/21 9:26	MD 文件	13 KB	

- 将源码拷贝到工程目录下，如`~/work/cmbacktrace-demo/code/components`
- 添加头文件`cm_backtrace.h cmb_cfg.h cmb_def.h`
- 添加源文件`cm_backtrace.c`
- 添加demo文件 `demos/non_os/stm32f10x/app/src/fault_test.c`

2.1 添加修改makefile：

方法一、修改添加`fault_handler/gcc/cmb_fault.S`为`fault_handler/gcc/cmb_fault.s`

```
ASM_SOURCES =  \
CMSIS/device/startup/startup_n32l40x_gcc.s \
components/cm_backtrace/fault_handler/gcc/cmb_fault.s # 添加这一行
```

方法二、将`cmb_fault.S`将入`makefile`编译选项

```
ASM_SOURCES = CMSIS/device/startup/startup_n32l40x_gcc.s
ASM_SOURCES2 = components/cm_backtrace/fault_handler/gcc/cmb_fault.S    # 此行为新增

# C源文件、汇编源文件的目标文件路径
C_OBJECTS = $(addprefix $(OUTPUT_DIR)/, $(C_SOURCES:.c=.o))
ASM_OBJECTS = $(addprefix $(OUTPUT_DIR)/, $(ASM_SOURCES:.s=.o)) \
          $(addprefix $(OUTPUT_DIR)/, $(ASM_SOURCES2:.S=.o))    # 此行为新增

$(OUTPUT_DIR)/%.o: %.s
```

```
    mkdir -p $(dir $@)
    $(CC) $(INCLUDE) $(CFLAGS) -c $< -o $@

$(OUTPUT_DIR)/%.o: %.S         # 新增 %.S
    mkdir -p $(dir $@)
    $(CC) $(INCLUDE) $(CFLAGS) -c $< -o $@
```

2.2 添加printf重定向：

```
int _write(int fd, char* pBuffer, int size)
{
    // 添加自己的发送函数
    return drv_serial_dma_write(ESERIAL_1, pBuffer, size);
}
```

2.3 修改文件：

- `cmb_cfg.h`配置文件：

```
#ifndef _CMB_CFG_H_
#define _CMB_CFG_H_

#include "log.h"

/* print line, must config by user */
#define cmb_println(...)  printf(__VA_ARGS__);printf("\r\n") /* e.g., printf(__VA_ARGS__);printf("\r\n")  or  SEGGER_RTT_printf(0, __VA_ARGS__);SEGGER_RTT_WriteString(0, "\r\n")  */
/* enable bare metal(no OS) platform */
#define CMB_USING_BARE_METAL_PLATFORM
/* enable OS platform */
/* #define CMB_USING_OS_PLATFORM */
/* OS platform type, must config when CMB_USING_OS_PLATFORM is enable */
/* #define CMB_OS_PLATFORM_TYPE           CMB_OS_PLATFORM_RTT or CMB_OS_PLATFORM_UCOSII or CMB_OS_PLATFORM_UCOSIII or CMB_OS_PLATFORM_FREERTOS or CMB_OS_PLATFORM_RTX5 */
/* cpu platform type, must config by user */
#define CMB_CPU_PLATFORM_TYPE   CMB_CPU_ARM_CORTEX_M4       /* CMB_CPU_ARM_CORTEX_M0 or CMB_CPU_ARM_CORTEX_M3 or CMB_CPU_ARM_CORTEX_M4 or CMB_CPU_ARM_CORTEX_M7 */
/* enable dump stack information */
#define CMB_USING_DUMP_STACK_INFO
/* language of print information */
#define CMB_PRINT_LANGUAGE    CMB_PRINT_LANGUAGE_ENGLISH        /* CMB_PRINT_LANGUAGE_ENGLISH(default) or CMB_PRINT_LANGUAGE_CHINESE */
#endif /* _CMB_CFG_H_ */
```

- 修改n32l40x_flash.ld链接文件如下：
- text段开始之前添加 _stext = .; 下面为例程：

```
/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  _stext = .;               # text段开始之前添加

  /* The program code and other data goes into FLASH */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    *(.glue_7)         /* glue arm to thumb code */
    *(.glue_7t)        /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))
```

- bss段开始之前添加 _sstack = .; 下面为例程：

```
  .bss :
  {
    /* This is used by the startup in order to initialize the .bss secion */
    _sbss = .;         /* define a global symbol at bss start */
    __bss_start__ = _sbss;
    *(.bss)
    *(.bss*)
    *(COMMON)

    . = ALIGN(4);
    _ebss = .;         /* define a global symbol at bss end */
    __bss_end__ = _ebss;
  } >RAM

  _sstack = .;              # stack段开始之前添加

  /* User_heap_stack section, used to check that there is enough RAM left */
  ._user_heap_stack :
  {
    . = ALIGN(4);
    PROVIDE ( end = . );
    PROVIDE ( _end = . );
    . = . + _Min_Heap_Size;
    . = . + _Min_Stack_Size;
    _estack = .;        /* end of RAM */
    . = ALIGN(4);
  } >RAM
```

2.4 储存错误信息：

- `cm_backtrace.c`文件修改，添加读写部分：

```
...
// 添加读写flash的地址
#include "dcd_user.h"
#define ERRORLOG_FLASH_BASIC_ADDR   USER_DATA_ADDR
#define ERRORLOG_FLASH_OFFSET       (0 * 1024)
#define ERRORLOG_FLASH_TARGET_ADDR  (ERRORLOG_FLASH_BASIC_ADDR +
ERRORLOG_FLASH_OFFSET)
#define ERRORLOG_FLASH_TARGET_SIZE  (2 * 1024)
/**
 * dump function call stack
 *
 * @param sp stack pointer
 */
static void print_call_stack(uint32_t sp) {
    size_t i, cur_depth = 0;
    uint32_t call_stack_buf[CMB_CALL_STACK_MAX_DEPTH] = {0};

    cur_depth = cm_backtrace_call_stack(call_stack_buf, CMB_CALL_STACK_MAX_DEPTH,
sp);

    for (i = 0; i < cur_depth; i++) {
        sprintf(call_stack_info + i * (8 + 1), "%08lx", (unsigned
long)call_stack_buf[i]);
        call_stack_info[i * (8 + 1) + 8] = ' ';
    }

    if (cur_depth) {
        call_stack_info[cur_depth * (8 + 1) - 1] = '\0';
        cmb_println(print_info[PRINT_CALL_STACK_INFO], fw_name,
CMB_ELF_FILE_EXTENSION_NAME, call_stack_info);

        // 添加部分，回溯字符串写到flash里。例：Show more call stack info by run:
addr2line -e CmBacktrace.elf -a -f 080154c2 0800a3b3 08009092
        uint8_t buff[512] = {0};
        snprintf((char *)buff, sizeof(buff), print_info[PRINT_CALL_STACK_INFO],
fw_name, CMB_ELF_FILE_EXTENSION_NAME, call_stack_info);
        dcd_port_erase(ERRORLOG_FLASH_TARGET_ADDR, ERRORLOG_FLASH_TARGET_SIZE);
        dcd_port_write(ERRORLOG_FLASH_TARGET_ADDR, (const uint32_t *)buff,
strlen((char *)buff) + 1);
    } else {
        cmb_println(print_info[PRINT_CALL_STACK_ERR]);
    }
}

// 读取错误信息
static void fault_read_string(void)
{
    uint8_t buff[512] = {0};
    dcd_port_read(ERRORLOG_FLASH_TARGET_ADDR, (uint32_t *)buff, sizeof(buff));
    buff[512-1] = 0;
    logPrintln("CmBacktrace hard fault = %s", buff);
}
SHELL_EXPORT_CMD(SHELL_CMD_PERMISSION(0)|SHELL_CMD_TYPE(SHELL_TYPE_CMD_FUNC)|SHEL
L_CMD_DISABLE_RETURN, fault_read_string, fault_read_string, fault_read_string);
...
```

2.5 注释掉原有的`HardFault_Handler`：

```
/**
 *\*\name   HardFault_Handler.
 *\*\fun    This function handles Hard Fault exception.
 *\*\param  none.
 *\*\return none.
 */
// void HardFault_Handler(void)
// {
//     /* Go to infinite loop when Hard Fault exception occurs */
//     while (1)
//     {
//     }
// }
```

2.6 主函数例程：

```
#include "cm_backtrace.h"
#define HARDWARE_VERSION               "V1.0.0"
#define SOFTWARE_VERSION               "V0.1.0"

extern void fault_test_by_unalign(void);
extern void fault_test_by_div0(void);

int main(void)
{
    main_system_init();
    cm_backtrace_init("CmBacktrace", HARDWARE_VERSION, SOFTWARE_VERSION);     //
在开启时钟，打印和看门狗之后就需要初始化

    fault_test_by_unalign();    # 字节对齐异常示例
    fault_test_by_div0();       # 除零异常示例

    while(1)
    {
    }
}
```

3. 编译出错后代码：

```
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: 20004ec8    data: 5a6d79ca
  addr: 20004ecc    data: f758b4b7
  addr: 20004ed0    data: 94cfc3fd
  addr: 20004ed4    data: a8ccaa51
  addr: 20004ed8    data: 61049ca6
  addr: 20004edc    data: e4e1b169
  addr: 20004ee0    data: b48e100d
  addr: 20004ee4    data: c44eb7ea
  addr: 20004ee8    data: 23d4e51e
  addr: 20004eec    data: 8527b7c0
  addr: 20004ef0    data: fd9d41f7
  addr: 20004ef4    data: f539e421
  addr: 20004ef8    data: 4ad52963
  addr: 20004efc    data: 4587b423
  addr: 20004f00    data: e000ed00
  addr: 20004f04    data: 00000000
  addr: 20004f08    data: 00000000
  addr: 20004f0c    data: 00000000
  addr: 20004f10    data: 200022cc
  addr: 20004f14    data: 00000000
  addr: 20004f18    data: 00000000
  addr: 20004f1c    data: 00000000
  addr: 20004f20    data: 00000000
  addr: 20004f24    data: 08009093
====================================
=================== Registers information ====================
  R0 : 20002ee9  R1 : 20002e4c  R2 : e000ed14  R3 : 2000253c
  R12: 0000000a  LR : 0800a3b3  PC : 080154c2  PSR: 61000000
==============================================================
Usage fault is caused by attempts to execute an undefined instruction
Show more call stack info by run: addr2line -e CmBacktrace.elf -a -f 080154c2
0800a3b3 08009092
```

转换为定位代码工具：

- linux环境下输入，app.elf 为你的工程编译文件，需在当前目录下：

```
addr2line -e app.elf -a -f 080154c2 0800a3b3 08009092
```

数据分析结果：

```
xuan@DESKTOP-A52B6V9:~/work/n5-mini-s-plus/code/app/build$ addr2line -e app.elf -a -f 080154c2 0800a3b3 08009092
0x080154c2
fault_test_by_unalign
/home/xuan/work/n5-mini-s-plus/code/app/components/cm_backtrace/fault_test.c:18
0x0800a3b3
main
/home/xuan/work/n5-mini-s-plus/code/app/application/main.c:30
0x08009092
LoopFillZerobss  # .bss段异常（未初始化的全局和静态变量）
/home/xuan/work/n5-mini-s-plus/code/app/CMSIS/device/startup/startup_n32l40x_gcc.s:113
```

3. 总结：

- CmBacktrace能快速方便的定位偶现的程序跑飞问题。
- 错误log全功能打印带储存代码占用为8K左右；去掉打印，保留最基本的功能代码占用空间仅4K左右。
- 储存代码段可优化，添加flash擦写均衡。