GR551x Fault Trace Module Application Note

Version: 2.0

Release Date: 2022-02-20

Shenzhen Goodix Technology Co., Ltd.

Preface

Purpose

This document introduces the functionalities, operating mechanisms, and applications of GR551x Fault Trace Module, to help users quickly get started with the Module.

Audience

This document is intended for:

- GR551x user
- GR551x developer
- GR551x tester
- Technical writer

Release Notes

This document is the seventh release of *GR551x Fault Trace Module Application Note*, corresponding to GR551x System-on-Chip (SoC) series.

Revision History

Version	Date	Description
1.3	2020-03-16	Initial release
1.5	2020-05-30	Updated descriptions on the source file *cortex_backtrace.c*, the initialization module for GR551x Fault Trace Module, and reading fault trace data by using GProgrammer in "Fault Trace Module in Application".
1.6	2020-06-30	Updated the document version based on Software Development Kit (SDK) changes.
1.7	2020-12-15	Updated GRToolbox UI screenshots based on software update.
1.8	2021-06-28	Updated SoC model descriptions.
1.9	2021-08-09	• Updated the section "Preparation". • Modified SoC model descriptions.
2.0	2022-02-20	Updated document descriptions based on SDK changes.

Contents

1 Introduction

GR551x Fault Trace Module aims to help developers identify problems during Bluetooth application development. When GR551x firmware fails to operate normally, GR551x 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.

GR551x 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
GR551x Developer Guide	Introduces GR551x Software Development Kit (SDK) and how to develop and debug applications based on the SDK.
GR551x Bluetooth Low Energy Stack User Guide	Introduces the layers and basic layer functionalities of a GR551x Bluetooth LE Protocol Stack, and how applications interact with the protocol stack using APIs on the stack.
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 GR551x System-on-Chips (SoCs).

2 Environment Setup

This chapter introduces how to rapidly set up an operating environment for GR551x Fault Trace Module.

2.1 Preparation

- **Hardware preparation**

Table 2-1 Hardware preparation

Name	Description
Development board	GR5515 Starter Kit Board (SK Board), the target device on which Fault Trace Module runs
Connection cable	Micro USB 2.0 serial cable
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 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`.

Note:

SDK_Folder is the root directory of GR551x SDK.

3 Fault Trace Module in Application

This chapter introduces how to add GR551x Fault Trace Module to a project and how to use the Module by taking ble_app_hrs as an example based on GR551x SDK.

3.1 Importing Data to Fault Trace Module to Target Project

Fault Trace Module is optional for running a GR551x-SoC–based project. Before using the Module, add the files of Fault Trace Module to the project directory of ble_app_hrs and enable the macro switch of the Module.

3.1.1 Add the Module

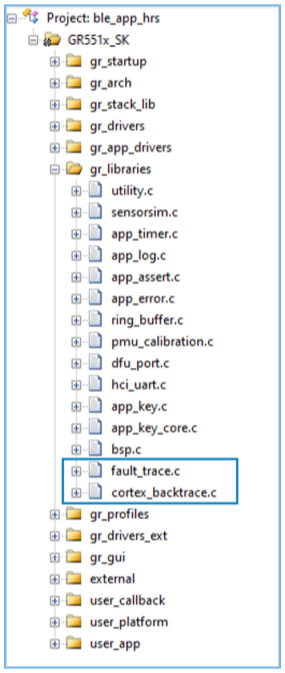
1. Open the heart rate example project ble_app_hrs.

   The source code and project file of the GR551x 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.

   Source files of Fault Trace Module are in `SDK_Folder\components\libraries\fault_trace` and `SDK_Folder\components\libraries\app_error`.

   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:



```
Project: ble_app_hrs
  GR551x_SK
    gr_startup
    gr_arch
    gr_stack_lib
    gr_drivers
    gr_app_drivers
    gr_libraries
      utility.c
      sensorsim.c
      app_timer.c
      app_log.c
      app_assert.c
      app_error.c
      ring_buffer.c
      pmu_calibration.c
      dfu_port.c
      hci_uart.c
      app_key.c
      app_key_core.c
      bsp.c
      fault_trace.c
      cortex_backtrace.c
    gr_profiles
    gr_drivers_ext
    gr_gui
    external
    user_callback
    user_platform
    user_app
```

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

3.1.2 Enable the Module

Open `user_app\custom_config.h` in the directory. Find out the macro SYS_FAULT_TRACE_ENABLE of the Module, and set SYS_FAULT_TRACE_ENABLE to 1.

3.1.3 Initialize the Module

Call fault_trace_db_init() in app_log_assert_init() in *user_periph_setup.c*, to initialize the Module.

```
static void app_log_assert_init(void)
{
    app_log_init_t log_init;

    log_init.filter.level                  = APP_LOG_LVL_DEBUG;
    log_init.fmt_set[APP_LOG_LVL_ERROR]    = APP_LOG_FMT_ALL & (~APP_LOG_FMT_TAG);
    log_init.fmt_set[APP_LOG_LVL_WARNING] = APP_LOG_FMT_LVL;
    log_init.fmt_set[APP_LOG_LVL_INFO]     = APP_LOG_FMT_LVL;
    log_init.fmt_set[APP_LOG_LVL_DEBUG]    = APP_LOG_FMT_LVL;

    app_log_init(&log_init, bsp_uart_send, bsp_uart_flush);

    fault_trace_db_init();
}
```

After adding the Module to the right directory, enabling, and initializing the Module, program the firmware generated based on the compiled program to the SK Board by following instructions in *GR551x Developer Guide.*

If a HardFault or an Assert fault occurs on the SK Board when running the project, relevant fault trace data will be stored in the NVDS of the GR551x 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:

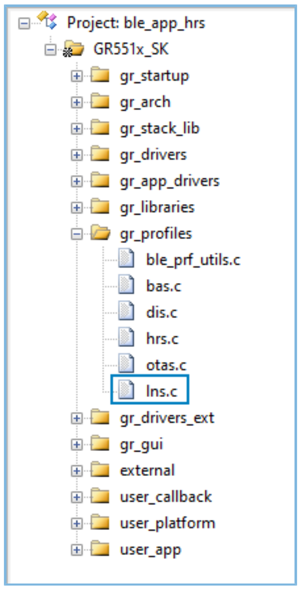


Figure 3-2 Adding the LNS file to the project directory

Call lns_service_init() to initialize the LNS.

Note:

The code below is available in `SDK_Folder\projects\ble\ble_peripheral\ble_app_hrs\Src\user\user_app.c` in the SDK, and available in `GR5515_SK\user_app\user_app.c` in the example project directory.

```
static void services_init(void)
{
...
    lns_service_init(NULL);

#if DFU_ENABLE
    dfu_service_init(NULL);
#endif
}
```

Note:

The code line in bold is the newly added services_init() function to initialize the LNS.

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.

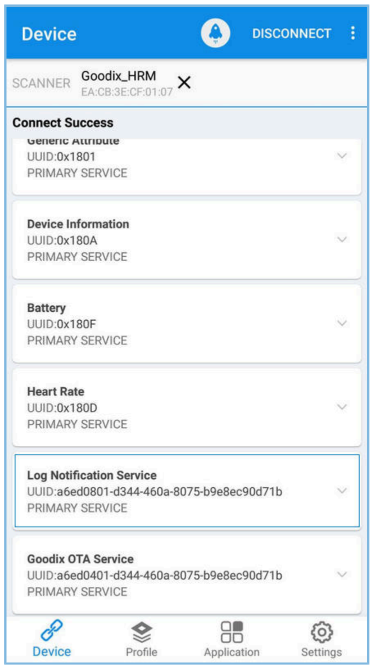


Figure 3-3 Successful discovery of LNS after connecting the phone to the Board on GRToolbox

2. Tap ⋮ in the upper-right corner, and choose **Read Log**, as shown in Figure 3-4.

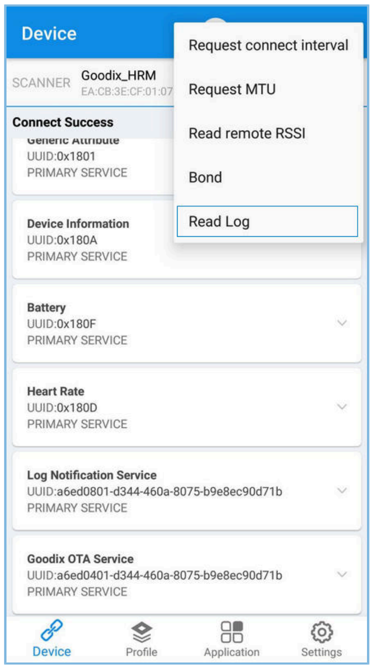
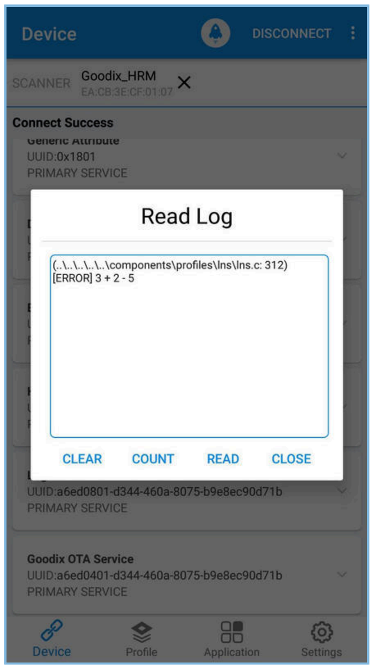


Figure 3-4 To read device logs

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



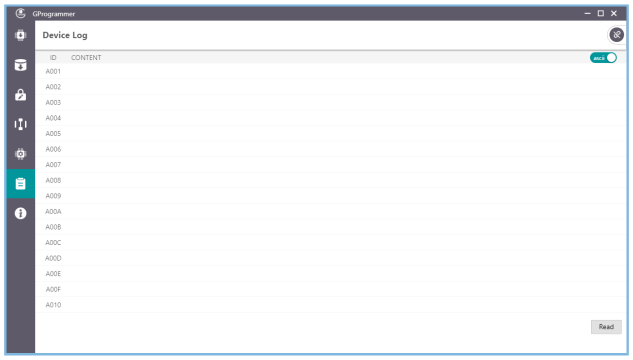
```
(..\..\..\..\..\components\profiles\lns\lns.c: 312)
[ERROR] 3 + 2 - 5
```

Figure 3-5 Successfully reading device 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.



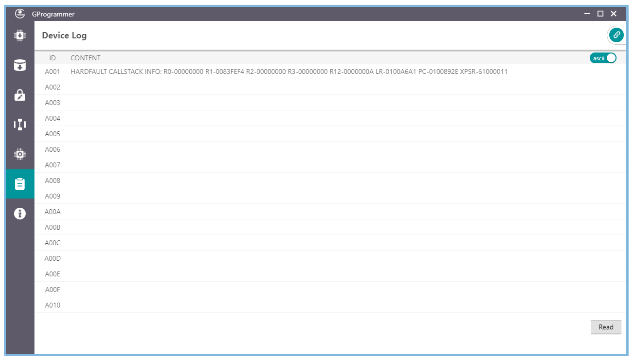
ID	CONTENT
A001	
A002	
A003	
A004	
A005	
A006	
A007	
A008	
A009	
A00A	
A00B	
A00C	
A00D	
A00E	
A00F	
A010	

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:


GProgrammer

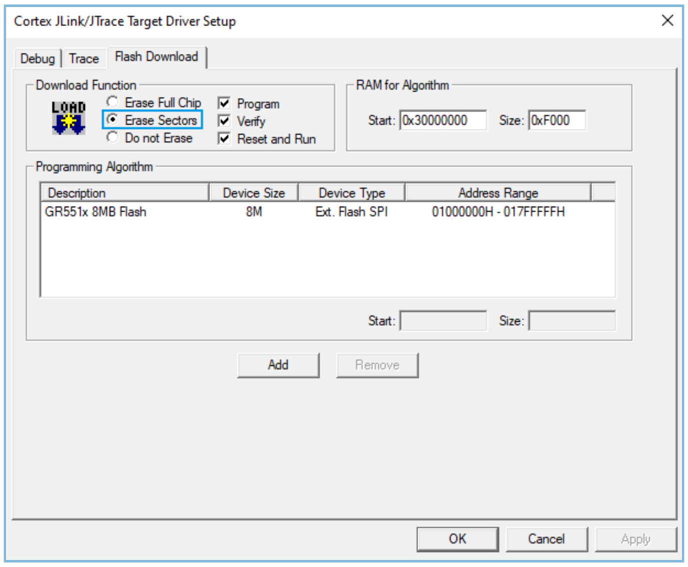
Device Log

ID	CONTENT
A001	HARDFAULT CALLSTACK INFO: R0-00000000 R1-0083FEF4 R2-00000000 R3-00000000 R12-0000000A LR-0100A6A1 PC-0100892E XPSR-61000011
A002	
A003	
A004	
A005	
A006	
A007	
A008	
A009	
A00A	
A00B	
A00C	
A00D	
A00E	
A00F	
A010	

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.


Cortex JLink/JTrace Target Driver Setup

Debug | Trace | Flash Download

Download Function: ( ) Erase Full Chip [✓ (•) Erase Sectors] ( ) Do not Erase; [✓] Program [✓] Verify [✓] Reset and Run

RAM for Algorithm: Start: 0x30000000 Size: 0xF000

Programming Algorithm

Description	Device Size	Device Type	Address Range
GR551x 8MB Flash	8M	Ext. Flash SPI	01000000H - 017FFFFFH

Start: Size:

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 Figure 3-8. Avoid selecting **Erase Full Chip**, which will also erase the fault trace data in the NVDS of the SK Board.

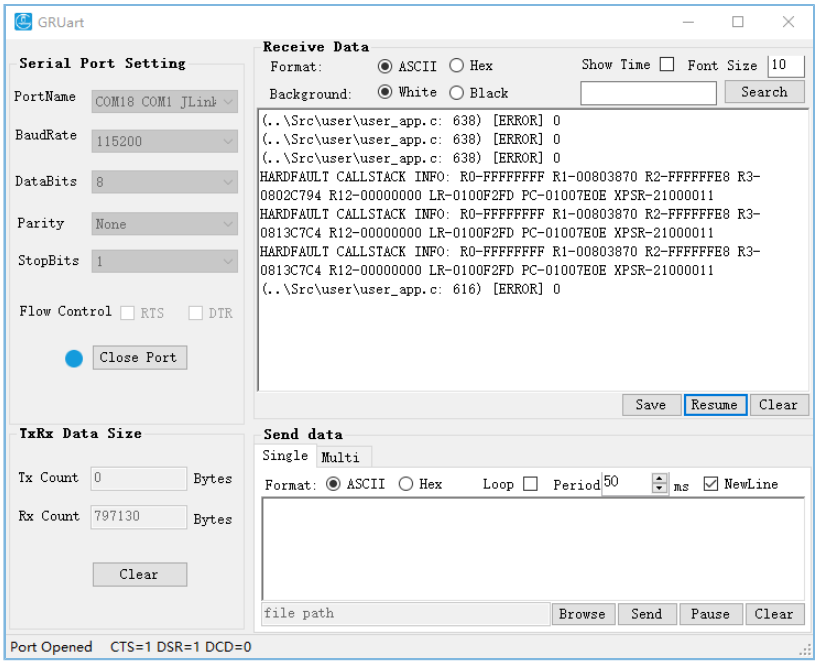
The code below is an example to read fault trace data by calling APIs in the project.

```
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 “Modify the main() Function” in *GR551x Developer Guide.*

An example of fault trace data obtained through GRUart is displayed in Figure 3-9.



PortName: COM18 COM1 JLink
BaudRate: 115200
DataBits: 8
Parity: None
StopBits: 1
Flow Control: ☐ RTS ☐ DTR

Format: [✓ ASCII] Hex   Show Time ☐   Font Size 10
Background: [✓ White] Black

```
(..\Src\user\user_app.c: 638) [ERROR] 0
(..\Src\user\user_app.c: 638) [ERROR] 0
(..\Src\user\user_app.c: 638) [ERROR] 0
HARDFAULT CALLSTACK INFO: R0-FFFFFFFF R1-00803870 R2-FFFFFFE8 R3-
0802C794 R12-00000000 LR-0100F2FD PC-01007E0E XPSR-21000011
HARDFAULT CALLSTACK INFO: R0-FFFFFFFF R1-00803870 R2-FFFFFFE8 R3-
0813C7C4 R12-00000000 LR-0100F2FD PC-01007E0E XPSR-21000011
HARDFAULT CALLSTACK INFO: R0-FFFFFFFF R1-00803870 R2-FFFFFFE8 R3-
0813C7C4 R12-00000000 LR-0100F2FD PC-01007E0E XPSR-21000011
(..\Src\user\user_app.c: 616) [ERROR] 0
```

Tx Count: 0 Bytes
Rx Count: 797130 Bytes

Format: [✓ ASCII] Hex   Loop ☐   Period 50 ms   ☑ NewLine

Port Opened   CTS=1 DSR=1 DCD=0

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\components\profiles\hrs\hrs.h` in the SDK, and available in `GR5515_SK\gr_profiles\hrs.c` in the example project directory.

The code lines in bold 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.

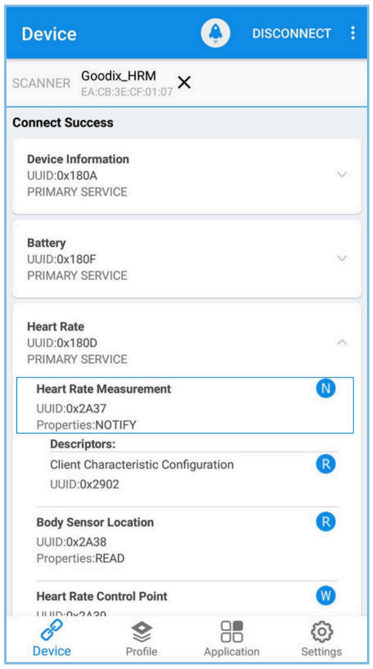
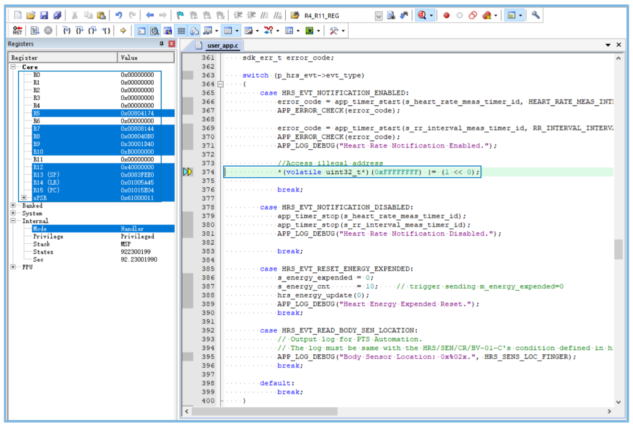


Figure 3-10 To view **Heart Rate** notification

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.

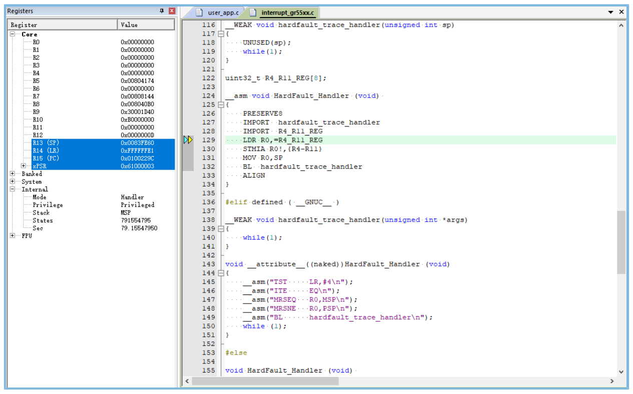


```
Registers
Register            Value
Core
  R0                0x00000000
  R1                0x00000000
  R2                0x00000000
  R3                0x00000000
  R4                0x00000000
  R5                0x00804174
  R6                0x00000000
  R7                0x00808144
  R8                0x008040B0
  R9                0x30001D40
  R10               0xB0000000
  R11               0x00000000
  R12               0x40000000
  R13 (SP)          0x0083FEE0
  R14 (LR)          0x01005A45
  R15 (PC)          0x01015E04
  xPSR              0x61000011
Banked
System
Internal
  Mode              Handler
  Privilege         Privileged
  Stack             MSP
  States            922300199
  Sec               92.23001990
FPU
```

```c
    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_INT
            APP_ERROR_CHECK(error_code);

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

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

            break;

        case HRS_EVT_NOTIFICATION_DISABLED:
            app_timer_stop(s_heart_rate_meas_timer_id);
            app_timer_stop(s_rr_interval_meas_timer_id);
            APP_LOG_DEBUG("Heart Rate Notification Disabled.");

            break;

        case HRS_EVT_RESET_ENERGY_EXPENDED:
            s_energy_expended = 0;
            s_energy_cnt      = 10;    // trigger sending m_energy_expended=0
            hrs_energy_update(0);
            APP_LOG_DEBUG("Heart Energy Expended Reset.");
            break;

        case HRS_EVT_READ_BODY_SEN_LOCATION:
            // Output log for PTS Automation.
            // The log must be same with the HRS/SEN/CR/BV-01-C's condition defined in h
            APP_LOG_DEBUG("Body Sensor Location: 0x%02x.", HRS_SENS_LOC_FINGER);
            break;

        default:
            break;
    }
```

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.

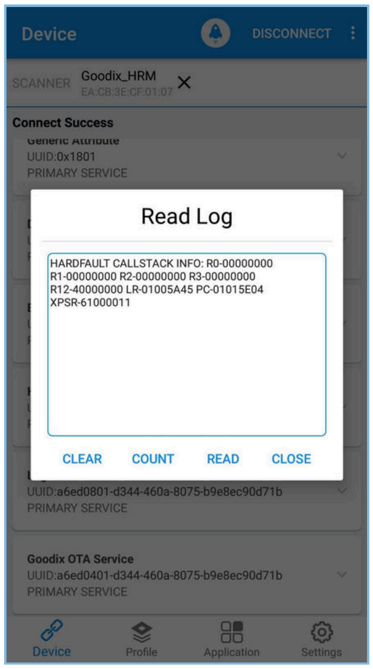


```
Registers
Register            Value
Core
  R0                0x00000000
  R1                0x00000000
  R2                0x00000000
  R3                0x00000000
  R4                0x00000000
  R5                0x00804174
  R6                0x00000000
  R7                0x00808144
  R8                0x008040B0
  R9                0x30001D40
  R10               0xB0000000
  R11               0x00000000
  R12               0x0000000D
  R13 (SP)          0x0083FE60
  R14 (LR)          0xFFFFFFE1
  R15 (PC)          0x0100229C
  xPSR              0x61000003
Banked
System
Internal
  Mode              Handler
  Privilege         Privileged
  Stack             MSP
  States            791554795
  Sec               79.15547950
FPU
```

```c
__WEAK void hardfault_trace_handler(unsigned int sp)
{
    UNUSED(sp);
    while(1);
}

uint32_t R4_R11_REG[8];

__asm void HardFault_Handler (void)
{
    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
    ALIGN
}

#elif defined ( __GNUC__ )

__WEAK void hardfault_trace_handler(unsigned int *args)
{
    while(1);
}

void __attribute__((naked))HardFault_Handler (void)
{
    __asm("TST     LR,#4\n");
    __asm("ITE     EQ\n");
    __asm("MRSEQ   R0,MSP\n");
    __asm("MRSNE   R0,PSP\n");
    __asm("BL      hardfault_trace_handler\n");
    while (1);
}

#else

void HardFault_Handler (void)
```

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:



```
HARDFAULT CALLSTACK INFO: R0-00000000
R1-00000000 R2-00000000 R3-00000000
R12-40000000 LR-01005A45 PC-01015E04
XPSR-61000011
```

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 taking GR551x SDK for an example, 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 included in HardFault_Handler() for exception handling.

Note:

The code below is available in `SDK_Folder\projects\ble\ble_peripheral\ble_app_hrs\Src\user\user_app.c` in the SDK, and available in `GR5515_SK\gr_arch\interrupt_gr55xx.c` in the example project directory.

```
uint32_t R4_R11_REG[8];

__asm void HardFault_Handler (void)
{
    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
    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\app\components\libraries\fault_trace\fault_trace.c` in the SDK, and available in `GR5515_SK\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));

    while(1);
}
```

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-FFFFFFFF R1-00803870 R2-FFFFFFE8 R3-0802C794 R12-00000000
LR-0100F2FD PC-01007E0E XPSR-21000011}
```

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 Assert 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\ap_assert\app_assert.h in the SDK, and available in GR5515_SK\gr_libraries\app_assert.h 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\ap_assert\app_assert.c in the SDK, and available in GR5515_SK\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.

```
/**@brief Assert callbacks.*/
typedef struct
{
    assert_err_cb_t   assert_err_cb;    /**< Assert error type callback. */
    assert_param_cb_t assert_param_cb;  /**< Assert parameter error type callback. */
    assert_warn_cb_t  assert_warn_cb;   /**< Assert warning type callback. */
}sys_assert_cb_t;
```

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 GR5515_SK\gr_libraries\fault_trace.c in the example project directory.

```
static void 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\app\components\libraries\app_assert\app_assert.c` in the SDK, and available in `GR5515_SK\gr_libraries\app_assert.c` in the example project directory.

```
static void lns_write_att_cb(uint8_t conn_idx,
                             const gatts_write_req_cb_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_cb() 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 calls the registered event handling function lns_evt_handler().

Note:

The code below is available in `SDK_Folder\components\libraries\ap_assert\ap_assert.c` in the SDK, and available in `GR5515_SK\gr_libraries\ap_assert.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.

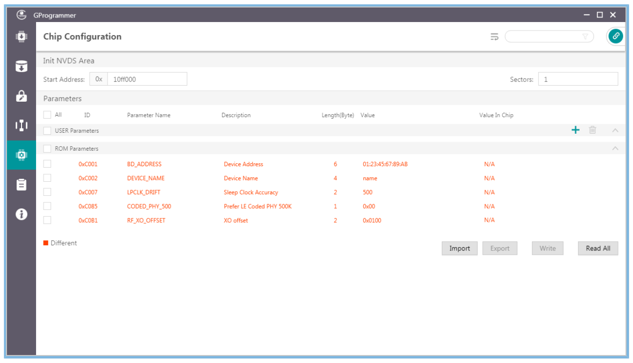
5 FAQ

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?



Init NVDS Area

Start Address: 0x 10ff000

Sectors: 1

Parameters

All	ID	Parameter Name	Description	Length(Byte)	Value	Value In Chip
USER Parameters						
ROM Parameters						
	0xC001	BD_ADDRESS	Device Address	6	01:23:45:67:89:AB	N/A
	0xC002	DEVICE_NAME	Device Name	4	name	N/A
	0xC007	LPCLK_DRIFT	Sleep Clock Accuracy	2	500	N/A
	0xC085	CODED_PHY_500	Prefer LE Coded PHY 500K	1	0x00	N/A
	0xC0B1	RF_XO_OFFSET	XO offset	2	0x0100	N/A

🟥 Different

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.

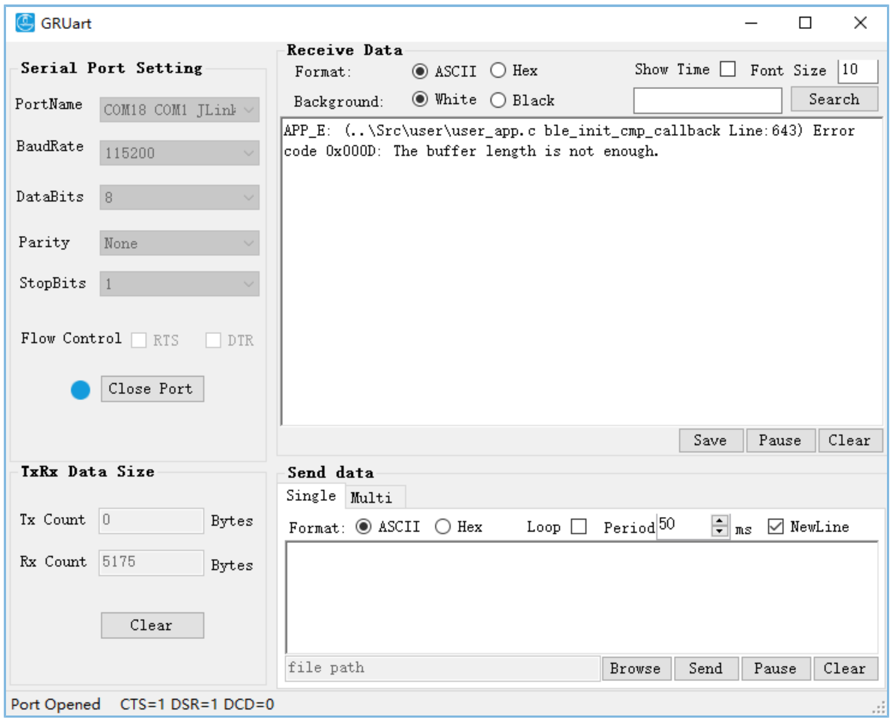
- Solution

  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:



```
APP_E: (..\Src\user\user_app.c ble_init_cmp_callback Line:643) Error
code 0x000D: The buffer length is not enough.
```

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.