浙江大学

本科实验报告

姓名：包博文

学院：生物医学工程与仪器科学学院

系：生物医学工程系

专业：生物医学工程

学号：3220105972

指导教师：周凡

2024 年 11 月 29 日

浙江大学实验报告

课程名称：嵌入式系统 实验类型：

实验项目名称：C 语言裸机编程

学生姓名：包博文 专业：生物医学工程 学号：3220105972

同组学生姓名：

指导老师：刘雪松

实验地点：创客空间 实验日期：2024 年 11 月 29 日

一、实验目的和要求

本实验的目的是通过 C 语言编写裸机代码，直接控制树莓派的硬件资源，进一步理解嵌入式系统的运行机制和硬件操作。具体目标包括：

学习并掌握树莓派 GPIO 引脚的控制方法，编写裸机程序实现 LED 灯的点亮和闪烁。

熟悉裸机编程环境的搭建和调试，了解 Makefile 的基础语法并进行代码编译。

独立完成裸机程序的编写、烧录和运行，掌握 GPIO 控制的基本原理。

二、实验内容和原理

1、裸机编程

裸机编程指的是在没有操作系统的情况下，直接控制硬件的运行。在本实验中，通过操作 GPIO 寄存器，点亮或控制 LED 灯闪烁。

2、GPIO寄存器总览

GPIO控制的关键寄存器有以下几类（基地址假设为 MMIO_BASE，以树莓派4为例）：

（1）功能选择寄存器（GPFSELx）：

• 控制GPIO引脚的模式（输入/输出/其他功能）：

000：输入模式

001：输出模式

010~111：特殊功能（Alt0~Alt5，例如UART、SPI、PWM等）

•每个寄存器控制10个GPIO，每个GPIO占3位。如GPFSEL0：控制GPIO 0~9（每个GPIO占3位，共30位），GPFSEL1：控制GPIO 10~19。

（2）设置引脚为高电平（GPSETx）：

• 设置某引脚的输出状态为高（1）。

• 每位对应一个GPIO。

（3）设置引脚为低电平（GPCLRx）：

• 设置某引脚的输出状态为低（0）。

（4）读取引脚状态（GPLEVx）：

• 用于读取GPIO的输入/输出状态。

（5）其他寄存器：

如中断检测寄存器、上拉/下拉寄存器等。

实验内容包括：

安装交叉编译工具：安装用于编译裸机程序的工具链 gcc-arm-none-eabi。

编写裸机代码：使用 C 语言直接操作树莓派 GPIO 寄存器，实现对 LED 的控制。

配置和使用 Makefile：编译 C 语言代码生成裸机镜像文件。

替换树莓派系统：将编译生成的裸机程序烧录至树莓派，观察实验效果。

三、主要仪器设备

树莓派开发板

配备 Ubuntu 系统的 PC（WSL2）

SD 卡及读卡器

LED 灯×1、杜邦线×2

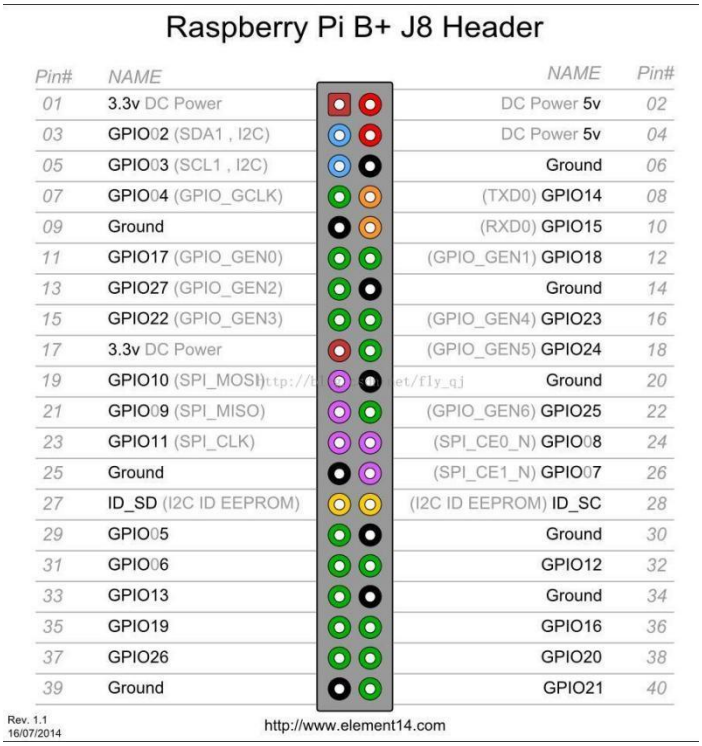
必要的软件工具：

gcc-arm-none-eabi

SCP 工具（用于传输文件）

GPIO 手册（树莓派参考手册）

四、操作方法和实验步骤



1. 阅读了解样例代码的基本原理
2. 修改样例代码控制 GPIO 引脚

参照树莓派手册对样例代码进行修改，实现对具体 GPIO 引脚的控制，使用 make 指令进行交叉编译

3. 替换原有树莓派系统

树莓派 SD 卡的 bootfs 分区中的内容，全部替换为 0.sd_boot 中的内容；用编译后生成的 kernel7.img 替换掉原有的 kernel7.img

4、观察实验结果

五、实验数据记录和处理

1、样例代码分析

由于程序的编译之后，需要进行链接，link 文件告诉了程序链接的规则。程序分为代码段(.text),数据段(.data)以及 bss 段(.bss)。首先将代码段的地址.= 0x8000;指向 0x8000 的地址处，因为默认情况下，树莓派默认启动后，会从 0x8000 这个地址处开始加载程序并启动。KEEP(*(.text.boot))表示首先将.text.boot 的内容放在第一个地址处，目前开始的地址是 0x8000。需要注意的是.bss 段包含的是初始化为零的数据，通过将这些数据放在一个单独的节中，

编译器可以在 elf 文件中省略一些空间。所以需要记录 bss_start 与 bss_end 段。并且将这段空间对齐。如果不对齐，一些函数访问的时候，将会出现异常数据。

main.c 是主程序文件。它初始化 UART，然后进入一个循环，不断从 UART 接收字符并将其发送回去。

peripheral.c 是外设函数的实现文件。它包含了 UART 和 GPIO 的初始化和控制函数。

Makefile 用于编译和链接项目。它定义了编译选项和编译规则。

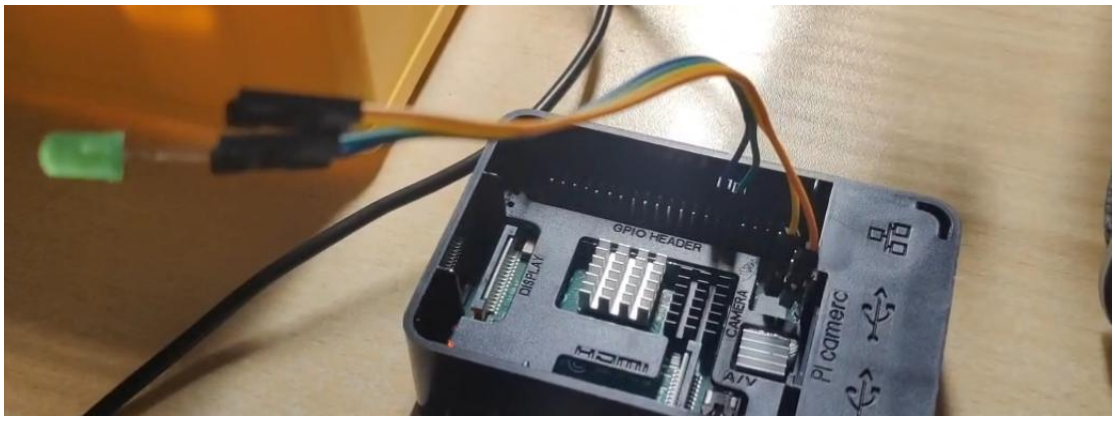
start.S 是启动代码。它设置 CPU 模式、禁用中断、初始化堆栈，并清除 .bss 段。

2、修改样例代码控制 GPIO 引脚

```
bubblevan@Bubbles:~/lab2_zju/code/code$ make
rm kernel7.elf kernel7.img *.o >/dev/null 2>/dev/null || true
arm-none-eabi-gcc -march=armv8-a -Wall -O2 -ffreestanding -nostdinc -nostdlib -nostartfiles -c start.S -o start.o
arm-none-eabi-gcc -march=armv8-a -Wall -O2 -ffreestanding -nostdinc -nostdlib -nostartfiles -c main.c -o main.o
arm-none-eabi-gcc -march=armv8-a -Wall -O2 -ffreestanding -nostdinc -nostdlib -nostartfiles -c peripheral.c -o periphera
l.o
# 使用 arm-none-eabi-ld 进行链接
# -nostdlib 禁用标准库的默认链接
# start.o 是启动代码，main.o peripheral.o 是所有编译后的目标文件
# -T link.ld 使用指定的链接脚本 link.ld 定义内存布局
arm-none-eabi-ld -nostdlib start.o main.o peripheral.o -T link.ld -o kernel7.elf
# 将 ELF 格式的文件转换为二进制镜像文件
arm-none-eabi-objcopy -O binary kernel7.elf kernel7.img
```

3、替换原有树莓派系统

4、观察实验结果



连接 GPIO12 与 GND，可见 LED 闪烁发光。

5、学习 Makefile 的使用

Makefile 的知识主要在学在浙大那本 gnu-make 中文手册中了解（顺带一提手册的页眉全是 GUN make）。下面进行 cmake 的学习：

（1）sudo apt-get install gcc-arm-none-eabi

（2）然后创建 raspi_toolchain.cmake，定义交叉编译器和目标架构

CMake 通过工具链文件（toolchain file）支持交叉编译：

CMAKE_SYSTEM_NAME：目标系统的名称，例如 Generic 用于裸机开发。

CMAKE_SYSTEM_PROCESSOR：目标系统的处理器架构，如 arm。

CMAKE_C_COMPILER：C 语言编译器的路径，如 arm-none-eabi-gcc。

CMAKE_CXX_COMPILER：C++语言编译器的路径，如 arm-none-eabi-g++。

CMAKE_TRY_COMPILE_TARGET_TYPE：指定编译类型，裸机开发中常用 STATIC_LIBRARY。

add_compile_options：指定编译选项，例如-nostdlib 避免链接标准库。

```
# raspi_toolchain.cmake

set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR arm)
set(CMAKE_C_COMPILER   arm-none-eabi-gcc)
set(CMAKE_CXX_COMPILER arm-none-eabi-g++)
set(CMAKE_ASM_COMPILER arm-none-eabi-gcc)
set(CMAKE_TRY_COMPILE_TARGET_TYPE "STATIC_LIBRARY")
set(CMAKE_C_COMPILER_WORKS TRUE)
set(CMAKE_CXX_COMPILER_WORKS TRUE)
set(CMAKE_C_STANDARD_LIBRARIES "")
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE NEVER)
```

（3）编写 CmakeLists.txt，管理源文件和依赖

```
# CMakeLists.txt


cmake_minimum_required(VERSION 3.5)

# Include the toolchain file

set(CMAKE_TOOLCHAIN_FILE ${CMAKE_SOURCE_DIR}/raspi_toolchain.cmake)

# Project setup

project(raspi_gpio)

add_executable(raspi_gpio

    main.c

    peripheral.c

    start.S

)

# Compiler flags

add_compile_options(-march=armv7-a -mfpu=vfpv3 -mfloat-abi=hard -nostdlib -ffreestanding -fno-builtin)

# Linker options

target_link_options(raspi_gpio PRIVATE

    -T ${CMAKE_SOURCE_DIR}/link.ld

    -Wl,--gc-sections

)

# Generate binary firmware

add_custom_target(firmware

    COMMAND ${CMAKE_OBJCOPY} -O binary ${CMAKE_BINARY_DIR}/raspi_gpio ${CMAKE_BINARY_DIR}/kernel.img

    DEPENDS raspi_gpio

)

# Enable verbose output for debugging

set(CMAKE_VERBOSE_MAKEFILE ON)
```

（4）编译项目

```
bubblevan@Bubbles:~/lab2_zju/code/code$ mkdir build
bubblevan@Bubbles:~/lab2_zju/code/code$ cd build
bubblevan@Bubbles:~/lab2_zju/code/code/build$ cmake ..
-- The C compiler identification is GNU 13.2.1
-- The CXX compiler identification is GNU 13.2.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/arm-none-eabi-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/arm-none-eabi-g++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (4.0s)
-- Generating done (0.0s)
-- Build files have been written to: /home/bubblevan/lab2_zju/code/code/build
bubblevan@Bubbles:~/lab2_zju/code/code/build$ make
/usr/bin/cmake -S/home/bubblevan/lab2_zju/code/code -B/home/bubblevan/lab2_zju/code/code/build --check-build-system CMakeFiles/Makefile.cmake 0
/usr/bin/cmake -E cmake_progress_start /home/bubblevan/lab2_zju/code/code/build/CMakeFiles /home/bubblevan/lab2_zju/code/code/build//CMakeFiles/progress.marks
make  -f CMakeFiles/Makefile2 all
make[1]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
make  -f CMakeFiles/raspi_gpio.dir/build.make CMakeFiles/raspi_gpio.dir/depend
make[2]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
cd /home/bubblevan/lab2_zju/code/code/build && /usr/bin/cmake -E cmake_depends "Unix Makefiles" /home/bubblevan/lab2_zju/code/code /home/bubblevan/lab2_zju/code/code /home/bubblevan/lab2_zju/code/code/build /home/bubblevan/lab2_zju/code/code/build /home/bubblevan/lab2_zju/code/code/build/CMakeFiles/raspi_gpio.dir/DependInfo.cmake "--color="
make[2]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
make  -f CMakeFiles/raspi_gpio.dir/build.make CMakeFiles/raspi_gpio.dir/build
make[2]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
[ 33%] Building C object CMakeFiles/raspi_gpio.dir/main.c.obj
/usr/bin/arm-none-eabi-gcc    -MD -MT CMakeFiles/raspi_gpio.dir/main.c.obj -MF CMakeFiles/raspi_gpio.dir/main.c.obj.d -o CMakeFiles/raspi_gpio.dir/main.c.obj -c /home/bubblevan/lab2_zju/code/code/main.c
[ 66%] Building C object CMakeFiles/raspi_gpio.dir/peripheral.c.obj
/usr/bin/arm-none-eabi-gcc    -MD -MT CMakeFiles/raspi_gpio.dir/peripheral.c.obj -MF CMakeFiles/raspi_gpio.dir/peripheral.c.obj.d -o CMakeFiles/raspi_gpio.dir/peripheral.c.obj -c /home/bubblevan/lab2_zju/code/code/peripheral.c
[100%] Linking C executable raspi_gpio
/usr/bin/cmake -E cmake_link_script CMakeFiles/raspi_gpio.dir/link.txt --verbose=1
/usr/bin/arm-none-eabi-gcc -T /home/bubblevan/lab2_zju/code/code/link.ld -Wl,--gc-sections CMakeFiles/raspi_gpio.dir/main.c.obj CMakeFiles/raspi_gpio.dir/peripheral.c.obj -o raspi_gpio
make[2]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
[100%] Built target raspi_gpio
make[1]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
/usr/bin/cmake -E cmake_progress_start /home/bubblevan/lab2_zju/code/code/build/CMakeFiles 0
```

kernel.img 的生成依赖于 firmware 目标，但 firmware 没有被包含在 all 默认目标中，所以我需要手动 make firmware ：

```
bubblevan@Bubbles:~/lab2_zju/code/code/build$ make firmware
/usr/bin/cmake -S/home/bubblevan/lab2_zju/code/code -B/home/bubblevan/lab2_zju/code/code/build --check-build-system CMakeFiles/Makefile.cmake 0
make  -f CMakeFiles/Makefile2 firmware
make[1]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
/usr/bin/cmake -S/home/bubblevan/lab2_zju/code/code -B/home/bubblevan/lab2_zju/code/code/build --check-build-system CMakeFiles/Makefile.cmake 0
/usr/bin/cmake -E cmake_progress_start /home/bubblevan/lab2_zju/code/code/build/CMakeFiles 3
make  -f CMakeFiles/Makefile2 CMakeFiles/firmware.dir/all
make[2]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
make  -f CMakeFiles/raspi_gpio.dir/build.make CMakeFiles/raspi_gpio.dir/depend
make[3]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
cd /home/bubblevan/lab2_zju/code/code/build && /usr/bin/cmake -E cmake_depends "Unix Makefiles" /home/bubblevan/lab2_zju/code/code /home/bubblevan/lab2_zju/code/code /home/bubblevan/lab2_zju/code/code/build /home/bubblevan/lab2_zju/code/code/build /home/bubblevan/lab2_zju/code/code/build/CMakeFiles/raspi_gpio.dir/DependInfo.cmake "--color="
make[3]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
make  -f CMakeFiles/raspi_gpio.dir/build.make CMakeFiles/raspi_gpio.dir/build
make[3]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
make[3]: Nothing to be done for 'CMakeFiles/raspi_gpio.dir/build'.
make[3]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
[100%] Built target raspi_gpio
make  -f CMakeFiles/firmware.dir/build.make CMakeFiles/firmware.dir/depend
make[3]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
cd /home/bubblevan/lab2_zju/code/code/build && /usr/bin/cmake -E cmake_depends "Unix Makefiles" /home/bubblevan/lab2_zju/code/code /home/bubblevan/lab2_zju/code/code /home/bubblevan/lab2_zju/code/code/build /home/bubblevan/lab2_zju/code/code/build /home/bubblevan/lab2_zju/code/code/build/CMakeFiles/firmware.dir/DependInfo.cmake "--color="
make[3]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
make  -f CMakeFiles/firmware.dir/build.make CMakeFiles/firmware.dir/build
make[3]: Entering directory '/home/bubblevan/lab2_zju/code/code/build'
/usr/bin/arm-none-eabi-objcopy -O binary /home/bubblevan/lab2_zju/code/code/build/raspi_gpio /home/bubblevan/lab2_zju/code/code/build/kernel.img
make[3]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
[100%] Built target firmware
make[2]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
/usr/bin/cmake -E cmake_progress_start /home/bubblevan/lab2_zju/code/code/build/CMakeFiles 0
make[1]: Leaving directory '/home/bubblevan/lab2_zju/code/code/build'
```

六、实验结果与分析

在将生成的 kernel7.img 替换树莓派原有的系统镜像后，树莓派能够正确加载裸机程序并启动，验证了编译、链接以及裸机启动流程的正确性。编写的裸机程序成功点亮和控制 LED 灯的闪烁，当 GPIO12 与 GND 连接时，LED 按设定的时间间隔持续亮灭，显示出 GPIO 引脚设置和控制的效果。

七、讨论、心得

1、思考题

这次安装的 arm-none-eabi-gcc 也是交叉编译器，和上次用的 aarch64-linux-gnu-gcc 有什么区别呢？

arm-none-eabi-gcc 的目标架构是 ARM 32 位架构（ARMv7 及更早版本），主要用于嵌入式系统开发，特别是那些没有操作系统（bare-metal）的系统。它通常用于编译裸机程序、固件、引导加载程序等。这个工具链不包含操作系统相关的库和头文件，如 libc、libm 等，而是包含一些基本的库，如 libgcc，用于支持基本的编译和链接。因此，它适用于开发嵌入式系统，特别是那些没有操作系统的系统，例如编译引导加载程序（bootloader）、固件（firmware）、裸机程序（bare-metal programs）等。

aarch64-linux-gnu-gcc 的目标架构是 ARM 64 位架构(ARMv8 及更高版本)，主要用于编译运行在 Linux 操作系统上的应用程序。它包含了 Linux 系统调用、

库和头文件，适用于开发运行在 Linux 上的应用程序。这个工具链包含完整的 Linux 系统库和头文件，如 libc、libm、libpthread 等，适用于开发需要使用 Linux 系统调用和库的应用程序。因此，它适用于开发运行在 ARM 64 位处理器上的 Linux 应用程序，例如编译运行在 ARM 64 位处理器上的服务器程序、桌面应用程序、嵌入式 Linux 系统等。

在工具链组件方面，arm-none-eabi-gcc 包含编译器（arm-none-eabi-gcc）、链接器（arm-none-eabi-ld）、调试器（arm-none-eabi-gdb）等，通常用于生成 ELF 格式的可执行文件，但不包含操作系统加载器。而 aarch64-linux-gnu-gcc 包含编译器（aarch64-linux-gnu-gcc）、链接器（aarch64-linux-gnu-ld）、调试器（aarch64-linux-gnu-gdb）等，用于生成 ELF 格式的可执行文件，并包含 Linux 加载器和库。

总结来说，arm-none-eabi-gcc 适用于 ARM 32 位架构的裸机开发，不包含操作系统支持，而 aarch64-linux-gnu-gcc 适用于 ARM 64 位架构的 Linux 应用程序开发，包含完整的 Linux 系统支持。选择哪个工具链取决于你的开发目标和目标平台的需求。如果你在开发嵌入式系统且不需要操作系统支持，arm-none-eabi-gcc 是合适的选择。如果你在开发运行在 ARM 64 位处理器上的 Linux 应用程序，aarch64-linux-gnu-gcc 是更好的选择。

2、实验感受

裸机开发需要直接操作寄存器，这要求对硬件手册非常熟悉。此外本实验的关键技术之一是交叉编译，我初步使用 Makefile 完成了裸机程序的编译，理解了目标文件生成的规则和依赖关系。CMake 的引入则更进一步，通过模块化配置和自动依赖管理，简化了交叉编译工具链的使用流程，特别适用于复杂项目的管理。

八、代码附件

```
void main()
{
    // set up serial console
    uart_init();
    uart_puts("uart init\n");
```

```
    led_gpio_init();
    uart_puts("LED GPIO initialized\n");

    // echo everything back
    while(1) {
        // uart_send(uart_getc());
        led_gpio_on();
        sleep(10000);
        led_gpio_off();
        sleep(10000);
    }
    //============================================================================
    // TODO: Comment out the above code and uncomment the following code to
    // replace it with an LED light control program.
    //============================================================================
    // led_gpio_init();
    // while(1) {
    //     sleep(10000);
    //     led_gpio_on();
    //     sleep(10000);
    //     led_gpio_off();
    // }
}
```

```
void led_gpio_init() {
    // TODO
    register unsigned int r;
    r = *GPFSEL1;         // 读取 GPFSEL1 寄存器的值
    r &= ~(7 << 6);       // 清除 GPIO 12 功能选择位（对应 6-8 位）
    r |= (1 << 6);        // 设置 GPIO 12 为输出模式（6-8 位为 001）
    *GPFSEL1 = r;         // 写入 GPFSEL1 寄存器


}
```

```
int led_gpio_on() {
    // TODO
    *GPSET0 = 1 << 12;    // 设置 GPIO 12 为高电平
    return 1;
}
```

```
int led_gpio_off() {
    // TODO
    *GPCLR0 = 1 << 12;    // 设置 GPIO 12 为低电平
```

```
    return 0;
}
```