GR5xx Fault Trace Module应用说明

版本： **3.2**

发布日期： **2023-11-06**

深圳市汇顶科技股份有限公司

版权所有 © 2023 深圳市汇顶科技股份有限公司。保留一切权利。

非经本公司书面许可，任何单位和个人不得对本手册内的任何部分擅自摘抄、复制、修改、翻译、传播，或将其全部或部分用于商业用途。

商标声明

GOODiX 和其他汇顶商标均为深圳市汇顶科技股份有限公司的商标。本文档提及的其他所有商标或注册商标，由各自的所有人持有。

免责声明

本文档中所述的器件应用信息及其他类似内容仅为您提供便利，它们可能由更新之信息所替代。确保应用符合技术规范，是您自身应负的责任。

深圳市汇顶科技股份有限公司（以下简称“GOODIX”）对这些信息不作任何明示或暗示、书面或口头、法定或其他形式的声明或担保，包括但不限于针对其使用情况、质量、性能、适销性或特定用途的适用性的声明或担保。GOODIX对因这些信息及使用这些信息而引起的后果不承担任何责任。

未经GOODIX书面批准，不得将GOODIX的产品用作生命维持系统中的关键组件。在GOODIX知识产权保护下，不得暗中或以其他方式转让任何许可证。

深圳市汇顶科技股份有限公司

总部地址：深圳市福田保税区腾飞工业大厦B座12-13层

电话：+86-755-33338828　　邮编：518000

网址：www.goodix.com

前言

编写目的

本文档描述GR5xx Fault Trace Module应用的作用、原理和使用方法，旨在帮助用户快速使用该模块。

读者对象

本文适用于以下读者：

- 芯片用户
- 开发人员
- 测试人员
- 文档工程师

版本说明

本文档为第4次发布，对应的产品为低功耗蓝牙GR5xx系列。

修订记录

版本	日期	修订内容
1.0	2023-01-10	首次发布
3.0	2023-03-30	新增支持多款芯片的相关描述
3.1	2023-08-08	更新“使用GProgrammer读取Fault Trace Data失败”章节中的问题分析和处理方法
3.2	2023-11-06	更新GProgrammer、GRUart、GRToolbox获取方式。

目录

前言 I
1 简介 1
2 环境搭建 2
2.1 准备工作 2
3 使用Fault Trace Module 3
3.1 导入Fault Trace Module 3
3.1.1 添加模块 3
3.1.2 使能模块 4
3.1.3 初始化模块 4
3.2 读取Fault Trace Data 5
3.2.1 蓝牙连接读取 5
3.2.2 GProgrammer读取 8
3.2.3 工程中调用API读取 9
3.3 实例展示 11
4 模块详解 15
4.1 HardFault Data Trace 15
4.2 Assert Fault Data Trace 16
4.3 蓝牙控制实现 18
5 常见问题 21
5.1 使用GProgrammer读取Fault Trace Data失败 21
5.2 工程中调用API读取Fault Trace Data失败 21

版权所有 © 2023 深圳市汇顶科技股份有限公司

1 简介

GR5xx Fault Trace Module是一个用于开发阶段辅助定位系统异常问题的应用模块。在GR5xx的固件运行失常时它可以将某些现场信息（Fault Trace Data）写入到Flash中的NVDS区域，之后通过特定方法从NVDS中导出Fault Trace Data，从而还原现场，帮助定位问题。

GR5xx Fault Trace Module支持以下两种场景将Fault Trace Data写入NVDS：

- 当发生HardFault时，将芯片内部寄存器的现场值写入NVDS中。
- 当使用Assert模块断言失败（Assert Fault）时，将现场的函数名、行数、参数名等信息写入NVDS中。

在进行操作前，可参考以下文档。

表 1-1 文档参考

名称	描述
对应芯片开发者指南	介绍GR5xx SDK以及基于SDK的应用开发和调试
J-Link用户指南	J-Link的使用说明：https://www.segger.com/downloads/jlink/UM08001_JLink.pdf
Keil用户指南	Keil的详细操作：https://www.keil.com/support/man/docs/uv4/
Bluetooth Core Spec	Bluetooth官方标准核心规范
Bluetooth GATT Spec	Bluetooth Profile和Service的详细信息查看地址：https://www.bluetooth.com/specifications/gatt
GProgrammer用户手册	GProgrammer软件的操作使用说明，包括固件下载、固件加密等。

版权所有 © 2023 深圳市汇顶科技股份有限公司

2 环境搭建

本章介绍如何快速搭建GR5xx Fault Trace Module应用的运行环境。

说明:

SDK_Folder为对应芯片SDK的根目录。

2.1 准备工作

- 硬件准备

表 2-1 硬件准备

名称	描述
开发板	对应芯片Starter Kit开发板（以下简称“开发板”）
连接线	USB Type-C（GR551x系列使用Micro USB 2.0连接线）
Android Phone	Android 5.0（KitKat）及以上版本

- 软件准备

表 2-2 软件准备

名称	描述
Windows	Windows 7/Windows 10操作系统
J-Link Driver	J-Link驱动程序，下载网址：http://www.segger.com/downloads/jlink/
Keil MDK-ARM IDE（Keil）	IDE工具，支持MDK-ARM 5.20 及以上版本，下载网址：https://www.keil.com/demo/eval/arm.htm
GProgrammer（Windows）	Programming工具，下载网址：www.goodix.com/zh/software_tool/gprogrammer_ble
GRUart（Windows）	串口调试工具，下载网址：www.goodix.com/zh/download?objectId=64&objectType=software
GRToolbox（Android）	Bluetooth LE调试工具，下载网址：www.goodix.com/zh/software_tool/grtoolbox

3 使用Fault Trace Module

本章将以心率示例工程ble_app_hrs为例，介绍GR5xx Fault Trace Module的导入以及使用方法。

3.1 导入Fault Trace Module

Fault Trace Module是一个独立的功能模块，在使用前需要在用户的应用工程中添加Fault Trace Module的文件、打开该模块的宏开关并进行初始化。

3.1.1 添加模块

1. 打开ble_app_hrs心率示例工程。

   心率示例工程的源代码和工程文件位于`SDK_Folder\projects\ble\ble_peripheral\ble_app_hrs`，其中工程文件位于Keil_5文件夹。

2. 在ble_app_hrs工程目录添加Fault Trace Module的相关源文件。

说明:

目前SDK包提供的ble_app_hrs示例工程，默认已添加Fault Trace Module源文件。

   Fault Trace Module源文件*fault_trace.c*和*cortex_backtrace.c*分别位于`SDK_Folder\components\libraries\fault_trace`和`SDK_Folder\components\libraries\app_error`。

   选中gr_libraries目录点击鼠标右键，选择“Add Existing Files to Group "gr_libraries"”将*fault_trace.c*以及*cortex_backtrace.c*文件手动添加至gr_libraries目录下。添加成功后如下图所示：

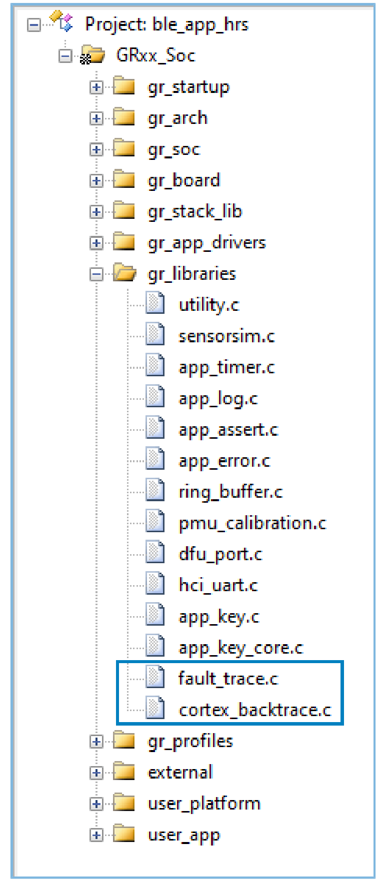

图 3-1 将Fault Trace Module文件添加到工程目录

3.1.2 使能模块

在工程目录中打开user_app\custom_config.h文件，找到关于该模块的宏开关SYS_FAULT_TRACE_ENABLE，将宏SYS_FAULT_TRACE_ENABLE设置为1。

3.1.3 初始化模块

在工程目录中打开user_platform\user_periph_setup.c文件，在app_periph_init()中调用函数fault_trace_db_init()，进行Fault Trace Module初始化。

```
void app_periph_init(void)
{
    SYS_SET_BD_ADDR(s_bd_addr);
#if DTM_TEST_ENABLE
    dtm_trigger_pin_init();
#endif

#if DTM_TEST_ENABLE
    if (dtm_test_enable)
    {
        pwr_mgmt_mode_set(PMR_MGMT_ACTIVE_MODE);
    }
    else
```

```
    {
        board_init();
        fault_trace_db_init();
        pwr_mgmt_mode_set(PMR_MGMT_SLEEP_MODE);
    }
#else
    board_init();
    fault_trace_db_init();
    pwr_mgmt_mode_set(PMR_MGMT_SLEEP_MODE);
#endif
}
```

说明:

DTM测试模式下为避免测试冲突不推荐使用Fault Trace模块。若使用Fault Trace模块，则在*custom_config.h*中将DTM_TEST_ENABLE宏置0。

ble_app_hrs示例工程默认已调用fault_trace_db_init()函数进行了初始化。

初始化完成后，可参考对应芯片开发者指南，将编译好的程序烧录到开发板中。

当运行该工程的开发板发生HardFault或Assert Fault时，相关现场信息就会被存入到对应芯片的NVDS中。除非对Flash进行整片擦除，否则该信息会一直存在。

3.2 读取Fault Trace Data

支持三种方式读取存储在NVDS中的Fault Trace Data：

1. 在手机端使用GRToolbox工具，通过蓝牙连接读取开发板的Fault Trace Data。
2. 利用GProgrammer读取开发板NVDS中的Fault Trace Data。
3. 在工程中直接调用相关API来读取Fault Trace Data。

说明:

蓝牙连接读取和工程中直接调用API读取的方式，要依赖Fault Trace Module，使用这两种方式读取时，需要确保开发板运行的固件中添加并使能了该模块。GProgrammer读取则不需要。

3.2.1 蓝牙连接读取

使用蓝牙连接读取的方式，除了对Fault Trace Module的依赖之外，还必须确保目标设备运行了LNS服务（Log Notification Service）。

若目标设备无LNS服务，则需要在设备运行的ble_app_hrs工程中添加LNS服务。

LNS服务源文件位于`SDK_Folder\components\profiles\lns`。

在ble_app_hrs工程目录中选中gr_profiles目录点击鼠标右键，选择“Add Existing Files to Group gr_profiles”将*lns.c*文件手动添加至gr_profiles目录下。添加成功后如图 3-2所示：

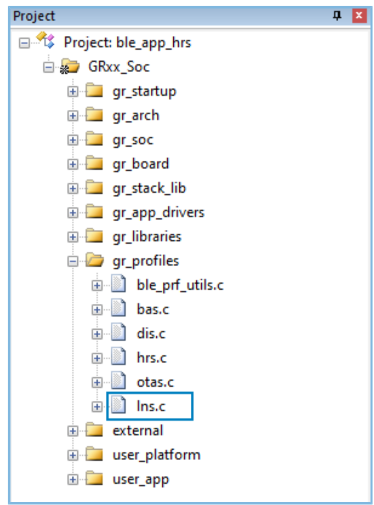

图 3-2 将LNS文件添加到工程目录

将*lns.c*添加到gr_profiles目录下后，还需在`SDK_Folder\projects\ble\ble_peripheral\ble_app_hrs\Src\user\user_app.c`的services_init()中调用初始化函数lns_service_init()完成LNS服务的初始化。

说明:

ble_app_hrs工程默认已添加并完成LNS服务的初始化。

配置好工程后，使用开发板运行该工程生成的固件。

1. 在手机端使用GRToolbox连接该开发板之后，可以发现LNS服务“Log Notification Service”，如图 3-3所示。

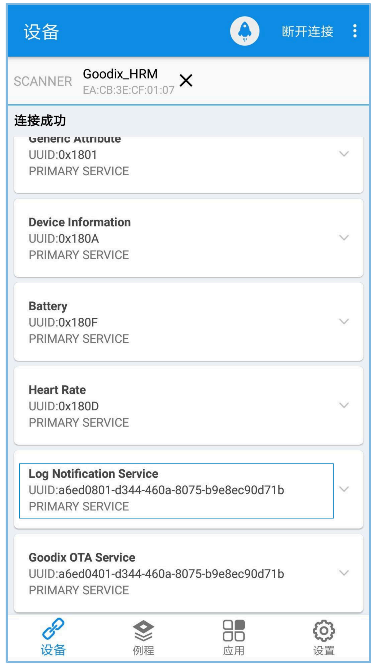

图 3-3 GRToolbox连接开发板后发现LNS服务

版权所有 © 2023 深圳市汇顶科技股份有限公司

📖 说明:

本文中GRToolbox的截图仅供用户了解操作步骤，实际界面请参考最新版本GRToolbox。

2. 点击右上角的⋮按钮，先选择“设置最大数据单元”将值设置为“512”，再选择“读取崩溃记录”获取FaultTrace Data，如图 3-4所示：

📖 说明:

设备间数据交互的最大数据单元（MTU）默认为23 Bytes，而FaultTrace Data大于23 Bytes。因此，获取FaultTrace Data前需先在手机端设置MTU值（建议值：512 Bytes）。

图 3-4 设置最大数据单元并读取崩溃记录

3. 在弹出的“读取崩溃记录”弹框中，点击“读取”按钮，即可读取出开发板存储的Fault Trace Data。

图 3-5 设备日志读取成功

3.2.2 GProgrammer读取

将PC连接需要读取Fault Trace Data的开发板后，运行GProgrammer工具。

在GProgrammer主界面点击 按钮进入“Device Log”界面。

图 3-6 GProgrammer工具Device Log界面

版权所有 © 2023 深圳市汇顶科技股份有限公司

说明:

本文中GProgrammer的截图仅供用户了解操作步骤，实际界面请参考最新版本GProgrammer。

点击芯片配置界面上的“Read”按钮，就可以读取开发板NVDS中的Fault Trace Data。如图 3-7所示：

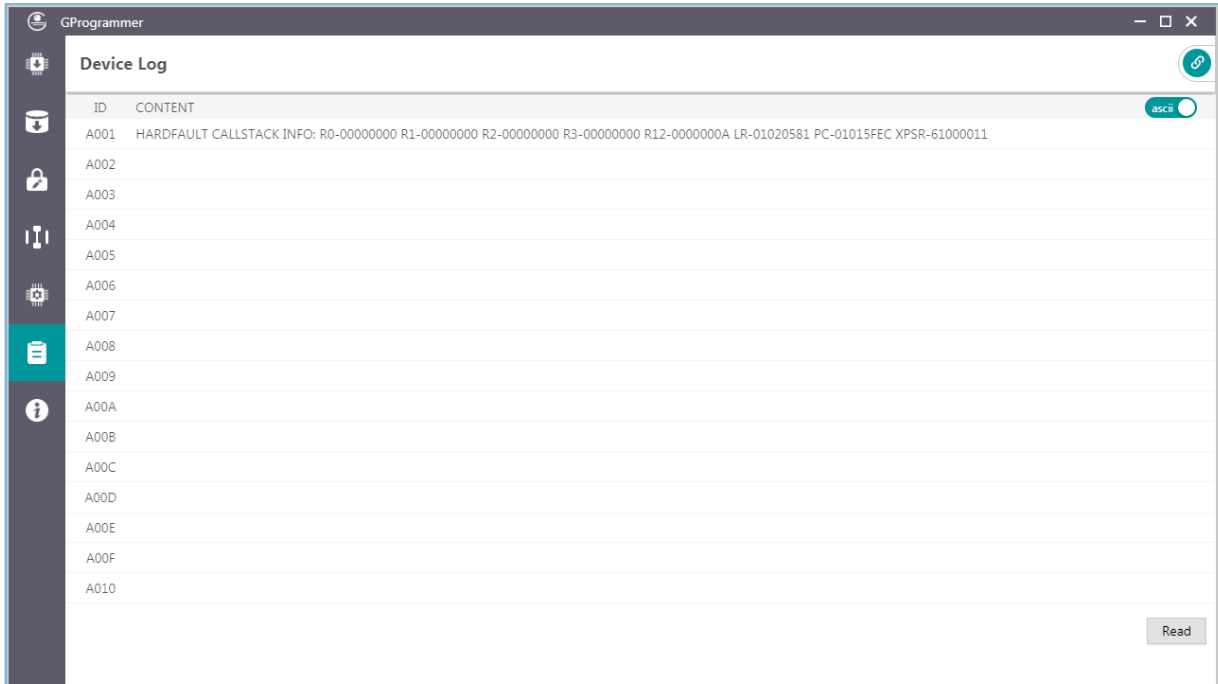

图 3-7 Device Log中的Fault Trace Data

3.2.3 工程中调用API读取

Fault Trace Module提供了读取数据的API，只要在工程中调用相关API，再使用串口输出等方式，就可以得到Fault Trace Data。以下以GR5526芯片为例进行说明，其它芯片同理。

版权所有 © 2023 深圳市汇顶科技股份有限公司

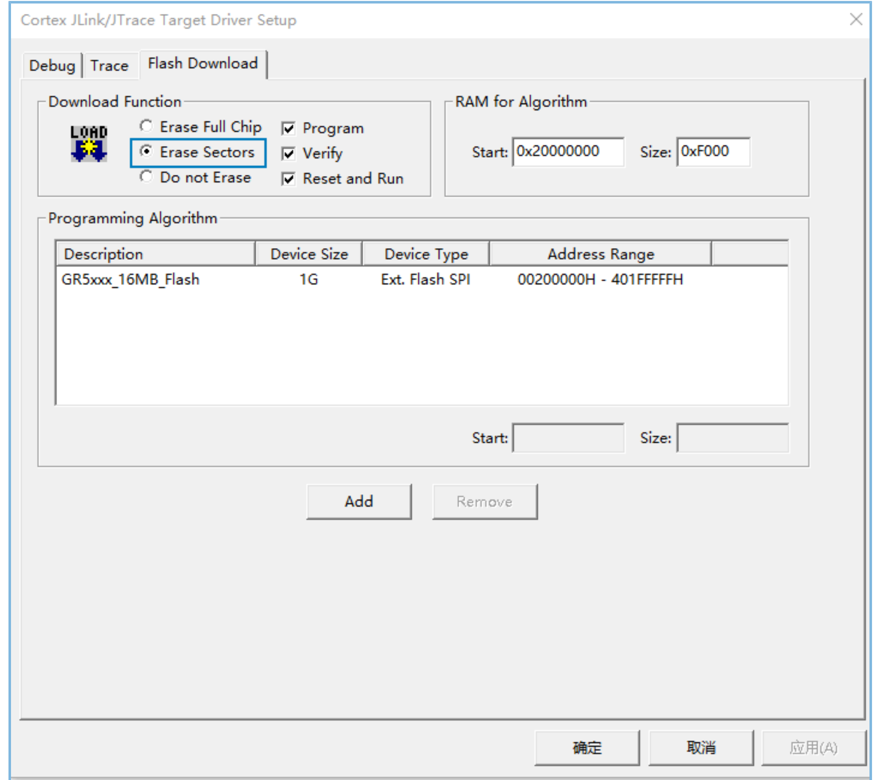

图 3-8 Erase选项设置界面

说明:

如果使用Keil集成环境烧录工程生成的固件，需要将Erase选项设置为“Erase Sectors”，如上图所示。如果选择的是“Erase Full Chip”，将会把Flash中NVDS区域里的Fault Trace Data也一并擦除掉。

在工程中调用API读取Fault Trace Data的实现可参考如下代码。

```
sdk_err_t error_code;
uint8_t fault_trace_data[1000] = {0};
uint32_t data_len = 1000;
error_code = fault_db_records_dump(fault_trace_data, &data_len);
APP_ERROR_CHECK(error_code);
for (uint32_t i = 0; i < data_len; i++)
  {
     APP_LOG_RAW_INFO("%c",fault_trace_data[i]);
  }
```

说明:

需要确保UART模块和APP LOG模块在这读取Fault Trace Data之前已经完成初始化。示例工程中，UART模块和APP LOG模块的初始化一般在app_periph_init()函数中进行，详情可参考对应芯片开发者指南，“修改主函数”。

使用GRUart串口输出工具得到的Fault Trace Data形式如下图所示。

版权所有 © 2023 深圳市汇顶科技股份有限公司

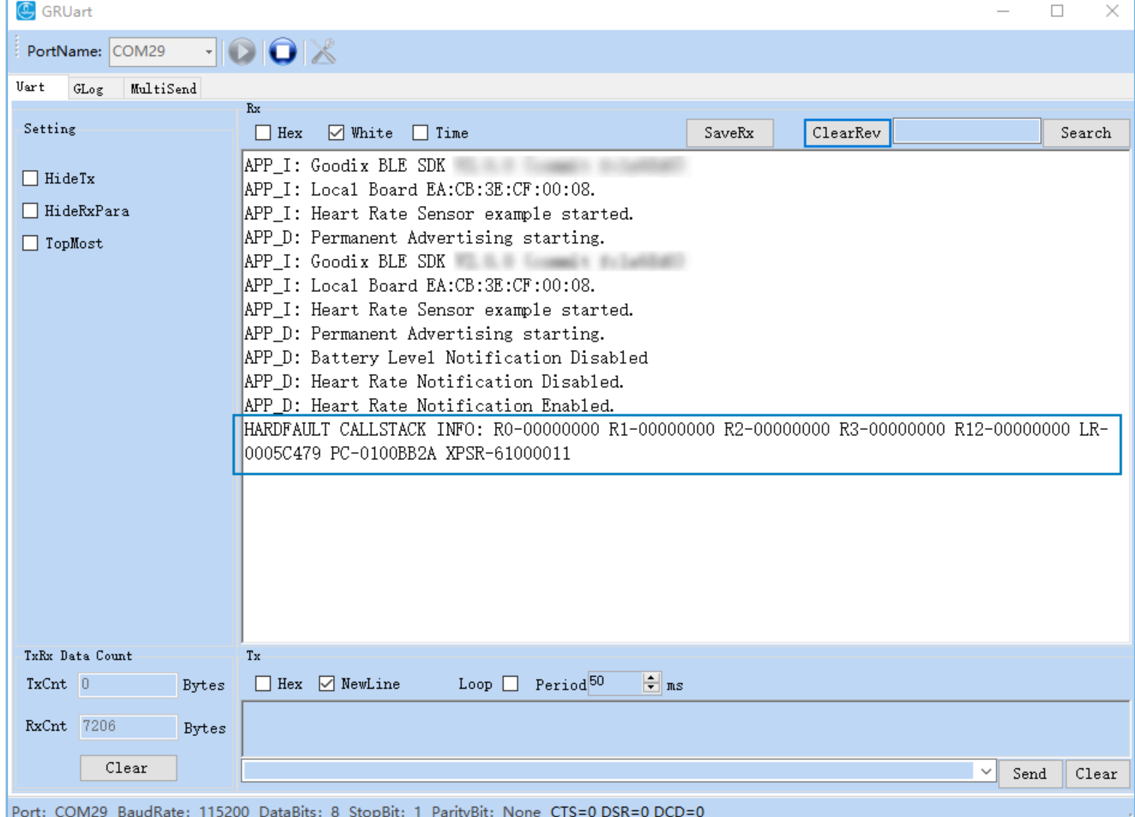

图 3-9 Fault Trace Data串口输出界面

3.3 实例展示

本节以比较常见的HardFault场景为例来展示该模块的作用及有效性。

1. 在ble_app_hrs示例工程代码中添加引起HardFault的代码。

```
static void heartrate_service_process_event(hrs_evt_t *p_hrs_evt)
{
    sdk_err_t error_code;
    switch (p_hrs_evt->evt_type)
    {
        case HRS_EVT_NOTIFICATION_ENABLED:
            error_code = app_timer_start(s_heart_rate_meas_timer_id,
                                         HEART_RATE_MEAS_INTERVAL, NULL);
            APP_ERROR_CHECK(error_code);

            error_code = app_timer_start(s_rr_interval_meas_timer_id,
                                         RR_INTERVAL_INTERVAL, NULL);
            APP_ERROR_CHECK(error_code);
            APP_LOG_DEBUG("Heart Rate Notification Enabled.");

            //Access illegal address
            *(volatile uint32_t*)(0xFFFFFFFF) |= (1 << 0);

            break;
```

版权所有 © 2023 深圳市汇顶科技股份有限公司

```
        ...
      }
}
```

📖 说明:

上方代码在SDK中的路径为：`SDK_Folder\projects\ble\ble_peripheral\ble_app_hrs\Src\user\user_app.c`；在示例工程中的路径为：`GRxx_Soc\user_app\user_app.c`。

其中，加粗部分“*(volatile uint32_t*)(0xFFFFFFFF) |= (1 << 0);”为添加的引起HardFault的代码。

在打开特征Heart Rate通知的处理模块中增加了一行访问非法地址的代码，一旦特征Heart Rate通知被Client端打开，就会引起HardFault异常。

2. 将工程编译生成固件，下载到开发板中。以Debug模式运行该工程，在添加的引起HardFault的代码位置打上断点，再在手机端使用GRToolbox连接该开发板，点击“Heart Rate Measurement”右边的 N 按钮打开Heart Rate通知。

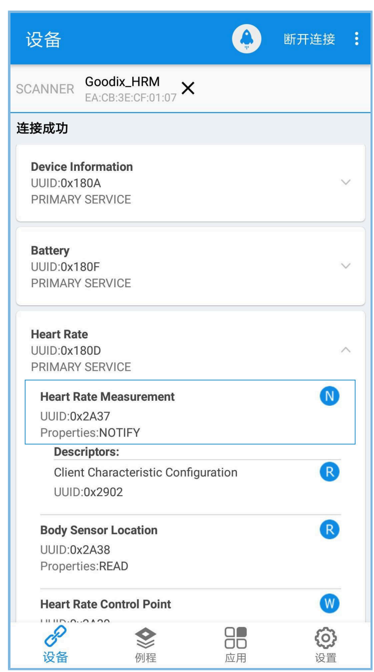

图 3-10 打开Heart Rate通知

如图 3-11所示，工程会在该行代码处停止运行，可以从左侧Register栏观察现场的寄存器值。

版权所有 © 2023 深圳市汇顶科技股份有限公司

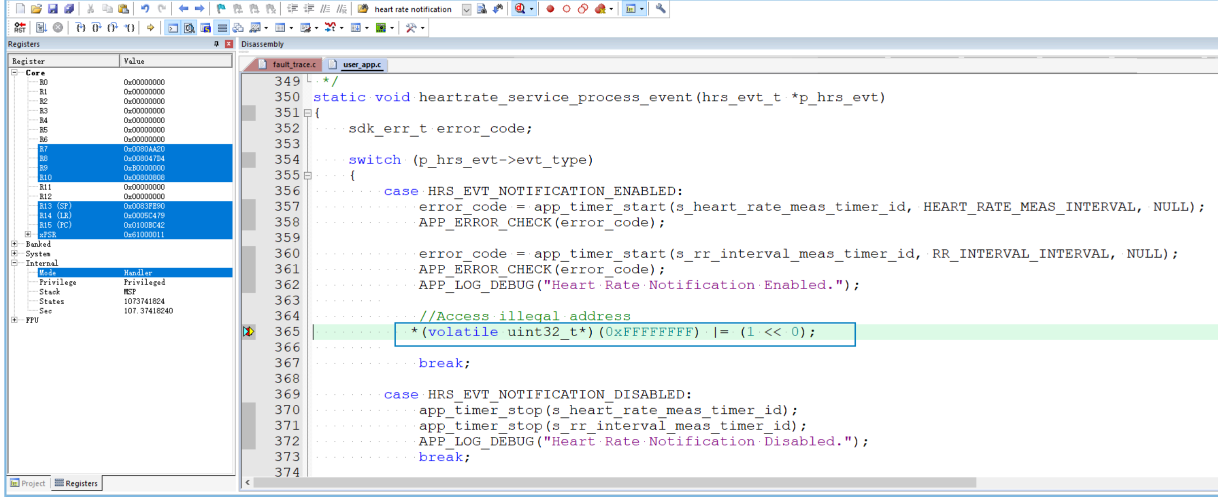

图 3-11 即将发生HardFault的调试界面

3. 按下快捷键F11单步运行，工程进入HardFault异常函数。HardFault前的现场数据将存入NVDS中。

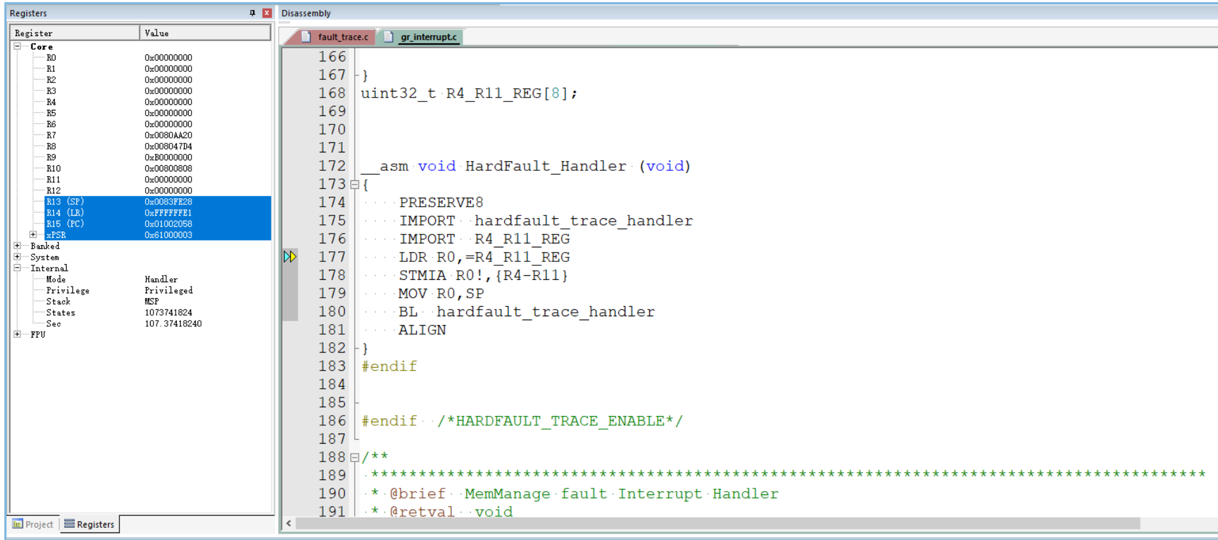

图 3-12 进入HardFault异常函数

4. 退出调试模式后重置开发板。按照3.2.1 蓝牙连接读取中的方法使用蓝牙连接读取该开发板的Fault Trace Data，得到如下结果：

图 3-13 蓝牙连接读取Fault Trace Data

对比调试界面中和Fault Trace Data中的寄存器值，可以发现两者是完全相同的。说明Fault Trace Module记录的是HardFault现场信息。

版权所有 © 2023 深圳市汇顶科技股份有限公司

4 模块详解

Fault Trace Module的读写实现是基于NVDS系统的一系列API来实现的，本章将重点介绍HardFault和Assert Fault的信息追踪方法，以及蓝牙控制模块的实现方法。

4.1 HardFault Data Trace

当HardFault异常发生时，现场的寄存器PSR、R15（PC）、R14（LR）、R3、R2、R1、R0由处理器硬件控制，被依次压入栈中，并进入异常处理函数HardFault_Handler。

说明:

下方代码在SDK中的路径为：`SDK_Folder\platform\soc\common\gr_interrupt.c`；在示例工程目录中的路径为：`GRxx_Soc\gr_arch\gr_interrupt.c`。

ENABLE_BACKTRACE_FEA宏默认设置为0，表示使用Fault Trace 模块记录异常信息；如将ENABLE_BACKTRACE_FEA设为1，则使用cortex_backtrace模块记录异常信息。

```
SECTION_RAM_CODE __asm void HardFault_Handler (void)
{
#if (ENABLE_BACKTRACE_FEA == 0)//use fault trace module
    PRESERVE8
    IMPORT  hardfault_trace_handler
    IMPORT  R4_R11_REG
    LDR R0,=R4_R11_REG
    STMIA R0!,{R4-R11}
    MOV R0,SP
    BL  hardfault_trace_handler
#elif (ENABLE_BACKTRACE_FEA == 1)//use cortex_backtrace module
    PRESERVE8
    IMPORT  cortex_backtrace_fault_handler
    MOV     r0, lr
    MOV     r1, sp
    BL      cortex_backtrace_fault_handler
#endif

Fault_Loop
    BL      Fault_Loop
    ALIGN
}
```

在HardFault_Handler中，寄存器R4 ～ R11的HardFault现场值被保存在全局数组R4_R11_REG中。SP指针被赋给R0寄存器，使SP指针作为接下来被调用的函数hardfault_trace_handler的参数。

说明:

下方代码在SDK中的路径为：`SDK_Folder\components\libraries\fault_trace\fault_trace.c`；在示例工程中的路径为：`GRxx_Soc\gr_libraries\fault_trace.c`。

```
void hardfault_trace_handler(unsigned int sp)
{
    unsigned int stacked_r0;
    unsigned int stacked_r1;
    unsigned int stacked_r2;
    unsigned int stacked_r3;
    unsigned int stacked_r12;
    unsigned int stacked_lr;
    unsigned int stacked_pc;
    unsigned int stacked_psr;

    stacked_r0  = ((unsigned long *)sp)[0];
    stacked_r1  = ((unsigned long *)sp)[1];
    stacked_r2  = ((unsigned long *)sp)[2];
    stacked_r3  = ((unsigned long *)sp)[3];
    stacked_r12 = ((unsigned long *)sp)[4];
    stacked_lr  = ((unsigned long *)sp)[5];
    stacked_pc  = ((unsigned long *)sp)[6];
    stacked_psr = ((unsigned long *)sp)[7];

    memset(s_fault_info, 0, FAULT_INFO_LEN_MAX);

    sprintf(s_fault_info,
           "HARDFAULT CALLSTACK INFO: R0-%08X R1-%08X R2-%08X R3-%08X R12-%08X LR-%08X
            PC-%08X XPSR-%08X\r\n",
            stacked_r0, stacked_r1, stacked_r2, stacked_r3, stacked_r12, stacked_lr,
            stacked_pc, stacked_psr);

    fault_db_record_add((uint8_t *)s_fault_info, strlen(s_fault_info));
}
```

hardfault_trace_handler函数根据其参数，即SP指针，从栈中取出存入的寄存器值。并将其写入数组s_fault_info中，再调用fault_db_record_add函数写入到NVDS中。

HardFault产生的Fault Trace Data形式如下所示：

```
HARDFAULT  CALLSTACK  INFO:  R0-00000000  R1-00000000  R2-00000000  R3-00000000  R12-00000000
 LR-0005C479 PC-0100BC42 XPSR-61000011
```

分别记录了HardFault现场的R0、R1、R2、R3、R12、R14（LR）、R15（PC）、PSR（XPSR）寄存器值。

说明:

在异常处理函数HardFault_Handler中，寄存器R4 ～ R11的HardFault现场值被保存在全局数组R4_R11_REG中。用户可根据自身需求，更改hardfault_trace_handler()函数，将其一起写入NVDS中。

4.2 Assert Fault Data Trace

断言（Assert）作为一种软件调试的方法，提供了一种在代码中进行正确性检查的机制。SDK中的Assert模块位于`SDK_Folder\components\libraries\app_assert`。

📖 **说明:**

下方代码在SDK中的路径为：SDK_Folder\components\libraries\app_assert\app_assert.c；在示例工程中的路径为：GRxx_Soc\gr_libraries\app_assert.c。

```
#define APP_ASSERT_CHECK(EXPR)                                    \
    do                                                            \
    {                                                             \
        if (!(EXPR))                                              \
        {                                                         \
            app_assert_handler(#EXPR, __FILE__, __LINE__);        \
        }                                                         \
    } while(0)
```

当调用APP_ASSERT_CHECK(EXPR)进行断言，且其参数EXPR值为0时，会调用处理函数app_assert_handler。

📖 **说明:**

下方代码在SDK中的路径为：SDK_Folder\components\libraries\app_assert\app_assert.c；在示例工程中的路径为：GRxx_Soc\gr_libraries\app_assert.c。

```
void app_assert_handler(const char *expr, const char *file, int line)
{
    if (s_assert_cbs.assert_err_cb)
    {
        s_assert_cbs.assert_err_cb(expr, file, line);
    }
}
```

处理函数中会调用回调函数，将现场信息通过串口输出。

```
static sys_assert_cb_t s_assert_cbs =
{
    .assert_err_cb   = app_assert_err_cb ,
    .assert_param_cb = app_assert_param_cb,
    .assert_warn_cb  = app_assert_warn_cb,
};
```

Assert模块实现了三种回调函数，对应不同的Assert参数格式及现场信息，（具体请参考app_assert源码，位于*app_assert.c*文件中）。Assert处理函数app_assert_handler中默认设置为调用assert_err_cb函数。用户可根据自身需求更改app_assert_handler函数中的实现，调用其他回调函数。

Assert模块的回调函数都会将现场信息通过串口输出。Fault Trace Module重新实现了Assert模块的三种回调函数（原回调函数的实现为weak函数），将现场信息存储到NVDS区域中，其中回调函数assert_err_cb实现如下。

版权所有 © 2023 深圳市汇顶科技股份有限公司

📖 说明:

下方代码在SDK中的路径为：SDK_Folder\components\libraries\fault_trace\fault_trace.c；在示例工程中的路径为：GRxx_Soc\gr_libraries\fault_trace.c。

```
SECTION_RAM_CODE static void app_assert_err_cb(const char *expr,
                                               const char *file, int line)
{
    __disable_irq();

    uint32_t       expre_len     = 0;
    uint32_t       file_name_len = 0;

    file_name_len=(ASSERT_FILE_NAME_LEN < strlen(file)) ?
                   ASSERT_FILE_NAME_LEN:strlen(file);
    expre_len = (ASSERT_EXPR_NAME_LEN < strlen(expr)) ?
                 ASSERT_EXPR_NAME_LEN : strlen(expr);

    memset(&s_assert_info, 0, sizeof(assert_info_t));
    memcpy(s_assert_info.file_name, file, file_name_len);
    memcpy(s_assert_info.expr, expr, expre_len);

    s_assert_info.assert_type = ASSERT_ERROR;
    s_assert_info.file_line   = line;

    assert_info_save(&s_assert_info);
    while(1);
}
```

assert_err_cb()函数将param实参名、调用APP_ASSERT_CHECK的函数名及路径、行数信息存入结构体，最终调用NVDS的API将其以特定格式存入NVDS中。

Assert Fault产生的Fault Trace Data示例如下所示：

```
(..\Src\user\user_app.c: 638) [ERROR] param
```

表明Assert Fault的位置（..\Src\user\user_app.c: 638）、类型（ERROR）、实参名（param）。

4.3 蓝牙控制实现

通过蓝牙连接来控制开发板上的Fault Trace Module，是基于LNS服务（Log Notification Service）实现的。LNS 提供了特定的特征（Characteristic）来完成控制命令的接收及数据发送。

LNS的特征包括Log Information、Log Control Point，如表 4-1 所示。

表 4-1 LNS Characteristic说明

Characteristic	UUID	Type	Support	Security	Properties
Log Information	A6ED0802-D344-460A-8075-B9E8EC90D71B	128 bits	Mandatory	None	Notify

Characteristic	UUID	Type	Support	Security	Properties
Log Control Point	A6ED0803-D344-460A-8075-B9E8EC90D71B	128 bits	Mandatory	None	Write, Indicate

- Log Information：用于发送Fault Trace Data（Notify）。
- Log Control Point：用于接收指令（Write）和返回信息（Indicate）。

下文将通过介绍LNS 的具体实现来说明蓝牙控制Fault Trace Module的原理。

说明:

下方代码在SDK中的路径为：SDK_Folder\components\profiles\lns\lns.c；在示例工程目录中的路径为：GRxx_Soc\gr_profiles\lns.c。

```
static void lns_write_att_evt_handler(uint8_t conn_idx,
                                      const ble_gatts_evt_write_t *p_param)
{
    ...
    switch (tab_index)
    {
        ...
        case LNS_IDX_LOG_CTRL_PT_VAL:
        {
            switch (p_param->value[0])
            {
                case LNS_CTRL_PT_TRACE_STATUS_GET:
                    event.evt_type = LNS_EVT_TRACE_STATUS_GET;
                    break;

                case LNS_CTRL_PT_TRACE_INFO_DUMP:
                    event.evt_type = LNS_EVT_TRACE_INFO_DUMP;
                    break;

                case LNS_CTRL_PT_TRACE_INFO_CLEAR:
                    event.evt_type = LNS_EVT_TRACE_INFO_CLEAR;
                    break;

                default:
                    break;
            }
        }
    ...
    if (BLE_ATT_ERR_INVALID_HANDLE != cfm.status && LNS_EVT_INVALID != event.evt_type)
    {
        lns_evt_handler(&event);
    }
}
```

lns_write_att_evt_handler()是LNS 被写入的回调函数。Client端向特征Log Control Point写入LNS_CTRL_PT_TRACE_STATUS_GET（0x01）、LNS_CTRL_PT_TRACE_INFO_DUMP（0x02）、LNS_CTRL_PT_TRACE_INFO_CLEAR（0x03），分别可以引起一种类型的事件（event.evt_type），并调用注册的事件处理函数lns_evt_handler()。

说明:

下方代码在SDK中的路径为：SDK_Folder\components\profiles\lns\lns.c，在示例工程中的路径为：GRxx_Soc\gr_profiles\lns.c。

```
static void lns_evt_handler(lns_evt_t *p_evt)
{
    uint8_t trace_log_num  = 0;

    switch (p_evt->evt_type)
    {
        case LNS_EVT_TRACE_STATUS_GET:
            trace_log_num = fault_db_records_num_get();
            lns_log_status_send(p_evt->conn_idx, trace_log_num);
            break;

        case LNS_EVT_TRACE_INFO_DUMP:
            lns_log_info_send(p_evt->conn_idx);
            break;

        case LNS_EVT_TRACE_INFO_CLEAR:
            fault_db_record_clear();
            break;
    }

    if (LNS_EVT_INVALID != p_evt->evt_type && s_lns_env.evt_handler)
    {
        s_lns_env.evt_handler(p_evt);
    }
}
```

在LNS的事件处理函数中，不同的事件类型将调用对应的功能函数。将Client端向特征Log Control Point的写入值与事件类型联系起来。

- 写入0x01，将调用fault_db_records_num_get()函数和lns_log_status_send()函数，分别实现读取Fault Trace Data的数目及发送该值给对端。
- 写入0x02，将调用lns_log_info_send()函数，读取并发送Fault Trace Data到对端。
- 写入0x03，将调用fault_db_record_clear()函数，清空Fault Trace Data。

如图 3-5所示，GRToolbox界面上的“计数”、“读取”、“清除”按键是向Slave LNS服务中的Log Control Point特征写入0x01，0x02，0x03。用户可以通过打开Log Information和Log Control Point特征的通知，再写入对应值，作为另一种等效的操作方案。

5 常见问题

本章描述了在使用Fault Trace Module时，可能出现的问题、原因及处理方法。

5.1 使用GProgrammer读取Fault Trace Data失败

- 问题描述

  使用GProgrammer读取Fault Trace Data失败，USER Parameters栏没有读取到信息。

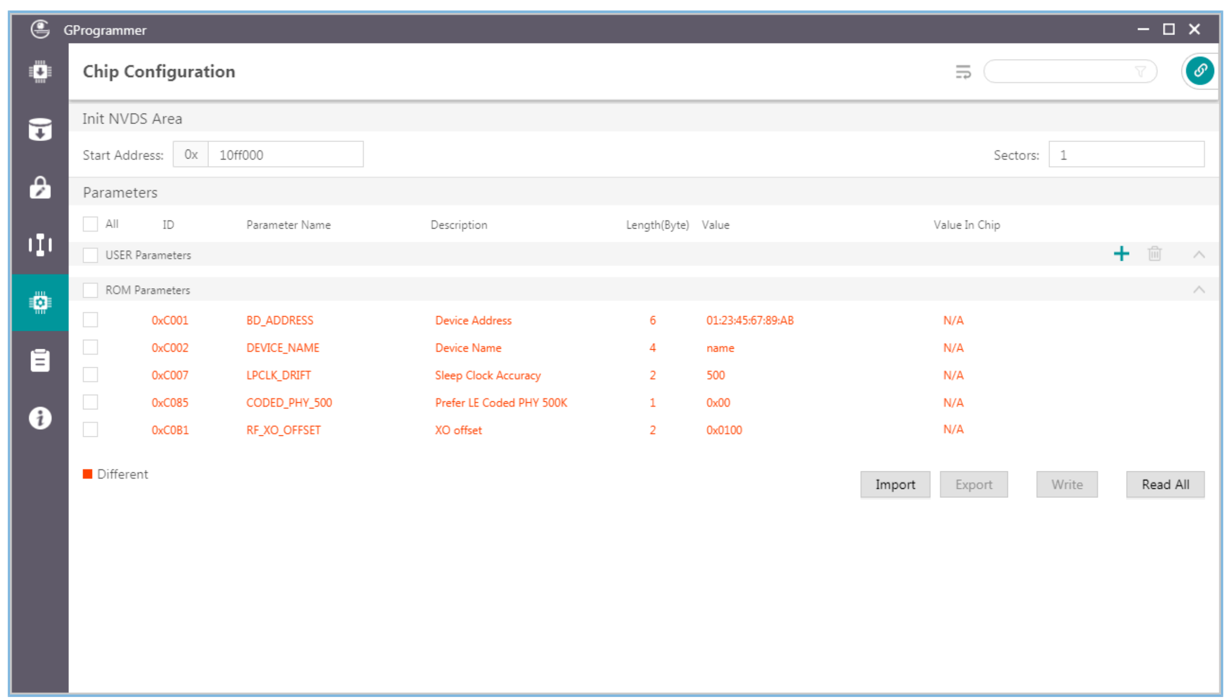

图 5-1 USER Parameters读取失败

- 问题分析

  界面上的“Start Address”与芯片实际的NVDS“Start Address”不一致。

- 处理方法

  检查界面上的“Start Address”是否为当前芯片真实的NVDS起始地址。GProgrammer启动时，会根据用户选择的芯片型号设置NVDS默认空间，GR551x默认为芯片Flash的最后4 K，其他型号芯片默认为芯片Flash的最后8 K。若用户重新分配了NVDS区域，则需要将“Start Address”值设置为对应值。

5.2 工程中调用API读取Fault Trace Data失败

- 问题描述

  工程中调用API读取Fault Trace Data失败，且串口工具GRUart上无串口信息输出。若对函数fault_db_records_dump返回值使用APP_ERROR_CHECK检查，串口会有如图 5-2所示输出：

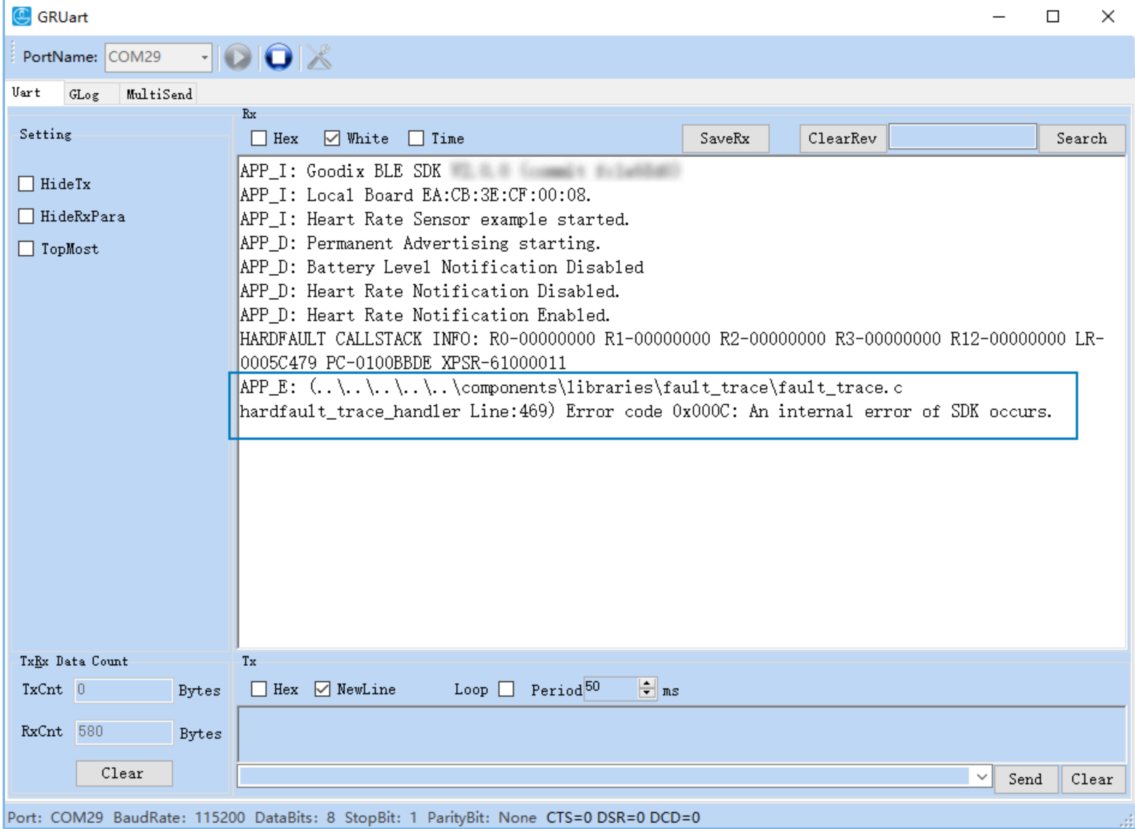

图 5-2 返回值检测失败的串口输出

- 问题分析

  用于存放Fault Trace Data的Buffer长度不够，导致读取失败。

- 处理方法

  增大用于存放Fault Trace Data的Buffer长度。

版权所有 © 2023 深圳市汇顶科技股份有限公司