



Nuclei™ N200 系列

处理器内核 SDK 使用说明

版权声明

版权所有 © 2018–2019 芯来科技（Nuclei System Technology）有限公司。保留所有权利。

Nuclei™是芯来科技公司拥有的商标。本文件使用的所有其他商标为各持有公司所有。

本文件包含芯来科技公司的机密信息。使用此版权声明为预防作用，并不意味着公布或披露。未经芯来科技公司书面许可，不得以任何形式将本文的全部或部分信息进行复制、传播、转录、存储在检索系统中或翻译成任何语言。

本文文件描述的产品将不断发展和完善；此处的信息由芯来科技提供，但不做任何担保。

本文件仅用于帮助读者使用该产品。对于因采用本文件的任何信息或错误使用产品造成的任何损失或损害，芯来科技概不负责。

联系我们

若您有任何疑问，请通过电子邮件 support@nucleisys.com 联系芯来科技。

修订历史

版本号	修订日期	修订的章节	修订的内容
1.0	2019/5/16	N/A	1. 初始版本

目录

版权声明.....	0
联系我们.....	0
修订历史.....	1
表格清单.....	4
图片清单.....	5
1. N200-SDK 下载地址	0
2. 基于 N200-SDK 的软件开发与运行	1
2.1. N200-SDK 简介	1
2.2. N200-SDK 代码结构	1
2.3. N200-SDK 板级支持包解析	3
2.3.1. 移植 Newlib 桩函数	3
2.3.2. 支持 printf 函数	4
2.3.3. 提供系统链接脚本	5
2.3.4. 系统启动引导程序	10
2.3.5. 异常、中断和 NMI 处理	15
2.3.6. 使用 newlib-nano	20
2.4. 使用 N200-SDK 开发和编译程序	21
2.4.1. 在 N200-SDK 环境中安装工具链	21
2.4.2. 在 N200-SDK 环境中开发程序	22
2.4.3. 编译使得程序从 Flash 直接运行	24
2.4.4. 编译使得程序从 ILM 中运行	25
2.4.5. 编译使得程序从 Flash 上载至 ILM 中运行	25
2.5. 使用 N200-SDK 下载程序	26
2.5.1. JTAG 调试器与开发板的连接	26
2.5.2. 设置 JTAG 调试器在 Linux 系统中的 USB 权限	27
2.5.3. 将程序下载至 FPGA 原型开发板	29
2.6. 在 FPGA 开发板上运行程序	29
2.6.1. 程序从 Flash 直接运行	30
2.6.2. 程序从 ILM 中运行	31
2.6.3. 程序从 Flash 上载至 ILM 中运行	32
2.7. 使用 GDB 远程调试程序	32
2.7.1. 调试器工作原理	32
2.7.2. GDB 常用操作示例	35
2.7.3. 使用 GDB 调试 Hello World 示例	36

3. 使用 N200-SDK 运行更多示例程序	41
3.1. DHRYSTONE 示例程序	41
3.1.1. Dhrystone 示例程序功能简介	41
3.1.2. Dhrystone 示例程序代码结构	41
3.1.3. 运行 Dhrystone	42
3.2. COREMARK 示例程序	42
3.2.1. CoreMark 示例程序功能简介	43
3.2.2. CoreMark 示例程序代码结构	43
3.2.3. 运行 CoreMark	44
3.3. DEMO_IASM 示例程序	44
3.3.1. Demo_iasm 示例程序功能简介	44
3.3.2. Demo_iasm 示例程序代码结构	45
3.3.3. 运行 Demo_iasm	45
3.4. DEMO_ECLIC 示例程序	46
3.4.1. Demo_eclic 示例程序功能简介	46
3.4.2. Demo_eclic 示例程序代码结构	47
3.4.3. Demo_eclic 示例程序源码分析	48
3.4.4. 运行 Demo_eclic	52
3.5. FREERTOS 示例程序	53
3.5.1. FreeRTOS 示例程序功能简介	53
3.5.2. FreeRTOS 示例程序代码结构	54
3.5.3. FreeRTOS 示例程序源码分析	55
3.5.4. 运行 FreeRTOS	55

表格清单

表 2-1 GDB 常用命令	36
----------------------	----

图片清单

图 2-1 编译 HELLO WORLD 程序使得程序从 FLASH 直接运行	24
图 2-2 虚拟机 LINUX 系统识别 USB 图标	29
图 2-3 将 USB 接口选择连接至虚拟机中	29
图 2-4 运行 HELLO WORLD 程序的输出	31
图 2-5 打开 OPENOCD 后的命令行界面	38
图 2-6 打开 GDB 的命令行界面	38
图 2-7 打开 GDB 显示设置的断点	38
图 2-8 通过 GDB 查看存储器中的数据	39
图 2-9 GDB 显示寄存器的值	39
图 2-10 GDB 显示程序停止于断点处	40
图 2-11 GDB 单步执行程序	40
图 3-1 运行 DEMO_IASM 示例后于主机串口终端上显示信息	46
图 3-2 运行 DEMO_ECLIC 示例后于主机串口终端上显示信息	47
图 3-3 运行 DEMO_ECLIC 示例后于主机串口终端上显示信息	53
图 3-4 FREERTOS 的中断示例打印信息	56

1. N200-SDK 下载地址

为了让用户能够快速熟悉使用 N200 系列处理器内核开发软件，N200 系列内核结合配套的 SoC 原型平台，开发了一套与之配套的软件开发平台（Software Development Kit），称之为 N200-SDK 平台。

为了便于用户随时跟踪状态和方便使用，N200-SDK 的所有源代码均开源托管于 GitHub 网站上，网址为 <https://github.com/nucleisys/n200-sdk>，本文将以“N200-SDK 项目”代指其 GitHub 上的具体网址。

注意：

- 本 SDK 是基于配套的样例 SoC 的一个参考 SDK，用户在集成 N200 处理器内核至其自有的 SoC 中之后，可能需要对 SDK 进行适当的修改以适配其 SoC。
- 本文档涉及到 Linux 操作和 Makefile 的基本知识，本文档将不做过多介绍。

2. 基于 N200-SDK 的软件开发与运行

2.1. N200-SDK 简介

N200-SDK 并不是一个软件，它本质上是由一些 Makefile、板级支持包（Board Support Package, BSP）、脚本和软件示例组成的一套开发环境。N200-SDK 基于 Linux 平台，使用标准的 RISC-V GNU 工具链对程序进行编译，使用 OpenOCD+GDB 将程序下载到硬件平台中并进行调试。N200-SDK 主要包含如下两个方面的内容。

- 板级支持包（Board Support Package, BSP）。
- 若干软件示例。

2.2. N200-SDK 代码结构

N200-SDK 平台的代码结构如下：

```
n200-sdk          // 存放 n200-sdk 的目录
|----bsp          // 存放板级支持包 (Board Support Package) 的目录
|----nuclei-n200  // 存放 Nuclei N200 处理器 Core 相关的 BSP 文件
|----n200
|----env          // 存放一些基本的支持性文件
|----drivers      // 存放处理器 Core 相关的函数和驱动文件
|----n200_func.c  // 处理器 Core 的常用函数
|----n200_func.h  // 处理器 Core 的头文件
|----n200_timer.h // TIMER 单元相关的头文件
|----n200_eclic.h // ECLIC 单元相关的头文件
|----riscv_encoding.h // RISC-V 架构相关的编码信息
|----stubs        // 存放移植 newlib 的底层桩函数
|----soc          // 存放配套 SoC 相关的 BSP 文件，用户可以替代成自己 SoC 的驱动
|----drivers      // 存放配套 SoC 相关的函数和驱动文件
|----n200_func.c  // SoC 的常用函数
|----n200_func.h  // SoC 的头文件
|----software     // 存放示例程序的源代码
|----hello_world  // hello_world 示例程序，见第 2.4.2 节
```

```

|----demo_iasm           //内嵌汇编示例程序，见第 3.3 节
|----dhrystone           //Dhrystone 跑分程序，见第 3.1 节
|----coremark            // CoreMark 跑分程序，见第 3.2 节
|----demo_eclic          //Demo_ECLIC 示例程序，见第 3.4 节
|----FreeRTOSv9.0.0      //FreeRTOS 示例程序，见第 3.5 节
|----work                //存放工具链的目录
|----Makefile            //主 Makefile 文件

```

各个主要的目录简述如下：

- **software** 目录主要用于存放软件示例，包括基本的 **hello_world** 示例程序、**demo_iasm** 示例程序、**dhrystone** 跑分程序和 **CoreMark** 跑分程序，以及 **FreeRTOS** 示例程序。每个示例均有单独的文件夹，包含了各自的源代码、**Makefile** 和编译选项（在 **Makefile** 中指定）等。
- **bsp/nuclei-n200/n200/drivers** 目录主要用于存放 **N200** 系列处理器内核的驱动程序代码，譬如 **ECLIC** 单元的底层驱动函数和代码。
- **bsp/nuclei-n200/soc/drivers** 目录主要用于存放配套 **SoC** 的驱动程序代码，譬如系统外设 **UART** 的底层驱动函数和代码。该文件夹可以被替代成为用户 **SoC** 自己的驱动。
- **bsp/nuclei-n200/n200/stubs** 目录主要用于存放一些移植 **Newlib** 所需的底层桩函数的具体实现。见第 2.3.1 节，了解 **Newlib** 移植桩函数的更多信息。
- **bsp/nuclei-n200/n200/env** 目录主要用于存放一些基本的支持性文件，简述如下。
 - **common.mk**: 调用 **GCC** 进行编译的 **Makefile** 脚本，也会指定编译相关的选项。
 - ***.lds**: 程序编译的链接脚本，见第 2.3.3 节了解其详情。
 - **start.S**: **Core** 的上电启动引导程序，见 2.3.4 节了解其详情。
 - **init.c**: **Core** 的上电初始化函数，见 2.3.4 节了解其详情。
 - **entry.S**: **Core** 的异常和中断入口函数，见 2.3.5 节了解其详情。
 - **handlers.c**: **Core** 的中断、异常、以及 **NMI** 处理函数，见 2.3.5 节了解其详情。

- *.cfg: OpenOCD 的配置文件。

2.3. N200-SDK 板级支持包解析

嵌入式平台通常会提供板级支持包（Board Support Package, BSP），使得应用开发人员无需关注底层的细节。N200-SDK 平台的板级支持包均存在于 BSP 目录下，下文将介绍该 BSP 如何解决嵌入式开发的几个基本问题。

注意：对于不想关注底层细节的应用开发人员可以略过此节，请直接参见第 2.4 节了解如何使用 N200-SDK 进行程序的开发。

2.3.1. 移植 Newlib 桩函数

Newlib 是嵌入式系统常用的 C 运行库。Newlib 的所有库函数都建立在 20 个桩函数的基础上，这 20 个桩函数完成具体操作系统和底层硬件相关的功能。

注意：不同的桩函数可能会被不同的 C 库函数所调用，所以如果嵌入式程序中使用到的 C 库函数不多的时候，便并不需要实现所有的 20 个桩函数。

N200-SDK 平台在板级支持包中完成了 Newlib 桩函数的实现。具体体现在 bsp/nuclei-n200/n200/stubs 目录下实现了如下几个桩函数：

- close.c: 实现了 _close 函数。
- _exit.c: 实现了 _exit 函数。
- fstat.c: 实现了 _fstat 函数。
- lseek.c: 实现了 _lseek 函数。
- read.c: 实现了 _read 函数。
- sbrk.c: 实现了 _sbrk 函数。
- write.c: 实现了 _write 函数。

注意：上述有的函数的函数体实现为空，因为在嵌入式程序中这些函数所支持的功能基本使用不到（譬如文件操作）。

上述的函数名称都是以下划线开始（譬如 `_write`），与原始的 Newlib 定义的桩函数名称（譬如 `write`）不一致。这是因为，在 Newlib 的底层桩函数中存在着多层嵌套，`wirte` 函数会调用名为 `write_r` 的可重入函数，然后 `write_r` 函数调用了最终的 `_write` 函数。

上述实现的桩函数将会在 `bsp/nuclei-n200/n200/env/common.mk` 脚本中作为普通源文件加入被编译的文件列表，`common.mk` 代码片段如下：

```
// bsp/nuclei-n200/n200/env/common.mk 脚本片段

//将桩函数加入源文件列表
C_SRCS += $(STUB_DIR)/_exit.c
C_SRCS += $(STUB_DIR)/write_hex.c
C_SRCS += $(STUB_DIR)/fstat.c
C_SRCS += $(STUB_DIR)/isatty.c
C_SRCS += $(STUB_DIR)/lseek.c
C_SRCS += $(STUB_DIR)/read.c
C_SRCS += $(STUB_DIR)/sbrk.c
C_SRCS += $(STUB_DIR)/write.c

.....

C_OBJS := $(C_SRCS:.c=.o)

.....

//调用 GCC 对源文件进行编译
$(C_OBJS): %.o: %.c $(HEADERS)
    $(CC) $(CFLAGS) $(INCLUDES) -include sys/cdefs.h -c -o $@ $<
```

综上，N200-SDK 平台通过实现桩函数的函数体并且将其与其他普通源文件进行一并编译，便实现了 Newlib 的移植和支持。由于这些桩函数作为源文件一起进行了编译，所以在链接阶段，链接器在链接 Newlib 的 C 库函数的时候能够找到这些桩函数一并进行链接（否则便会报错称找不到桩函数的实现）。

2.3.2. 支持 printf 函数

`printf` 函数在嵌入式早期开发阶段对于分析程序行为非常有帮助，因此对 `printf` 的支持必不可少。嵌入式平台中通常需要将 `printf` 的输出重定位到 UART 接口传输至主机 PC 的显示器上。

由于 `printf` 函数属于典型的 C 标准函数，因此会调用 Newlib C 运行库中的库函数，而在 Newlib

的 `printf` 库函数中，最终将字符逐个的输出是依靠的底层桩函数 `write` 函数。因此，对于 `printf` 函数的移植归根结底在于对于 `Newlib` 桩函数 `write` 函数的实现。

`N200-SDK` 平台在 `BSP` 中完成了 `write` 桩函数的实现。如本章第 2.3.1 节中所述，`write` 函数最终调用 `_write` 函数，而该函数被实现在 `bsp/nuclei-n200/n200/stubs/write.c` 文件中，其代码片段如下：

```
.....

//write.c 函数片段

ssize_t _write(int fd, const void* ptr, size_t len)
{
    const uint8_t * current = (const char *)ptr;

    if (isatty(fd)) {
        for (size_t jj = 0; jj < len; jj++) {
            while (UART0_REG(UART_REG_TXFIFO) & 0x80000000) ;//等待 UART 的 TXFIFO 有空
            //向 UART 的 TXFIFO 寄存器写入字符，从而使得字符通过 UART 输出
            UART0_REG(UART_REG_TXFIFO) = current[jj];
            if (current[jj] == '\n') {
                while (UART0_REG(UART_REG_TXFIFO) & 0x80000000) ;
                UART0_REG(UART_REG_TXFIFO) = '\r';
            }
        }
        return len;
    }

    return _stub(EBADF);
}
```

从 `_write` 函数体中可以看出，该函数通过向 `SoC` 的 `UART0` 的 `TXFIFO` 写入字符，最终将输出字符重定向至 `UART` 将其输出，最终能够显示在主机 `PC` 的显示屏幕上（借助主机 `PC` 的串口调试助手软件）。

综上，`N200-SDK` 平台通过实现桩函数 `_write` 便实现了 `printf` 的移植。

2.3.3. 提供系统链接脚本

嵌入式系统中需要关注“链接脚本”为程序分配合适的存储器空间，譬如程序段放在什么区间、数据段放在什么区间等等。有关 `GCC` 的“链接脚本（`Link Scripts`）”的语法和说明请用户自行查阅其他资料学习。

`N200-SDK` 平台提供三个不同的“链接脚本”，在编译时，可以通过 `Makefile` 的命令行指定不

同的“链接脚本”作为 GCC 的链接脚本，从而实现不同的运行方式。三个不同的“链接脚本”分别在后文介绍。

■ 程序存放在 Flash 中且从 Flash 中直接执行

使用链接脚本 `bsp/nuclei-n200/n200/env/link_flashxip.lds`，可以将程序存放在 Flash 中并且直接从 Flash 中进行执行。该链接脚本代码片段及解释如下：

//bsp/nuclei-n200/n200/env/link_flashxip.lds 代码片段

ENTRY(_start) //指明程序入口为_start 标签

MEMORY

{

//定义了两块地址区间，分别名为 flash 和 ram，对应 Flash 和 DLM 的地址区间

flash (rxai!w) : ORIGIN = 0x20000000, LENGTH = 4M

ram (wxa!ri) : ORIGIN = 0x90000000, LENGTH = 64K

}

SECTIONS

{

__stack_size = DEFINED(__stack_size) ? __stack_size : 2K;

.init :

{

*(.vtable) //中断向量表

KEEP (*(SORT_NONE(.init)))

} >flash AT>flash

.ilalign :

{

. = ALIGN(4);

PROVIDE(_ilm_lma = .); //创建一个标签名为_ilm_lma，地址为 flash 地址区间

//的起始地址

} >flash AT>flash

.ialign :

{

PROVIDE(_ilm = .); //创建一个标签名为_ilm，地址也为 flash 地址区间

//的起始地址

} >flash AT>flash

.text :

{

(.text.unlikely .text.unlikely.)

(.text.startup .text.startup.)

(.text .text.)

(.gnu.linkonce.t.)

} >flash AT>flash

//注意：由于此“链接脚本”意图是让程序存储在 Flash 之中，且直接从 Flash 中进行运行，所以其物理

//地址和运行地址相同，所以，上述.text 代码段的物理地址是 flash 区间，而运行地址为 flash 区间

```
.data      :
{
    *(.rdata)
    *(.rodata .rodata.*)
    *(.gnu.linkonce.r.*)
    *(.data .data.*)
    *(.gnu.linkonce.d.*)
    . = ALIGN(8);

    PROVIDE( __global_pointer$ = . + 0x800 );//创建一个标签名为__global_pointer$
    *(.sdata .sdata.* .sdata*)
    *(.gnu.linkonce.s.*)
    . = ALIGN(8);
    *(.srodata.cst16)
    *(.srodata.cst8)
    *(.srodata.cst4)
    *(.srodata.cst2)
    *(.srodata .srodata.*)
} >ram AT>flash
```

//注意：由于此“链接脚本”意图是让数据存储在 Flash 之中，而将数据段上载至 DLM 中进行运行，所以数据段物理地址和运行地址不同，所以，上述.data 数据段的物理地址是 flash 区间，而运行地址为 ram 区间

■ 程序存放在 ILM 中且从 ILM 中直接执行

使用链接脚本 bsp/nuclei-n200/n200/env/link_ilm.lds，可以将程序存放在 ILM 中并且直接从 ILM 中进行执行。该链接脚本代码片段及解释如下：

//bsp/nuclei-n200/n200/env/link_ilm.lds 代码片段

```
ENTRY( _start ) //指明程序入口为_start 标签
```

```
MEMORY
```

```
{
//定义了两块地址区间，分别名为 ilm 和 ram，对应 ILM 和 DLM 的地址区间
    ilm (rxai!w) : ORIGIN = 0x80000000, LENGTH = 64K
    ram (wxa!ri) : ORIGIN = 0x90000000, LENGTH = 64K
}
```

```
SECTIONS
```

```
{
    __stack_size = DEFINED(__stack_size) ? __stack_size : 2K;

    .init      :
    {
        *(.vtable) //中断向量表
        KEEP (*(SORT_NONE(.init)))
    } >ilm AT>ilm
```

```
.ilalign      :
{
    . = ALIGN(4);
```

```
PROVIDE( _ilm_lma = . );//创建一个标签名为_ilm_lma, 地址为 ilm 地址区间的起始地址
} >ilm AT>ilm
```

```
.ialign      :
{
PROVIDE( _ilm = . ); //创建一个标签名为_ilm, 地址也为 ilm 地址区间
                        //的起始地址
} >ilm AT>ilm
```

```
.text      :
{
    *(.text.unlikely .text.unlikely.*)
    *(.text.startup .text.startup.*)
    *(.text .text.*)
    *(.gnu.linkonce.t.*)
} >ilm AT>ilm
```

//注意: 由于此“链接脚本”意图是让程序存储在 ILM 之中, 且直接从 ILM 中进行运行, 所以其物理地址和运行地址相同, 所以, 上述 .text 代码段的物理地址是 ilm 区间, 而运行地址也为 ilm 区间

```
.data      :
{
    *(.rdata)
    *(.rodata .rodata.*)
    *(.gnu.linkonce.r.*)
    *(.data .data.*)
    *(.gnu.linkonce.d.*)
    . = ALIGN(8);

    PROVIDE( __global_pointer$ = . + 0x800 );//创建一个标签名为__global_pointer$
    *(.sdata .sdata.* .sdata*)
    *(.gnu.linkonce.s.*)
    . = ALIGN(8);
    *(.srodata.cst16)
    *(.srodata.cst8)
    *(.srodata.cst4)
    *(.srodata.cst2)
    *(.srodata .srodata.*)
} >ram AT>ilm
```

//注意: 由于此“链接脚本”意图是让数据存储在 ILM 之中, 而将数据段上载至 DLM 中进行运行, 所以数据段物理地址和运行地址不同, 所以, 上述 .data 数据段的物理地址是 ilm 区间, 而运行地址为 ram 区间

■ 程序存放在 Flash 中但是上电后上载至 ILM 中进行执行

使用链接脚本 `bsp/nuclei-n200/n200/env/link_flash.lds`, 可以将程序存放在 Flash 中但是上电后上载至 ILM 中进行执行。该链接脚本代码片段及解释如下:

```
//bsp/nuclei-n200/n200/env/link_flash.lds 代码片段

ENTRY( _start ) //指明程序入口为_start 标签

MEMORY
{
```



```
//定义了三块地址区间, 分别名为 flash, ilm 和 ram, 对应 Flash, ILM 和 DLM 的地址区间
flash (rxai!w) : ORIGIN = 0x20000000, LENGTH = 4M
ilm (rxai!w) : ORIGIN = 0x80000000, LENGTH = 64K
ram (wxa!ri) : ORIGIN = 0x90000000, LENGTH = 64K
}
```

SECTIONS

```
{
    __stack_size = DEFINED(__stack_size) ? __stack_size : 2K;
```

```
.init      :
{
    KEEP (*(SORT_NONE(.init)))
} >flash AT>flash
```

//注意: 上述语法中 AT 前的一个 flash 表示该段的运行地址, AT 后的 flash 表示该段的物理地址。有关此语法的详细细节请用户自行搜索学习 GCC Link 脚本语法。

//物理地址是该程序要被存储的存储器地址 (调试器下载程序之时会遵从此物理地址进行下载), 运行地址却是指程序真正运行起来后所处于的地址, 所以程序中的相对寻址都会遵从此运行地址。

//注意: 上述 .init 段为上电引导程序所处的段, 所以它直接在 Flash 里面执行, 所以其运行地址和物理地址相同, 都是 flash 区间。

```
.ilalign    :
{
    . = ALIGN(4);
PROVIDE( _ilm_lma = . ); //创建一个标签名为_ilm_lma, 地址为 flash 地址区间
                        //的起始地址
```

```
} >flash AT>flash
```

```
.ialign     :
{
PROVIDE( _ilm = . ); //创建一个标签名为_ilm, 地址为 ilm 地址区间
                        //的起始地址
```

```
} >ilm AT>flash
```

```
.text      :
{
    *(.vtable_ilm) //中断向量表 (ILM)
    *(.text.unlikely .text.unlikely.*)
    *(.text.startup .text.startup.*)
    *(.text .text.*)
    *(.gnu.linkonce.t.*)
} >ilm AT>flash
```

//注意: 由于此 “链接脚本” 意图是让程序存储在 Flash 之中, 而上载至 ILM 中进行运行, 所以其物理地址和运行地址不同, 所以, 上述 .text 代码段的物理地址是 flash 区间, 而运行地址为 ilm 区间

```
.data      :
{
    *(.rdata)
    *(.rodata .rodata.*)
    *(.gnu.linkonce.r.*)
}
```

```

* (.data .data.*)
* (.gnu.linkonce.d.*)
. = ALIGN(8);

PROVIDE( __global_pointer$ = . + 0x800 ); //创建一个标签名为__global_pointer$
* (.sdata .sdata.* .sdata*)
* (.gnu.linkonce.s.* )
. = ALIGN(8);
* (.srodata.cst16)
* (.srodata.cst8)
* (.srodata.cst4)
* (.srodata.cst2)
* (.srodata .srodata.*)
} >ram AT>flash

```

//注意：由于此“链接脚本”意图是让数据存储在 Flash 之中，而将数据段上载至 DLM 中进行运行，所以数据段物理地址和运行地址不同，所以，上述 .data 数据段的物理地址是 flash 区间，而运行地址为 ram 区间

2.3.4. 系统启动引导程序

嵌入式系统上电后执行的第一段软件代码是引导程序，该程序往往由用汇编语言编写。

N200-SDK 平台的引导程序为 bsp/nuclei-n200/n200/env/start.S，该程序由汇编语言编写。

start.S 代码中主要完成一些基本配置，如果有需要还会将代码从 Flash 上载至 ILM 中（即将 Flash 中的代码搬运到 ILM 中）。

■ start.S 代码解读

start.S 代码片段和功能解释如下：

```

// start.S 文件代码片段

.section .init //声明此处的 section 名为 .init
.globl _start //指明标签_start 的属性为全局性的
.type _start,@function

_start: //标签名_start 处于此处

/*置位 CSR_MMISC_CTL 寄存器设置 mtvec 的值为 NMI 的基地址
li t0, (0x1 << 9);
csrs CSR_MMISC_CTL, t0

/*初始化 mtvt 寄存器*/
la t0, vector_base
csrw CSR_MTVT, t0

/* 初始化 mtvt2 寄存器并且使能*/
la t0, irq_entry
csrw CSR_MTVT2, t0
csrs CSR_MTVT2, 0x1

```

```

/*初始化 mvec 为 NMI 和异常共用的基地址*/
la t0, trap_entry
csrw CSR_MTVEC, t0

//下列代码通过将 CSR 寄存器 MSTATUS 的 FS 域设置为非零值, 从而将 FPU 打开使能。

#ifdef __riscv_flen //只有定义了此宏 (即意味着支持浮点指令) 时, 才需要执行下列打开 FPU
    //的操作。
    li t0, MSTATUS_FS
    csrs mstatus, t0 //向 MSTATUS 的 FS 域设置非零值
    csrw fcsr, x0 //初始化 fcsr 的值为 0
#endif

.option push
.option norelax
//设置全局指针
la gp, __global_pointer$ //将标签__global_pointer$所处的地址赋值给 gp 寄存器
//注意: 标签__global_pointer 在链接脚本中定义参见链接脚本的__global_pointer$标签
.option pop

//设置堆栈指针
la sp, _sp //将标签_sp 所处的地址赋值给 sp 寄存器
//注意: 标签_sp 在链接脚本中定义, 参见链接脚本的_sp 标签

//下列代码判断_ilm_lma 与_ilm 标签的地址值是否相同:
// 如果相同则意味着代码直接从 Flash 中进行执行 (link_flashxip.lds 中定义的_ilm_lma 与_ilm 标签地址相
等), 那么直接跳转到后面数字标签 2 所在的代码继续执行;
//如果不相同则意味着代码需要从 Flash 中上载至 ILM 中进行执行 (link_flash.lds 中定义的_ilm_lma 与_ilm 标签地
址不相等), 因此使用 lw 指令逐条地将指令从 Flash 中读取出来, 然后使用 sw 指令逐条地写入 ILM 中, 通过此方式完成指
令的上载至 ILM 中。

la a0, _ilm_lma//将标签_ilm_lma 所处的地址赋值给 a0 寄存器
//注意: 标签_ilm_lma 在链接脚本中定义, 参见链接脚本的_ilm_lma 标签
la a1, _ilm//将标签_ilm 所处的地址赋值给 a1 寄存器
//注意: 标签 a1 在链接脚本中定义, 参见链接脚本的 a1 标签
beq a0, a1, 2f //a0 和 a1 的值分别为标签_ilm_lma 和_ilm 标签的地址, 判断其
//是否相等, 如果相等则直接跳到后面的数字 "2" 标签所在的地方,
//如果不等则继续向下执行

la a2, _eilm //将标签_eilm 所处的地址赋值给 a2 寄存器
//注意: 标签_eilm 在链接脚本中定义, 参见链接脚本的_eilm 标签

//通过一个循环, 将指令从 Flash 中搬到 ILM 中
bgeu a1, a2, 2f //如果_ilm 标签地址比_eilm 标签地址还大, 则属于不正常的配置,
//放弃搬运, 直接跳转到后面数字标签 2 所在的位置

```

1:

```

    lw t0, (a0)          //从地址指针 a0 所在的位置 (Flash 中) 读取 32 位数
    sw t0, (a1)          //将读取的 32 位数写入地址指针 a1 所在的位置 (ILM 中)
    addi a0, a0, 4       //将地址指针 a0 寄存器加 4 (即 32 位)
    addi a1, a1, 4       //将地址指针 a0 寄存器加 4 (即 32 位)
    bltu a1, a2, 1b      //跳转回之前数字标签 1 所在的位置
2:

/* 使用与上述相同的原理, 通过一个循环, 将 数据从 Flash 中搬运到 DLM 中*/
la a0, _data_lma
la a1, _data
la a2, _edata
bgeu a1, a2, 2f
1:
    lw t0, (a0)
    sw t0, (a1)
    addi a0, a0, 4
    addi a1, a1, 4
    bltu a1, a2, 1b
2:

//BSS 段是链接器预留的未初始化变量所处的地址段, 引导程序必须对其初始化为 0
//此处通过一个循环来初始化 BSS 段
la a0, __bss_start
la a1, _end
bgeu a0, a1, 2f
1:
    sw zero, (a0)
    addi a0, a0, 4
    bltu a0, a1, 1b
2:

/* 以下调用 Newlib 全局的构造函数 (Global constructors) */
//
la a0, __libc_fini_array //将标签 __libc_fini_array 的值赋给 a0 作为函数参数
call atexit              //调用 atexit 函数 (Newlib 的函数)
call __libc_init_array   //调用 __libc_init_array (Newlib 的函数)
//
//注意: 上述的 __libc_fini_array, atexit 和 __libc_init_array 函数都是 Newlib C
//      运行库的特殊库函数, 用于处理一些 C/C++ 程序中的全局性的构造和析构函数。本文档在
//      此对其不做详细介绍, 请用户自行查阅相关资料学习。
//
//值得注意的是: __libc_init_array 函数中会调用一个名为 _init 的函数, N200-SDK
//环境中的 _init 函数定义在 bsp/nuclei-n200/n200/env/init.c 中, 因此此处会执行该函数, 后
//文对此 _init.c 文件将进行进一步介绍。

//调用 main 函数
//根据 ABI 调用原则, 函数调用时由 a0 和 a1 寄存器传递参数, 因此此处赋参数值给 a0 和 a1
/* argc = argv = 0 */
li a0, 0
li a1, 0
call main //调用 main 函数, 开始执行 main 函数
tail exit //如果完成了 main 函数后, 调用 exit 函数 (Newlib 桩函数之一, 参见第 2.3.1

```

```

//节了解 Newlib 桩函数的更多信息)

1:
    j 1b    //最后的死循环，程序理论上不可能执行到此处

.global disable_mcycle_minstret
disable_mcycle_minstret:    //用于控制计数器的关闭(低功耗考虑)
    csrsi CSR_MCOUNTINHIBIT, 0x5
    ret

.global enable_mcycle_minstret
enable_mcycle_minstret:    //用于控制计数器的开启
    csrci CSR_MCOUNTINHIBIT, 0x5
    ret

.global core_wfe
core_wfe:    //休眠时，Wait for Event
    csrc CSR_MSTATUS, MSTATUS_MIE
    csrs CSR_WFE, 0x1

    wfi    //WFI 的休眠模式
    csrc CSR_WFE, 0x1
    csrs CSR_MSTATUS, MSTATUS_MIE
    ret

```

■ init.c 代码解读

如 **start.S** 代码中所述，在执行 `__libc_init_array` 函数时会调用一个名为 `_init` 的函数，而 N200-SDK 平台中的 `_init` 函数定义在 `bsp/nuclei-n200/n200/env/init.c` 中。

`init.c` 文件中的 `_init` 函数定义和功能解释如下：

```

//bsp/nuclei-n200/n200/env/init.c 代码片段

//_init 函数声明
void _init()
{
    #ifndef NO_INIT
        soc_init(); //调用 soc_init 函数进行配套 SoC 的初始化。soc_init 函数主要调用 uart_init
                    // 函数，对 UART 模块进行设置。如第 2.3.2 节中所述，
                    // 由于 UART 是支持 printf 函数输出的物理接口，所以必须对 UART
                    // 进行正确的设置。

        //打印当前 core 的运行频率，此处调用了 get_cpu_freq() 函数来计算当前运行频率。参见后文
        //对此函数的详解。
        printf("*****\n");
        printf("*****\n");
        printf("**                *\n");
        printf("Core freq at %d Hz\n", get_cpu_freq());
        printf("**                *\n");
        printf("*****\n");
    #endif
}

```

```

    printf("*****\n");

//ECLIC 初始化并且使能 eclic
eclic_init(ECLIC_NUM_INTERRUPTS);
eclic_mode_enable();

disable_mcycle_minstret(); //在进入main 函数之前，将计数器关闭。(低功耗考虑)
#endif
}

_init 函数中调用到的若干功能函数解释如下：

//uart_init 函数实现（来自于 bsp/nuclei-n200/soc/drivers/soc_func.c 代码片段）
static void uart_init(size_t baud_rate)//参数为波特率
{
    //设置 UART0 相关的寄存器和 GPIO 相关寄存器
    GPIO_REG(GPIO_IOF_SEL) &= ~IOF0_UART0_MASK;
    GPIO_REG(GPIO_IOF_EN) |= IOF0_UART0_MASK;
    UART0_REG(UART_REG_DIV) = get_cpu_freq() / baud_rate - 1;
    UART0_REG(UART_REG_TXCTRL) |= UART_TXEN;
    UART0_REG(UART_REG_RXCTRL) |= UART_RXEN;
}

//get_cpu_freq 函数实现（来自于 bsp/nuclei-n200/n200/drivers/n200_func.c 代码片段）
unsigned long get_cpu_freq()
{
    uint32_t cpu_freq;

    // warm up
    measure_cpu_freq(1);
    // measure for real
    cpu_freq = measure_cpu_freq(100); //调用 measure_cpu_freq 函数

    return cpu_freq;
}

//measure_cpu_freq 函数实现（来自于 bsp/nuclei-n200/n200/drivers/n200_func.c 代码片段）
static unsigned long __attribute__((noinline)) measure_cpu_freq(size_t n)
{
    unsigned long start_mtime, delta_mtime;
    unsigned long mtime_freq = get_timer_freq();

    // Don't start measuring until we see an mtime tick
    unsigned long tmp = mtime_lo();
    do {
        start_mtime = mtime_lo();
    } while (start_mtime == tmp); //不断观察 MTIME 计数器并将其值作为初始时间值

    //通过读取 CSR 寄存器 MCYCLE 得到当前时钟周期，并作为 初始计数值
    unsigned long start_mcycle = read_csr(mcycle);

    do {
        delta_mtime = mtime_lo() - start_mtime;
    } while (delta_mtime < n); //不断观察 MTIME 计数器直到其值等于函数参数设定的目标值

    //通过读取 CSR 寄存器 MCYCLE 得到当前时钟周期，并与初始计数值相减得到这段时间消耗的时钟周期

```

```

    unsigned long delta_mcycle = read_csr(mcycle) - start_mcycle;

//由于 MTIME 计数器的频率是 Always-On Domain 的参考频率 (譬如 32.768KHz), 而 Core 的运行频率与 CSR 寄存器
MCYCLE 的值一致。有关 N200 系列配套 SoC 的时钟域划分, 请参见单独文档《Nuclei_N200 系列配套 SoC 介绍》。
//因此可以通过 MCYCLE 和 MTIME 的相对关系计算出当前 Core 的时钟频率。
    return (delta_mcycle / delta_mtime) * mtime_freq
        + ((delta_mcycle % delta_mtime) * mtime_freq) / delta_mtime;
}

//eclic_init 函数实现 (来自于 bsp/nuclei-n200/soc/drivers/soc_func.c 代码片段)
void eclic_init ( uint32_t num_irq )
{
    typedef volatile uint32_t vuint32_t;

    //clear cfg register
    *(volatile uint8_t*)(ECLIC_ADDR_BASE+ECLIC_CFG_OFFSET)=0;//清除 cfg 寄存器

    //clear minthresh register
    *(volatile uint8_t*)(ECLIC_ADDR_BASE+ECLIC_MTH_OFFSET)=0;//清除 minthresh 寄存器

    //清除所有中断源的 IP/IE/ATTR/CTRL 位域
    vuint32_t * ptr;

    vuint32_t * base = (vuint32_t*)(ECLIC_ADDR_BASE + ECLIC_INT_IP_OFFSET);
    vuint32_t * upper = (vuint32_t*)(base + num_irq*4);

    for (ptr = base; ptr < upper; ptr=ptr+4){
        *ptr = 0;
    }
}

//eclic_mode_enable 函数实现 (来自于 bsp/nuclei-n200/soc/drivers/soc_func.c 代码片段)
void eclic_mode_enable() { //使能 eclic 模式 (区别于使能 eclic 向量模式)
    uint32_t mtvec_value = read_csr(mtvec);
    mtvec_value = mtvec_value & 0xFFFFF0;
    mtvec_value = mtvec_value | 0x00000003;
    write_csr(mtvec,mtvec_value);
}

```

2.3.5. 异常、中断和 NMI 处理

本节需要了解 N200 处理器内核的中断、异常和 NMI 相关知识, 请参见《Nuclei_N200 系列指令架构手册》了解详情。

N200-SDK 平台的 BSP 中已经将中断和异常处理的基础框架实现, 使得普通应用开发人员无需关心底层这些细节。

板级支持包中对于中断和异常处理基础框架实现的相关源代码介绍如下。

■ 设置 mtvec、mtvt、mtvt2 寄存器的值

如《Nuclei_N200 系列指令架构手册》中所述：

- N200 系列内核在程序执行过程中，一旦遇到异常或者 NMI，则终止当前的程序流，处理器被强行跳转到新的 PC 地址（异常处理程序的入口），该地址由 **mtvec** 寄存器指定。

- 中断向量表的起始地址由 CSR 寄存器 **mtvt** 指定，通常可以将 **mtvt** 寄存器设置为 0（上电复位默认值），即，将中断向量表起始地址设置在整个存储器空间的 0 地址，中断向量表之后才是真正的上电开机启动程序代码。

- 如果配置 CSR 寄存器 **mtvt2** 的最低位为 0（上电复位默认值），则所有非向量中断共享的入口地址由 CSR 寄存器 **mtvec** 的值（忽略最低 2 位的值）指定。由于 **mtvec** 寄存器的值也指定异常的入口地址，因此，意味着在这种情况下，异常和所有非向量中断共享入口地址。

- 如果配置 CSR 寄存器 **mtvt2** 的最低位为 1，则所有非向量中断共享的入口地址由 CSR 寄存器 **mtvt2** 的值（忽略最低 2 位的值）指定。为了让中断以尽可能快的速度被响应和处理，推荐将 CSR 寄存器 **mtvt2** 的最低位设置为 1，即，由 **mtvt2** 指定一个独立的入口地址供所有非向量中断专用，和异常的入口地址（由 **mtvec** 的值指定）彻底分开。

因此，在系统启动引导程序中需要设置 **mtvec**、**mtvt2**、**mtvt** 寄存器的值，使其指向异常、中断处理函数的入口和中断向量表地址。

■ 异常入口程序 **trap_entry**

如《Nuclei_N200 系列指令架构手册》中所述：

- 由于 N200 系列内核进入异常和退出异常机制中没有硬件自动保存和恢复上下文的操作，因此需要软件明确地使用（汇编语言编写的）指令进行上下文的保存和恢复。

trap_entry 函数即为使用汇编语言编写的异常入口程序，该函数位于 **bsp/nuclei-n200/n200/env/entry.S** 中，主要用于上下文的保存和恢复，其代码如下：

```
// bsp/nuclei-n200/n200/env/entry.S 代码片段

//该宏用于保存 ABI 定义的“调用者应保持的寄存器 (Caller saved register)” 进入堆栈
.macro SAVE_CONTEXT

    STORE x1, 0*REGBYTES(sp)
    STORE x5, 1*REGBYTES(sp)
    STORE x6, 2*REGBYTES(sp)
    STORE x7, 3*REGBYTES(sp)
    STORE x10, 4*REGBYTES(sp)
```



```

STORE x11, 5*REGBYTES(sp)
STORE x12, 6*REGBYTES(sp)
STORE x13, 7*REGBYTES(sp)
STORE x14, 8*REGBYTES(sp)
STORE x15, 9*REGBYTES(sp)
STORE x16, 10*REGBYTES(sp)
STORE x17, 11*REGBYTES(sp)
STORE x28, 12*REGBYTES(sp)
STORE x29, 13*REGBYTES(sp)
STORE x30, 14*REGBYTES(sp)
STORE x31, 15*REGBYTES(sp)
.endm

//该宏用于从堆栈中恢复 ABI 定义的“调用者应保存的寄存器 (Caller saved register)”
.macro RESTORE_CONTEXT
LOAD x1, 0*REGBYTES(sp)
LOAD x5, 1*REGBYTES(sp)
LOAD x6, 2*REGBYTES(sp)
LOAD x7, 3*REGBYTES(sp)
LOAD x10, 4*REGBYTES(sp)
LOAD x11, 5*REGBYTES(sp)
LOAD x12, 6*REGBYTES(sp)
LOAD x13, 7*REGBYTES(sp)
LOAD x14, 8*REGBYTES(sp)
LOAD x15, 9*REGBYTES(sp)
LOAD x16, 10*REGBYTES(sp)
LOAD x17, 11*REGBYTES(sp)
LOAD x28, 12*REGBYTES(sp)
LOAD x29, 13*REGBYTES(sp)
LOAD x30, 14*REGBYTES(sp)
LOAD x31, 15*REGBYTES(sp)

.endm

#该宏用于从堆栈中恢复(除了 x5)ABI 定义的“调用者应保存的寄存器 (Caller saved register)”
#restore caller registers
.macro RESTORE_CONTEXT_EXCPT_X5
LOAD x1, 0*REGBYTES(sp)
LOAD x6, 2*REGBYTES(sp)
LOAD x7, 3*REGBYTES(sp)
LOAD x10, 4*REGBYTES(sp)
LOAD x11, 5*REGBYTES(sp)
LOAD x12, 6*REGBYTES(sp)
LOAD x13, 7*REGBYTES(sp)
LOAD x14, 8*REGBYTES(sp)
LOAD x15, 9*REGBYTES(sp)
LOAD x16, 10*REGBYTES(sp)
LOAD x17, 11*REGBYTES(sp)
LOAD x28, 12*REGBYTES(sp)
LOAD x29, 13*REGBYTES(sp)
LOAD x30, 14*REGBYTES(sp)
LOAD x31, 15*REGBYTES(sp)
.endm

#该宏用于从堆栈中恢复 ABI 定义的 x5 寄存器
.macro RESTORE_CONTEXT_ONLY_X5
LOAD x5, 1*REGBYTES(sp)
.endm

//该宏用于保存 MEPC 和 MSTATUS 寄存器进入堆栈
.macro SAVE_EPC_STATUS

```

```
csrr x5, CSR_MEPC
STORE x5, 16*REGBYTES(sp)
csrr x5, CSR_MSTATUS
STORE x5, 17*REGBYTES(sp)
csrr x5, CSR_MSUBM
STORE x5, 18*REGBYTES(sp)
.endm

//该宏用于从堆栈中恢复 MEPC 和 MSTATUS 寄存器
.macro RESTORE_EPC_STATUS
LOAD x5, 16*REGBYTES(sp)
csrw CSR_MEPC, x5
LOAD x5, 17*REGBYTES(sp)
csrw CSR_MSTATUS, x5
LOAD x5, 18*REGBYTES(sp)
csrw CSR_MSUBM, x5
.endm

.section .text.trap
.align 6 // In CLIC mode, the trap entry must be 64bytes aligned
.global trap_entry

.weak trap_entry //指定该标签为 weak 类型，标签为“弱 (weak)”属性。“弱 (weak)”属性是
// C/C++语法中定义的一种属性，一旦有具体的“非弱”性质同名函数存在，将
//会覆盖此函数。

trap_entry: //定义标签名 trap_entry，该标签名作为函数入口

//更改堆栈指针，分配 19 个单字（32 位）的空间用于保存寄存器
addi sp, sp, -19*REGBYTES
//进入异常处理函数之前必须先保存处理器的上下文
SAVE_CONTEXT
//此处调用 SAVE_CONTEXT 保存 ABI 定义的“调用者应存储的寄存器 (Caller saved
//register)”进入堆栈

SAVE_EPC_MSTATUS
//此处调用 SAVE_EPC_MSTATUS 保存 MEPC 和 MSTATUS 寄存器进入堆栈

//调用 handle_trap 函数
//根据 ABI 调用原则，函数调用时由 a0 和 a1 寄存器传递参数，因此此处赋参数值给 a0 和 a1
csrr a0, mcause //参数 1
mv a1, sp //参数 2
call handle_trap //调用 handle_trap 函数

//在退出异常处理函数之后需要恢复之前保存的处理器上下文
RESTORE_EPC_MSTATUS
//此处调用 RESTORE_EPC_MSTATUS 从堆栈中恢复 MEPC 和 MSTATUS 寄存器
RESTORE_CONTEXT
//调用 RESTORE_CONTEXT 从堆栈中恢复 ABI 定义的“调用者应存储的寄存器 (Caller saved
//register)”

//恢复寄存器之后，更改堆栈指针，回收 19 个单字（32 位）的空间
addi sp, sp, 19*REGBYTES
mret //使用 mret 指令从异常模式返回
```

■ 异常处理函数 handle_trap

handle_trap 函数是使用 C/C++ 语言编写的中断和异常处理函数，该函数位于 bsp/nuclei-n200/n200/env/handlers.c 中，其代码如下：

```
//bsp/nuclei-n200/n200/env/handlers.c 代码片段

//该函数理论上需要由用户自行填写，此处的函数内容仅作为一个示例以打印进入异常后的信息。所以此处
//指定该标签为 weak 类型，标签为“弱 (weak)”属性。“弱 (weak)”属性是 C/C++ 语法中定义的一种属性，一旦用户定义有具体的“非弱”性质同名函数存在，将会覆盖此函数。__attribute__((weak)) uintptr_t uintptr_t

__attribute__((weak)) uintptr_t handle_trap(uintptr_t mcause, uintptr_t sp)
{
    if(mcause == 0xFFFF) {
        handle_nmi;
    }
    write(1, "trap\n", 5);
    printf("In trap handler, the mcause is %d\n", mcause);
    printf("In trap handler, the mepc is 0x%x\n", read_csr(mepc));
    printf("In trap handler, the mtval is 0x%x\n", read_csr(mbadaddr));
    _exit(mcause);
    return 0;
}
```

■ 中断入口程序 irq_entry

如《Nuclei_N200 系列指令架构手册》中所述：

- 由于 N200 系列内核进入中断和退出中断机制中没有硬件自动保存和恢复上下文的操作，因此需要软件明确地使用（汇编语言编写的）指令进行上下文的保存和恢复。

irq_entry 函数即为使用汇编语言编写的异常入口程序，该函数位于 bsp/nuclei-n200/n200/env/entry.S 中，主要用于上下文的保存和恢复，其代码如下：

```
// bsp/nuclei-n200/n200/env/entry.S 代码片段

.section      .text.irq
.align 2
.global irq_entry
.weak irq_entry //指定该标签为 weak 类型，标签为“弱 (weak)”属性。“弱 (weak)”属性是
// C/C++ 语法中定义的一种属性，一旦有具体的“非弱”性质同名函数存在，将
//会覆盖此函数。
irq_entry:    //定义标签名 irq_entry，该标签名作为函数入口

//更改堆栈指针，分配 19 个单字（32 位）的空间用于保存寄存器
    addi sp, sp, -19*REGBYTES

//进入中断处理函数之前必须先保存处理器的上下文
```

```

SAVE_CONTEXT
//此处调用 SAVE_CONTEXT 保存 ABI 定义的“调用者应存储的寄存器 (Caller saved
//register)” 进入堆栈

//-----This special CSR read operation, which is actually use mcause as operand to directly
store it to memory
//以指令的方式保存 MCAUSE 寄存器
csrrwi x0, CSR_PUSHMCAUSE, 16
//以指令的方式保存 MEPC 寄存器
csrrwi x0, CSR_PUSHMEPC, 17
//以指令的方式保存 MSUBM 寄存器
csrrwi x0, CSR_PUSHMSUBM, 18

service_loop:
// 响应当前等待级别最高的中断，进入对应的中断处理函数
csrrw ra, CSR_JALMNXTI, ra

RESTORE_CONTEXT_EXCPT_X5

DISABLE_MIE # 关闭全局中断

LOAD x5, 18*REGBYTES(sp)
csrw CSR_MSUBM, x5
LOAD x5, 17*REGBYTES(sp)
csrw CSR_MEPC, x5
LOAD x5, 16*REGBYTES(sp)
csrw CSR_MCAUSE, x5

RESTORE_CONTEXT_ONLY_X5

//恢复寄存器之后，更改堆栈指针，回收 19 个单字（32 位）的空间
addi sp, sp, 19*REGBYTES
mret //使用 mret 指令从异常模式返回

```

■ NMI 处理函数 handle_nmi

handle_nmi 函数是使用 C/C++ 语言编写的 NMI 处理函数，该函数位于 bsp/nuclei-n200/n200/env/handlers.c 中，其代码如下：

```

//bsp/nuclei-n200/n200/env/handlers.c 代码片段

//该函数理论上需要由用户自行实现，此处的函数内容仅作为一个示例以打印进入 NMI 后的信息。所以此处指定该标签为
weak 类型，标签为“弱 (weak)”属性。“弱 (weak)”属性是 C/C++ 语法中定义的一种属性，一旦用户定义有具体的“非弱”
性质同名函数存在，将会覆盖此函数。
__attribute__((weak)) uintptr_t handle_nmi()
{
    write(1, "nmi\n", 5);
    _exit(1);
    return 0;
}

```

2.3.6. 使用 newlib-nano

`newlib-nano` 是一个特殊的 `newlib` 版本，它提供了更加精简版本的 `malloc` 和 `printf` 函数的实现，并且对所有库函数使用 GCC 的 `-Os`（对于代码体积“Code Size”的优化）选项进行编译优化。

因此，在嵌入式系统中，推荐使用 `newlib-nano` 版本作为 C 运行库。如果需要使用 `newlib-nano` 版本，需要如下步骤：

- 在 GCC 的链接步骤时使用选项（`--specs=nano.specs`）来指定使用 `newlib-nano` 作为链接库。
- 如果不需要使用系统调用，还可以在链接时添加选项（`--specs=nosys.specs`）来指定使用空的桩函数来进行链接。
- 默认的 `newlib-nano` 的精简版 `printf` 是不支持浮点数的，如果需要输出浮点数，那么需要额外再加上一个选项（`-u _printf_float`）来指定支持浮点数的格式输出。注意：添加此选项后会造成代码体积一定的膨胀，因为它需要链接更多的浮点相关的函数库。

在 N200-SDK 平台中使用的是 `newlib-nano` 版本，可以控制是否支持浮点数的格式输出。相关脚本的代码片段和解释如下：

```
//n200-sdk/目录下的 Makefile 片段

PFLOAT      := 1      //在此 Makefile 中有一个变量控制是否需要 newlib-nano 版本的
                        //printf 支持浮点数，默认为 1
```

2.4. 使用 N200-SDK 开发和编译程序

2.4.1. 在 N200-SDK 环境中安装工具链

因为编译程序需要使用到 RISC-V GCC 交叉编译工具链，所以本节先介绍如何在 N200-SDK 环境中安装预先编译好的 GCC 工具链，步骤如下：

// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置：

- （1）使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。
 - （2）由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统
- 有关如何安装 VMware 以及 Ubuntu 操作系统本文档不做介绍，有关 Linux 的基本使用本文档也不做介绍，请用户自行查阅资料学习。

// 步骤二：将 N200-SDK 项目下载到本机 Linux 环境中，使用如下命令：

```
git clone https://github.com/nucleisys/n200-sdk.git
// 经过此步骤将项目克隆下来，本机上即可具有有如第 2.2 节中所述完整的
// n200-sdk 目录文件夹，假设该目录为<your_sdk_dir>，后文将使用该缩
// 写指代。

// 步骤三：由于编译软件程序需要使用到 GNU 工具链，假设使用完整的 riscv-tools 来自己编译 GNU 工具链则费时费力，因此本文档推荐使用预先已经编译好的 GCC 工具链。用户可以在芯来科技公司网站的下载中心，下载到最新的 RISC-V GCC 工具链，下载中心地址记载于 https://github.com/nucleisys/n200-sdk/tree/master/prebuilt\_tools 中。
// 用户在下载中心，可以下载到压缩包
// gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz 和
// gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-centos64.tgz，然后按照如下步骤解压使用（注意：
// 下载中心内的工具链可能会不断更新，用户请注意自行判断使用最新日期的版本，下列步骤仅为特定版本的示例）。

cp gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz ~/
cp gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-centos64.tgz ~/
// 将两个压缩包均拷贝到用户的根目录下

cd ~/
tar -xzf gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz
tar -xzf gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-centos64.tgz
// 进入根目录并解压上述两个压缩包，解压后可以看到一个生成的 gnu-mcu-eclipse 文件夹

cd <your_sdk_dir>
// 进入 n200-sdk 目录文件夹
mkdir -p work/build/openocd/prefix
// 在 n200-sdk 目录下创建上述这个 prefix 目录
cd work/build/openocd/prefix
// 进入到 prefix 该目录

ln -s ~/gnu-mcu-eclipse/openocd/0.10.0-6-20180112-1448/bin bin
// 将用户根目录下解压的 OpenOCD 目录下的 bin 目录作为软链接链接到该 prefix 目录下

cd <your_sdk_dir>
// 再次进入到 n200-sdk 目录文件夹
mkdir -p work/build/riscv-gnu-toolchain/riscv32-unknown-elf/prefix/
// 在 n200-sdk 目录下创建上述这个 prefix 目录
cd work/build/riscv-gnu-toolchain/riscv32-unknown-elf/prefix
// 进入到 prefix 该目录

ln -s ~/gnu-mcu-eclipse/riscv-none-gcc/7.2.0-2-20180111-2230/bin bin
// 将用户根目录下解压的 GNU Toolchain 目录下的 bin 目录作为软链接链接到该 prefix 目录下
```

2.4.2. 在 N200-SDK 环境中开发程序

本节以一个简单的 Hello World 程序为例，介绍如何在 N200-SDK 环境中开发一个应用程序。其步骤如下：

- 步骤一：在 n200-sdk/software 目录下创建一个 hello_world 的文件夹
- 步骤二：在 n200-sdk/software/hello_world 目录下创建一个文件 hello_world.c，其内容如下：

```
#include <stdio.h>

int main(void)
{
//简单的 Printf 输出 Hello World 字符串
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    printf("Hello World!" "\n");
    return 0;
}
```

- 步骤三：在 n200-sdk/software/hello_world 目录下创建一个文件 Makefile，其内容如下：

```
TARGET = hello_world //指明生成的 elf 文件名

CFLAGS += -O2 //指明程序所需要的特别的 GCC 编译选项，此处指明使用 GCC 的 O2 优化级别

BSP_BASE = ../../bsp

C_SRCS += hello_world.c //指明程序所需要的 C 源文件

//调用板级支持包 (bsp) 目录下的 common.mk
include $(BSP_BASE)/$(BOARD)/n200/env/common.mk
```

经过上述步骤之后，Hello World 程序在 n200-sdk 的相关代码结构如下所示。

```
n200-sdk          // 存放 n200-sdk 的目录
|----software     // 存放示例程序的源代码
|----hello_world  // Hello World 示例程序目录
|----hello_world.c //Hello World 源代码
|----Makefile     //Makefile 脚本
```


2.4.3. 编译使得程序从 Flash 直接运行

如第 2.3.3 节中所述,通过系统链接脚本 `link_flashxip.lds` 可以控制将程序段存放在 Flash 中,并且使得代码段的物理地址和运行地址完全一致,那么在上电系统引导程序(参见 2.3.4 节所述 `start.S`)中便不会将程序上载至 ILM 中运行,而是直接在 Flash 中运行。

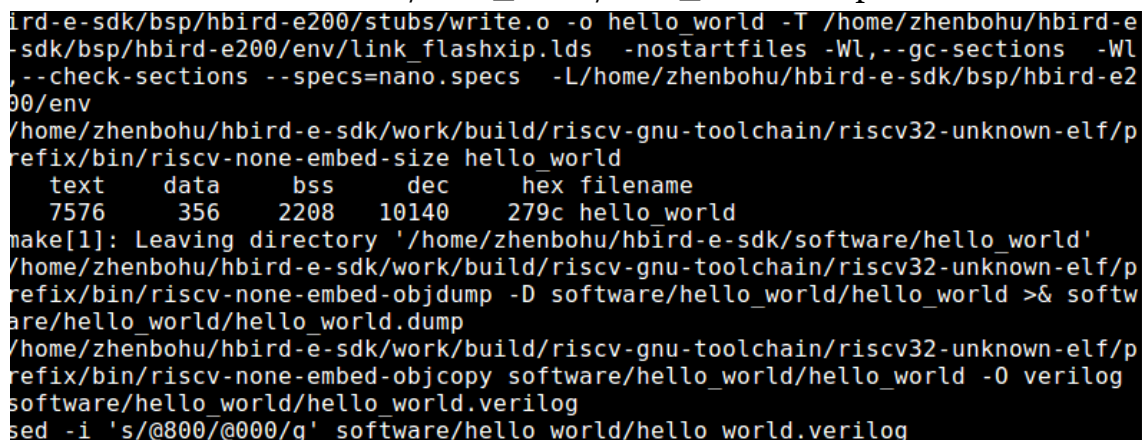
以 2.4.2 节中开发的 Hello World 程序为例,在 N200-SDK 平台中进行编译时,使用如下命令选项将会使用链接脚本 `link_flashxip.lds`:

```
// 注意: 确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链, 请参见本文档第 2.4.1 了解其详情。

cd n200-sdk
    // 确保目前处于 n200-sdk 目录下。

make dasm PROGRAM=hello_world BOARD=nuclei-n200 CORE=n201 DOWNLOAD=flashxip
NANO_PFLOAT=0
// 上述命令使用到了如下几个 Makefile 参数, 分别解释如下:
//    dasm: 该选项表示对程序进行编译, 并且对可执行文件(elf 文件)进行反汇编(生成.dump 文件)。
//    DOWNLOAD=flashxip: 指明采用“将程序从 Flash 直接运行的方式”进行编译, 即选择使用链接脚本
link_flashxip.lds。
//    PROGRAM=hello_world: 指定需要编译 software/hello_world 目录下的示例程序。
//    BOARD=nuclei-n200: 确定 BSP 型号, 指明需要使用 bsp/nuclei-n200 目录下的 BSP。
//    CORE=n201: 指明 N200 系列的具体处理器内核型号, 此处的示例是指明 n201, 但是用户需要按照具体型号进行
指明, 譬如如果使用 N201 的用户此处需要指明 CORE=n201。
//    NANO_PFLOAT=0: 由于 Hello World 程序的 printf 函数不需要输出浮点数, 指明 newlib-nano 的 printf
函数无需支持浮点数, 请参见本文档第 2.3.6 节了解相关信息。
```

编译成功后在终端的显示信息如图 2-1 中所示,从其中可以看出代码尺寸信息(7576),并且可以看到反汇编文件生成在 `software/hello_world/hello_world.dump` 中。



```
bird-e-sdk/bsp/hbird-e200/stubs/write.o -o hello_world -T /home/zhenbohu/hbird-e
-sdk/bsp/hbird-e200/env/link_flashxip.lds -nostartfiles -Wl,--gc-sections -Wl
,--check-sections --specs=nano.specs -L/home/zhenbohu/hbird-e-sdk/bsp/hbird-e2
00/env
/home/zhenbohu/hbird-e-sdk/work/build/riscv-gnu-toolchain/riscv32-unknown-elf/p
refix/bin/riscv-none-embed-size hello_world
text    data    bss    dec    hex filename
7576    356    2208    10140    279c hello_world
make[1]: Leaving directory '/home/zhenbohu/hbird-e-sdk/software/hello_world'
/home/zhenbohu/hbird-e-sdk/work/build/riscv-gnu-toolchain/riscv32-unknown-elf/p
refix/bin/riscv-none-embed-objdump -D software/hello_world/hello_world >& softw
are/hello_world/hello_world.dump
/home/zhenbohu/hbird-e-sdk/work/build/riscv-gnu-toolchain/riscv32-unknown-elf/p
refix/bin/riscv-none-embed-objcopy software/hello_world/hello_world -O verilog
software/hello_world/hello_world.verilog
sed -i 's/@800/@000/g' software/hello_world/hello_world.verilog
```

图 2-1 编译 Hello World 程序使得程序从 Flash 直接运行

如果使用此种方式进行编译，按照第 2.5 节中所述的步骤下载程序至开发板，然后按照第 2.6.12.6.1 节中所述的步骤在开发板上运行程序，通过其打印到 PC 中断上的字符串显示速度可以看出其运行速度非常之慢，这是因为程序直接从 Flash 中运行需要每次都从 Flash 中取指令，取指时间较长，影响了程序的执行速度。

2.4.4. 编译使得程序从 ILM 中运行

如第 2.3.3 节中所述，通过系统链接脚本 `link_ilm.lds` 可以控制将程序段存放在 ILM 中，并且使得代码段的物理地址和运行地址完全一致，那么在上电系统引导程序（参见第 2.3.4 节所述 `start.S`）中便不会将程序进行重复上载，而是直接在 ILM 中运行。

以 2.4.2 节中开发的 Hello World 程序为例，在 N200-SDK 平台中进行编译时，使用如下命令选项将会使用链接脚本 `link_ilm.lds`：

```
// 注意：确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链，请参见本文档第 2.4.1 了解其详情。

cd n200-sdk
    // 确保目前处于 n200-sdk 目录下。

make dasm PROGRAM=hello_world BOARD=nuclei-n200 CORE=n201 DOWNLOAD=ilm
NANO_PFLOAT=0
// 上述命令使用到的 Makefile 参数与第 2.4.3 节中所述的几乎一致，除了如下参数：
    // DOWNLOAD=ilm：指明采用“将程序从 ILM 中运行的方式”进行编译，即选择使用链接脚本 link_ilm.lds。
```

编译成功后在终端的显示信息与**错误!未找到引用源。**中所示几乎一致，从其中同样可以看出代码尺寸信息，并且可以看到反汇编文件生成在 `software/hello_world/hello_world.dump` 中。

如果使用此种方式进行编译，按照第 2.5 节中所述的步骤下载程序至开发板，然后按照第 2.6.1 节中所述的步骤在开发板上运行程序，通过其打印到 PC 中断上的字符串显示速度可以看出其运行速度非常之快，这是因为程序直接从 ILM 中运行时每次都从 ILM 中取指令，能够做到每一个周期取一条指令，所以执行速度很快。

2.4.5. 编译使得程序从 Flash 上载至 ILM 中运行

如第 2.3.3 节中所述，通过系统链接脚本（`link_flash.lds`）可以控制将程序段存放在 Flash 中，

但是使得代码段的物理地址和运行地址不一致，那么在上电系统引导程序（参见第 2.3.4 节所述 start.S）中便会将程序上载至 ILM 中运行。

以 2.4.2 节中开发的 Hello World 程序为例，在 N200-SDK 平台中进行编译时，使用如下命令选项将会使用链接脚本 link_flash.lds:

```
// 注意：确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链，请参见本文档第 2.4.1 了解其详情。

cd n200-sdk
    // 确保目前处于 n200-sdk 目录下。

make dasm PROGRAM=hello_world BOARD=nuclei-n200 CORE=n201 DOWNLOAD=flash
NANO_PFLOAT=0
// 上述命令使用到的 Makefile 参数与第 2.4.3 节中所述的几乎一致，除了如下参数：
    //   DOWNLOAD=flash：指明采用“将程序从 Flash 上载至 ILM 中运行的方式”进行编译，即选择使用链接脚本
link_flash.lds。
    //   注意：DOWNLAD 选项的默认值在 Makefile 中被设定成了 flash。所以，如果不指明 DOWNLOAD 选项，则默认
采用“将程序从 Flash 上载至 ILM 进行执行的方式”进行编译。
```

编译成功后在终端的显示信息与**错误!未找到引用源。**中所示几乎一致，从其中同样可以看出代码尺寸信息，并且可以看到反汇编文件生成在 software/hello_world/hello_world.dump 中。

如果使用此种方式进行编译，按照第 2.5 节中所述的步骤下载程序至开发板，然后按照第 2.6.1 节中所述的步骤在开发板上运行程序，通过其打印到 PC 中断上的字符串显示速度可以看出其运行速度非常之快，这是因为程序上载至 ILM 中运行后，运行时每次都从 ILM 中取指令，能够做到每一个周期取一条指令，所以执行速度很快。

2.5. 使用 N200-SDK 下载程序

2.5.1. JTAG 调试器与开发板的连接

Nuclei N200 定制了专用的 JTAG 调试器，该调试器具有如下特性：

- 调试器的一端为普通 U 盘接口，便于直接将其插入主机 PC 的 USB 接口；另一端为标准的 4 线 JTAG 接口 和 2 线 UART 接口。
- 调试器具备 USB 转 JTAG 功能，通过标准的 4 线 JTAG 接口可与配套 SoC 原型开发板连接。

- N200 处理器内核支持标准的 JTAG 接口, 通过此接口可以进行程序的下载与交互式调试。
- 调试器具备 UART 转 USB 功能, 通过标准的 2 线 UART 接口可与配套 SoC 原型开发板连接。
- 由于嵌入式系统往往没有配备显示屏, 因此常用 UART 口连接主机 PC 的 COM 口 (或者将 UART 转换为 USB 后连接主机 PC 的 USB 口) 进行调试, 这样便可以将嵌入式系统中的 printf 函数重定向打印至主机的显示屏。

Nuclei N200 定制了专用的 FPGA 开发板, 作为 MCU 原型平台。

有关 Nuclei N200 定制的专用 JTAG 调试器和 MCU 原型开发板的详细介绍请参见单独文档《Nuclei_N200 系列配套 FPGA 实现》。

2.5.2. 设置 JTAG 调试器在 Linux 系统中的 USB 权限

如果使用 Linux 操作系统, 需要按照如下步骤保证正确的设置 JTAG 调试器的 USB 权限。

// 步骤一: 准备好自己的电脑环境, 可以在公司的服务器环境中运行, 如果是个人用户, 推荐如下配置:

(1) 使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。

(2) 由于 Linux 操作系统的版本众多, 推荐使用 Ubuntu 16.04 版本的 Linux 操作系统。

有关如何安装 VMware 和 Ubuntu 操作系统本文档不做介绍, 有关 Linux 的基本使用本文档也不做介绍, 请用户自行查阅资料学习。

// 步骤二: 将“专用 JTAG 调试器”插入电脑 PC 的 USB 接口。

注意:

务必使该 USB 接口被虚拟机的 Linux 系统识别 (而非被 Windows 识别), 如图 2-2 中圆圈所示, 若 USB 图标在虚拟机中显示为高亮, 则表明 USB 被虚拟机中 Linux 系统正确识别 (而非被 Windows 识别)。

若 USB 图标在虚拟机中显示为灰色, 则表明 USB 没有被虚拟机中的 Linux 系统正确识别, 如图 2-3 中所示, 可以使用鼠标点中 USB 图标, 选择将其“连接 (与主机的连接)”, 将其连接至 Linux 系统 (而非外部 Windows)。

// 步骤三: 使用如下命令查看 USB 设备的状态:

```
lsusb // 运行该命令后会显示如下信息。
```

```
...
```

```
Bus 001 Device 003: ID 0403:6010 Future Technology Devices International, Ltd FT2232C Dual USB-UART/FIFO IC
```

// 步骤四：使用如下命令设置 udev rules 使得该 USB 设备能够被 plugdev group 所访问：

```
sudo vi /etc/udev/rules.d/99-openocd.rules
// 用 vi 打开该文件，然后添加以下内容至该文件中，然后保存退出。
SUBSYSTEM=="usb", ATTR{idVendor}=="0403",
ATTR{idProduct}=="6010", MODE="664", GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403",
ATTRS{idProduct}=="6010", MODE="664", GROUP="plugdev"
```

// 步骤五：使用如下命令查看该 USB 设备是否属于 plugdev group：

```
ls /dev/ttyUSB*           // 运行该命令后会显示类似如下信息。
/dev/ttyUSB0 /dev/ttyUSB1

ls -l /dev/ttyUSB1       // 运行该命令后会显示类似如下信息。
crw-rw-r-- 1 root plugdev 188, 1 Nov 28 12:53 /dev/ttyUSB1
```

// 步骤六：将你自己的用户添加到 plugdev group 中：

```
whoami
// 运行该命令能显示自己的用户名，假设你的用户名显示为 your_user_name。
// 运行如下命令将 your_user_name 添加到 plugdev group 中。
sudo usermod -a -G plugdev your_user_name
```

// 步骤七：确认自己的用户是否属于 plugdev group：

```
groups           // 运行该命令后会显示类似如下信息。
... plugdev ...
// 只要从显示的 groups 中看到 plugdev 则意味着自己的用户属于该组，表示设置成功。
```

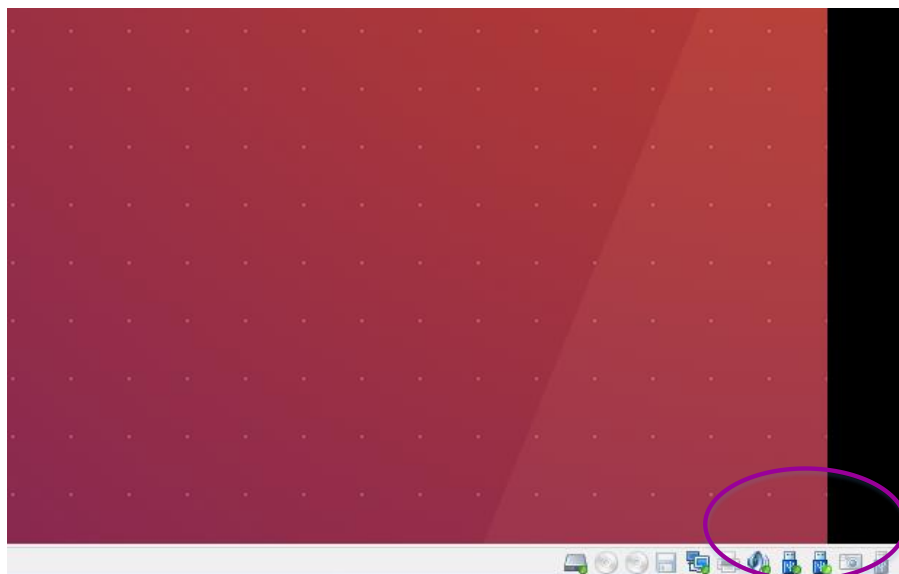


图 2-2 虚拟机 Linux 系统识别 USB 图标

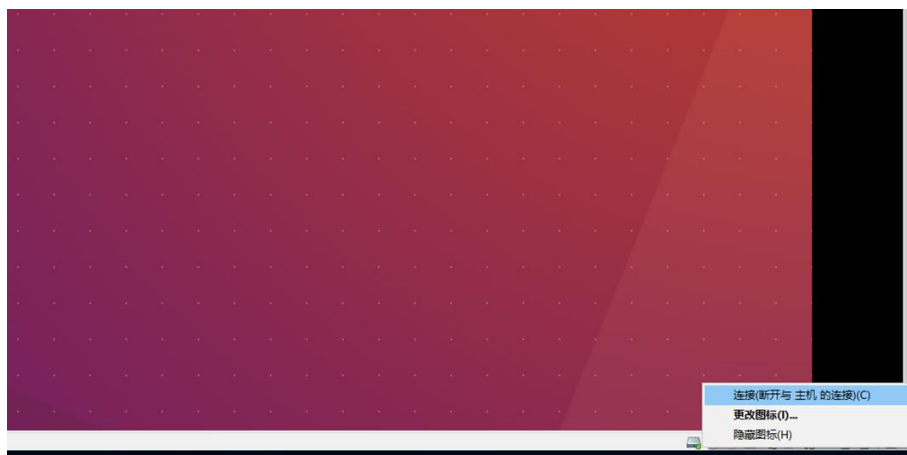


图 2-3 将 USB 接口选择连接至虚拟机中

2.5.3. 将程序下载至 FPGA 原型开发板

以 2.4.2 节中开发的 Hello World 程序为例，在 N200-SDK 平台中，使用如下命令将编译好的 hello_world 程序下载至 FPGA 原型开发板中。

```
// 注意：确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链，请参见本文档第 2.4.1 了解其详情。
// 注意：确保 FPGA 开发板与 JTAG 调试器正确的进行了连接，请参见本文档第 2.5.1 了解其详情。
// 注意：确保在 Linux 系统中正确设置了 JTAG 调试器的 USB 权限，请参见本文档第 2.5.2 了解其详情。

//将编译好的 hello_world 程序下载至 FPGA 原型开发板中，使用如下命令：

make upload PROGRAM=hello_world BOARD=nuclei-n200 CORE=n201
// 上述命令使用到了如下几个 Makefile 参数，分别解释如下：
//   upload: 该选项表示对程序进行下载。
//   PROGRAM=hello_world: 指定需要下载 software/hello_world 目录下的示例程序。
//   BOARD=nuclei-n200: 确定开发板型号，指明需要使用 bsp/nuclei-n200 目录下的板级支持包。
//   CORE=n201: 指明使用的 N200 系列的具体处理器内核型号，此处的示例是指明 n201，但是用户需要按照具体型号进行指明，譬如如果使用 N201 的用户此处需要指明 CORE=n201。
```

2.6. 在 FPGA 开发板上运行程序

由于程序将通过 UART（转换为 USB 口）连接至主机 PC，成为一个串口，打印一个 Log 符号到主机 PC 的显示屏上。因此需要先将串口显示终端准备好，在 Ubuntu 的命令行终端中使用如下命令。

```
sudo screen /dev/ttyUSB1 115200
// 该命令将设备 ttyUSB1 设置为串口显示的来源，波特率为 115200
// 若该命令执行成功的话，Ubuntu 的该命令行终端将被锁定，用于显示串口发送的字符。
// 注意：
// 若该命令无法执行成功，请检查如下几项。
// (1) 确保已按照第 2.5.2 节中所述方法将 USB 的权限设置正确。
// (2) 确保已按照第 2.5.2 节中所述方法将 USB 被 Linux 虚拟机识别（右下角
//      显示为高亮）。
// (3) 按照 2.5.2 节中所述使用命令“ls /dev/ttyUSB*”查看 USB 被识别成为
//      ttyUSB1 还是 ttyUSB2，若被识别成为 ttyUSB2，则应使用命令 sudo screen
//      /dev/ttyUSB2 115200
```

将主机 PC 的串口显示终端准备好之后，按照后续小节所介绍的方法运行程序。

2.6.1. 程序从 Flash 直接运行

以 2.4.2 节中开发的 Hello World 程序为例，将程序按照第 2.4.3 节中所介绍的编译方式（使得程序从 Flash 中直接运行）进行编译之后，然后按照第 2.5.3 节中所介绍的方法将程序下载至开发板后，便可以在开发板上运行程序。

可以通过按开发板上的 RESET 按键，让处理器复位重新执行程序。注意：由于程序是被烧写在 Flash 中，因此，需要保证 N200 系列处理器复位后从外部 Flash 开始执行，因此需要将 SoC 引脚 BOOTROM_N 设置为高电平。请参见单独文档《Nuclei_N200 系列配套 SoC 介绍》了解更多 SoC 信息。

Hello World 程序正确执行后将字符串打印至主机 PC 的串口显示终端上，如图 2-4 所示。通过其打印到 PC 中断上的字符串显示速度可以看出其运行速度非常之慢，这是因为程序直接从 Flash 中运行需要每次都从 Flash 中取指令，取指时间较长，影响了程序的执行速度。

由于程序被烧写进入了 Flash 中，因此程序不会掉电丢失。

2.6.3. 程序从 Flash 上载至 ILM 中运行

以 2.4.2 节中开发的 Hello World 程序为例，将程序按照第 2.4.5 节中所介绍的编译方式（使得程序从 Flash 上载至 ILM 中运行）进行编译之后，然后按照第 2.5.3 节中所介绍的方法将程序下载至开发板后，便可以在开发板上运行程序。

可以通过按开发板上的 RESET 按键，让处理器复位重新执行程序。注意：由于程序是被烧写在 Flash 中，因此，需要保证 N200 系列处理器复位后从外部 Flash 开始执行，因此需要将 SoC 引脚 BOOTROM_N 设置为高电平。请参见单独文档《Nuclei_N200 系列配套 SoC 介绍》了解更多 SoC 的信息。

Hello World 程序正确执行后将字符串打印至主机 PC 的串口显示终端上，如图 2-4 所示。通过其打印到 PC 中断上的字符串显示速度可以看出其运行速度非常之快，这是因为程序从 Flash 上载至 ILM 中运行后，运行时每次都从 ILM 中取指令，能够做到每一个周期取一条指令，所以执行速度很快。

由于程序被烧写进入了 Flash 中，因此程序不会掉电丢失。

2.7. 使用 GDB 远程调试程序

GDB 是常用的远程调试工具，本节将介绍如何使用 GDB 在 N200-SDK 平台中进行远程调试。

2.7.1. 调试器工作原理

不同于普通的固定电路芯片，处理器运行的是软件程序。因此，处理器对于运行于其上的软件程序提供调试能力是至关重要的。

对于处理器的调试功能而言，常用的两种是“交互式调试”和“追踪调试”。本节将对此两种调试的功能及原理加以简述。感兴趣的用户可以参见中文书籍《手把手教你设计 CPU——RISC-V 处理器篇》的第 14 章了解更多调试机制的详细介绍。

■ 交互调试概述

交互调试（Interactive Debug）功能是处理器提供的最常见的一种调试功能，从最低端的处理器到最高端的处理器，交互调试几乎是必备的功能。交互调试是指调试器软件（譬如常见的调试软件 GDB）能够直接对处理器取得控制权，进而对其以一种交互的方式进行调试，譬如通过调试软

件对处理器。

- 下载或者启动程序。
- 通过设定各种特定条件来停止程序。
- 查看处理器的运行状态。包括通用寄存器的值、存储器地址的值等。
- 查看程序的状态。包括变量的值、函数的状态等。
- 改变处理器的运行状态。包括通用寄存器的值、存储器地址的值等。
- 改变程序的状态。包括变量的值、函数的状态等。

对于嵌入式平台而言，调试器软件一般是运行于主机 PC 端的一款软件，而被调试的处理器往往是在嵌入式开发板之上，这是交叉编译和远程调试的一种典型情形。调试器软件为何能够取得处理器的控制权，从而对其进行调试呢？可想而知，需要硬件的支持才能做到。在处理器核的硬件中，往往需要一个硬件调试模块。该调试模块通过物理介质（譬如 JTAG 接口）与主机端的调试软件进行通信接受其控制，然后调试模块对处理器核进行控制。

为了帮助用户进一步理解，以交互式调试中常见的一种调试情形为例来阐述此过程。假设调试软件 GDB 试图对程序中的某个 PC 地址设置一个断点，然后希望程序运行到此处之后停下来，之后 GDB 能够读取处理器当时的某个寄存器的值。调试软件和调试模块便会进行如下协同操作：

- 开发人员通过运行于主机端的 GDB 软件在其软件界面上设置某行程序的断点，GDB 软件通过底层驱动 JTAG 接口访问远程处理器的调试模块，对其下达命令，告诉其希望于某 PC 设置一个断点。
- 调试模块得令即开始对处理器核进行控制，首先它会请求处理器核停止；然后修改存储器中那个 PC 地址的指令，将其替换成一个 Breakpoint 指令；最后将处理器核放行，让其恢复执行。
- 当处理器恢复执行后，执行到那个 PC 地址时，由于碰到了 Breakpoint 指令，会产生异常进入调试模式的异常服务程序。调试模块探测到处理器核进入了调试模式的异常服务程序，并将此信息显示出来。主机端的 GDB 软件一直在监测调试模块的状态从而得知此信息，便得知处理器核已经运行到断点处停止了下來，并显示在 GDB 软件界面上。
- 开发人员通过运行于主机端的 GDB 软件在其软件界面上设置读取某个寄存器的值，

GDB 软件通过底层驱动 JTAG 接口访问远程处理器的调试模块，对其下达命令，告诉其希望读取某个寄存器的值。

- 调试模块得令即开始对处理器核进行控制，从处理器核中将那个寄存器的值读取出来，并将此信息显示出来。主机端的 GDB 软件一直在监测调试模块的状态，从而得知此信息，便通过 JTAG 接口将读取的值返回到主机 PC 端，并显示在 GDB 软件界面上。

注意：以上采用极为通俗的语言来描述此过程，以帮助用户理解，但难免失之严谨，请以具体的调试机制文档为准。

从上述过程中可以看出，调试机制是一套复杂的软硬件协同工作机制，需要调试软件和硬件调试模块的精密配合。

同时，也可以看出交互式调试对于处理器的运行往往是具有打扰性（**Intrusive**）的。调试单元会在后台偷偷地控制住处理器核，时而让其停止，时而让其运行。由于交互式调试对处理器运行的程序具有影响，甚至会改变其行为，尤其是对时间先后性有依赖的程序，有时候交互式调试并不能完整地重现某些程序的 **Bug**。最常见的情形便是处理器在全速运行某个程序时会出现 **Bug**，当开发人员使用调试软件对其进行交互式调试时，**Bug** 又不见了。其主要原因往往就是交互式调试过程的打扰性（**Intrusive**），使得程序在调试模式和全速运行下的结果出现了差异。

■ 跟踪调试概述

上一节中论述了交互式调试的一个缺点是对处理器的运行具有打扰性，为了克服此种缺陷，便引入了跟踪调试（**Trace Debug**）机制。

跟踪调试，即调试器只跟踪记录处理器核执行过的所有程序指令，而不会打断干扰处理器的执行过程。跟踪调试同样需要硬件的支持才能做到，相比交互式调试的实现难度更大。由于处理器的运行速度非常快，每秒钟能执行上百万条指令，如果长时间运行某个程序，其产生的信息量十分巨大。跟踪调试器的硬件单元需要跟踪记录下所有的指令，对于处理速度的要求，数据的压缩、传输和存储等都是极大挑战。跟踪调试器的硬件实现会涉及相比交互调试而言更加复杂的技术，同时硬件开销也更大，因此跟踪调试器往往只在比较高端的处理器中使用。

注意：N200 系列内核不支持跟踪调试。

2.7.2. GDB 常用操作示例

GDB 能够用于调试 C、C++、Ada 等等各种语言编写的程序，它提供如下功能。

- 下载或者启动程序。
- 通过设定各种特定条件来停止程序。
- 查看处理器的运行状态，包括通用寄存器的值、存储器地址的值等。
- 查看程序的状态，包括变量的值、函数的状态等。
- 改变处理器的运行状态，包括通用寄存器的值、存储器地址的值等。
- 改变程序的状态，包括变量的值、函数的状态等。

GDB 可以用于在主机 PC 的 Linux 系统中调试运行的程序，同时也能用于调试嵌入式硬件。在嵌入式硬件的环境中，由于资源有限，一般的嵌入式目标硬件上无法直接构建 GDB 的调试环境（譬如显示屏和 Linux 系统等），这时可以通过 GDB+GdbServer 的方式进行远程（Remote）调试，通常 GdbServer 在目标硬件上运行，而 GDB 则在主机 PC 上运行。

为了能够支持 GDB 对其进行调试，Nuclei N200 系列配套 SoC 使用 OpenOCD 作为其 GdbServer，与 GDB 进行配合。OpenOCD（Open On-Chip Debugger）是一款开源的免费调试软件，由社区共同维护。由于其开放开源的特点，众多的公司和个人使用其作为调试软件，支持大多数主流的 MCU 和硬件开发板。通过编写 OpenOCD 的底层驱动文件能够使其通过 JTAG 接口连接 Nuclei N200 系列配套 SoC，并利用其硬件调试特性对 Nuclei N200 系列配套 SoC 进行调试。

为了能够完全支持 GDB 的功能，在使用 GCC 对源代码进行编译时，需要使用 -g 选项，例如：‘gcc -g -o hello hello.c’。-g 选项会将调试所需信息加入编译所得的可执行程序中，因此该选项会增大可执行程序的大小，在正式发布的版本中通常不使用该选项。

GDB 虽然可以使用一些前端工具实现图形化界面，但是更常见的是使用命令行直接对其进行操作。常用的 GDB 命令介绍如[错误!未找到引用源。](#)所示。

命令	介绍
load file	动态链入 file 文件，并读取它的符号表
jump	使当前执行的程序跳转到某一行，或者跳转到某个地址
info br	使用该指令可查看断点信息，br 是断点 break 的缩写，

	GDB 具有自动补齐功能，此命令等效于 <code>info break</code>
<code>info source</code>	使用该指令可查看当前程序的信息
<code>info stack</code>	使用该指令可查看程序的调用层次关系
<code>list function-name</code>	使用该指令可列出某个函数
<code>list line-number</code>	列出某行附近的代码
<code>break function</code> <code>break line-number</code>	在指定的函数，或者行号处设置断点
<code>break *address</code>	在指定的地址处设置断点，一般在没有源代码时使用
<code>continue</code>	恢复程序运行，直到遇到下一个断点
<code>step</code>	进入下一行代码的执行，会进入函数内部
<code>step number</code>	等效于连续执行 <code>number</code> 次 <code>step</code> 命令
<code>next</code>	执行下一行代码，但不会进入函数内部
<code>next number</code>	等效于连续执行 <code>number</code> 次 <code>next</code> 命令
<code>until</code> <code>until line-number</code>	继续运行直到到达指定行号，或者函数、地址等
<code>stepi</code> <code>nexti</code>	<code>stepi/nexti</code> 命令与 <code>step/next</code> 的区别在于其执行下一条汇编指令，而不是下一行代码（譬如 C/C++ 中的一行代码）
<code>x address</code>	打印指定存储器地址中的值
<code>p variable</code>	打印指定变量的值

表 2-1 GDB 常用命令

2.7.3. 使用 GDB 调试 Hello World 示例

以 2.4.4 节中开发的 Hello World 程序为例（从 ILM 中直接执行），在 N200-SDK 平台中，按照如下步骤使用 GDB 和 OpenOCD 对基于 Nuclei N200 系列配套 SoC 原型开发板进行调试。

```
// 注意：确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链，请参见本文档第 2.4.1 节了解其详情。
// 注意：确保 FPGA 开发板与 JTAG 调试器正确的进行了连接，请参见本文档第 2.5.1 了解其详情。
// 注意：确保在 Linux 系统中正确设置了 JTAG 调试器的 USB 权限，请参见本文档第 2.5.2 节了解其详情。

// 确保位于 n200-sdk 目录。
cd n200-sdk

// 步骤一：从第一个 Terminal 中打开 OpenOCD
// 首先使用如下命令打开 OpenOCD
make run_openocd PROGRAM=hello_world BOARD=nuclei-n200 CORE=n201 DOWNLOAD=ilm
// 上述命令使用到的 Makefile 参数请参见第 2.4.3 节了解详情。
// 运行该命令会来打开 OpenOCD，并与开发板相连。
// 如果该步骤执行成功，则如图 2-5 所示。
```

// 步骤二：新开一个 Terminal，打开 GDB

// 由于命令行界面已经被 OpenOCD 挂住，因此需要重新开启一个新的 Terminal 终端，

// 注意：再次强调，此处是重新开启一个新的 Terminal 终端。

// 在新的 Terminal 终端下，同样确保位于 n200-sdk 目录。

```
cd n200-sdk
```

// 然后使用如下命令打开 GDB

```
make run_gdb PROGRAM=hello_world BOARD=nuclei-n200 CORE=n201 DOWNLOAD=ilm
```

// 上述命令使用到的 Makefile 参数请参见第 2.4.3 节了解详情。

// 运行该命令会自动打开 GDB 来调试 hello_world 示例程序。

// 如果该步骤执行成功，则进入了 GDB 的调试命令行界面，如图 2-6 所示。

// 步骤三：演示使用 GDB 命令：

// 接下来便可使用 GDB 的常用命令进行调试。

```
b main
```

// 在 main 函数的入口处设置断点。

```
info b
```

// 查看目前程序设置的断点，显示如图 2-7 所示。

```
x 0x20000000
```

```
x 0x20000004
```

```
x 0x20000008
```

// 查看存储器地址 0x20000000/0x20000004/0x20000008 中的数值，显示如图 2-8

// 所示。

```
info reg
```

```
info reg mstatus
```

// 查看当前处理器的通用寄存器的值和 CSR 寄存器 mstatus 的值，显示如图 2-9 所示。

```
info reg csr768
```

// 查看当前处理器的地址 768 的 CSR 寄存器的值。

// 注意：编号 768 为十进制数，对应十六进制为 0x300，对应 mstatus 寄存器的 CSR

// 地址。参见《Nuclei_N200 系列指令架构手册》了解 RSIC-V 架构的 CSR 寄存器列表和地址。

```
info reg mcause
```

```
info reg mepc
```

```
info reg mtval
```

// 查看当前处理器的 CSR 寄存器 mcause, mepc 和 mtval 的值。

// 注意：当程序出现了异常（程序运行结果显示结果为 Trap）时，可以通过 GDB 查看此

// 三个寄存器的值有效的定位异常的原因和发生位置。有关 mcause, mepc 和 mtval

// 寄存器的详情，请参见《Nuclei_N200 系列指令架构手册》。

```
jump main
```

// 从程序的 main 入口开始执行，将停于设置的第一个断点处，显示如图 2-10 所示。

```
ni
```

// 单步执行，显示如图 2-11 所示。

```
continue
```

// 继续执行，将停于下一个断点处，若无断点则一直执行至程序结束处。


```
Info : Exposing additional CSR 3069
Info : Exposing additional CSR 3070
Info : Exposing additional CSR 3071
Info : Examined RISC-V core; XLEN=32, misa=0x40001105
Info : Listening on port 3333 for gdb connections
Info : [0] Found 1 triggers
halted at 0x200002ea due to debug interrupt
Info : Found flash device 'micron n25q128' (ID 0x0018ba20)
cleared protection for sectors 0 through 255 on flash bank 0
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : accepting 'gdb' connection on tcp/3333
```

图 2-5 打开 OpenOCD 后的命令行界面

```
one-embed-gdb software/hello_world/hello_world -ex "set remotetimeout 240" -ex "target ext
-remote localhost:3333"
GNU gdb (GNU MCU Eclipse RISC-V Embedded GCC, 64-bits) 8.0.50.20170724-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-unknown-linux-gnu --target=riscv-none-embed".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from software/hello_world/hello_world...done.
Remote debugging using localhost:3333
0x8000027e in _exit (code=0) at /home/zhenbohu/hbird-e-sdk/bsp/hbird-e200/stubs/_exit.c:12
12      write(STDERR_FILENO, "\n", 1);
```

图 2-6 打开 GDB 的命令行界面

```
one-embed-gdb software/hello_world/hello_world -ex "set remotetimeout 240" -ex "target ext
-remote localhost:3333"
GNU gdb (GNU MCU Eclipse RISC-V Embedded GCC, 64-bits) 8.0.50.20170724-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-unknown-linux-gnu --target=riscv-none-embed".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from software/hello_world/hello_world...done.
Remote debugging using localhost:3333
0x8000027e in _exit (code=0) at /home/zhenbohu/hbird-e-sdk/bsp/hbird-e200/stubs/_exit.c:12
12      write(STDERR_FILENO, "\n", 1);
(gdb) b main
Breakpoint 1 at 0x80000004: file hello_world.c, line 6.
(gdb) info b
Num      Type           Disp Enb Address      What
1        breakpoint      keep y   0x80000004   in main at hello_world.c:6
(gdb)
```

图 2-7 打开 GDB 显示设置的断点

```

Reading symbols from software/hello world/hello_world...done.
Remote debugging using localhost:3333
0x8000027e in _exit (code=0) at /home/zhenbohu/hbird-e-sdk/bsp/hbird-e200/stubs/_exit.c:12
12      write(STDERR_FILENO, "\n", 1);
(gdb) b main
Breakpoint 1 at 0x80000004: file hello_world.c, line 6.
(gdb) info b
Num      Type           Disp Enb Address      What
1        breakpoint     keep y   0x80000004 in main at hello_world.c:6
(gdb) x 0x20000000
0x20000000 <_start>: 0x70001197
(gdb) x 0x20000004
0x20000004 <_start+4>: 0xec018193
(gdb) x 0x20000008
0x20000008 <_start+8>: 0x70010117
(gdb) █

```

图 2-8 通过 GDB 查看存储器中的数据

```

(gdb) info reg
ra      0x8000027e      -2147483010
sp      0x9000ffc0      -1878982720
gp      0x9000ec0       -1879044416
tp      0x00000000      0
t0      0x80002e32      -2147471822
t1      0x69786520      1769497888
t2      0x00000000      0
fp      0x00000000      0
s1      0x00000000      0
a0      0x00000001      1
a1      0x90000029      -1879048151
a2      0x0000000d      13
a3      0x0000000a      10
a4      0x10013000      268513280
a5      0xffffffff      -1
a6      0x90000029      -1879048151
a7      0x20646574      543450484
s2      0x00000000      0
s3      0x00000000      0
s4      0x00000000      0
s5      0x00000000      0
s6      0x00000000      0
s7      0x00000000      0
s8      0x00000000      0
s9      0x00000000      0
s10     0x00000000      0
s11     0x00000000      0
t3      0x73616820      1935763488
t4      0x6d617267      1835102823
t5      0x6f72500a      1869762570
t6      0x00000000      0
pc      0x8000027e      -2147483010
priv    0x80000203      prv:3 [Machine]
(gdb) info reg mstatus
mstatus 0x00001800      SD:0 VM:0 MXR:0 PUM:0 MPRV:0 XS:0 FS:0 MPP:3 HPP:0 SPP:0 MPIE:0
HPIE:0 SPIE:0 UPIE:0 MIE:0 HIE:0 SIE:0 UIE:0
(gdb) █

```

图 2-9 GDB 显示寄存器的值

```

pc      0x8000027e      -2147483010
priv    0x80000203      prv:3 [Machine]
(gdb) info reg mstatus
mstatus 0x00001800      SD:0 VM:0 MXR:0 PUM:0 MPRV:0 XS:0 FS:0 MPP:3 HPP:0 SPP:0 MPIE:0
HPIE:0 SPIE:0 UPIE:0 MIE:0 HIE:0 SIE:0 UIE:0
(gdb) jump main
Line 6 is not in `_exit'. Jump anyway? (y or n) y
Continuing at 0x80000004.

Breakpoint 1, main () at hello_world.c:6
6      printf("Hello World!" "\n");
(gdb) █

```

图 2-10 GDB 显示程序停止于断点处

```
(gdb) info reg mcause
mcause      0x00000000      0
(gdb) info reg mepc
mepc        0x00000000      0
(gdb) info reg mtval
mtval       0x00000000      0
(gdb) ni
0x80000008   6      printf("Hello World!" "\n");
(gdb) ni
5      {
(gdb) ni
6      printf("Hello World!" "\n");
(gdb) ni
7      printf("Hello World!" "\n");
(gdb) ni
0x80000016   7      printf("Hello World!" "\n");
(gdb) ni
8      printf("Hello World!" "\n");
(gdb) continue
Continuing.
```

图 2-11 GDB 单步执行程序

3. 使用 N200-SDK 运行更多示例程序

在本文档上一章中介绍了基于 N200-SDK 平台开发一个简单的 Hello World 程序,并且下载、运行调试的方法。本章将进一步解析几个功能更加丰富的示例程序,以便于用户巩固和加深理解。

3.1. Dhrystone 示例程序

3.1.1. Dhrystone 示例程序功能简介

Dhrystone 是一个综合的处理器 Benchmark Program (跑分程序),由 Reinhold P. Weicker 于 1984 年开发,用于衡量处理器的整数运算处理性能。

有关 Dhrystone 跑分程序详细的背景知识和计算方法,用户可以参阅中文书籍《手把手教你设计 CPU——RISC-V 处理器篇》的第 20 章。本文档在此不做赘述,仅讲解如何使用 N200-SDK 运行 Dhrystone。

3.1.2. Dhrystone 示例程序代码结构

Dhrystone 示例程序的相关代码结构如下所示。

```
n200-sdk          // 存放 n200-sdk 的目录
|----software      // 存放示例程序的源代码
|----dhrystone    // Dhrystone 程序目录
|----dhry_1.c      //源代码
|----dhry_2.c      //源代码
|----dhry_stubs.c  //源代码
|----Makefile      //Makefile 脚本
```

Makefile 为主控制脚本,其代码片段如下:

```
//指明生成的 elf 文件名
TARGET = dhrystone

//指明 Dhrystone 程序所需要的特别的 GCC 编译选项
CFLAGS := -O2 -fno-inline -fno-common -falign-labels=4 -falign-functions=4 -falign-jumps=4
-falign-loops=4
```

```
BSP_BASE = ../../bsp

//指明 Dhrystone 程序所需要的 C 源文件
C_SRCS := dhry_stubs.c dhry_1.c dhry_2.c
HEADERS := dhry.h

//调用板级支持包 (bsp) 目录下的 common.mk
include $(BSP_BASE)/$(BOARD)/n200/env/common.mk
```

3.1.3. 运行 Dhrystone

Dhrystone 跑分程序示例可运行于 Nuclei N200 系列配套 SoC 平台中,使用本文档第 2.4 节中介绍的方法按照如下步骤运行:

// 步骤一: 参照本文档第 2.4 节中描述的方法, 编译 Dhrystone 示例程序, 使用如下命令:

```
make dasm PROGRAM=dhrystone BOARD=nuclei-n200 CORE=n201 NANO_PFLOAT=1
                                         DOWNLOAD=flash
```

// 上述命令使用到的 Makefile 参数请参见第 2.4.3 节了解详情。

//注意: 由于 Dhrystone 程序的 printf 函数需要输出浮点数, 上述选项 NANO_PFLOAT=1 指明 newlib-nano 的 printf 函数需要支持浮点数, 请参见本文档第 0 节了解相关信息。

//注意: 此处指定 DOWNLOAD=flash 选项, 则采用“将程序从 Flash 上载至 ILM 进行执行的方式”进行编译, 请参见本文档第 2.4.5 节了解更多详情。

// 步骤二: 参照本文档第 2.5 节中描述的方法, 将编译好的 Dhrystone 程序下载至 FPGA 原型开发板中, 使用如下命令:

```
make upload PROGRAM=dhrystone BOARD=nuclei-n200 CORE=n201 DOWNLOAD=flash
```

// 步骤三: 参照本文档第 2.6 节中描述的方法, 在 FPGA 原型开发板上运行 Dhrystone 程序:

```
// 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考第 2.6 节中
// 所述方法将串口显示电脑屏幕设置好, 使得程序的打印信息能够显示在电脑屏幕上。
//
// 由于步骤二已经将程序烧写进 FPGA 开发板的 Flash 之中, 因此每次按 FPGA 开发板的
// RESET 按键, 则处理器复位开始执行 Dhrystone 程序, 并将字符串打印至主机 PC
// 的串口显示终端上, 从其打印的结果我们可以看出处理器内核运行 Dhrystone 程
// 序的结果性能指标。
```

3.2. CoreMark 示例程序

3.2.1. CoreMark 示例程序功能简介

CoreMark 也是一个综合的处理器 Benchmark 程序，由非盈利组织 EEMBC（Embedded Microprocessor Benchmark Consortium）的 Shay Gal-On 于 2009 年开发。

有关 CoreMark 跑分程序详细的背景知识和计算方法，用户可以参阅中文书籍《手把手教你设计 CPU——RISC-V 处理器篇》的第 20 章。本文档在此不做赘述，仅讲解如何使用 N200-SDK 运行 CoreMark。

3.2.2. CoreMark 示例程序代码结构

CoreMark 示例程序的相关代码结构如下所示。

```
n200-sdk          // 存放 n200-sdk 的目录
|----software      // 存放示例程序的源代码
|----coremark      // CoreMark 示例程序目录
|----core_list_join.c //Coremark 的源代码
|----core_main.c
|----core_matrix.c
|----core_state.c
|----core_util.c
|----core_portme.c
|----Makefile      //Makefile 脚本
```

Makefile 为主控制脚本，其代码片段如下：

```
//指明生成的 elf 文件名
TARGET := coremark

//指明 CoreMark 程序所需要的 C 源文件
C_SRCS := \
    core_list_join.c \
    core_main.c \
    core_matrix.c \
    core_state.c \
    core_util.c \
    core_portme.c \

HEADERS := \
    coremark.h \
    core_portme.h \

//指明 CoreMark 程序所需要的特别的 GCC 编译选项
CFLAGS := -O3 -funroll-all-loops -finline-limit=600 -ftree-dominator-opts -fno-if-conversion2
-fselective-scheduling -fno-code-hoisting -fno-common -funroll-loops -finline-functions
-falign-functions=4 -falign-jumps=4 -falign-loops=4
CFLAGS += -DFLAGS_STR=\"$(CFLAGS)\"
```

```
CFLAGS += -DITERATIONS=10000 -DPERFORMANCE_RUN=1

BSP_BASE = ../../bsp
//调用板级支持包 (bsp) 目录下的 common.mk
include $(BSP_BASE)/$(BOARD)/n200/env/common.mk
```

3.2.3. 运行 CoreMark

CoreMark 跑分程序示例可运行于 Nuclei N200 系列配套 SoC 平台中，使用本文档第 2.4 节中介绍的方法按照如下步骤运行：

// 注意：确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链，请参见本文档第 2.4.1 了解其详情。

// 步骤一：参照本文档第 2.4 节中描述的方法，编译 CoreMark 示例程序，使用如下命令：

```
make dasm PROGRAM=coremark BOARD=nuclei-n200 CORE=n201 NANO_PFLOAT=1
                                           DOWNLOAD=flash
```

// 上述命令使用到的 Makefile 参数请参见第 2.4.3 节了解详情。

//注意：由于 CoreMark 程序的 printf 函数需要输出浮点数，上述选项 NANO_PFLOAT=1 指明 newlib-nano 的 printf 函数需要支持浮点数，请参见本文档第 0 节了解相关信息。

//注意：此处指定 DOWNLOAD=flash 选项，则采用“将程序从 Flash 上载至 ILM 进行执行的方式”进行编译，请参见本文档第 2.4.5 节了解更多详情。

// 步骤二：参照本文档第 2.5 节中描述的方法，将编译好的 CoreMark 程序下载至 FPGA 原型开发板中，使用如下命令：

```
make upload PROGRAM=coremark BOARD=nuclei-n200 CORE=n201 DOWNLOAD=flash
```

// 步骤三：参照本文档第 2.6 节中描述的方法，在 FPGA 原型开发板上运行 CoreMark 程序：

```
// 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考第 2.6 节中
// 所述方法将串口显示电脑屏幕设置好，使得程序的打印信息能够显示在电脑屏幕上。
//
// 由于步骤二已经将程序烧写进 FPGA 开发板的 Flash 之中，因此每次按 FPGA 开发板的
// RESET 按键，则处理器复位开始执行 CoreMark 程序，并将字符串打印至主机 PC
// 的串口显示终端上，从其打印的结果我们可以看出处理器内核运行 CoreMark 程
// 序的结果性能指标。
```

3.3. Demo_iasm 示例程序

3.3.1. Demo_iasm 示例程序功能简介

Demo_iasm 程序是一个完整的示例程序，用于演示在 C/C++ 程序中直接嵌入汇编程序的执行结果。

3.3.2. Demo_iasm 示例程序代码结构

Demo_iasm 示例程序的相关代码结构如下所示。

```
n200-sdk          // 存放 n200-sdk 的目录
|----software      // 存放示例程序的源代码
|----demo_iasm    // demo_iasm 示例程序目录
|----demo_iasm.c   //demo_iasm 源代码
|----Makefile      //Makefile 脚本
```

Makefile 为主控制脚本，其代码片段如下：

```
//指明生成的 elf 文件名
TARGET = demo_iasm

//指明 Demo_iasm 程序所需要的特别的 GCC 编译选项
CFLAGS += -O2

BSP_BASE = ../.. /bsp

//指明 Demo_iasm 程序所需要的 C 源文件
C_SRCS += demo_iasm.c

//调用板级支持包 (bsp) 目录下的 common.mk
include $(BSP_BASE)/$(BOARD)/n200/env/common.mk
```

其中 demo_iasm.c 为源代码，下节对其源码和功能进行详述。

3.3.3. 运行 Demo_iasm

Demo_iasm 示例可运行于 Nuclei N200 系列配套 SoC 平台中，使用本文档第 2.4 节中介绍的方法按照如下步骤运行：

// 注意：确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链，请参见本文档第 2.4.1 解其详情。

// 步骤一：参照本文档第 2.4 节中描述的方法，编译 Demo_iasm 示例程序，使用如下命令：

```
make dasm PROGRAM=demo_iasm BOARD=nuclei-n200 CORE=n201 NANO_PFLOAT=0
                                         DOWNLOAD=flash
```

// 上述命令使用到的 Makefile 参数请参见第 2.4.3 节了解详情。

//注意：由于 Demo_iasm 程序的 printf 函数不需要输出浮点数，上述选项 NANO_PFLOAT=0 指明 newlib-nano 的 printf 函数无需支持浮点数，请参见本文档第 0 节了解相关信息。

//注意：此处指定 DOWNLOAD=flash 选项，则采用“将程序从 Flash 上载至 ILM 进行执行的方式”进行编译，请参见本文档第 2.4.5 节了解更多详情。

// 步骤二：参照本文档第 2.5 节中描述的方法，将编译好的 Demo_iasm 程序下载至 FPGA 原型开发板中，使用如下命令：

```
make upload PROGRAM=demo_iasm BOARD=nuclei-n200 CORE=n201 DOWNLOAD=flash
```

// 步骤三：参照本文档第 2.6 节中描述的方法，在 FPGA 原型开发板上运行 Demo_iasm 程序：

```
// 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考第 2.6 节中
// 所述方法将串口显示电脑屏幕设置好，使得程序的打印信息能够显示在电脑屏幕上。
//
// 由于步骤二已经将程序烧写进 FPGA 开发板的 Flash 之中，因此每次按开发板的
// RESET 按键，则处理器复位开始执行 Demo_iasm 程序，并将字符串打印至主机 PC
// 的串口显示终端上，如图 3-1 所示，程序运行的结果为 PASS，意味着达到了预期。
```

```
#####
#####
This is a Demo to use the inline ASM to conduct ADD operations
We will use the inline assembly 'add' instruction to add two operands 100 and 200
The expected result is 300

If the result is 300, then we print PASS, otherwise FAIL
!!! PASS !!!
Program has exited with code:0x00000000
```

图 3-1 运行 Demo_iasm 示例后于主机串口终端上显示信息

3.4. Demo_eclic 示例程序

3.4.1. Demo_eclic 示例程序功能简介

Demo_eclic 程序是一个完整的示例程序，相比 Dhrystone 和 Coremark 这样纯粹的跑分程序，Demo_eclic 更加接近一个常见的嵌入式应用程序，它使用到了 SoC 系统中的外设，调用了中断处理函数等，其功能简述如下：

- 通过 printf 函数输出一串 RISC-V 的字符，printf 输出将会通过 UART 串口重定向至主机 PC 的屏幕上，如图 3-3 所示。

- 等待通过 `getc` 函数输入一个字符,然后将得到的字符通过 `printf` 输出值主机 PC 的屏幕上,如图 3-3 所示。
- 进入死循环不断地对 SoC 的 GPIO 13 的输出管脚进行翻转,如果使用示波器观测此 GPIO 输出管脚,可以看到其产生规律的输出方波。
- 设置计时器,使其先等待 5 秒钟之后开始触发计时器中断。在计时器中断中配置计时器的下一次触发时间是 0.5 秒以后,每次触发计时器中断都会在处理函数中对 GPIO 的输出管脚(对应三色灯的红灯)进行翻转。所以观察到的现象便是:刚开始等待 5 秒钟,之后开发板上红灯便开始以 1 秒钟的固定周期进行闪烁。
- 开发板上的两个用户按键连接到了 SoC 的 GPIO 管脚,这两个 GPIO 管脚各自作为一个 `eclic` 的外部中断,在其中断处理函数中会将 GPIO 的输出管脚(对应三色灯的蓝灯和绿灯)进行设置,从而造成开发板上三色灯的颜色发生变化。

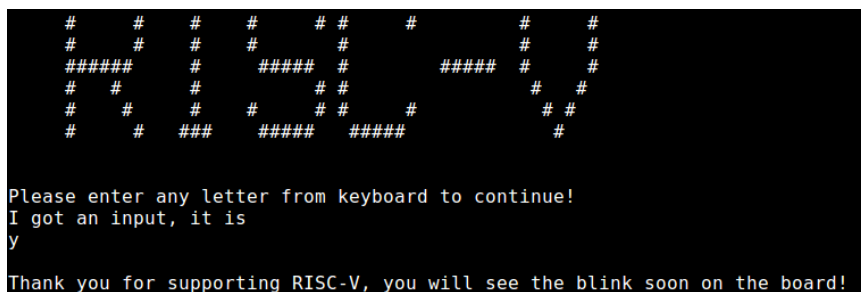


图 3-2 运行 Demo_eclic 示例后于主机串口终端上显示信息

3.4.2. Demo_eclic 示例程序代码结构

Demo_eclic 示例程序的相关代码结构如下所示。

```
n200-sdk          // 存放 n200-sdk 的目录
|----software      // 存放示例程序的源代码
|----demo_eclic    // demo_eclic 示例程序目录
|----demo_eclic.c  // demo_eclic 源代码
|----Makefile      // Makefile 脚本
```

Makefile 为主控制脚本,其代码片段如下:

```
//指明生成的 elf 文件名
TARGET = demo_eclic
//指明 Demo_eclic 程序所需要的特别的 GCC 编译选项
```

```
CFLAGS += -O2

BSP_BASE = ../../bsp

//指明 Demo_eclic 程序所需要的 C 源文件
C_SRCS += demo_eclic.c

//调用板级支持包 (bsp) 目录下的 common.mk
include $(BSP_BASE)/$(BOARD)/env/common.mk
```

其中 `demo_eclic.c` 为源代码，下节对其源码和功能进行详述。

3.4.3. Demo_eclic 示例程序源码分析

■ 主函数

`Demo_eclic` 的主程序位于 `software/demo_eclic/demo_eclic.c` 中，其源代码片段如下：

```
//software/demo_eclic/demo_eclic.c 代码片段

//主函数的入口
int main(int argc, char **argv)
{

    //设置开发板上按键相关的 GPIO 寄存器

    //通过“与”操作将 GPIO_OUTPUT_EN 寄存器某些位清 0，即将开发板按键对应的 GPIO 输出使能关闭
    GPIO_REG(GPIO_OUTPUT_EN) &=
        ~(
            (0x1 << BUTTON_1_GPIO_OFFSET) |
            (0x1 << BUTTON_2_GPIO_OFFSET)
        );

    //通过“与”操作将 GPIO_PULLUP_EN 寄存器某些位清 0，即将开发板按键对应的 GPIO 输入上拉关闭
    GPIO_REG(GPIO_PULLUP_EN) &=
        ~(
            (0x1 << BUTTON_1_GPIO_OFFSET) |
            (0x1 << BUTTON_2_GPIO_OFFSET)
        );

    //通过“或”操作将 GPIO_INPUT_EN 寄存器某些位设置为 1，即将开发板按键对应的 GPIO 输入使能关闭
    GPIO_REG(GPIO_INPUT_EN) |=
        (
            (0x1 << BUTTON_1_GPIO_OFFSET) |
            (0x1 << BUTTON_2_GPIO_OFFSET)
        );

    //通过“或”操作将 GPIO_RISE_IE 寄存器某些位设置为 1，即将开发板按键对应的 GPIO 管脚设置为上升沿触发的中断来源
    GPIO_REG(GPIO_RISE_IE) |= (1 << BUTTON_1_GPIO_OFFSET);
    GPIO_REG(GPIO_RISE_IE) |= (1 << BUTTON_2_GPIO_OFFSET);
```



```

//设置开发板上三色灯相关的 GPIO 寄存器

//通过“与”操作将 GPIO_INPUT_EN 寄存器某些位清 0，即将开发板三色灯对应的 GPIO 输入使能关闭
GPIO_REG(GPIO_INPUT_EN)    &=
    ~(
        (0x1<< RED_LED_GPIO_OFFSET) |
        (0x1<< GREEN_LED_GPIO_OFFSET) |
        (0x1 << BLUE_LED_GPIO_OFFSET)
    );

//通过“或”操作将 GPIO_INPUT_EN 寄存器对应的位置 1，即将开发板三色灯对应的 GPIO 输出使能打开
GPIO_REG(GPIO_OUTPUT_EN)    |=
    (
        (0x1<< RED_LED_GPIO_OFFSET) |
        (0x1<< GREEN_LED_GPIO_OFFSET) |
        (0x1 << BLUE_LED_GPIO_OFFSET)
    );

//通过“或”操作将 GPIO_OUTPUT_EN 寄存器对应的位置 1，即将开发板三色灯中的红色灯对应的 GPIO 输出值设置为 1，这
意味着将三色灯的红色灯打开。
GPIO_REG(GPIO_OUTPUT_VAL)    |=    (0x1 << RED_LED_GPIO_OFFSET) ;

//通过“与”操作将 GPIO_OUTPUT_EN 寄存器某些清 0，即将开发板三色灯中的蓝色和绿色灯对应的 GPIO 输出值设置为 0，
这意味着将三色灯的蓝色和绿色灯关闭，所以只会显示红色。
GPIO_REG(GPIO_OUTPUT_VAL)    &=    ~( (0x1<<    BLUE_LED_GPIO_OFFSET)    |    (0x1<<
GREEN_LED_GPIO_OFFSET) ) ;
// 输出特殊字符串至屏幕
printf ("%s",printf_instructions_msg);

    // 提示输入任意字符
printf ("%s","\nPlease enter any letter from keyboard to continue!\n");

    // 进入循环等待用户从键盘输入任意字符（使用_getc 函数），得到输入后跳出循环。
    // 注意：_getc 函数的函数体定义在 demo_eclic.c 文件中，通过 UART0 的输入通道抓取字符。
char c;
// Check for user input
while(1){
    if (_getc(&c) != 0){
        printf ("%s","I got an input, it is\n\r");
        break;
    }
}
_putchar(c);
printf ("\n\r");
printf ("%s","\nThank you for supporting RISC-V, you will see the blink soon on the board!\n");

    // 配置 eclic 并且将其管理的各个外部中断的中断处理函数入口地址进行初始化，参见后文对此函数的介绍
config_eclic_irqs ();

    // 配置计时器（通过 mtime 寄存器），参见后文对此函数的介绍
setup_mtime();

    //使能全局中断
set_csr(mstatus, MSTATUS_MIE);

```

```

/*****/
//接下来进入死循环，每个循环中都使用原子操作对 bitbang_mask 对应的 GPIO 管脚输出值进行反转。

// 设置位操作 (bitbang) 的指示位
uint32_t bitbang_mask = 0;
bitbang_mask = (1 << 13); //bitbang_mask 对应 GPIO 13 管脚。

//将位操作 (bitbang) 对应的 GPIO 管脚设置为输出
GPIO_REG(GPIO_OUTPUT_EN) |= bitbang_mask;

while (1){
    // 进入死循环不断地对某个 GPIO 的输出管脚进行翻转 (使用原子操作库函数)，如果使用示波器观测此 GPIO 输出管脚，可以看到其产生规律的输出方波。
    GPIO_REG(GPIO_OUTPUT_VAL) ^= bitbang_mask;
}

return 0;
}

```

■ TIMER 初始化

N200 系列内核 TIMER 单元中的 MTIME 寄存器和 MTIMECMP 寄存器充当计时器，这两个寄存器均为存储器地址映射 (Memory Address Mapped)，可以通过配置这两个寄存器对计时器进行初始化。

在 Demo_eclic 函数中通过 setup_mtime 函数来对计时器进行配置，其源代码如下：

```

void setup_mtime () {

//定义两个 volatile 类型的指针，分别指向 MTIME 和 MTIMECMP 寄存器的地址
volatile uint64_t * mtime      = (uint64_t*) (TMR_CTRL_ADDR + TMR_MTIME);
volatile uint64_t * mtimecmp   = (uint64_t*) (TMR_CTRL_ADDR + TMR_MTIMECMP);

//由于计时器默认一直在进行计数，所以需要通过读取 MTIME 寄存器得到当前的计数值
uint64_t now = *mtime;

//在目前的计数值基础上加上 2 秒钟的计数值，将其赋值给 MTIMECMP 寄存器，这意味着经过计时器的不断自增计数，2 秒钟后 MTIME 的值就会大于 MTIMECMP 的值，从而产生计时器中断。
uint64_t then = now + 2*RTC_FREQ;
*mtimecmp = then;
}

```

■ Eclic 配置

如第 2.3.5 节中所述，从上述代码可以看出，config_eclic_irqs 负责中断的使能和中断级别的设置其代码片段如下：

```

void config_eclic_irqs () {

```

```
//通过配置 ECLIC 的寄存器使能“计时器中断”和“开发板两个按键的 GPIO 中断”。注意: eclic_enable_interrupt
的函数原型定义在 shared/bsp/core.c 中
eclic_enable_interrupt (ECLIC_INT_MTIP);
eclic_enable_interrupt (ECLIC_INT_DEVICE_BUTTON_1);
eclic_enable_interrupt (ECLIC_INT_DEVICE_BUTTON_2);

//注意: eclic_set_nbits 的函数原型定义 bsp/nuclei-n200/n200/drivers/n200_func.c 中
eclic_set_nbits(3); // N200 only support 8 levels (3bits wide encoding)

// 通过配置的 eclic 设置 “计时器中断” 和 “开发板两个按键的 GPIO 中断” 的 eclic 中断优先级, 由高至低依次为:
// Button2 中断的优先级为 3
// Button1 中断的优先级为 2
// 计时器中断的优先级为 1
eclic_set_irq_lvl_abs(ECLIC_INT_MTIP, 1);
eclic_set_irq_lvl_abs(ECLIC_INT_DEVICE_BUTTON_1, 2);
eclic_set_irq_lvl_abs(ECLIC_INT_DEVICE_BUTTON_2, 3);

//button1 使用向量模式
eclic_set_vmode(ECLIC_INT_DEVICE_BUTTON_1);
```

■ 计时器中断处理函数

由于 Demo_eclic 会使用到计时器中断, 所以定义了 MTIME_HANDLER 函数作为其中断处理函数, 其代码如下:

```
void MTIME_HANDLER(){

    printf ("%s","Begin mtime handler\n");
    //将对应开发板上“三色灯中红灯”GPIO 管脚的输出值取反。由于每隔 0.5 秒进入中断处理函数后会对红灯取反(亮或者灭),
    所以在开发板上观察到的就是每隔 1 秒钟, 红灯闪烁一次。
    GPIO_REG(GPIO_OUTPUT_VAL) ^= (0x1 << RED_LED_GPIO_OFFSET);

    //在目前的计数值基础上加上 0.5 秒钟的计数值, 将其赋值给 MTIMECMP 寄存器, 这意味着经过计时器的不断自增计数, 0.5
    秒钟后 MTIME 的值就会大于 MTIMECMP 的值, 从而再次产生计时器中断。通过这个方法便可以产生固定周期为 0.5 秒钟计时
    器中断。
    uint64_t now = TMR_REG64(TMR_MTIME_OFFSET);
    uint64_t then = now + 0.5 * TMR_FREQ;
    TMR_REG64(TMR_MTIMECMP_OFFSET) = then;
    wait_seconds(5); // 延时 5 秒等待按键外部中断打断

    printf ("%s","End mtime handler\n");
}
```

■ 外部中断处理函数

如第 3.4.1 节所述, 开发板上的两个按键连接到了 GPIO 的管脚, 这两个 GPIO 管脚各自作为一个 eclic 的外部中断。Demo_eclic 使用这两个中断, 所以定义了 BUTTON_1_HANDLER 和

BUTTON_2_HANDLER 分别作为它们的中断处理函数，其代码如下：

```
void __attribute__((interrupt)) BUTTON_1_HANDLER(void) {

    SAVE_STATUS_IRQ_VECTOR;

    printf ("%s", "----Begin button1 handler\n");

    // 点亮绿灯
    GPIO_REG(GPIO_OUTPUT_VAL) |= (1 << GREEN_LED_GPIO_OFFSET);

    GPIO_REG(GPIO_RISE_IP) = (0x1 << BUTTON_1_GPIO_OFFSET);

    // 等待 5s
    wait_seconds(5);

    printf ("%s", "----End button1 handler\n");

    RESTORE_STATUS_IRQ_VECTOR;

};

void BUTTON_2_HANDLER(void) {

    printf ("%s", "-----Begin button2 handler\n");

    // 点亮蓝灯
    GPIO_REG(GPIO_OUTPUT_VAL) |= (1 << BLUE_LED_GPIO_OFFSET);

    GPIO_REG(GPIO_RISE_IP) = (0x1 << BUTTON_2_GPIO_OFFSET);

    // 等待 5s
    wait_seconds(5);

    printf ("%s", "-----End button2 handler\n");

};

//保存 mcause,mepc,msubm 寄存器，打开全局中断
#define SAVE_STATUS_IRQ_VECTOR \
    uint32_t mcause = read_csr(mcause);\
    uint32_t mepc = read_csr(mepc);\
    uint32_t msubm = read_csr(0x7c4);\
    set_csr(mstatus, MSTATUS_MIE);

//关闭全局中断，恢复 mcause,mepc,msubm 寄存器
#define RESTORE_STATUS_IRQ_VECTOR \
    clear_csr(mstatus, MSTATUS_MIE);\
    write_csr(0x7c4, msubm);\
    write_csr(mepc, mepc);\
    write_csr(mcause, mcause);
```

3.4.4. 运行 Demo_eclic

Demo_eclic 示例可运行于 Nuclei N200 系列配套 SoC 平台中，使用本文档第 2.4 节中介绍的方法按照如下步骤运行：

// 注意：确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链，请参见本文档第 2.4.1 解其详情。

// 步骤一：参照本文档第 2.4 节中描述的方法，编译 Demo_eclic 示例程序，使用如下命令：

```
make dasm PROGRAM=demo_eclic BOARD=nuclei-n200 CORE=n201 OCDCFG=hbird DOWNLOAD=flash
```

// 上述命令使用到的 Makefile 参数请参见第 2.4.3 节了解详情。

//注意：此处指定 DOWNLOAD=flash 选项，则采用“将程序从 Flash 上载至 ILM 进行执行的方式”进行编译，请参见本文档第 2.4.5 节了解更多详情。

// 步骤二：参照本文档第 2.5 节中描述的方法，将编译好的 Demo_eclic 程序下载至 FPGA 原型开发板中，使用如下命令：

```
make upload PROGRAM=demo_eclic BOARD=nuclei-n200 CORE=n201 OCDCFG=hbird DOWNLOAD=flash
```

// 步骤三：参照本文档第 2.6 节中描述的方法，在 FPGA 原型开发板上运行 Demo_eclic 程序：

// 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考第 2.6 节中

// 所述方法将串口显示电脑屏幕设置好，使得程序的打印信息能够显示在电脑屏幕上。

//

// 由于步骤二已经将程序烧写进 FPGA 开发板的 Flash 之中，因此每次按 MCU 开发板的

// RESET 按键，则处理器复位开始执行 Demo_eclic 程序，并将 RISC-V 字符串打印至主

//机 PC 的串口显示终端上，如图 3-3 所示，然后用户可以输入任意字符（譬如字母 y），

//程序继续运行，开发板上将会以固定频率进行闪灯。

```
# # # # # # # #
# # # # # # # #
##### # #####
# # # # # # # #
# # # # # # # #
# # ### ##### ##### #

Please enter any letter from keyboard to continue!
I got an input, it is
y

Thank you for supporting RISC-V, you will see the blink soon on the board!
```

图 3-3 运行 Demo_eclic 示例后于主机串口终端上显示信息

3.5. FreeRTOS 示例程序

3.5.1. FreeRTOS 示例程序功能简介

由于 RTOS 需要占用一定系统资源，只有少数 RTOS 支持在小内存的 MCU 上运行，FreeRTOS 是一款迷你型实时操作系统内核，功能包括：任务管理、时间管理、信号量、消息队列、内存管理等功能，可基本满足较小系统的需要。相对于 VxWorks、uc/os-II 等商业操作系统，FreeRTOS 完

全免费，具有源码公开、可移植、可裁剪、任务调度灵活等特点，可以方便地移植到各种 MCU 上运行。

请参见《Nuclei_N200 系列快速应用手册》了解更多关于移植 FreeRTOS 的详情。

3.5.2. FreeRTOS 示例程序代码结构

FreeRTOS 示例程序的相关代码结构如下所示。

```
n200-sdk          // 存放 n200-sdk 的目录
|----software      // 存放示例程序的源代码
|----FreeRTOSv9.0.0 // FreeRTOS 示例程序目录
|----Source        //FreeRTOS 内核源代码
|----Demo          //FreeRTOS 的示例 Demo 程序代码
|----Makefile      //Makefile 脚本
```

Makefile 为主控制脚本，其代码片段如下：

```
//指明 Demo 的文件夹名称，默认为 Demo
DEMO :=Demo
DEMO_DIR :=${DEMO}
//指明生成的 elf 文件名
TARGET = FreeRTOSv9.0.0
//指明 FreeRTOS 程序所需要的特别的 GCC 编译选项，默认选择 Code Size 优化 (-Os)
CFLAGS += -Os

BSP_BASE = ../../bsp

//指明 FreeRTOS 程序所需要的 C 源文件
C_SRCS += Source/croutine.c
C_SRCS += Source/list.c
C_SRCS += Source/queue.c
C_SRCS += Source/tasks.c
C_SRCS += Source/timers.c
C_SRCS += Source/event_groups.c
C_SRCS += Source/portable/MemMang/heap_4.c
C_SRCS += Source/portable/GCC/N200/port.c

C_SRCS += ${Demo}/main.c

INCLUDES += -ISource/include
INCLUDES += -I${Demo}
INCLUDES += -ISource/portable/GCC/N200

ASM_SRCS += Source/portable/GCC/N200/portasm.S

//调用板级支持包 (bsp) 目录下的 common.mk
```

```
include $(BSP_BASE)/$(BOARD)/n200/env/common.mk
```

3.5.3. FreeRTOS 示例程序源码分析

请参见《Nuclei_N200 系列快速应用手册》了解更多关于移植 FreeRTOS 的源代码详情。

3.5.4. 运行 FreeRTOS

FreeRTOS 示例可运行于 Nuclei N200 系列配套 SoC 平台中，该 Demo 中演示了中断嵌套以及中断中切换任务和切换 LED 灯状态等操作。

■ 首先编译 Demo 示例使用本文档第 2.4 节中介绍的方法按照如下步骤运行：

// 注意：确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链，请参见本文档第 2.4.1 了解其详情。

// 步骤一：参照本文档第 2.4 节中描述的方法，编译 FreeRTOS 示例程序，使用如下命令：

```
make dasm PROGRAM=FreeRTOSv9.0.0 BOARD=nuclei-n200 CORE=n201 ODCCFG=hbird DOWNLOAD=flash
// 上述命令使用到的 Makefile 参数请参见第 2.4.3 节了解详情。
```

//注意：此处指定 DOWNLOAD=flash 选项，则采用“将程序从 Flash 上载至 ILM 进行执行的方式”进行编译，请参见本文档第 2.4.5 节了解更多详情。

// 步骤二：参照本文档第 2.5 节中描述的方法，将编译好的 FreeRTOS 程序下载至 FPGA 原型开发板中，使用如下命令：

```
make upload PROGRAM=FreeRTOSv9.0.0 BOARD=nuclei-n200 CORE=n201 ODCCFG=hbird DOWNLOAD=flash
```

// 步骤三：参照本文档第 2.6 节中描述的方法，在 FPGA 原型开发板上运行 FreeRTOS 程序：

// 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考第 2.6 节中

// 所述方法将串口显示电脑屏幕设置好，使得程序的打印信息能够显示在电脑屏幕上。

//

// 由于步骤二已经将程序烧写进 FPGA 开发板的 Flash 之中，给 MCU 开发板重新上电，串口输出如图 3-4

//以收到任务循环执行的打印输出，先按下按键 1，按键 1 的中断输出打印信息，并观察开发板

//LED 灯的亮灭现象

//按键 1 中断执行完前（大约需要 5s 之内），按下按键 2，会马上进入按键 2 中断，显示按键 2

//中断的打印信息。进行任务切换，打印输出 “Higher level”

//等按键 2 中断执行完后，再执行完按键 1 的中断。最后继续执行各任务。

```
*****
*
Core freq at 15999959 Hz
*
*****
*****
Before StartScheduler
task_1
task1_running....
task_2
task2_running....
----Begin button1 handler
----red LED off or on
-----Begin button2 handler
-----Higher level
-----End button2 handler
----End button1 handler
timers Callback
task1_running....
task2_running....
timers Callback
task1_running....
task2_running....
timers Callback
task1_running....
task2_running....
timers Callback
task1_running....
task2_running....
timers Callback
task1_running....
task2_running....
timers Callback
task1_running....
task2_running....
timers Callback
```

图 3-4 FreeRTOS 的中断示例打印信息