GR5xx Fault Trace Module Application Note

Version: 3.0

Release Date: 2023-03-30

Shenzhen Goodix Technology Co., Ltd.

Trademarks and Permissions

GOODiX and other Goodix trademarks are trademarks of Shenzhen Goodix Technology Co., Ltd. All other trademarks and trade names mentioned in this document are the property of their respective holders.

Disclaimer

Information contained in this document is intended for your convenience only and is subject to change without prior notice. It is your responsibility to ensure its application complies with technical specifications.

Shenzhen Goodix Technology Co., Ltd. (hereafter referred to as “Goodix”) makes no representation or guarantee for this information, express or implied, oral or written, statutory or otherwise, including but not limited to representation or guarantee for its application, quality, performance, merchantability or fitness for a particular purpose. Goodix shall assume no responsibility for this information and relevant consequences arising out of the use of such information.

Without written consent of Goodix, it is prohibited to use Goodix products as critical components in any life support system. Under the protection of Goodix intellectual property rights, no license may be transferred implicitly or by any other means.

Shenzhen Goodix Technology Co., Ltd.

Headquarters: Floor 12-13, Phase B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828 Zip Code: 518000

Website: www.goodix.com

Preface

Purpose

This document introduces the functionalities, operating mechanisms, and applications of Bluetooth Low Energy (Bluetooth LE) GR5xx Fault Trace Module, to help users quickly get started with the Module.

Audience

This document is intended for:

- GR5xx user
- GR5xx developer
- GR5xx tester
- Technical writer

Release Notes

This document is the second release of *GR5xx Fault Trace Module Application Note*, corresponding to Bluetooth LE GR5xx System-on-Chip (SoC) series.

Revision History

Version	Date	Description
1.0	2023-01-10	Initial release
3.0	2023-03-30	Updated descriptions about GR5xx SoCs.

Contents

1 Introduction

GR5xx Fault Trace Module aims to help developers identify problems during Bluetooth application development. When GR5xx firmware fails to operate normally, GR5xx Fault Trace Module can write fault trace data to the Non-Volatile Data Storage (NVDS) in Flash, and export the fault trace data from NVDS, so as to restore the failure scenario and help identify problems.

GR5xx Fault Trace Module can write fault trace data to NVDS in the following two scenarios:

- When a HardFault occurs, the Fault Trace Module can write the current values of the internal registers to the NVDS.
- When Assert faults occur, the Fault Trace Module can write the function names, the number of code lines, parameter names, and other relevant information to the NVDS.

Before getting started, you can refer to the following documents.

Table 1-1 Reference documents

Name	Description
Developer guide of the specific GR5xx SoC	Introduces GR5xx Software Development Kit (SDK) and how to develop and debug applications based on the SDK.
J-Link/J-Trace User Guide	Provides J-Link operational instructions. Available at https://www.segger.com/downloads/jlink/UM08001_JLink.pdf .
Keil User Guide	Offers detailed Keil operational instructions. Available at https://www.keil.com/support/man/docs/uv4/.
Bluetooth Core Spec	Offers official Bluetooth standards and core specification from Bluetooth SIG.
Bluetooth GATT Spec	Provides details about Bluetooth profiles and services. Available at https://www.bluetooth.com/specifications/gatt.
GProgrammer User Manual	Lists GProgrammer operational instructions, including downloading firmware to and encrypting firmware on GR5xx System-on-Chips (SoCs).

2 Environment Setup

This chapter introduces how to rapidly set up an operating environment for GR5xx Fault Trace Module.

Note:

SDK_Folder is the root directory of the GR5xx SDK in use.

2.1 Preparation

- **Hardware preparation**

Table 2-1 Hardware preparation

Name	Description
Development board	Starter Kit Board (SK Board) of the corresponding SoC
Connection cable	USB Type C cable (Micro USB 2.0 cable for GR551x SoCs)
Android phone	Phones running on Android 5.0 (KitKat) and later versions

- **Software preparation**

Table 2-2 Software preparation

Name	Description
Windows	Windows 7/Windows 10
J-Link driver	A J-Link driver. Available at https://www.segger.com/downloads/jlink/.
Keil MDK-ARM IDE (Keil)	An integrated development environment (IDE). MDK-ARM Version 5.20 or later is required. Available at https://www.keil.com/demo/eval/arm.htm.
GProgrammer (Windows)	A programming tool. Available in SDK_Folder\tools\GProgrammer.
GRUart (Windows)	A serial port debugging tool. Available in SDK_Folder\tools\GRUart.
GRToolbox (Android)	A Bluetooth LE debugging tool. Available in SDK_Folder\tools\GRToolbox.

3 Fault Trace Module in Application

This chapter introduces how to add GR5xx Fault Trace Module to a project and how to use the Module by taking the heart rate example project ble_app_hrs for example.

3.1 Importing Data to Fault Trace Module to Target Project

Fault Trace Module is optional for running a GR5xx-SoC-based project. Before using the Module, add the files of Fault Trace Module to the project directory of a user application, enable the macro switch of the Module, and initialize the Module.

3.1.1 Adding the Module

1. Open the heart rate example project ble_app_hrs.

   The source code and project file of the heart rate example project are in SDK_Folder\projects\ble\ble_peripheral\ble_app_hrs, and project file is in the Keil_5 folder.

2. Add the source files of Fault Trace Module to the project directory of ble_app_hrs.

Note:

By default, the source files of Fault Trace Module are added to the project directory of ble_app_hrs provided in GR5xx SDK currently.

   The source files *fault_trace.c* and *cortex_backtrace.c* of Fault Trace Module are in SDK_Folder\components\libraries\fault_trace and SDK_Folder\components\libraries\app_error respectively.

   Select and right-click gr_libraries. Choose **Add Existing Files to Group "gr_libraries"** to add *fault_trace.c* and *cortex_backtrace.c* to gr_libraries. The directory is then shown as below:

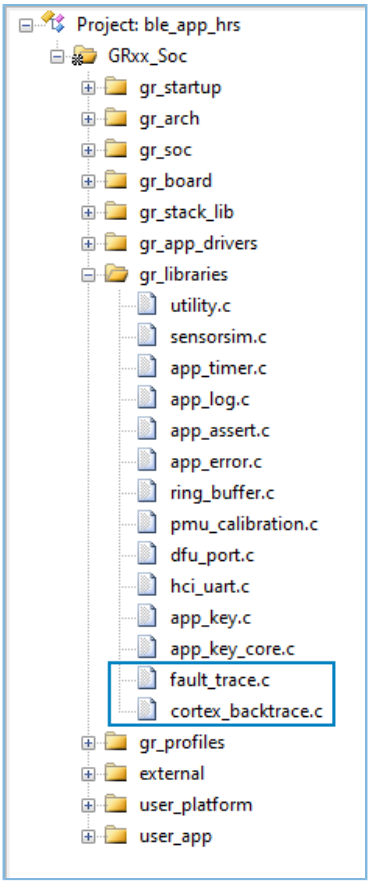

Figure 3-1 Adding source files of Fault Trace Module to the project directory

3.1.2 Enabling the Module

Open `user_app\custom_config.h` in the directory, and set SYS_FAULT_TRACE_ENABLE to 1 to enable the Module.

3.1.3 Initializing the Module

To initialize the Module, open `user_platform\user_periph_setup.c` in the project directory, and call fault_trace_db_init() in app_periph_init().

```
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
}
```

Note:

It is not recommended to use the Fault Trace Module for DTM tests, to avoid test conflicts. If you need to use the Module, set DTM_TEST_ENABLE to 0 in *custom_config.h*.

By default, the Module has been initialized by calling fault_trace_db_init() in ble_app_hrs.

After initializing the Module, program the firmware generated based on the compiled program to the SK Board by following instructions in the developer guide of the specific GR5xx SoC.

If a HardFault or an Assert fault occurs on the SK Board during project running, relevant fault trace data will be stored in the NVDS of the GR5xx SoC. The data will be kept until you do global erase on the Flash of the SoC.

3.2 Reading Fault Trace Data

You can read the fault trace data in the NVDS by following any of the three approaches:

1. Open GRToolbox on an Android phone, and get the fault trace data from the NVDS of the SK Board via Bluetooth.
2. Read the fault trace data from the NVDS of the SK Board by using GProgrammer on a PC.
3. Call the relevant APIs in the project to read the fault trace data.

Note:

Before you get fault trace data via Bluetooth or by calling the APIs, make sure the Fault Trace Module is added to the firmware of the SK Board. This is not required if you choose to get the data through GProgrammer.

3.2.1 Reading Data via Bluetooth

To read data via Bluetooth, make sure the Fault Trace Module is added and the Log Notification Service (LNS) is running on the SK Board.

If LNS is not available on the SK Board, add LNS to ble_app_hrs.

The source file of LNS is in SDK_`Folder\components\profile\lns`.

Select and right-click gr_profiles under the project directory of ble_app_hrs. Choose **Add Existing Files to Group gr_profiles** to add *lns.c* to gr_profiles. The directory is then shown as below:

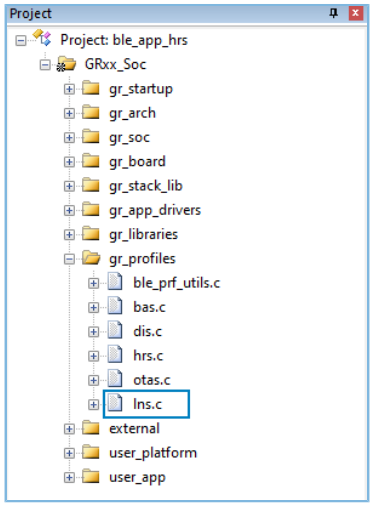

Figure 3-2 Adding the LNS file to the project directory

Call lns_service_init() in services_init() to initialize the LNS. services_init() is in SDK_Folder\projects\ble\ble_peripheral\ble_app_hrs\Src\user\user_app.c.

Note:

By default, the LNS is added to and initialized in ble_app_hrs.

After the project is properly configured, run the firmware generated from the example project on SK Board.

1. Open GRToolbox on an Android phone and connect the phone with the SK Board. **Log Notification Service** is then displayed, as shown in Figure 3-3.

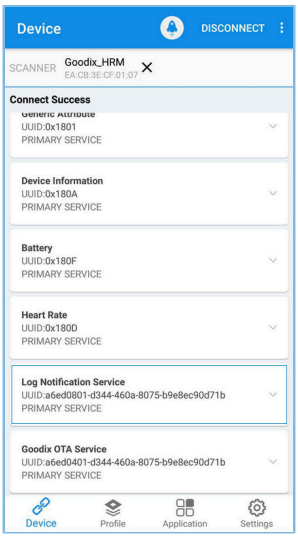

Figure 3-3 Successful discovery of LNS after connecting the phone to the Board on GRToolbox

Note:

Screenshots of GRToolbox in this document are for reference only, to help users better understand the software operation. In the case of interface differences due to version changes, the interface of GRToolbox in practice shall prevail.

2. Tap ⋮ in the upper-right corner; choose **Request MTU** to set the value to **512**, and then choose **Dump Crash Log**, as shown in Figure 3-4.

Note:

The Maximum Transmission Unit (MTU) of data interaction between devices is 23 bytes by default, whereas the fault trace data exceeds 23 bytes. Therefore, you need to set MTU larger (recommended: 512 bytes) before obtaining the fault trace data.

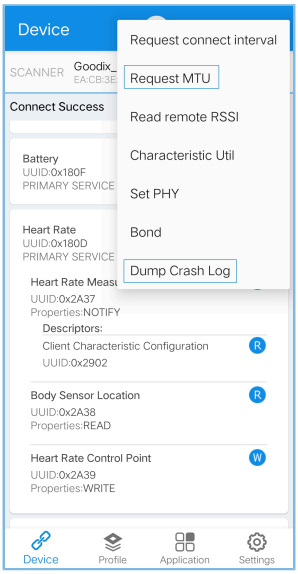

Figure 3-4 To set MTU and read crash logs

3. Tap **READ** in the **Dump Crash Log** pop-up box, to read the fault trace data stored on the SK board.

Figure 3-5 Successfully reading crash logs

3.2.2 Reading Data Through GProgrammer

Connect the PC with the SK Board that you wish to read fault trace data from, and start GProgrammer on the PC.

Click on the left bar of the main user interface of GProgrammer, to enter the **Device Log** page.

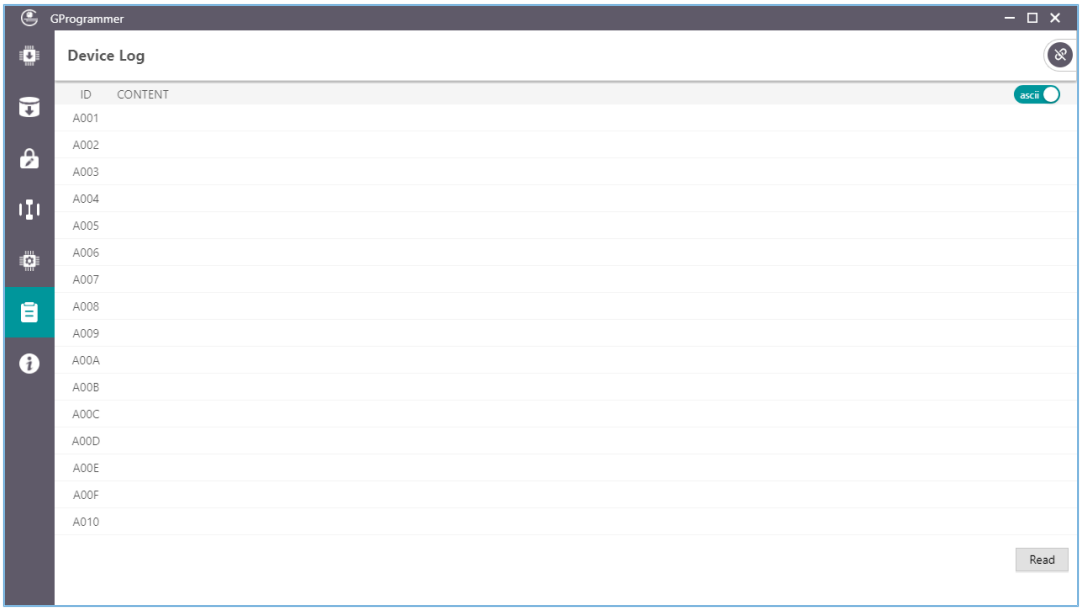

Figure 3-6 **Device Log** on GProgrammer

Note:

GProgrammer screenshots in this document are used to help users better understand operational steps only. The user interface of GProgrammer in actual use prevails.

Click **Read** in the bottom-right corner of the page, to read the fault trace data from the NVDS of the SK Board, as shown in Figure 3-7:

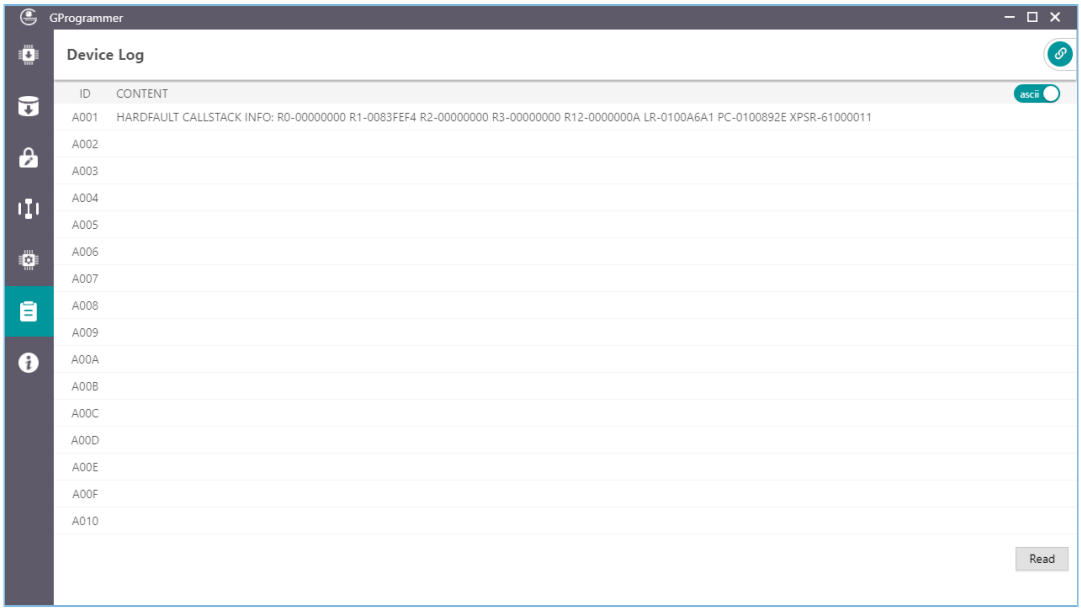

Figure 3-7 Fault trace data displayed on the **Device Log** page

3.2.3 Reading Data by Calling APIs in the Project

Fault Trace Module provides you with APIs to read the data. To get the fault trace data, you need to call the relevant APIs in the project, and output data through serial ports. This process is explained by taking a GR5526 SoC as an example, which also applies to other SoCs.

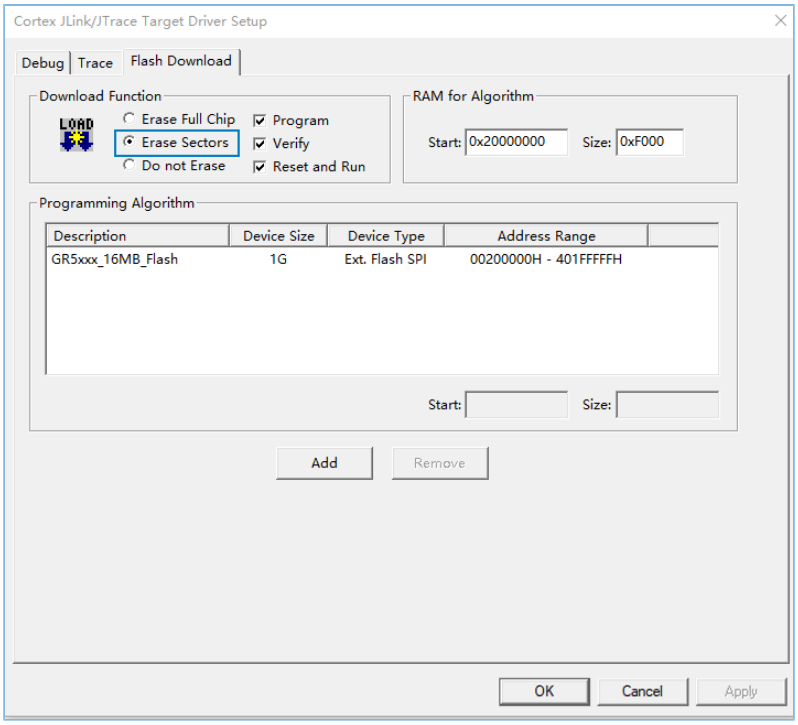

Figure 3-8 Setting the erase type

Note:

If the firmware is programmed with the project in Keil MDK, select **Erase Sectors** for erase, as shown in the figure above. Avoid selecting **Erase Full Chip**, which will also erase the fault trace data in the NVDS of the SK Board.

The code below is an example to read fault trace data by calling APIs in the project.

```
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]);
}
```

Note:

Make sure the UART module and the APP LOG module have been initialized before reading fault trace data. In the example project, the two modules are initialized by running app_periph_init(). For more information, see “Modifying the main() Function” in the developer guide of the specific GR5xx SoC.

An example of fault trace data obtained through GRUart is displayed in the figure below.

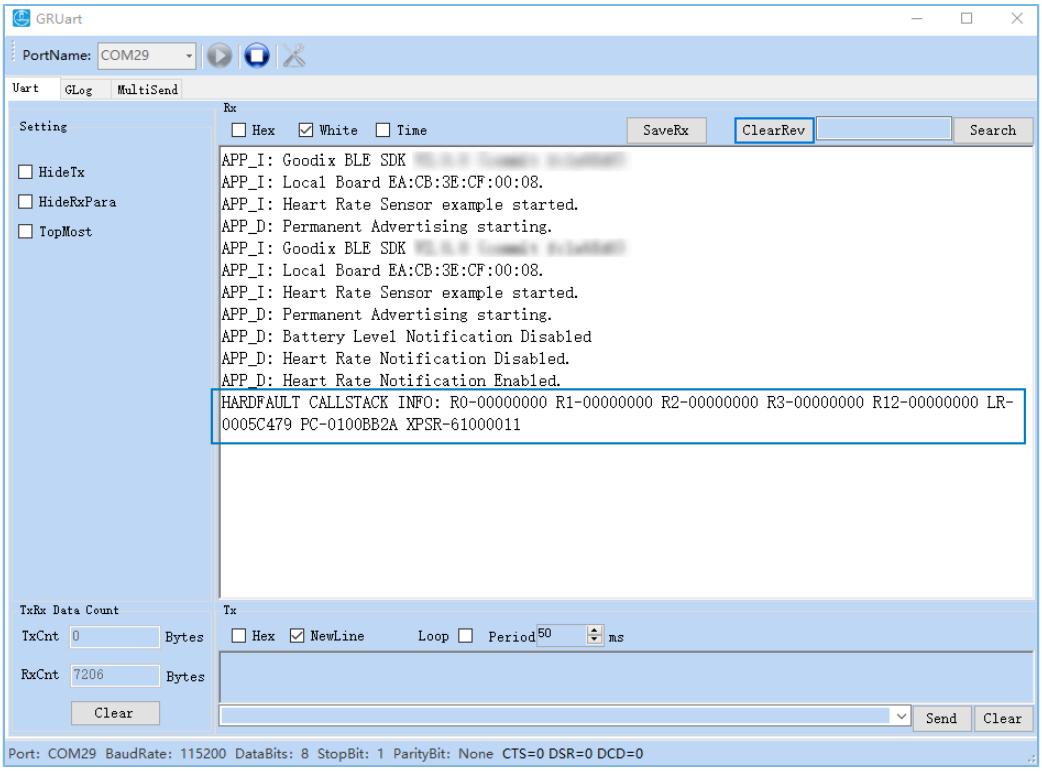

Figure 3-9 Fault trace data displayed on GRUart

3.3 Operation Demonstration

This section explains the functionalities of Fault Trace Module by taking the HardFault scenario, a common scenario as an example.

1. Add the code that causes the HardFault to the code of ble_app_hrs.

```
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;
     ...
    }
}
```

Note:

The code above is available in SDK_Folder\projects\ble\ble_peripheral\ble_app_hrs\Src\user\user_app.c in the SDK, and available in GRxx_Soc\user_app\user_app.c in the example project directory.

The code lines in bold ***(volatile uint32_t*)(0xFFFFFFFF) |= (1 << 0);** are newly added which explain the cause of the HardFault.

Enable the **Heart Rate** notification and a new code line that leads to access to the invalid address is displayed. A HardFault will occur after you view the **Heart Rate** notification on GRToolbox.

2. Compile the project and generate the firmware. Download the firmware to the SK Board. Run the project in debug mode, and set breakpoints on the code line where the HardFault is caused. Connect the phone with the SK Board on GRToolbox, and tap (N) on the right of **Heart Rate Measurement** to view **Heart Rate** notification.

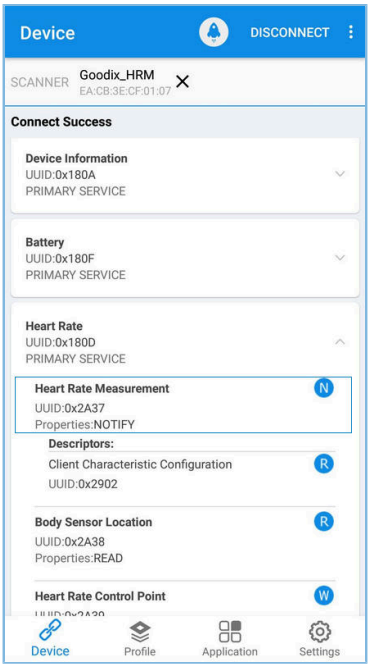

Figure 3-10 To view **Heart Rate** notification

By doing so, the project stops running at the breakpoint. Values of the registers are shown in the left pane, as shown in Figure 3-11.

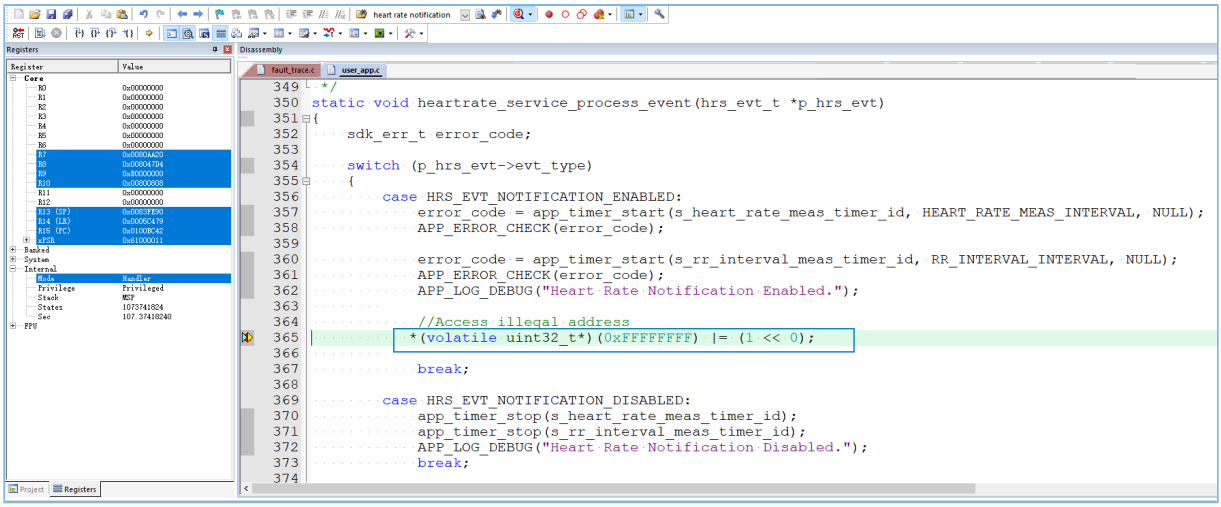

Figure 3-11 Debug interface before the HardFault occurs

3. Press F11 to step through code running. The project runs into the function that causes the HardFault. Fault trace data before the HardFault occurs is stored in the NVDS.

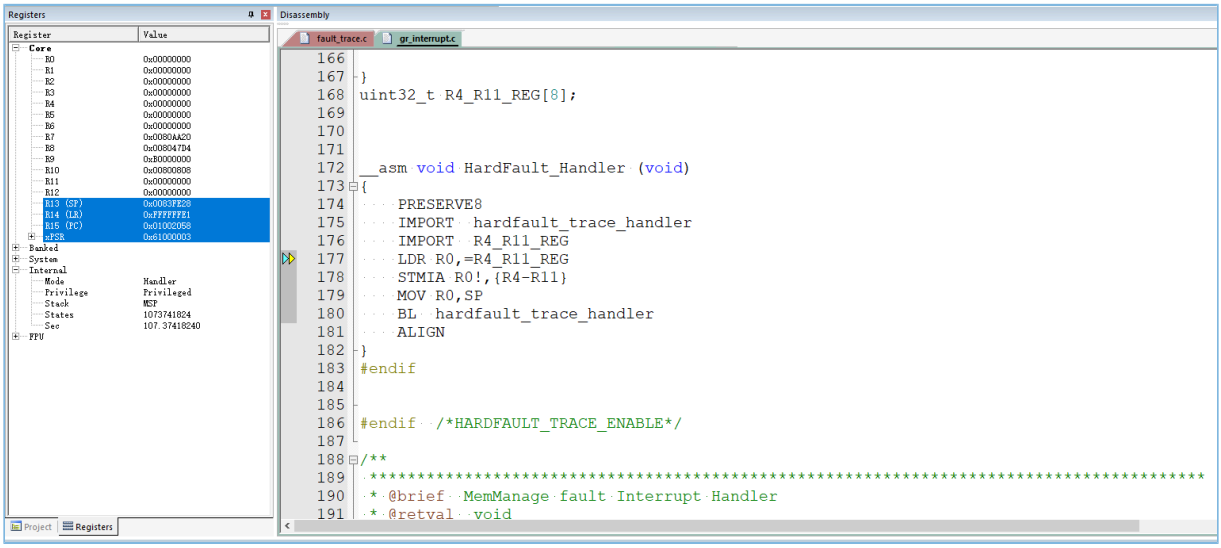

Figure 3-12 Entering the function causing HardFault

4. Reset the SK Board after the Board exiting the debug mode. Read the fault trace data from the Board through Bluetooth (by following the instructions in “Section 3.2.1 Reading Data via Bluetooth”). The data is displayed as below:

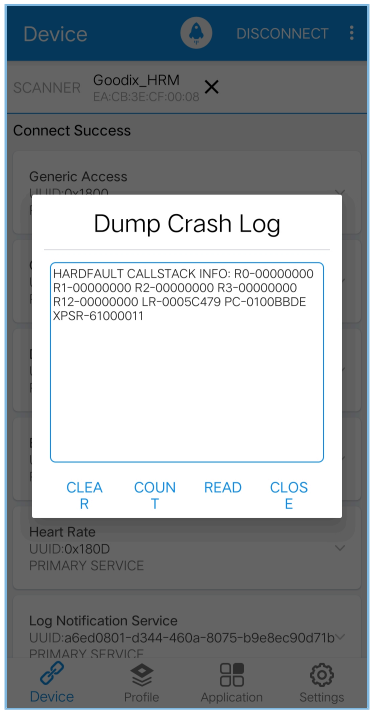

Figure 3-13 Reading fault trace data through Bluetooth

The register values on the debug interface (Figure 3-12) match with that shown in fault trace data (Figure 3-13), which proves that the Fault Trace Module records the fault trace data of the HardFault.

4 Module Details

The read and write of Fault Trace Module are enabled by the APIs for NVDS. This chapter elaborates on the mechanisms for tracing HardFaults and Assert faults, and the mechanism for data tracing through Bluetooth connection.

4.1 HardFault Data Tracing

When a HardFault occurs, registers PSR, R15 (PC), R14 (LR), R3, R2, R1, and R0, which are controlled by the processor at the hardware level, are pushed onto stack in order, and are passed into HardFault_Handler() for exception handling.

Note:

The code below is available in SDK_Folder\platform\soc\common\gr_interrupt.c in the SDK, and available in GRxx_Soc\gr_arch\gr_interrupt.c in the example project directory.

By default, ENABLE_BACKTRACE_FEA is set to 0, indicating that Fault Trace Module records exception information. Setting ENABLE_BACKTRACE_FEA to 1 indicates that the cortex_backtrace module records exception information.

```
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
}
```

HardFault_Handler() enables saving the values of R4 to R11 in the global array R4_R11_REG when a HardFault occurs. The pointer SP is assigned to R0, and SP will be a parameter in hardfault_trace_handler(), the function that will be called in the next step.

Note:

The code below is available in SDK_Folder\components\libraries\fault_trace\fault_trace.c in the SDK, and available in GRxx_Soc\gr_libraries\fault_trace.c in the example project directory.

```
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));
}
```

The parameter sp enables hardfault_trace_handler() to read the register values from the stack, and write the values to s_fault_info. fault_db_record_add() is then called to write the values to NVDS.

The fault trace data format of a HardFault is shown below:

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

The fault trace data above corresponds to the values of registers: R0, R1, R2, R3, R12, R14 (LR), R15 (PC), and PSR (XPSR) when a HardFault occurs.

Note:

HardFault_Handler() enables saving the values of registers from R4 to R11 in the global array R4_R11_REG when a HardFault occurs. You can use hardfault_trace_handler() on demand, and write the obtained values into NVDS.

4.2 Asserting Fault Data Tracing

The Assert method is used to debug software by identifying errors in code. The Assert module is available in SDK_Folder\components\libraries\app_assert in the SDK.

Note:

The code below is available in SDK_Folder\components\libraries\app_assert\app_assert.c in the SDK, and available in GRxx_Soc\gr_libraries\app_assert.c in the example project directory.

```
#define APP_ASSERT_CHECK(EXPR)                                  \
    do                                                            \
```

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

When APP_ASSERT_CHECK(EXPR) is called and EXPR = 0, app_assert_handler() will be called.

Note:

The code below is available in SDK_Folder\components\libraries\app_assert\app_assert.c in the SDK, and available in GRxx_Soc\gr_libraries\app_assert.c in the example project directory.

```
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);
    }
}
```

Callback functions will be called in the handler, to output fault trace data through serial ports.

```
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,
};
```

Three callbacks are called in the Assert module, with each corresponding to a different Assert parameter format and fault trace data (see the source code of app_assert in *app_assert.c*). By default, app_assert_handler() calls assert_err_cb(). You can use app_assert_handler() on demand, and call other callback functions.

Callback functions of the Assert module can output fault trace data through serial ports. Fault Trace Module helps the three callbacks of the Assert module cover previous implementation (the three callbacks were implemented as weak functions) and save fault trace data in NVDS. assert_err_cb() is implemented as below:

Note:

The code below is available in SDK_Folder\components\libraries\fault_trace\fault_trace.c in the SDK, and available in GRxx_Soc\gr_libraries\fault_trace.c in the example project directory.

```
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);
}
```

You can save actual parameter names (param), function names and paths for calling APP_ASSERT_CHECK(), and number of code lines to the structure, by using assert_err_cb(), and save the data to NVDS in designated format by calling the APIs for NVDS.

An example of the fault trace data of Assert faults is shown below:

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

The fault trace data shows the path for the Assert fault (`..\Src\user\user_app.c: 638`), fault type (ERROR), and name of the actual parameter (param).

4.3 Data Tracing Through Bluetooth

You can control the Fault Trace Module on the SK Board through Bluetooth, which is enabled by LNS. LNS provides specific characteristics to receive control commands and send data.

LNS characteristics include Log Information and Log Control Point, with details listed in Table 4-1.

Table 4-1 LNS Characteristics

Characteristic	UUID	Type	Support	Security	Property
Log Information	A6ED0802-D344-460A-8075-B9E8EC90D71B	128 bits	Mandatory	None	Notify
Log Control Point	A6ED0803-D344-460A-8075-B9E8EC90D71B	128 bits	Mandatory	None	Write, Indicate

- Log Information: used to send fault trace data (Notify)
- Log Control Point: used to receive commands (Write) and return information (Indicate)

This section elaborates on the mechanism that Bluetooth controls Fault Trace Module by introducing the implementation of LNS.

Note:

The code below is available in `SDK_Folder\components\profiles\lns\lns.c` in the SDK, and available in `GRxx_Soc\gr_profiles\lns.c` in the example project directory.

```
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() is a callback function written to LNS. The Client writes LNS_CTRL_PT_TRACE_STATUS_GET (0x01), LNS_CTRL_PT_TRACE_INFO_DUMP (0x02), and LNS_CTRL_PT_TRACE_INFO_CLEAR (0x03) to Log Control Point. Any one of the three can trigger a type of event (event.evt_type), and call the registered event handler lns_evt_handler().

Note:

The code below is available in SDK_Folder\components\profiles\lns\lns.c in the SDK, and available in GRxx_Soc\gr_profiles\lns.c in the example project directory.

```
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);
    }
}
```

The corresponding function in the event handling function of LNS is called for each type of event. Each value written to Log Control Point by the Client is associated with an event type.

- When the Client writes 0x01, fault_db_records_num_get() and lns_log_status_send() are called, to read the number of fault trace data entries and send the number to the peer device.
- When the Client writes 0x02, lns_log_info_send() is called, to read the fault trace data and send the data to the peer device.
- When the Client writes 0x03, fault_db_record_clear() is called, to clear the fault trace data.

As shown in Figure 3-5, **COUNT**, **READ**, and **CLEAR** on the GRToolbox interface are implemented by writing 0x01, 0x02, and 0x03 to Log Control Point of the Slave LNS. You can also implement these functionalities by enabling notifications of Log Information and Log Control Point, and writing the corresponding value.

5 FAQ

This chapter describes the possible problems when using Fault Trace Module, analyzes the causes, and provides solutions.

5.1 Why Do I Fail to Read Fault Trace Data on GProgrammer?

- Description

  Why do I fail to read fault trace data on GProgrammer, and no data for **USER Parameters** is obtained?

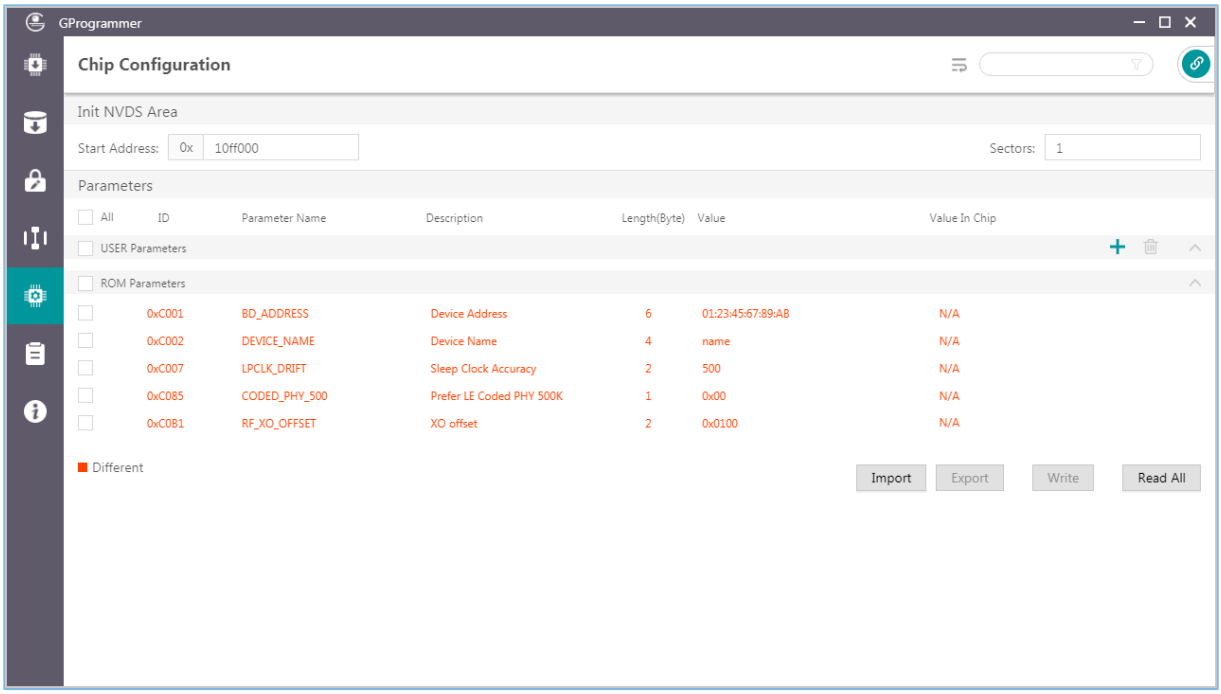

Figure 5-1 Failing to obtain **USER Parameters**

- Analysis

  This may be because the **Start Address** on the interface has not been reset. The start address of NVDS shall be reset every time when GProgrammer is started.

- Solution

  Enter “010FF000” in the **Start Address** field. If the NVDS is reallocated, set the start address accordingly.

5.2 Why Do I Fail to Read Fault Trace Data by Calling APIs in the Project?

- Description

  Why do I fail to read fault trace data by calling APIs in the project, and obtain no data output on GRUart?
  If I check the returned value from fault_db_records_dump() by using APP_ERROR_CHECK(), GRUart shows information as follows:

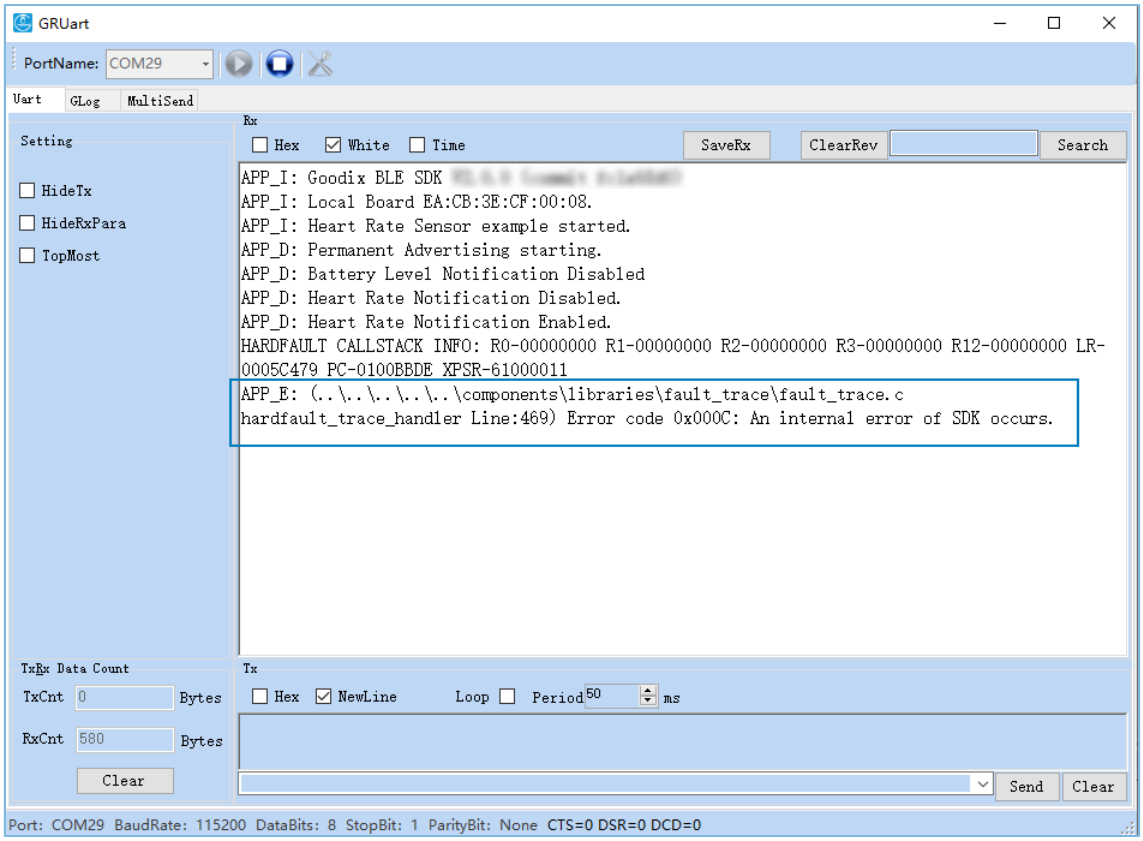

Figure 5-2 Serial port output when checking returned value in case of data read failure

- Analysis

  The buffer size for storing fault trace data is insufficient.

- Solution

  Increase the buffer size for storing fault trace data.