

浙江大学

本科实验报告

课程名称: 嵌入式系统

姓 名: Jessie Peng

学 院: 计算机科学与技术学院

系:

专 业:

学 号:

指导教师: 王总辉

2019 年 5 月 10 日

浙江大学实验报告

课程名称: 嵌入式系统 实验类型: 综合

实验项目名称: 实验 4: 尝试 Boot Loader

学生姓名: Jessie Peng 专业: _____ 学号: _____

实验地点: 曹西 501 实验日期: 2019 年 5 月 10 日 指导老师: 王总辉

一、实验目的和要求

- 了解 U-Boot 的一般功能
- 掌握 U-Boot 的基本工作原理
- 掌握编译并烧录 U-Boot 到树莓派上
- 编写简易的 Boot Loader

二、实验内容和原理

【实验器材——硬件】

- STM32F103 核心板
- 树莓派

【实验器材——软件】

- MDK 软件及扩展
 - mdk_513
 - Keil.STM32F1xx_DFP.2.0.0 pack
 - stm32cubemx

- GCC 交叉编译软件

【实验器材——其它资料】

- <https://blog.csdn.net/hengwei0412/article/details/81173482>
- <http://blog.csdn.net/u010216127/article/details/8794716>

【实验内容——交叉编译 U-Boot，并下载到树莓派，通过 U-Boot 启动 Linux】

- 下载 U-Boot 并交叉编译 U-Boot
 - 烧录 U-Boot
 - 配置 U-Boot
 - 验证

【实验内容——写一个简易的 Boot Loader，下载到 STM32 板子上，并验证】

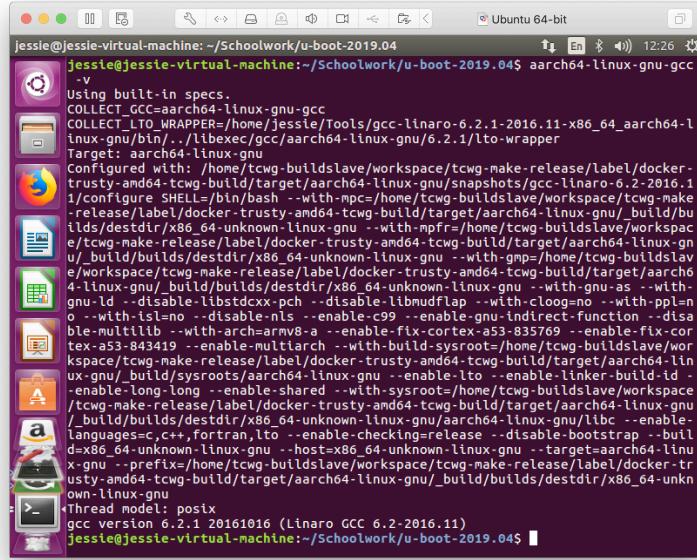
- 自行写一个简易 Boot Loader，能通过串口执行两条最简单的指令
 - peek addr 以一个字为单位读取内存中 addr 位置的数据（addr 是 4 字节对齐，十六进制的形式，长度为 8 位十六进制，例如 0x00008000），并以十六进制的形式输出
 - poke addr data 以一个字为单位修改内存中 addr 位置的数据为 data（addr 是 4 字节对齐，十六进制的形式，长度为 8 位十六进制，data 也是十六进制的形式，长度为 8 位十六进制）
 - 自行输入 6 组不同地址进行验证这两条命令

三、实验步骤和结果记录

【下载并交叉编译 U-Boot】

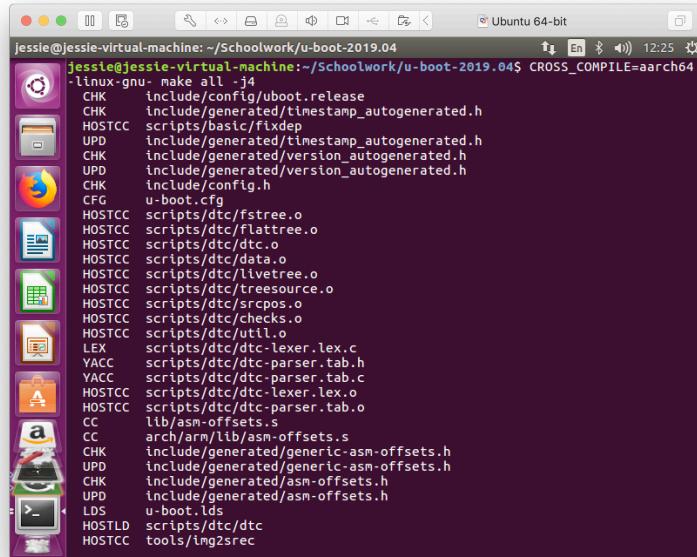
执行支持树莓派的配置:

下载交叉编译工具链并配置环境变量，查看版本信息（U-Boot 2019.04 要求
GCC 的版本必须在 6.0 以上，否则编译报错，这里用的是 6.2.1）：



```
Using built-in specs.
COLLECT_GCC=aarch64-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/home/jessie/Tools/gcc-linaro-6.2.1-2016.11-x86_64_aarch64-1
linux-gnu/bin/.libexec/gcc/aarch64-linux-gnu/6.2.1/lto-wrapper
Target: aarch64-linux-gnu
Configured with: /home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-and64-tcwg-build/target/aarch64-linux-gnu/snapshots/gcc-linaro-6.2-2016.1
1/configure SHELL=/bin/bash --with-npc=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-and64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu --with-mpfr=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-and64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gmp=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-and64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gnu-as --with-gnu-ld --disable-libstdcxx-pch --disable-cloog-no --with-ppl=no --with-isl=no --disable-nls --enable-c99 --enable-gnu-indirect-function --disable-multilib --with-arch-armv8-a --enable-fix-cortex-a53-835761 --enable-fix-cortex-a53-843419 --enable-multiarch --with-build-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-and64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu --enable-lto --enable-linker-build-id --enable-long-long --enable-shared --with-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-and64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu/aarch64-linux-gnu/lbis --enable-languages=c,c++,fortran,lto --enable-checking=release --disable-bootstrap --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=aarch64-linux-gnu --prefix=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-and64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu/aarch64-linux-gnu/lbis --enable-languages=c,c++,fortran,lto --enable-checking=release --disable-bootstrap --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=aarch64-linux-gnu
Thread model: posix
gcc version 6.2.1 20161016 (Linaro GCC 6.2-2016.11)
jessie@jessie-virtual-machine:~/Schoolwork/u-boot-2019.04$
```

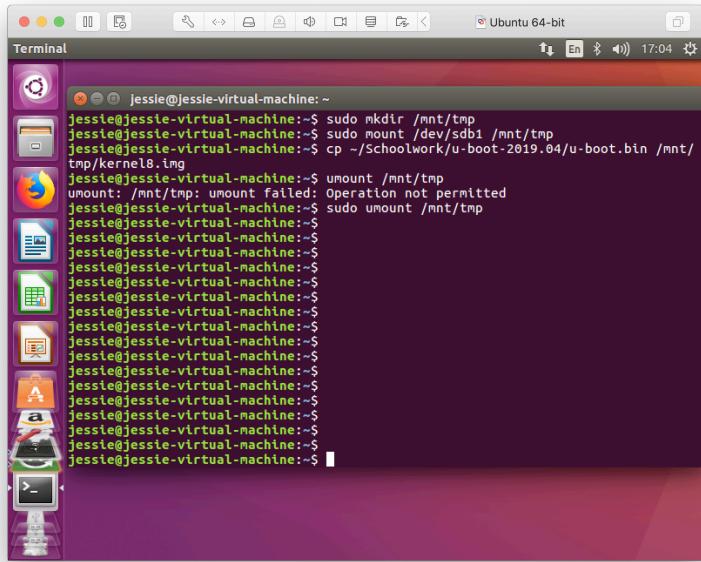
用交叉编译工具链进行编译：



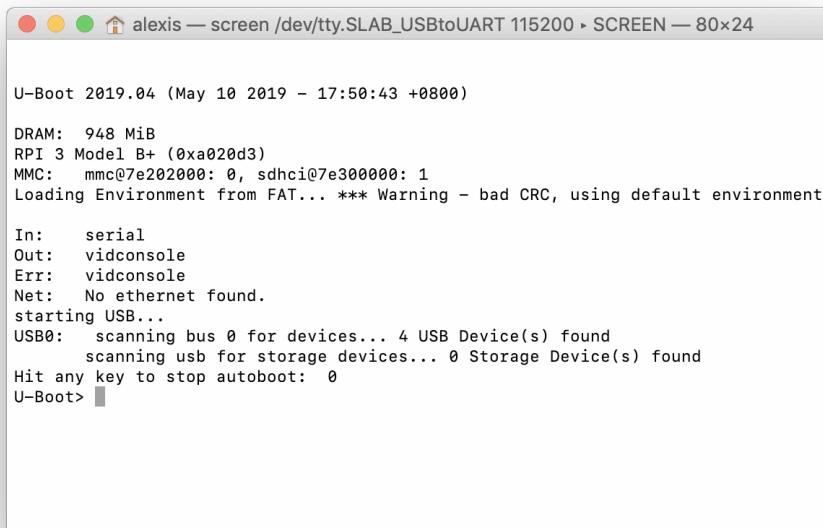
```
jessie@jessie-virtual-machine:~/Schoolwork/u-boot-2019.04
jessie@jessie-virtual-machine:~/Schoolwork/u-boot-2019.04$ CROSS_COMPILE=aarch64
-linu...
CHK include/config/uboot.release
CHK include/generated/timestamp autogenerated.h
HOSTCC scripts/basic/fixedep
UPD include/generated/timestamp autogenerated.h
CHK include/generated/version autogenerated.h
UPD include/generated/version autogenerated.h
CHK include/config.h
CFG u-boot.cfg
HOSTCC scripts/dtc/fstree.o
HOSTCC scripts/dtc/flattree.o
HOSTCC scripts/dtc/dtc.o
HOSTCC scripts/dtc/data.o
HOSTCC scripts/dtc/livetree.o
HOSTCC scripts/dtc/treeroot.o
HOSTCC scripts/dtc/srcpos.o
HOSTCC scripts/dtc/checks.o
HOSTCC scripts/dtc/util.o
LEX scripts/dtc/dtc-lexer.lex.c
YACC scripts/dtc/dtc-parser.tab.h
YACC scripts/dtc/dtc-parser.tab.c
HOSTCC scripts/dtc/dtc-lexer.lex.o
HOSTCC scripts/dtc/dtc-parser.tab.o
CC lib/asm-offsets.s
CC arch/arm/lib/asm-offsets.s
CHK include/generated/generic-asm-offsets.h
UPD include/generated/generic-asm-offsets.h
CHK include/generated/asm-offsets.h
UPD include/generated/asm-offsets.h
LDS u-boot.lds
HOSTLD scripts/dtc/dtc
HOSTCC tools/img2rec
```

【下载 U-Boot 至树莓派】

将编译好的 u-boot.bin 文件拷贝到树莓派的 SD 卡上并重命名为 kernel8.img，
树莓派在启动时会自动寻找最高版本的内核加载，这样 U-Boot 就可以代替原本
的 Raspbian 内核 kernel7.img 被优先加载：



树莓派插入 SD 卡，上电启动，从串口打印出信息并进入 U-Boot 命令行，说明 U-Boot 的编译烧录和启动都是成功的：



【U-Boot 引导启动 Linux】

(本部分尚未完成，目前已经成功进入 Linux 内核的启动过程，但是启动时报错，怀疑是 U-Boot 传递给内核的启动参数没有设置正确，仍在探索中。)

首先使用 U-Boot 附带的工具将 zImage 格式的 Linux 内核映像 kernel7.img 制

作成 `uImage`, 即在其头部增加 40 字节的说明信息。这么做是因为 U-Boot 需要根据这些说明信息来把内核加载到正确的地址, 以及定位到正确的入口地址等。

将 `uImage` 拷贝到 SD 卡的 `boot` 分区, 在 U-Boot 命令行中使用 `fatls mmc 0` 指令来查看第 0 号 MMC 设备(即 SD 卡的 `boot` 分区)中的文件, 可以看到有 `uImage`。

使用指令 `fatload mmc 0 0x30008000 uImage` 来将 `uImage` 下载到地址 `0x30008000` 中, 可以看到打印出读取到多少字节的信息。这里的地址由我们自己指定, 指定的地址是否有限制或者其他讲究还有待考证。

接下来使用 `setenv bootargs 'dwc_otg.lpm_enable=0 console=serial0,115200 console=tty1 root=PARTUUID=64a5b41d-02 rootfstype=ext4 elevator=deadline fsck.repair=yes rootwait'` 这样的语句来设置 `bootargs` 环境变量, 之后可以 `saveenv` 保存环境变量。`bootargs` 是传递给内核的启动参数, 我这里直接拷贝了之前树莓派内核启动的 `cmdline.txt` 中的内容, 不一定是正确的。

然后使用 `bootm 0x30008000` 指令告诉 U-Boot 去刚才那个地址启动内核, 此时通过串口打印的信息可以确定正确找到了 `uImage` 文件并进入了内核的启动过程。

【STM32 串口轮询接收测试】

在 STM32 上的 `boot loader` 需要通过串口收发实现命令行交互, 因此首先使用最简单的轮询方式测试串口的接收功能。

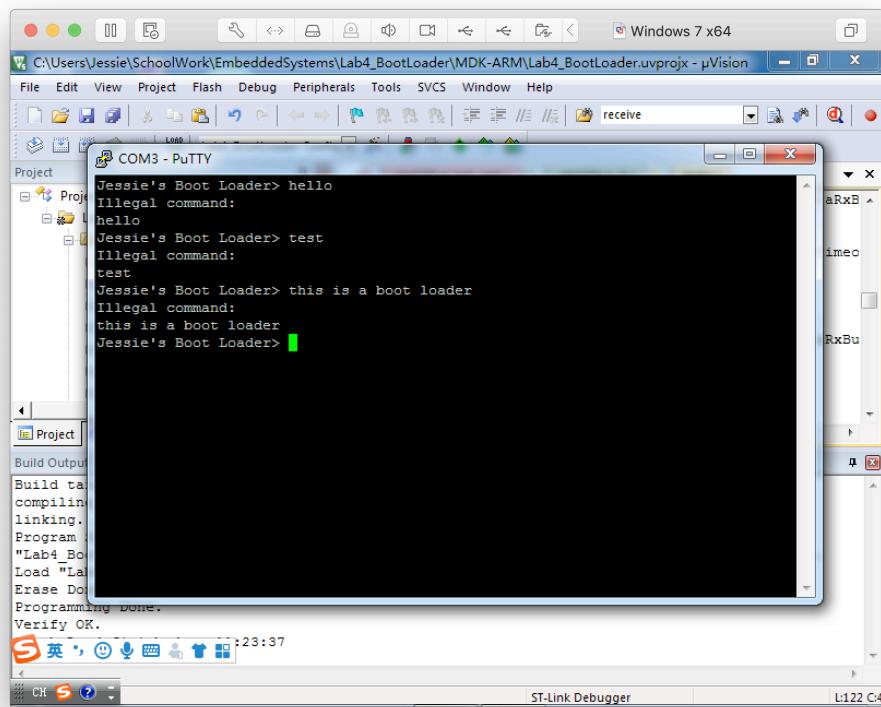
使用函数 `HAL_UART_Receive` 从串口接收指定字节的数据并存储在指定的位置。如果没有收到足够长度的数据, 将会阻塞在该语句处, 直到超时返回; 一旦接收到的数据达到指定长度, 则返回 `HAL_OK`。

使用函数 `HAL_UART_Transmit` 通过串口发送指定位置的指定字节的数据, 也可以重定向 `printf` 函数至该函数。

配置好 `UART1` 引脚, 使用 `asynchronous` 模式, 修改 `main` 函数代码:

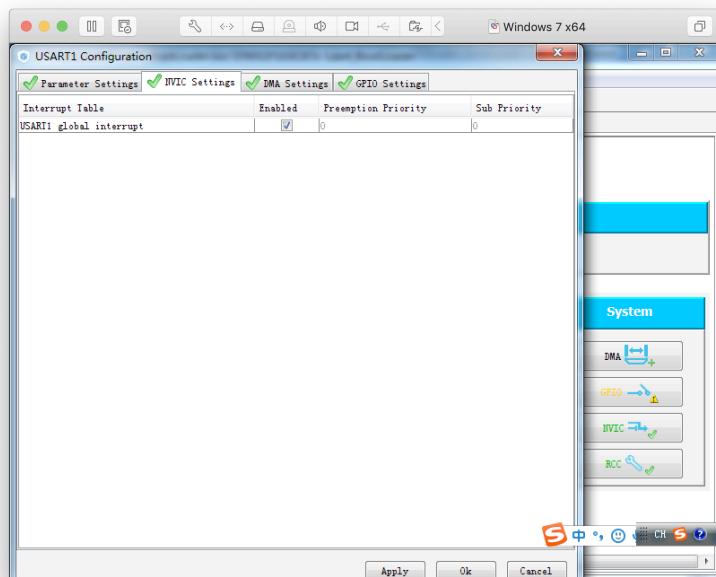
```
1. /* USER CODE BEGIN 2 */
2. // print header: boot loader>
3. HAL_UART_Transmit(&huart1, header, sizeof(header)-1, 0x1FFFFFF);
4. /* USER CODE END 2 */
5.
6. /* Infinite loop */
7. /* USER CODE BEGIN WHILE */
8. while (1)
9. {
10. /* USER CODE END WHILE */
11.
12. /* USER CODE BEGIN 3 */
13.     // receive one char at a time
14.     if (HAL_UART_Receive(&huart1, aRxBuffer, 1, 0x1FFFFFF) != HAL_OK)
15.     {
16.         // receiving time out
17.         HAL_UART_Transmit(&huart1, timeout, sizeof(timeout)-1, 0x1FFFFFF);
18.     }
19.     else
20.     {
21.         // print the received char
22.         HAL_UART_Transmit(&huart1, aRxBuffer, 1, 0x1FFFFFF);
23.         // save to command string
24.         command[cmdSize++] = aRxBuffer[0];
25.
26.         // end of a command
27.         if (aRxBuffer[0] == '\r')
28.         {
29.             // new line
30.             char c = '\n';
31.             command[cmdSize++] = c;
32.             HAL_UART_Transmit(&huart1, (uint8_t*)&c, 1, 0x1FFFFFF);
33.
34.             // print warning: illegal command
35.             HAL_UART_Transmit(&huart1, warning, sizeof(warning)-
36.                               1, 0x1FFFFFF);
37.             HAL_UART_Transmit(&huart1, command, cmdSize, 0x1FFFFFF);
38.
39.             // reset command buffer
40.             cmdSize = 0;
41.
42.             // print header: boot loader>
43.             HAL_UART_Transmit(&huart1, header, sizeof(header)-
44.                               1, 0x1FFFFFF);
45.         }
46.     /* USER CODE END 3 */
```

下载验证，可以通过串口模拟简单命令行交互界面了：



【STM32 串口中断接收测试】

下面改为用中断的方式来接收串口数据。首先配置好串口引脚，并使能串口全局中断（USART1 global interrupt）：



与函数 `HAL_UART_Receive` 类似，使用中断的接收函数 `HAL_UART_Receive_IT`

从串口接收指定字节的数据并存储在指定的位置，它接收三个参数，分别是串口结构体指针，要传输的数据指针以及待传输数据的大小。

通过代码可以看出 `HAL_UART_Receive_IT` 实质上是对中断接收的一个配置函数，它指定当接收到 `Size` 大小的字节后就产生一次中断进入中断处理。（注意其中串口结构体的 `pRxBuffPtr` 和 `RxXferCount` 分别被赋值为接收数据所存放位置的指针和接收数据总大小）：

`stm32f1xx_hal_uart.c`

```

1. /**
2.  * @brief Receives an amount of data in non blocking mode
3.  * @param huart: Pointer to a UART_HandleTypeDef structure that contains
4.  *                 the configuration information for the specified UART module.
5.  * @param pData: Pointer to data buffer
6.  * @param Size: Amount of data to be received
7.  * @retval HAL status
8. */
9. HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData,
10.                                       uint16_t Size)
11. {
12.     uint32_t tmp_state = 0;
13.     tmp_state = huart->State;
14.     if((tmp_state == HAL_UART_STATE_READY) || (tmp_state == HAL_UART_STATE_BUSY_TX))
15.     {
16.         if((pData == NULL ) || (Size == 0))
17.         {
18.             return HAL_ERROR;
19.         }
20.
21.         /* Process Locked */
22.         __HAL_LOCK(huart);
23.
24.         huart->pRxBuffPtr = pData;
25.         huart->RxXferSize = Size;
26.         huart->RxXferCount = Size;
27.
28.         huart->ErrorCode = HAL_UART_ERROR_NONE;
29.         /* Check if a transmit process is ongoing or not */
30.         if(huart->State == HAL_UART_STATE_BUSY_TX)
31.         {
32.             huart->State = HAL_UART_STATE_BUSY_RX_RX;
33.         }
34.         else
35.         {
36.             huart->State = HAL_UART_STATE_BUSY_RX;
37.         }
38.
39.         /* Process Unlocked */
40.         __HAL_UNLOCK(huart);
41.
42.         /* Enable the UART Parity Error Interrupt */
43.         __HAL_UART_ENABLE_IT(huart, UART_IT_PE);
44.
45.         /* Enable the UART Error Interrupt: (Frame error, noise error, overrun error) */
46.         __HAL_UART_ENABLE_IT(huart, UART_IT_ERR);
47.
48.         /* Enable the UART Data Register not empty Interrupt */
49.         __HAL_UART_ENABLE_IT(huart, UART_IT_RXNE);
50.
51.         return HAL_OK;
52.     }
53.     else
54.     {
55.         return HAL_BUSY;
56.     }
57. }

```

进入中断以后，由函数 USART1_IRQHandler 来处理串口中断，它调用了库函数 HAL_UART_IRQHandler：

stm32f1xx_it.c

```
1. /**
2. * @brief This function handles USART1 global interrupt.
3. */
4. void USART1_IRQHandler(void)
5. {
6. /* USER CODE BEGIN USART1_IRQHandler_0 */
7.
8. /* USER CODE END USART1_IRQHandler_0 */
9. HAL_UART_IRQHandler(&huart1);
10. /* USER CODE BEGIN USART1_IRQHandler_1 */
11.
12. /* USER CODE END USART1_IRQHandler_1 */
13. }
```

再看库函数 HAL_UART_IRQHandler 里面，在接收器模式下，它调用了一个 UART_Receive_IT（注意不是之前的 HAL_UART_Receive_IT）函数来处理接收到的数据：

stm32f1xx_hal_uart.c

```

1. /**
2.  * @brief This function handles UART interrupt request.
3.  * @param huart: Pointer to a UART_HandleTypeDef structure that contains
4.  *                 the configuration information for the specified UART module.
5.  * @retval None
6.  */
7. void HAL_UART_IRQHandler(UART_HandleTypeDef *huart)
8. {
9.     uint32_t tmp_flag = 0, tmp_it_source = 0;
10.
11.    tmp_flag = __HAL_UART_GET_FLAG(huart, UART_FLAG_PE);
12.    tmp_it_source = __HAL_UART_GET_IT_SOURCE(huart, UART_IT_PE);
13.    /* UART parity error interrupt occurred -----*/
14.    if((tmp_flag != RESET) && (tmp_it_source != RESET))
15.    {
16.        __HAL_UART_CLEAR_PFLAG(huart);
17.
18.        huart->ErrorCode |= HAL_UART_ERROR_PE;
19.    }
20.
21.    tmp_flag = __HAL_UART_GET_FLAG(huart, UART_FLAG_FE);
22.    tmp_it_source = __HAL_UART_GET_IT_SOURCE(huart, UART_IT_ERR);
23.    /* UART frame error interrupt occurred -----*/
24.    if((tmp_flag != RESET) && (tmp_it_source != RESET))
25.    {
26.        __HAL_UART_CLEAR_FEFLAG(huart);
27.
28.        huart->ErrorCode |= HAL_UART_ERROR_FE;
29.    }
30.
31.    tmp_flag = __HAL_UART_GET_FLAG(huart, UART_FLAG_NE);
32.    /* UART noise error interrupt occurred -----*/
33.    if((tmp_flag != RESET) && (tmp_it_source != RESET))
34.    {
35.        __HAL_UART_CLEAR_NEFLAG(huart);
36.
37.        huart->ErrorCode |= HAL_UART_ERROR_NE;
38.    }
39.
40.    tmp_flag = __HAL_UART_GET_FLAG(huart, UART_FLAG_ORE);
41.    /* UART Over-Run interrupt occurred -----*/
42.    if((tmp_flag != RESET) && (tmp_it_source != RESET))
43.    {
44.        __HAL_UART_CLEAR_OREFLAG(huart);
45.
46.        huart->ErrorCode |= HAL_UART_ERROR_ORE;
47.    }
48.
49.    tmp_flag = __HAL_UART_GET_FLAG(huart, UART_FLAG_RXNE);
50.    tmp_it_source = __HAL_UART_GET_IT_SOURCE(huart, UART_IT_RXNE);
51.    /* UART in mode Receiver -----*/
52.    if((tmp_flag != RESET) && (tmp_it_source != RESET))
53.    {
54.        UART_Receive_IT(huart);
55.    }
56.
57.    tmp_flag = __HAL_UART_GET_FLAG(huart, UART_FLAG_TXE);
58.    tmp_it_source = __HAL_UART_GET_IT_SOURCE(huart, UART_IT_TXE);
59.    /* UART in mode Transmitter -----*/

```

```
60. if((tmp_flag != RESET) && (tmp_it_source != RESET))
61. {
62.     UART_Transmit_IT(huart);
63. }
64.
65. tmp_flag = __HAL_UART_GET_FLAG(huart, UART_FLAG_TC);
66. tmp_it_source = __HAL_UART_GET_IT_SOURCE(huart, UART_IT_TC);
67. /* UART in mode Transmitter end -----
--*/
68. if((tmp_flag != RESET) && (tmp_it_source != RESET))
69. {
70.     UART_EndTransmit_IT(huart);
71. }
72.
73. if(huart->ErrorCode != HAL_UART_ERROR_NONE)
74. {
75.     /* Set the UART state ready to be able to start again the process */
76.     huart->State = HAL_UART_STATE_READY;
77.
78.     HAL_UART_ErrorCallback(huart);
79. }
80. }
```

UART_Receive_IT 函数读取接收寄存器，将数据按照长度、校验位进行处理，然后将去除校验位的数据存在 pRxBufferPtr 的位置，清除中断标志，并将 pRxBufferPtr 向后移动，将计数值 RxXferCount 减一。如果数据接收完成（之前规定的 Size 个字节），它就会禁止中断并调用 HAL 库中的回调函数 HAL_UART_RxCpltCallback:

stm32f1xx_hal_uart.c

```

1. /**
2.  * @brief Receives an amount of data in non blocking mode
3.  * @param huart: Pointer to a UART_HandleTypeDef structure that contains
4.  *                 the configuration information for the specified UART module.
5.  * @retval HAL status
6. */
7. static HAL_StatusTypeDef USART_Receive_IT(USART_HandleTypeDef *huart)
8. {
9.     uint16_t* tmp;
10.    uint32_t tmp_state = 0;
11.
12.    tmp_state = huart->State;
13.    if((tmp_state == HAL_USART_STATE_BUSY_RX) || (tmp_state == HAL_USART_STATE_BUSY_TX_RX))
14.    {
15.        if(huart->Init.WordLength == USART_WORDLENGTH_9B)
16.        {
17.            tmp = (uint16_t*) huart->pRxBuffPtr;
18.            if(huart->Init.Parity == USART_PARITY_NONE)
19.            {
20.                *tmp = (uint16_t)(huart->Instance->DR & (uint16_t)0x01FF);
21.                huart->pRxBuffPtr += 2;
22.            }
23.            else
24.            {
25.                *tmp = (uint16_t)(huart->Instance->DR & (uint16_t)0x00FF);
26.                huart->pRxBuffPtr += 1;
27.            }
28.        }
29.        else
30.        {
31.            if(huart->Init.Parity == USART_PARITY_NONE)
32.            {
33.                *huart->pRxBuffPtr++ = (uint8_t)(huart->Instance-
34.                >DR & (uint8_t)0x00FF);
35.            }
36.            else
37.            {
38.                *huart->pRxBuffPtr++ = (uint8_t)(huart->Instance-
39.                >DR & (uint8_t)0x007F);
40.            }
41.        }
42.        if(--huart->RxXferCount == 0)
43.        {
44.            __HAL_USART_DISABLE_IT(huart, USART_IT_RXNE);
45.            /* Check if a transmit process is ongoing or not */
46.            if(huart->State == HAL_USART_STATE_BUSY_TX_RX)
47.            {
48.                huart->State = HAL_USART_STATE_BUSY_TX;
49.            }
50.            else
51.            {
52.                /* Disable the USART Parity Error Interrupt */
53.                __HAL_USART_DISABLE_IT(huart, USART_IT_PE);
54.
55.                /* Disable the USART Error Interrupt: (Frame error, noise error, overrun error) */
56.                __HAL_USART_DISABLE_IT(huart, USART_IT_ERR);
57.
58.                huart->State = HAL_USART_STATE_READY;
59.            }
60.            HAL_USART_RxCpltCallback(huart);
61.

```

```
62.     return HAL_OK;
63.   }
64.   return HAL_OK;
65. }
66. else
67. {
68.   return HAL_BUSY;
69. }
70. }
```

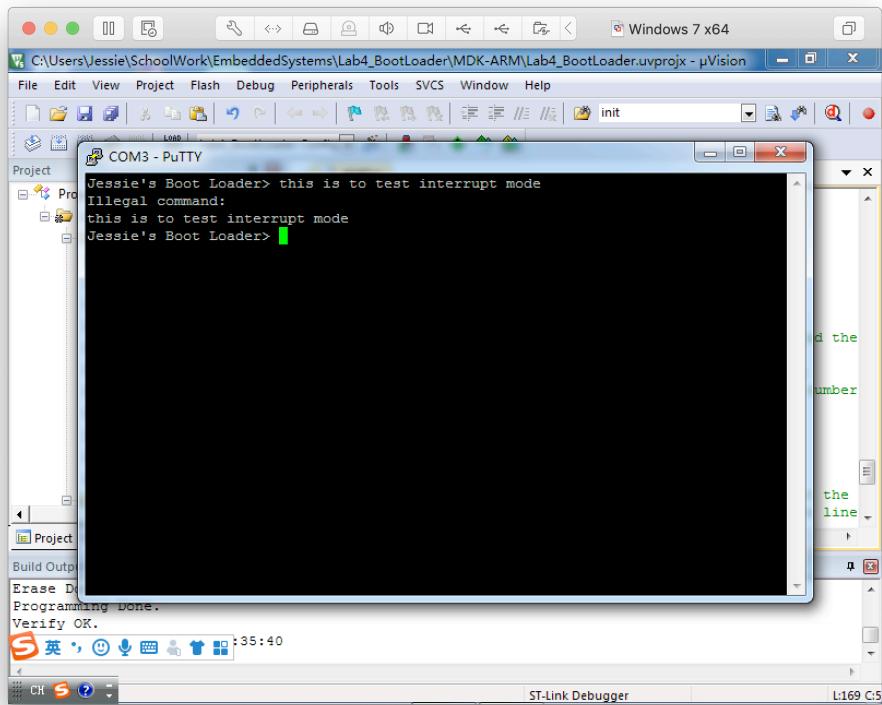
该回调函数是`_weak`修饰的：

stm32f1xx_hal_uart.c

```
1. /**
2.  * @brief Rx Transfer completed callbacks.
3.  * @param huart: Pointer to a UART_HandleTypeDef structure that contains
4.  *                 the configuration information for the specified UART module.
5.  * @retval None
6. */
7. __weak void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
8. {
9.   /* NOTE: This function should not be modified, when the callback is needed
10.    *
11.    *       the HAL_UART_RxCpltCallback can be implemented in the user file
12. */
13. }
```

因此我们在 `main.c` 中重写该回调函数，用于实现我们需要的功能。在回调函数的最后我们还需要再调用最开始的 `HAL_UART_Receive_IT` 来重新使能中断，从而继续接收数据。

回调函数的逻辑与刚才轮询的主循环类似，最终效果：



【编写 boot loader 实现 peek 和 poke 功能】

主函数中开辟一个数组表示内存空间，在最开始打印出内存首地址和大小，然后进入命令行模式，接收输入：

```

1. /* USER CODE BEGIN 2 */
2. char str[100];
3. int cnt = sprintf(str, "Memory starting at address 0x%p, total size: %d\r\n"
   , mem, MEM_SIZE);
4. HAL_UART_Transmit(&huart1, (uint8_t*)str, cnt, 0x1FFFFFFF);
5.
6. HAL_UART_Transmit(&huart1, header, sizeof(header)-1, 0x1FFFFFFF);
7. HAL_UART_Receive_IT(&huart1, aRxBuffer + bufRear, 1);
8. /* USER CODE END 2 */

```

使用一个循环队列来储存串口接收的数据，并将处理指令的逻辑移到主循环中去。这样，回调函数就只负责将每次接收的一个字节放入队尾，然后再次调用 HAL_UART_Receive_IT：

```
1. void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
2. {
3.     char c = aRxBuffer[bufRear];
4.     HAL_UART_Transmit(huart, (uint8_t*)&c, 1, 0x1FFFFFFF);
5.
6.     if (++bufRear == BUFFER_SIZE)
7.     {
8.         bufRear = 0;
9.     }
10.
11.    if (c == '\r')
12.    {
13.        c = '\n';
14.        aRxBuffer[bufRear] = c;
15.        HAL_UART_Transmit(huart, (uint8_t*)&c, 1, 0x1FFFFFFF);
16.
17.        if (++bufRear == BUFFER_SIZE)
18.        {
19.            bufRear = 0;
20.        }
21.    }
22.
23.    while (HAL_UART_Receive_IT(huart, aRxBuffer + bufRear, 1) != HAL_OK);
24. }
```

主循环中，每次从队首取出一个字节，如果已经取出了完整的一条指令（读到了换行符），则调用自己编写的 `respond_command` 函数来判断指令的合法性，如果合法则取出操作数并执行（`peek addr` 或者 `poke addr data`）：

```

1. /* Infinite loop */
2. /* USER CODE BEGIN WHILE */
3. while (1)
4. {
5. /* USER CODE END WHILE */
6.
7. /* USER CODE BEGIN 3 */
8.     if (bufRear != bufFront)
9.     {
10.         char c = aRxBuffer[bufFront];
11.         if (++bufFront == BUFFER_SIZE)
12.         {
13.             bufFront = 0;
14.         }
15.
16.         int ret;
17.         switch (c)
18.         {
19.             case '\n':
20.                 ret = respond_command();
21.                 cmdSize = 0;
22.                 if (ret == -1)
23.                 {
24.                     HAL_UART_Transmit(&huart1, warning, sizeof(warning)-
1, 0xFFFFFFFF);
25.                 }
26.                 HAL_UART_Transmit(&huart1, header, sizeof(header)-
1, 0xFFFFFFFF);
27.                 break;
28.             case '\b':
29.                 if (cmdSize > 0)
30.                 {
31.                     cmdSize--;
32.                 }
33.                 break;
34.             default:
35.                 command[cmdSize++] = c;
36.             }
37.         }
38.     }
39. /* USER CODE END 3 */

```

四、实验结果分析

【STM32 串口中断处理流程】

```

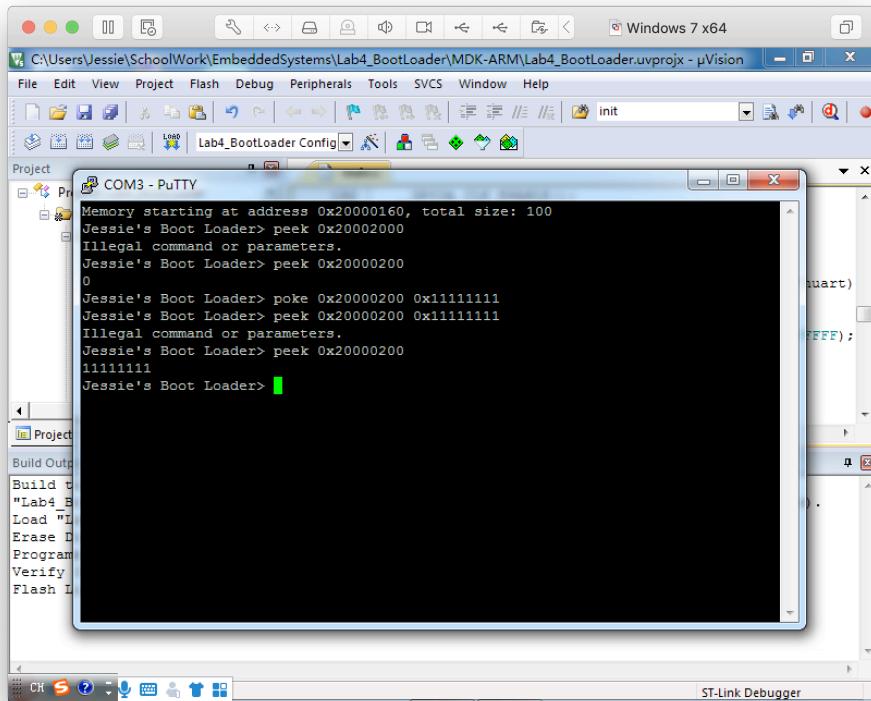
HAL_UART_Receive_IT => USART1_IRQHandler =>
HAL_UART_IRQHandler => UART_Receive_IT =>
HAL_UART_RxCpltCallback => HAL_UART_Receive_IT => ...

```

而在用户视角，就是调用 `HAL_UART_Receive_IT` 指定以中断的方式接收数据，接收完成后就进入回调函数 `HAL_UART_RxCpltCallback` 进行后续处理。

【实验结果】

实现了最基本的 peek 和 poke 功能：



五、讨论与心得

本次实验在将 u-boot 移植到树莓派上以后，在引导启动 Linux 内核阶段遇到了较大的困难：一开始是无法进入内核启动程序，后来通过制作 uImage 并手动下载内核到指定地址，告诉 u-boot 到该地址去启动 uImage 的方式顺利进入了内核的启动程序，此时的控制权已经交给了 Linux。但是内核启动过程中出现了同步错误，启动失败，暂时还未找到解决方法。

第二部分单片机实验，编写 boot loader 比较简单，但是在实现串口异步收发和串口全局中断的过程中也遇到了一些困难：由于旧的工程生成的中间文件未完全清理干净等原因，即使代码和配置都正确，回调函数仍然不能正确进入，后来完全新建了工程就能正常运行了。