

实验 2 基于 GPIO 的基本输入输出

实验目的：利用 STM32F407 GPIO 的输入输出功能读取和控制按键、LED、LCD 等外设，熟悉并掌握 GPIO 的设置和使用方法。

实验内容：

- 通过 GPIO 口的高低电平输出控制，实现一个经典的流水灯程序。
- 了解 STM32F407 的 GPIO 口作为输出使用的方法。
- 初步掌握 STM32F407 作为基本 IO 的使用方法。

2.1 STM32F407 IO 功能简介

STM32F407 每组通用 I/O 端口（16 个引脚）包括 4 个 32 位配置寄存器（MODER、OTYPER、OSPEEDR 和 PUPDR）、2 个 32 位数据寄存器（IDR 和 ODR）、1 个 32 位置位/复位寄存器（BSRR）、1 个 32 位锁定寄存器(LCKR) 和 2 个 32 位复用功能选择寄存器（AFRH 和 AFRL）等。

其中常用的有 4 个配置寄存器、2 个数据寄存器、2 个复用功能选择寄存器，共 8 个。

```
#define PERIPH_BASE          ((uint32_t)0x40000000)
#define AHB1PERIPH_BASE     (PERIPH_BASE + 0x00020000)
#define GPIOA_BASE          (AHB1PERIPH_BASE + 0x0000)
#define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)

typedef struct
{
    __IO uint32_t MODER;
    __IO uint32_t OTYPER;
    __IO uint32_t OSPEEDR;
    __IO uint32_t PUPDR;
    __IO uint32_t IDR;
    __IO uint32_t ODR;
    __IO uint16_t BSRR;
    __IO uint16_t BSRRH;
    __IO uint32_t LCKR;
    __IO uint32_t AFR[2];
```

```
} GPIO_TypeDef;  
GPIOA->ODR = 0x01;
```

如果在使用的時候，每次都直接用匯編操作寄存器配置 IO，過程比較繁瑣，所以一般用庫函數配置 IO。

同 STM32F1 一樣，STM32F4 的 IO 可以由軟件配置成如下 8 種模式中的任何一種：

- 1、輸入浮空；2、輸入上拉；3、輸入下拉；4、模擬輸入；5、開漏輸出；6、推挽輸出；
- 7、推挽复用功能；8、開漏复用功能。

2.2 新工程的配置流程

基於 C 的外設控制相關工程，需要用到大量的庫函數和相應頭文件，為了使工程層次清晰避免混亂，需要建立不同的子目錄存放各類庫文件，具體操作流程如下。

1. 工程目錄結構配置

在工程文件夾中新建 5 個文件夾為 CORE、OBJ、FWLIB、SYSTEM、USER（工程文件存放該目錄）（這個步驟可以在新建工程嚮導前實施），如圖 1.1 所示，下面我們對各文件夾存放的文件及各自的功能依次進行說明。

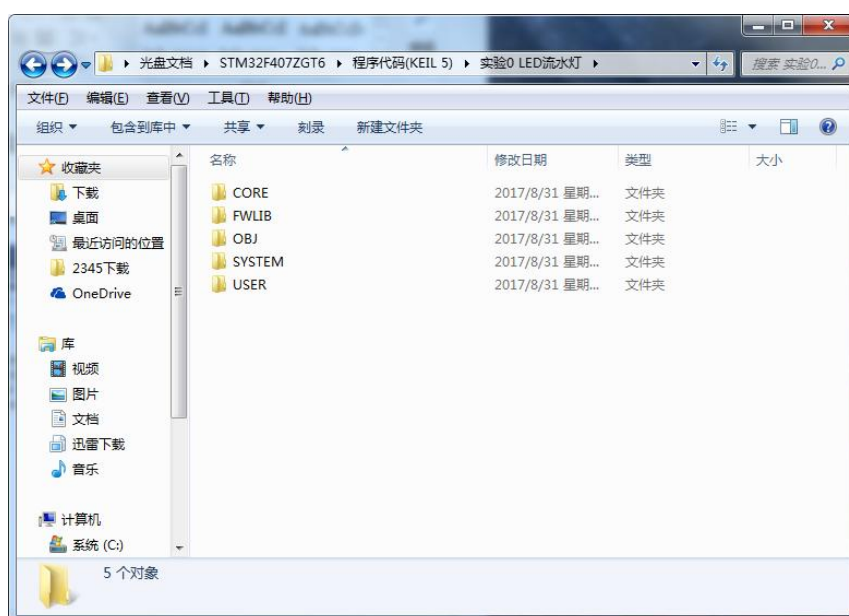


圖 1.1 工程目錄預覽






（1） CORE 文件夾下面存放的是固件庫必須的核心文件和啟動文件，這裡面的文件用戶不需要修改，可以根據自己的芯片型號選擇對應的啟動文件。

core_cm4.h: 內核功能的定義，NVIC 相關寄存器的結構體和 SysTick 配置；

core_cm4_simd.h: 包含與編譯器相關的處理；

core_cmFunc.h: 內核核心功能接口的頭文件；

core_cmInstr.h: 包含一些内核核心专用指令;

名称	修改日期	类型	大小
 core_cm4.h	2016/4/15 21:58	C/C++ Header	107 KB
 core_cm4_simd.h	2016/4/15 21:58	C/C++ Header	23 KB
 core_cmFunc.h	2016/4/15 21:58	C/C++ Header	17 KB
 core_cmInstr.h	2016/4/15 21:58	C/C++ Header	21 KB
 startup_stm32f40_41xxx.s	2016/4/15 21:58	Assembler Source	29 KB

启动文件

图 1.2 CORE 目录中文件

(2) FWLIB 文件夹下面存放的是 ST 官方提供的固件库函数，每一个源文件 stm32f4xx_xx.c 都对应一个头文件 stm32f4xx_xx.h，代表着一个外设或者接口的功能函数集合，文件夹下分 src 和 inc 子文件夹分别存放源文件和头文件。分组内的文件我们可以根据工程需要添加和删除，但是一定要注意如果你引入了某个源文件，一定要在头文件 stm32f4xx_conf.h 文件中确保对应的头文件也已经添加。比如我们第一个流水灯实验，我们只添加了 5 个源文件，那么对应的头文件我们必须确保在 stm32f4xx_conf.h 内也包含进来，否则工程会报错。












































名称	修改日期	类型	大小	名称	修改日期	类型
 misc.h	2016/4/15 21:58	C/C++ Header	7 KB	 misc.c	2016/4/15 21:58	C Source
 stm32f4xx_adc.h	2016/4/15 21:58	C/C++ Header	33 KB	 stm32f4xx_adc.c	2016/4/15 21:58	C Source
 stm32f4xx_can.h	2016/4/15 21:58	C/C++ Header	27 KB	 stm32f4xx_can.c	2016/4/15 21:58	C Source
 stm32f4xx_crc.h	2016/4/15 21:58	C/C++ Header	3 KB	 stm32f4xx_crc.c	2016/4/15 21:58	C Source
 stm32f4xx_cryp.h	2016/4/15 21:58	C/C++ Header	15 KB	 stm32f4xx_cryp.c	2016/4/15 21:58	C Source
 stm32f4xx_dac.h	2016/4/15 21:58	C/C++ Header	15 KB	 stm32f4xx_cryp_aes.c	2016/4/15 21:58	C Source
 stm32f4xx_dbgmcu.h	2016/4/15 21:58	C/C++ Header	5 KB	 stm32f4xx_cryp_des.c	2016/4/15 21:58	C Source
 stm32f4xx_dcmi.h	2016/4/15 21:58	C/C++ Header	13 KB	 stm32f4xx_cryp_tdes.c	2016/4/15 21:58	C Source
 stm32f4xx_dma.h	2016/4/15 21:58	C/C++ Header	29 KB	 stm32f4xx_dac.c	2016/4/15 21:58	C Source
 stm32f4xx_dma2d.h	2016/4/15 21:58	C/C++ Header	20 KB	 stm32f4xx_dbgmcu.c	2016/4/15 21:58	C Source
 stm32f4xx_exti.h	2016/4/15 21:58	C/C++ Header	8 KB	 stm32f4xx_dcmi.c	2016/4/15 21:58	C Source
 stm32f4xx_flash.h	2016/4/15 21:58	C/C++ Header	24 KB	 stm32f4xx_dma.c	2016/4/15 21:58	C Source
 stm32f4xx_flash_ramfunc.h	2016/4/15 21:58	C/C++ Header	4 KB	 stm32f4xx_dma2d.c	2016/4/15 21:58	C Source
 stm32f4xx_fsmc.h	2016/4/15 21:58	C/C++ Header	44 KB	 stm32f4xx_exti.c	2016/4/15 21:58	C Source
 stm32f4xx_fsmc.h	2016/4/15 21:58	C/C++ Header	27 KB	 stm32f4xx_flash.c	2016/4/15 21:58	C Source
 stm32f4xx_gpio.h	2016/4/15 21:58	C/C++ Header	23 KB	 stm32f4xx_flash_ramfunc.c	2016/4/15 21:58	C Source
 stm32f4xx_hash.h	2016/4/15 21:58	C/C++ Header	10 KB	 stm32f4xx_fmc.c	2016/4/15 21:58	C Source
 stm32f4xx_i2c.h	2016/4/15 21:58	C/C++ Header	32 KB	 stm32f4xx_fsmc.c	2016/4/15 21:58	C Source
 stm32f4xx_iwdg.h	2016/4/15 21:58	C/C++ Header	5 KB	 stm32f4xx_gpio.c	2016/4/15 21:58	C Source
 stm32f4xx_itdc.h	2016/4/15 21:58	C/C++ Header	21 KB	 stm32f4xx_hash.c	2016/4/15 21:58	C Source
 stm32f4xx_pwr.h	2016/4/15 21:58	C/C++ Header	8 KB			
 stm32f4xx_rcc.h	2016/4/15 21:58	C/C++ Header	30 KB			
 stm32f4xx_rng.h	2016/4/15 21:58	C/C++ Header	4 KB			

图 1.3 FWLIB 文件夹下 inc 和 src 中文件

(3) SYSTEM 文件夹中提供一些各类工程都需要用到的共用代码减少重复开发的麻烦，包括延时、串口调用、位带定义等。




名称	修改日期	类型
 delay	2017/9/25 14:58	文件夹
 sys	2017/9/25 14:58	文件夹
 usart	2017/9/25 14:58	文件夹

图 1.4 SYSTEM 文件夹下的共用功能文件

(4) OBJ 文件夹用于存放编译结果的 obj 中间文件和二进制 hex 文件等。

(5) USER 文件夹下面存放的主要是用户代码，还有一些重要的头文件 **stm32f4xx.h** 和 **system_stm32f4xx.h** 以及 **main.c**, **stm32f4xx_conf.h**, **stm32f4xx_it.c** (里面存放的是中断服务函数), **stm32f4xx_it.h**, **system_stm32f4xx.c**, 这些文件和工程文件是在同一目录下可以直接调用或者引用。其中 **stm32f4xx.h** 是非常重要的头文件，是片上外设访问头文件，里面包含大量结构体和宏定义（寄存器定义声明及内存访问操作的封装）。

它的说明如下@brief CMSIS Cortex-M4 Device Peripheral Access Layer Header File. This file contains all the peripheral register's definitions, bits definitions and memory mapping for STM32F4xx devices. The file is the unique include file that the application programmer is using in the C source code, usually in main.c. This file contains:

- Configuration section that allows to select:
- The device used in the target application
- To use or not the peripheral's drivers in application code(i.e. code will be based on direct access to peripheral's registers rather than drivers API), this option is controlled by `"#define USE_STDPERIPH_DRIVER"`
- To change few application-specific parameters such as the HSE crystal frequency
- Data structures and the address mapping for all peripherals
- Peripheral's registers declarations and bits definition
- Macros to access peripheral's registers hardware

另外 **stm32f4xx_conf.h** 也是非常重要的头文件，是外设驱动配置文件，可以根据需要 include 相关的外设头文件，它与 **stm32f4xx.h** 的关系我们后面会介绍。

名称	修改日期	类型	大小
JLinkLog.txt	2019/12/19 13:30	文本文档	6 KB
JLinkSettings.ini	2016/4/15 21:58	配置设置	1 KB
LED.uvguix.Administrator	2017/9/27 18:33	ADMINISTRATOR ...	139 KB
LED.uvguix.stephen_zhu_macair	2019/11/28 14:41	STEPHEN_ZHU_M...	69 KB
LED.uvguix.ustc01	2020/1/6 21:29	USTC01 文件	139 KB
LED.uvoptx	2020/1/6 21:29	UVOPTX 文件	15 KB
LED.uvprojx	2019/12/24 15:49	磬ision5 Project	18 KB
main.c	2017/9/18 13:56	C Source	3 KB
stm32f4xx.h	2016/4/15 21:58	C/C++ Header	688 KB
stm32f4xx_conf.h	2016/4/15 21:58	C/C++ Header	5 KB
stm32f4xx_it.c	2016/4/15 21:58	C Source	5 KB
stm32f4xx_it.h	2016/4/15 21:58	C/C++ Header	3 KB
system_stm32f4xx.c	2016/4/15 21:58	C Source	47 KB
system_stm32f4xx.h	2016/4/15 21:58	C/C++ Header	3 KB

图 1.5 USER 文件夹下的主要文件

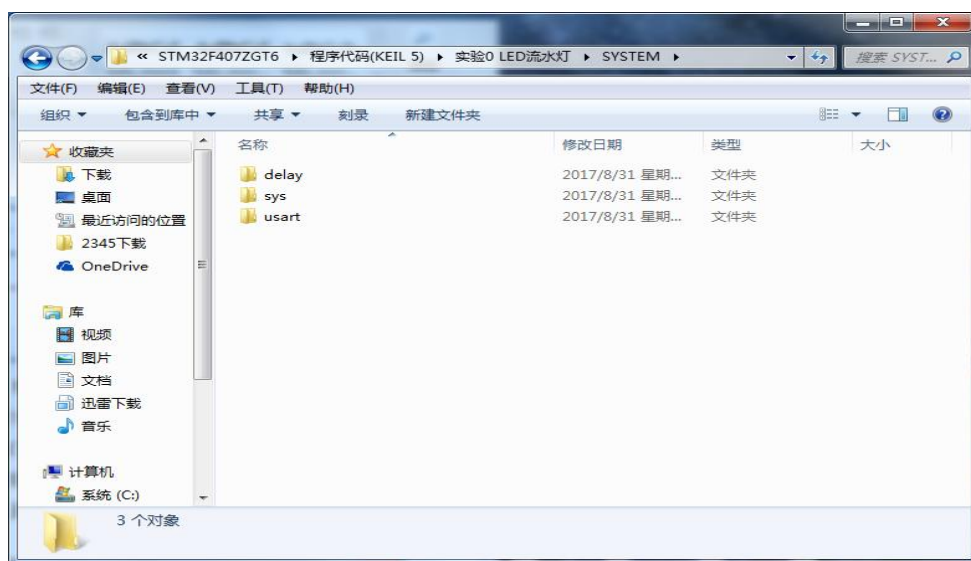


图 1.6 SYSTEM 目录文件浏览

注意： 以上的步骤只是把需要的固件库等相关文件复制到了工程目录下，下面还需要将这些文件加入到工程中去才能使用。

2. 工程文件添加及管理

右键点击 Target1，选择 Manage Project Items，如图 1.7 所示。

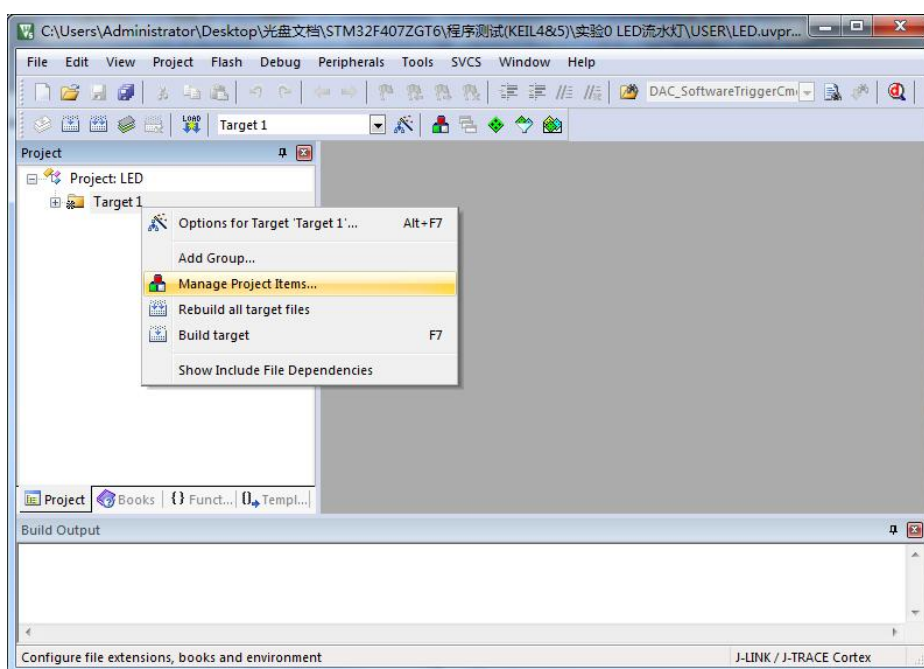


图 1.7 文件的添加设置

选择 Project Targets，在 Groups 一栏删掉 Source Group1，建立 4 个 Groups: USER, CORE, FWLIB, SYSTEM。然后点击 OK，如图 1.8 可以看到我们的 Target 名字以及 Groups 情况。

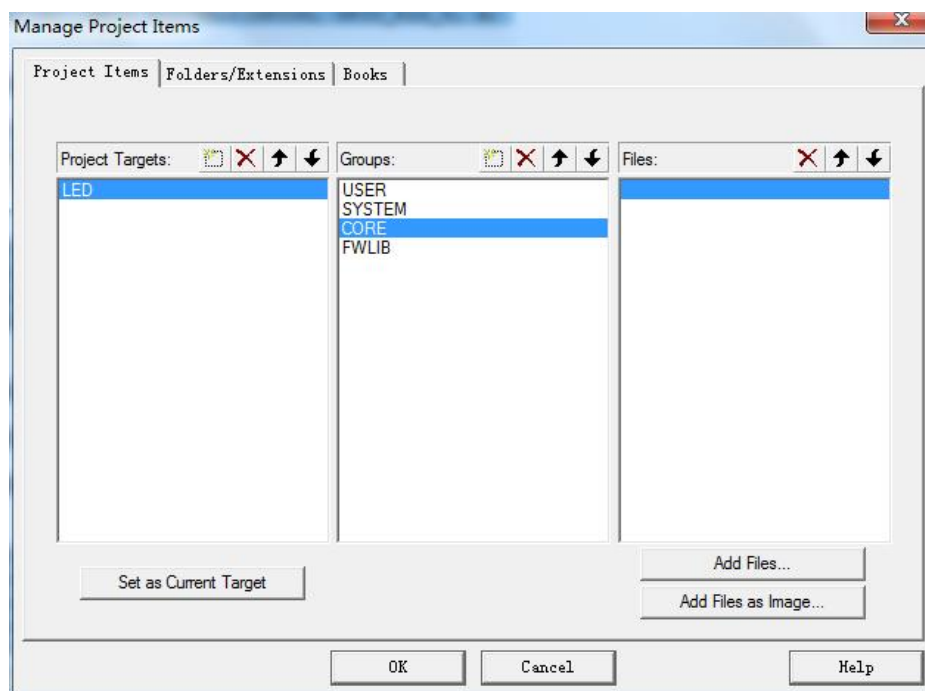


图 1.8 新建组

下面在 Group 添加需要的文件（以 LED 流水灯为例）。

第一步选择 FWLIB，然后点击右边的 Add Files，定位刚才建立的目录 FWLIB/src 下面，如果将所有文件全部添加，可以将里面所有的文件选中(Ctrl+A)，然后点击 Add，然后 Close。但是一般不会这样，我们只选择需要的文件，可以看到流水灯工程中添加的文件如图 1.9。

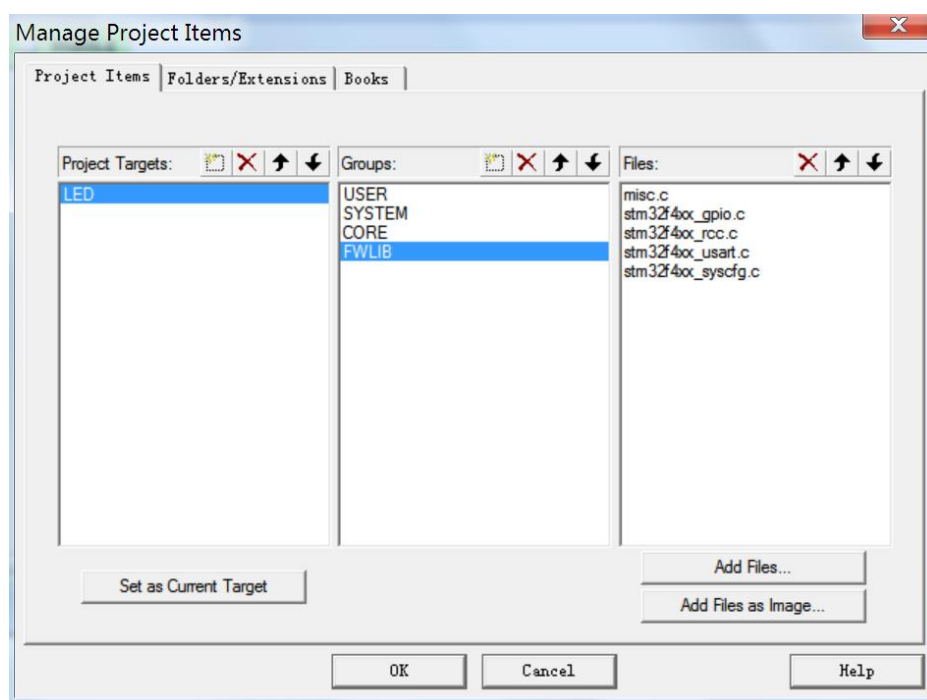


图 1.9 组 FWLIB 下添加的文件

用同样的方法，将 Groups 定位到 CORE 和 USER 下面，添加需要的文件。CORE 下面需要添加的文件为 startup_stm32f40_41xxx.s，USER 目录下面需要添加的文件为 stm32f4xx_it.c，system_stm32f4xx.c 以及 main.c，如图 1.10 和 1.11 所示。

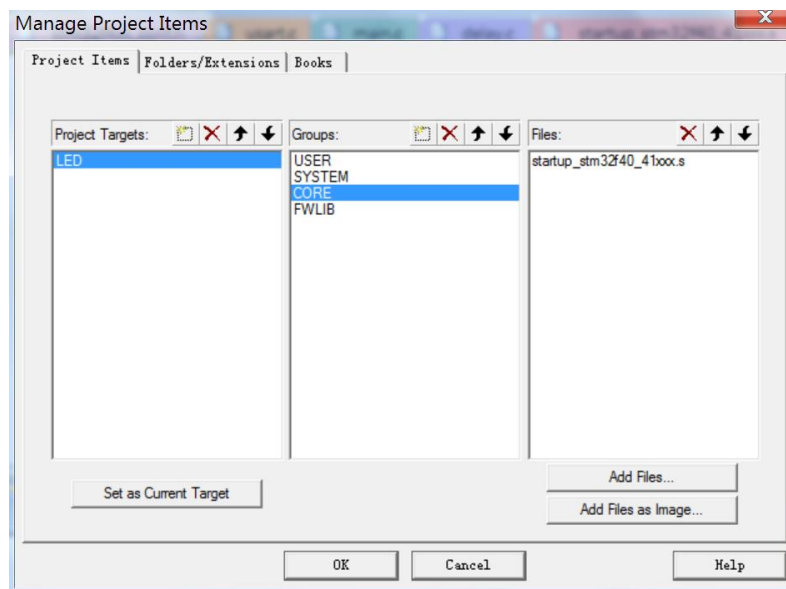


图 1.10 组 CORE 下添加的文件

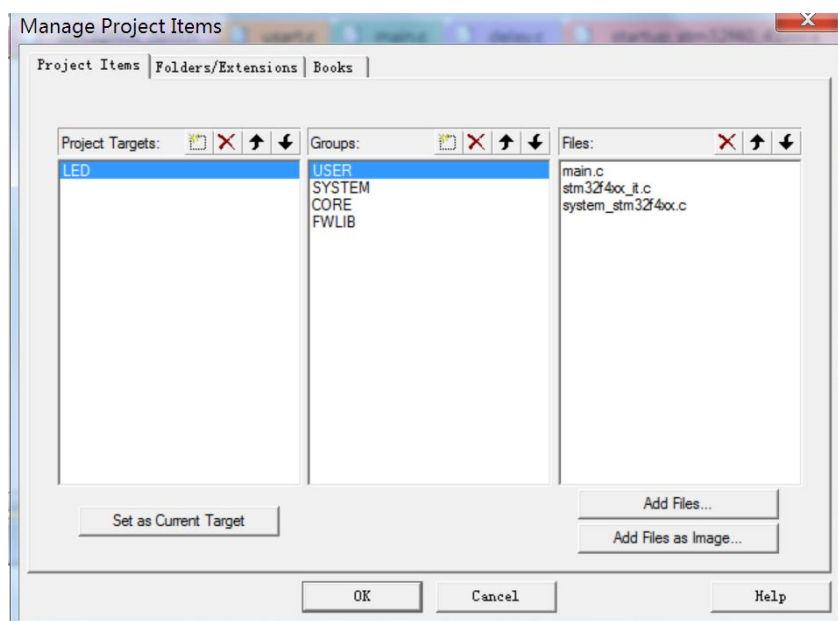


图 1.11 组 USER 下添加的文件

这样需要添加的文件已经添加到工程中了，最后点击 OK，回到工程主界面。

注意：组 SYSTEM 和 CORE 中添加的文件在所有类型的工程中都不需要改变，组 USER 中添加的文件基本不变，只有 main.c 会随着工程需求而修改代码，组 FWLIB 会随着工程需求而选择添加不同的库文件。可以参考 D 盘实验目录中的 LED 示例工程。

工程的配置

编译工程，在编译之前首先要选择编译中间文件编译后存放目录。点击魔术棒，然后选择“Output”选项下面的“Select folder for objects...”,然后选择目录为新建的 OBJ 目录，如图 1.12 所示。注意，如果不设置 Output 路径，那么默认的编译中间文件存放目录就是 MDK 自动生成的 Objects 目录和 Listings 目录。

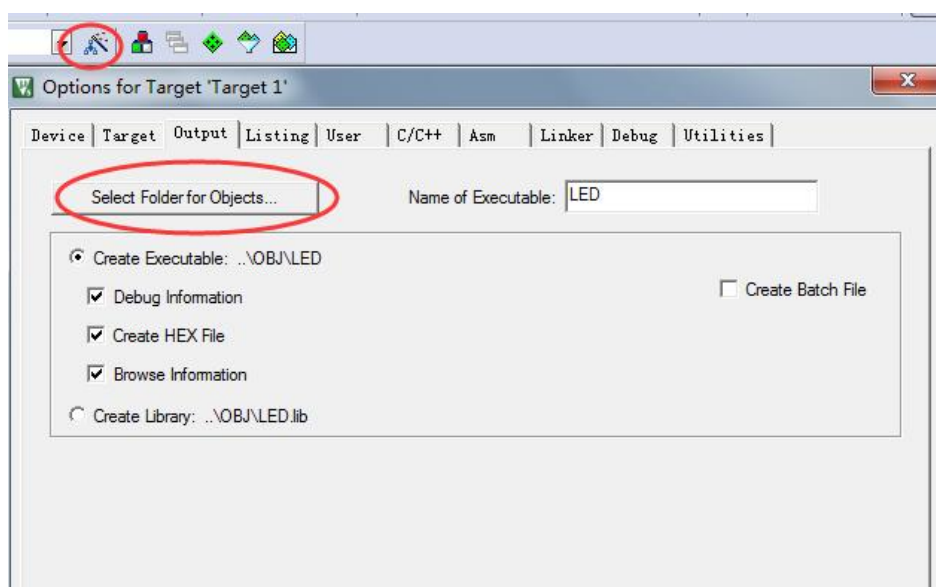


图 1.12 设置 OBJ 目录

下面设置头文件路径，也就是头文件目录。这里要注意，对于任何一个工程，都需要把工程中引用到的所有头文件的路径都包含进来。

回到工程主菜单，点击魔术棒，然后点击 c/c++选项.然后点击 Include Paths 右边的按钮，弹出一个添加包含头文件目录的对话框，将如图 1.13 的路径添加进去，这些路径都是拷贝过.h 头文件的路径。Keil 只会在一级目录查找，所以如果你的目录下面还有子目录，记得 path 一定要定位到最后一级子目录。然后点击 OK。

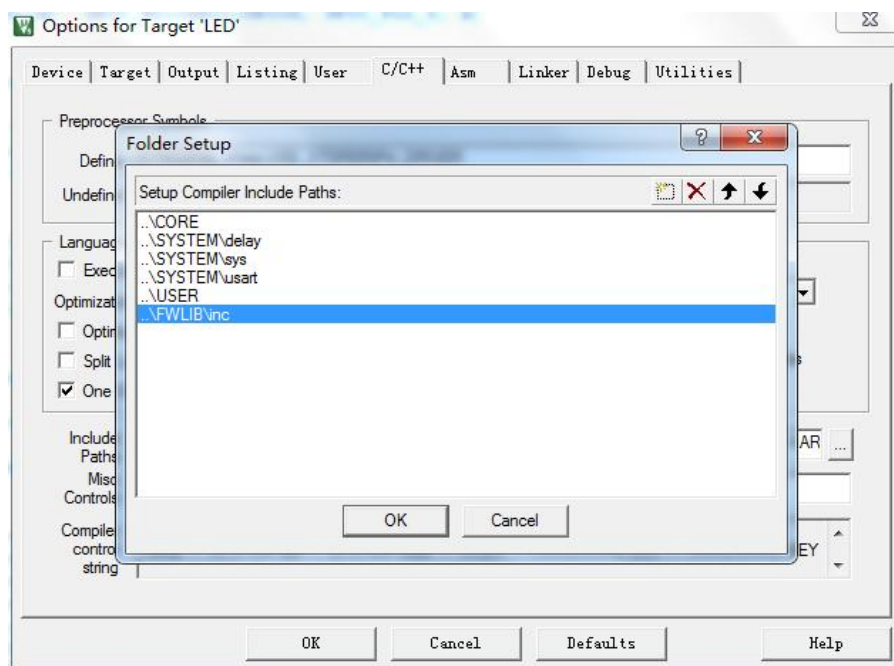


图 1.13 工程设置

因为库函数在配置和选择外设的时候通过宏定义来选择的，所以需要配置全局的宏定义变量。点击魔术棒，定位到 C/C++ 界面，然后填写“STM32F40_41xxx, USE_STDPERIPH_DRIVER”到 Define 输入框里面。注意这里是两个标识符 STM32F40_41xxx 和 USE_STDPERIPH_DRIVER，之间是用逗号隔开的。

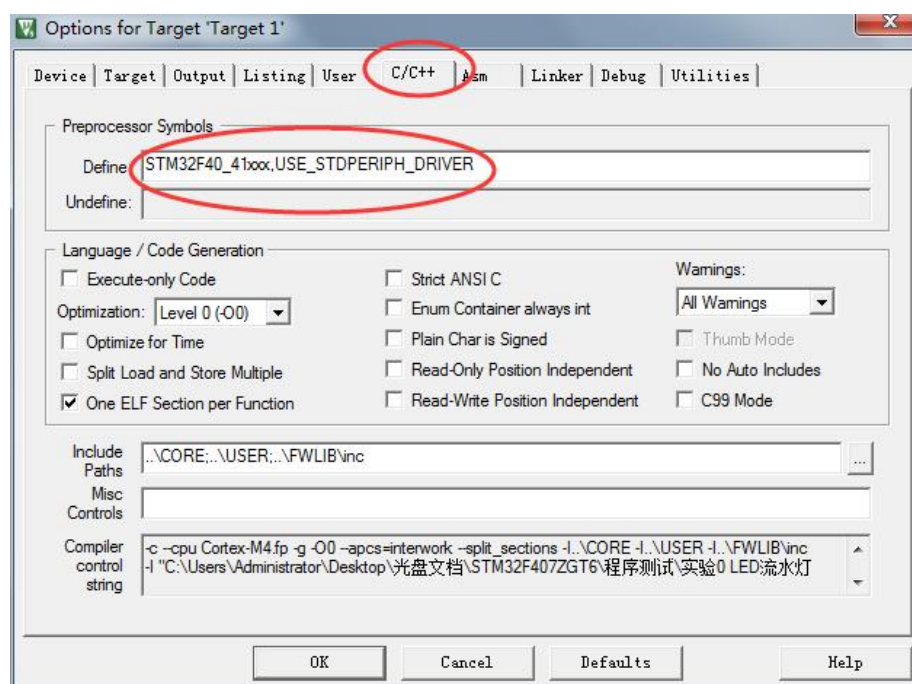


图 1.14 工程设置-宏定义的添加

注：关于宏 USE_STDPERIPH_DRIVER 的作用，可以在 stm32f4xx.h 中找到如图所示的条件编译指令：

```

9135 #ifndef USE_STDPERIPH_DRIVER
9136     #include "stm32f4xx_conf.h"
9137 #endif /* USE_STDPERIPH_DRIVER */
9138
9139 /** @addtogroup Exported_macro
9140     * @{
9141     */
9142

```

点击‘Output’项勾选生产 HEX 文件,然后点击 OK 完成工程的配置。

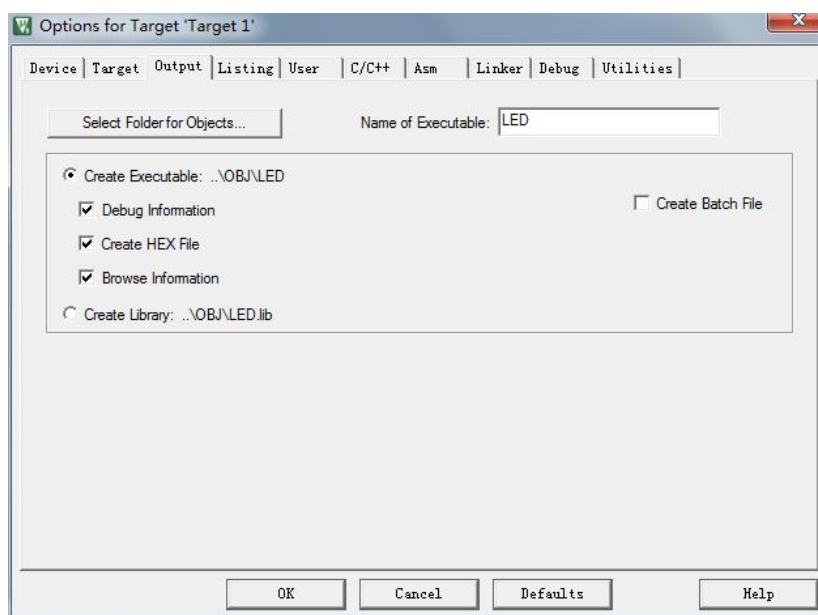


图 1.15 工程配置-文件输出

做完这些就可以在 main 函数中编写代码，在 main 函数中添加实验代码，然后进行下一步操作。

配置好的工程可以作为后续所有实验的模板，可以保存后备用。

2.3 利用库函数配置 GPIO 流程

GPIO 相关的函数和定义分布在固件库文件 stm32f4xx_gpio.c 和头文件 stm32f4xx_gpio.h 文件中（头文件在 stm32f4xx.h 中包含）。

在固件库开发中，操作配置寄存器初始化 GPIO 是通过

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
```

这个初始化函数完成的。这个函数有两个参数，第一个参数是用来指定需要初始化的 GPIO 对应的 GPIO 组，取值范围为 GPIOA~GPIOK；第二个参数为初始化参数结构体指针，结构体类型为 GPIO_InitTypeDef。下面我们看看这个结构体的定义，在 FWLIB/src 中可以找到 stm32f4xx_gpio.c 文件，定位到 GPIO_Init 函数体处，双击入口参数类型

GPIO_InitTypeDef 后右键选择“Go to definition of ”可以查看结构体的定义（或者在 main.c 中找 GPIO_InitTypeDef，选中后 F12 快捷键找到该结构体的定义）

该定义在 stm32f4xx_gpio.h 中。

typedef struct

```
{
    uint32_t GPIO_Pin;
    GPIO_Mode_TypeDef GPIO_Mode;
    GPIO_Speed_TypeDef GPIO_Speed;
    GPIO_OType_TypeDef GPIO_OType;
    GPIO_PuPd_TypeDef GPIO_PuPd;
}GPIO_InitTypeDef;
```

下面通过一个 GPIO 初始化实例来理解成员变量的含义，通过初始化结构体初始化 GPIO 的常用格式是：

```
GPIO_InitTypeDef  GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9//GPIO Pin9
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOF, &GPIO_InitStructure);//初始化 GPIO
```

上面代码的意思是设置 GPIOF 的第 9 个引脚设为推挽输出模式，同时速度为 100M，上拉。从上面初始化代码可以看出，结构体 GPIO_InitStructure 的第一个成员变量 GPIO_Pin 用来设置是要初始化哪个或者哪些 IO 口；第二个成员变量 GPIO_Mode 是用来设置对应 IO 端口的输出输入端口模式，这个值实际就是配置的 MODER 寄存器的值，只需要选择对应的值即可：

typedef enum

```
{
    GPIO_Mode_IN    = 0x00, /*!< GPIO Input Mode */
    GPIO_Mode_OUT   = 0x01, /*!< GPIO Output Mode */
    GPIO_Mode_AF    = 0x02, /*!< GPIO Alternate function Mode */
    GPIO_Mode_AN    = 0x03  /*!< GPIO Analog Mode */
}GPIO_Mode_TypeDef;
```

GPIO_Mode_IN 是用来设置复位状态的输入，GPIO_Mode_OUT 是通用输出模式，GPIO_Mode_AF 是复用功能模式，GPIO_Mode_AN 是模拟输入模式。

第三个参数 GPIO_Speed 是 IO 口输出速度设置，有四个可选值，实际上这就是配置的 GPIO 对应的 OSPEEDR 寄存器的值，通过枚举类型定义：

```
typedef enum
{
    GPIO_Low_Speed      = 0x00, /*!< Low speed      */
    GPIO_Medium_Speed   = 0x01, /*!< Medium speed */
    GPIO_Fast_Speed      = 0x02, /*!< Fast speed    */
    GPIO_High_Speed      = 0x03  /*!< High speed    */
}GPIOSpeed_TypeDef;
/* Add legacy definition */
#define GPIO_Speed_2MHz    GPIO_Low_Speed
#define GPIO_Speed_25MHz   GPIO_Medium_Speed
#define GPIO_Speed_50MHz   GPIO_Fast_Speed
#define GPIO_Speed_100MHz  GPIO_High_Speed
```

第四个参数 GPIO_OType 是 GPIO 的输出类型设置，实际上是配置的 GPIO 的 OTYPER 寄存器的值，同样是通过枚举类型定义：

```
typedef enum
{
    GPIO_OType_PP = 0x00,
    GPIO_OType_OD = 0x01
}GPIOOType_TypeDef;
```

如果需要设置为输出推挽模式，那么选择值 GPIO_OType_PP，如果需要设置为输出开漏模式，那么设置值为 GPIO_OType_OD。

第五个参数 GPIO_PuPd 用来设置 IO 口的上下拉，实际上就是设置 GPIO 的 PUPDR 寄存器的值，同样通过一个枚举类型列出：

```
typedef enum
{
    GPIO_PuPd_NOPULL = 0x00,
    GPIO_PuPd_UP      = 0x01,
    GPIO_PuPd_DOWN    = 0x02
}GPIOPuPd_TypeDef;
```

这三个值的意思很好理解，GPIO_PuPd_NOPULL 为不使用上下拉，GPIO_PuPd_UP 为上拉，GPIO_PuPd_DOWN 为下拉，我们根据我们需要设置相应的值即可。

以上操作是利用库函数配置 GPIO 的参数配置寄存器，还可以利用库函数配置 GPIO 输入输出电平控制相关的寄存器（涉及到的寄存器是 ODR、IDR、BSRR、AFRH 和 AFRL）。

流水灯工程中使用的函数 void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction BitVal)，功能是对指定引脚进行置位或者复位的操作。参数 GPIOx 是用来指定需要初始化的 GPIO 对应的 GPIO 组，取值范围为 GPIOA~GPIOK；参数 GPIO_Pin 用来指定引脚；参数 BitVal 是一个枚举类型 BitAction，定义如下：

```
typedef enum
{
    Bit_RESET = 0, //复位
    Bit_SET //置位
}BitAction;
```

另有 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)，用于对指定引脚复位（即置低电平 0），与其对应的函数是 void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)，用于对指定引脚置位（即置高电平 1）。

GPIO_WriteBit 与 GPIO_SetBits 的区别在于前者是对单个 IO 口置 0 或 1，例如：GPIO_WriteBit(GPIOA,GPIO_Pin_8,0)；而后者可以同时多个 IO 口置 1，例如：GPIO_SetBits(GPIOD,GPIO_Pin_0 | GPIO_Pin_5 | GPIO_Pin_6)；对于单个引脚的操作，两个函数没有区别。

GPIO 配置操作步骤可以总结为：

（1）使能 IO 口时钟，调用函数为 void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState)；

例：RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG|RCC_AHB1Periph_GPIOD, ENABLE); //使能 GPIOG 和 GPIOD 引脚的外设时钟；

（2）初始化 IO 参数，调用函数 GPIO_Init()；

（3）函数控制 IO 实现输入输出；

2.4 利用 GPIO 实现 LCD 控制及字符显示

通过 GPIO 口控制 LCD 并发送字符进行显示，可以初步掌握 STM32F407 与 LCD 的连接及通信协议，发送命令字控制 LCD 进行字符显示。

2.4.1 LCD 液晶显示简介

LCD 是指液晶显示器，是 Liquid Crystal Display 的简称。LCD 的构造是在两片平行玻璃基板当中放置液晶盒，下基板玻璃上设置 TFT（薄膜晶体管），上基板玻璃上设置彩色滤光片，通过 TFT 上的信号与电压改变来控制液晶分子的转动方向，从而达到控制每个像素点偏振光出射与否达到显示的目的。

2.4.2 LCD12864 引脚功能与接口时序

LCD12864 液晶显示屏是分辨率为 128（列） \times 64（行）点阵型液晶显示器。中文汉字图形点阵液晶显示模块，可显示汉字及图形，内置 8192 个中文汉字（16X16 点阵）、128 个字符（8X16 点阵），可以显示 8 \times 4 行 16 \times 16 点阵的汉字。液晶模块不仅可以显示字符、数字，还可以显示各种图形、曲线以及汉字，并且可实现屏幕上下左右滚动、动画功等，其原理是控制 LCD12864 点阵中的点的亮暗，亮和暗的点阵按一定规律可以组成汉字，组成一幅图形和曲线等（一个字节 8 个点）。LCD12864 与 MCU 有串行和并行两种通讯方式，本实验平台的硬件连接为串行连接（PSB 引脚接低电平则为串行方式）。

LCD12864 硬件连接如图 1.16 所示：

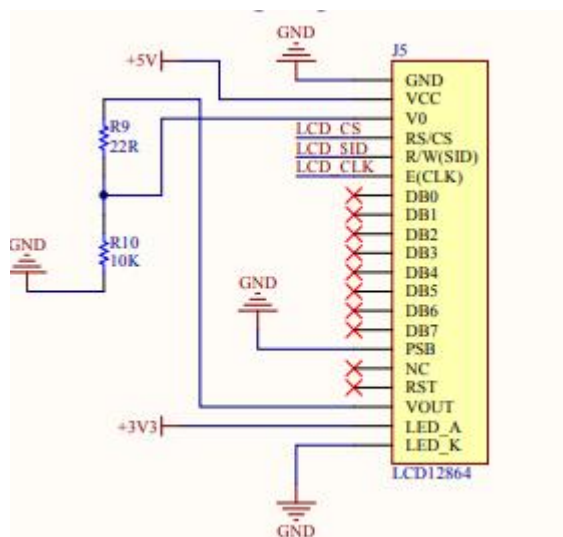
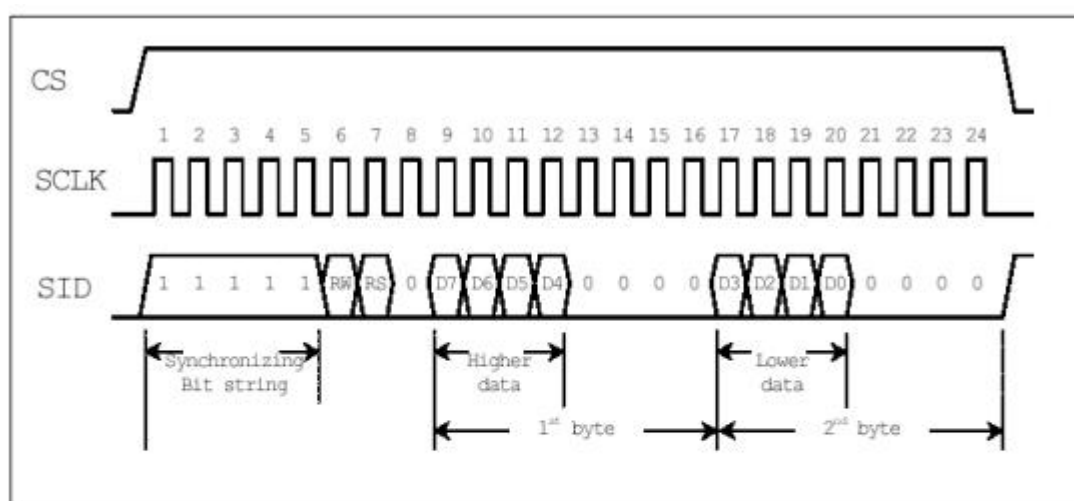


图 1.16 LCD12864 硬件连线图

LCD12864 各引脚功能图如下表所示：

引脚号	引脚名称	方向	功能说明
1	GND	-	模块的电源地
2	VCC	-	模块的电源正端
3	V0	-	LCD 驱动电压输入端
4	RS(CS)	H/L	数据选择信号/并行的指令；串行的片选信号
5	R/W(SID)	H/L	并行的读写选择信号；串行的数据口
6	E(CLK)	H/L	并行的使能信号；串行的同步时钟
7	DB0	H/L	数据 0
8	DB1	H/L	数据 1
9	DB2	H/L	数据 2
10	DB3	H/L	数据 3
11	DB4	H/L	数据 4
12	DB5	H/L	数据 5
13	DB6	H/L	数据 6
14	DB7	H/L	数据 7
15	PSB	H/L	并/串行接口选择：H-并行；L-串行
16	NC		空脚
17	/RST	H/L	复位 低电平有效
18	VOOUT		倍压输出脚（VDD=+3.3V 有效）
19	LED_A	(LED+5V)	背光源正极
20	LED_K	(LED-OV)	背光源负极

LCD12864 串行接口时序如图 1.17 所示：



串行数据传送共分三个字节完成：

第一字节：串口控制—格式 11111ABC

A 为数据传送方向控制：H 表示数据从 LCD 到 MCU，L 表示数据从 MCU 到 LCD

B 为数据类型选择：H 表示数据是显示数据，L 表示数据是控制指令

C 固定为 0

第二字节：(并行)8 位数据的高 4 位—格式 DDDD0000

第三字节：(并行)8 位数据的低 4 位—格式 0000DDDD

串行接口时序参数：(测试条件：T=25℃ VDD=4.5V)

图 1.17 LCD12864 串行接口时序

由时序图 1.17，串行模式传输过程总结如下：

1. CS 片选一直为高电平，期间 LCD 才可以接受数据或指令

2. MCU 要给出数据传输起始位，这里是以 5 个连续的“1”作数据起始位，如模块接收到连续的 5 个“1”，则内部传输被重置并且串行传输将被同步

3. 紧接的两个位指定传输方向（RW 用于选择数据的传输方向，1 是读数据，0 是写数据）以及传输性质（RS，用于选择内部数据寄存器或指令寄存器，0 是命令寄存器，1 是数据寄存器），最后的第 8 位固定为“0”。到此第一个字节 / 数据传输起始位发送完成

4. 在接收到起始位及“RW”和“RS”的第 1 个字节后，之后便开始传输指令或者数据，在传输过程中会进行拆分处理，该字节将被分为 2 个字节来传输或接收

5. 发送的数据或指令的高 4 位，被放在发送的第 2 个字节串行数据的高 4 位，其低 4 位则置为“0”；数据或指令的低 4 位被放在第 3 个字节的高 4 位，其低 4 位也置为“0”，如此完成一个字节指令或数据的传送

6. 完成一个字节数据的发送大约需要 24 个时钟周期，因为 1 个字节实际是发送了 3 个字节

7. 只有在时钟线 SCLK 拉低时，数据线 SID 上的数据才允许变化，在时钟线 SCLK 高电平时，SID 上的数据必须保持稳定(不能变化)

这点与 IIC 是相同的

例：需要发送的数据为“A”对应 16 进制 0x41，对应二进制 0100 0001

那么发送的顺序就是：

1. 先发送 0xFA (1111 010)；五个 1，RW=0，RS=1

2. 发送 0100 0000；高四位为“A”对应的高四位，低 4 位补 0

3. 发送 0001 0000；高四位为“A”对应的低四位 低 4 位补 0

到此一个字节发送完成。

所以写指令之前，必须先发送 1111 000 （即 0xF8）

写数据之前，必须先发送 1111 010 （即 0xFA）

2.4.3 LCD 显示原理

LCD 的字符显示实际是控制写入字符代码的缓存器 DDRAM，只要将要显示的中文字符编码或其他字符编码写入 DDRAM(显示数据)，也就是串行模式下发送一个字节数据，硬件将依照编码自动从 CGROM (2M 中文字型 ROM)、HCGROM (16K ASCII 码 ROM)

CGRAM（自定义字形 RAM）三种字形中自动辨别选择对应的是那种字形的哪个字符/汉字编码，再将要显示的字符/汉字编码显示在屏幕上。也就是字符显示是通过将字符显示编码写入字符显示 RAM（DDRAM）实现的。

模块内部的 RAM 提供 64×16 的显示空间，最多可以显示 4 行 8 字（32 个汉字）或 64 个 ASCII 码字符的显示。DDRAM 一共有 32 个字符显示区域，当然，字符显示的 RAM 的地址与 32 个字符显示区域有着一一对应的关系，字符显示时，DDRAM 地址与液晶屏的位置如下图：



通过写入不同的地址，就可以实现字符显示在不同的位置。

2.4.4 LCD 控制指令

1. 清屏指令

清除屏幕字符，也就是对整块屏幕写入空字符并且将光标移到开始位置，在使用清屏时，需要加上一定的延时等待液晶稳定。

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	1

2. 功能设定

DL = 0/1 : 4/8 位数据，使用 8 位，所以为 1；RE=1 时，使用扩充指令集，RE=0 时，使用基本指令集，我们正常使用基本指令集 所以 RE 需要为 0，一般配置为 0X30。

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	DL	X	RE	X	X

3. 读取 BF 忙标志和地址

读取忙碌标志（BF）可以确认内部动作是否完成，同时可以读出地址计数器（AC）的值，BF 标志提供内部工作状况，BF=1 表示模块内部正在进行操作，此时模块不接受指令和数据。BF=0 模块为准备接受状态，可以接受指令和数据。

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0

4. LCD 初始化流程

- (1) 在开电之后，首先要等待 40ms 以上，等待液晶自检，使 LCD 系统复位完成
- (2) 功能的设定：选择基本指令集或者扩充指令集，随后延时等待 100us 以上
- (3) 功能设定 2：选择 8bit 数据流或者 4bit 数据流 随后延时 37us 以上
- (4) 开关显示：是否打开显示开关，随后延时 100us 以上
- (5) 清屏：清空 RAM 并初始化光标位置，随后延时 10us 以上
- (6) 进入模式选择：也就是设定游标相对于上一个位置的移位，默认为地址自动+1

5. LCD 画图流程

- (1) 切换到扩充指令
- (2) 关闭绘图显示功能
- (3) 先将垂直的坐标(Y)写入 CGRAM 地址
- (4) 再将水平的位元组坐标(X)写入 CGRAM 地址
- (5) 将高位字节 D15—D8 写入 RAM 中
- (6) 将低位字节 D7—D0 写入到 RAM 中，重复 3-6 步，完成图片各个部分的写入 先写上半屏，再写下半屏
- (7) 打开绘图显示功能
- (8) 切换回基本指令

也就是先打开 CGRAM(自定义字形 RAM) 然后把定义好的图片编码写入 CGRAM 然后再对 DDRAM 正常写入该图片编码，这时硬件将依照编码从 CGRAM(自定义字形 RAM) 读取之前写入的图片编码，然后显示该图片。

注意：在显示一幅图片之后，要加上 2s 左右延时，否则不会有图片显示。

6. LCD 显示注意事项

①欲在某一个位置显示中文字符时，应先设定显示字符位置，即先设定显示地址，再写入中文字符编码。

②显示 ASCII 字符过程与显示中文字符过程相同。不过在显示连续字符时，只须设定一次显示地址，由模块自动对地址加 1 指向下一个字符位置，否则，显示的字符中将会有一个空 ASCII 字符位置。

③当字符编码为 2 字节时，应先写入高位字节，再写入低位字节。

④模块在接收指令前，向处理器必须先确认模块内部处于非忙状态，即读取 BF 标志时 BF 需为“0”，方可接受新的指令。如果在送出一个指令前不检查 BF 标志，则在前一个指令和这个指令中间必须延迟一段较长的时间，即等待前一个指令确定执行完成。指令执行的时间请参考指令表中的指令执行时间说明。

⑤“RE”为基本指令集与扩充指令集的选择控制位。当变更“RE”后，以后的指令集将维持在最后的状态，除非再次变更“RE”位，否则使用相同指令集时，无需每次均重设“RE”位。

完整指令集见《J12864 中文液晶使用说明》

2.4.5 LCD 控制实验步骤

1. LCD 液晶模块与 GPIO 引脚对应关系表：

LCD-CS	PG1
LCD-SID	PF15
LCD-CLK	PF14

2. 如上节中工程所示，在 main.c 中如果包含头文件 `stm32f4xx.h` 后，可以将外设的初始化、设置等操作在 main.c 中完成。但是如果外设增加，多个外设的初始化及设置都在主程序中完成会使主程序变得臃肿、结构凌乱。所以，在实际操作中我们一般会在库目录中再添加外设子目录，子目录名称可以任选如取名 `HARDWARE`，如图 1.18 所示：

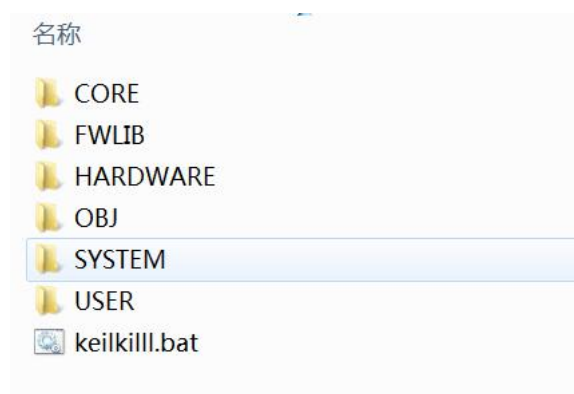


图 1.18 通用工程目录结构

在子目录 HARDWARE 中可以按照不同外设再创建不同的子目录，本次实验中外设为 LCD，所以在 HARDWARE 中新建子目录 12864 如图 1.19 所示，子目录中文件结构如图 1.20 所示，存放 12864 的头文件和源文件。



图 1.19 外设子目录结构

图 1.20 外设子目录文件结构



图 1.20 外设子目录包含文件

3. 添加新文件后需进行工程配置

工程中添加新的外设文件，如图 1.21 所示

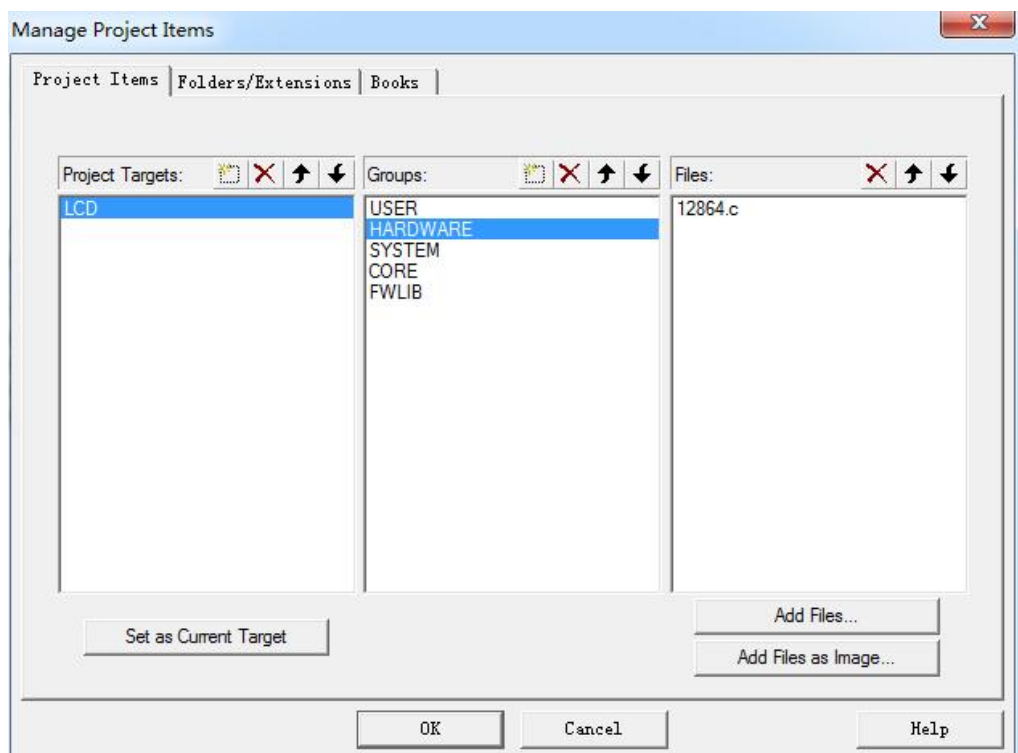


图 1.21 工程中添加外设源文件

在路径中添加外设头文件路径，如图 1.22 所示



图 1.22 工程中添加外设头文件路径

4. 添加代码实现

初始化液晶模块与 MUC 连接的三个引脚，通过控制指令初始化 LCD。添加 12864.c 和 12864.h 对控制主函数进行编写。

1. LCD 显示函数代码

```
//12864.c
#include "12864.h"
#include "delay.h"

void LCD_GPIO_Init()
{
    ??? //GPIO 初始化代码
    CS=1;
    SID=1;
    SCLK=1;
}

void SendByte(u8 byte)
{
    u8 i;
```

```

    for(i = 0;i < 8;i++)
    {
        if((byte << i) & 0x80) //0x80(1000 0000)
        {

            SID = 1;
        }
        else
        {
            SID = 0;
        }

        SCLK = 0;
        delay_us(5);
        SCLK = 1;
    }
}

void Lcd_WriteCmd(u8 Cmd )
{
    delay_ms(1);
    SendByte(WRITE_CMD);           //11111,RW(0),RS(0),0
    SendByte(0xf0&Cmd);
    SendByte(Cmd<<4);
}

void Lcd_WriteData(u8 Dat )
{
    delay_ms(1);
    SendByte(WRITE_DAT);
    SendByte(0xf0&Dat);
    SendByte(Dat<<4);
}

```



```

void LCD_Init(void)
{
    delay_ms(50);
    Lcd_WriteCmd(0x30);

    delay_ms(1);
    Lcd_WriteCmd(0x30);

    delay_ms(1);
    Lcd_WriteCmd(0x0c);

    delay_ms(1);
    Lcd_WriteCmd(0x01);

    delay_ms(30);
    Lcd_WriteCmd(0x06);
}

void LCD_Display_Words(uint8_t x,uint8_t y,uint8_t*str)
{
    Lcd_WriteCmd(LCD_addr[x][y]);
    while(*str>0)
    {
        Lcd_WriteData(*str);
        str++;
    }
}

void LCD_Display_Picture(uint8_t *img)
{
    uint8_t x,y,i;
    Lcd_WriteCmd(0x34);
    Lcd_WriteCmd(0x34);
    for(i = 0; i < 2; i++)
    {

```

```

        for(y=0;y<32;y++)
        {
            for(x=0;x<8;x++)
            {
                Lcd_WriteCmd(0x80 + y);
                Lcd_WriteCmd(0x80 + x+i*0x08);

                Lcd_WriteData(*img ++);

                Lcd_WriteData(*img ++);
            }
        }
    }
    Lcd_WriteCmd(0x36);
    Lcd_WriteCmd(0x30);
}

```

```

void LCD_Clear(void)
{
    Lcd_WriteCmd(0x01);
    delay_ms(2);
}

```

//12864.h 头文件

```

#ifndef __lcd12864_H_
#define __lcd12864_H_

```

```

#include "sys.h"

```

```

static u8 LCD_addr[4][8]={
    {0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87},
    {0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97},
    {0x88, 0x89, 0x8A, 0x8B, 0x8C, 0x8D, 0x8E, 0x8F},
    {0x98, 0x99, 0x9A, 0x9B, 0x9C, 0x9D, 0x9E, 0x9F}
}

```

```
};
```

```
#define WRITE_CMD 0xF8//写命令
```

```
#define WRITE_DAT 0xFA//写数据
```

```
#define SID PFout(15)
```

```
#define SCLK PFout(14)
```

```
#define CS PGout(1)
```

```
??? //相关函数声明
```

```
#endif
```

2. 主程序中代码

```
#include "sys.h"
```

```
#include "delay.h"
```

```
#include "usart.h"
```

```
#include "12864.h"
```

```
int main()
```

```
{
```

```
    delay_init(168);
```

```
    LCD_GPIO_Init();
```

```
    LCD_Init();
```

```
    while(1)
```

```
    {
```

```
        //LCD_Display_Picture(ImageData);
```

```
        //delay_ms(2000);
```

```
        LCD_Display_Words(0,0,"理");
```

```

        LCD_Display_Words(1,0,"实");
        LCD_Display_Words(2,0,"交");
        LCD_Display_Words(3,0,"融");
    }
}

```

附录：图形数据，可添加至 12864.h 中

```

uint8_t ImageData[]={
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x1E,0x00,0x00,0x00,0x00,0x00,0x07,0xF0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x7F,0x80,0x00,0x00,0x00,0x00,0x18,0x0C,0x00,0x00,0x01,0x00,0x00,0x00,0x00,
0x01,0xFF,0x80,0x00,0x00,0x00,0x00,0x20,0x00,0x00,0x00,0x13,0x10,0x03,0xFE,0x00,
0x03,0xFF,0xC0,0x00,0x00,0x00,0x00,0x58,0x00,0x00,0x00,0x3F,0x30,0x1F,0xFF,0xC0,
0x03,0xFF,0xE0,0x00,0x00,0x00,0x00,0x8C,0x03,0xF0,0x00,0x7F,0xE0,0x7C,0x01,0xE0,
0x03,0xFF,0xF0,0x00,0x00,0x00,0x01,0x36,0x06,0xC0,0x00,0x5F,0xC0,0xFF,0xFC,0x60,
0x01,0xFF,0xF0,0x00,0x00,0x00,0x02,0x1B,0x0F,0x80,0x00,0xFF,0x01,0xFE,0x0F,0x30,
0x00,0xEF,0xF0,0x00,0x00,0x00,0x02,0x6D,0x9F,0x00,0x00,0x3E,0x03,0xFF,0xF1,0x90,
0x00,0xFF,0xF8,0x00,0x00,0x00,0x04,0x36,0xFE,0x00,0x01,0xFF,0x07,0xFF,0xFC,0x90,
0x00,0xEF,0xFF,0xFF,0x80,0x00,0x04,0xDB,0x7E,0x00,0x03,0xFF,0x87,0xFF,0xFC,0xD
0,0x00,0x0F,0xFF,0xFF,0xC0,0x00,0x04,0x6D,0xFC,0x00,0x07,0xFF,0x8F,0xFF,0xFE,0x
50,0x00,0x0F,0xFF,0xFF,0xE0,0x00,0x04,0x36,0xFC,0x10,0x07,0xFF,0x8F,0xFF,0xFE,0x
90,0x00,0x0F,0xFF,0xFF,0xE0,0x00,0x04,0x1B,0xF8,0x10,0x07,0xFF,0xCF,0xFF,0xFE,0x
80,0x00,0x0F,0xFF,0xFF,0xF0,0x00,0x04,0x0F,0xF8,0x10,0x07,0xFF,0xFF,0xFF,0xFA,0x
00,0x00,0x07,0xFF,0xFF,0xF0,0x00,0x04,0x07,0xF0,0x10,0x07,0xFF,0xFF,0xFF,0xFA,0x
00,0x00,0xFF,0xFF,0xFF,0xF8,0x00,0x02,0x03,0xF0,0x20,0x07,0xFF,0xFF,0xFF,0xBA,0x
00,0x00,0xFD,0xFF,0xFF,0xFC,0x00,0x02,0x03,0xF0,0x20,0x03,0xFF,0xFF,0xDF,0xB8,0
x00,0x00,0xC1,0xC0,0x3F,0xFC,0x00,0x01,0x01,0xE0,0x40,0x00,0xFF,0xFF,0xDF,0xB0,
0x00,0x00,0x81,0xC0,0x3F,0xCE,0x00,0x00,0x81,0xE0,0x80,0x00,0x7F,0xFF,0xDF,0xA0,
0x00,0x00,0x81,0x80,0x1D,0xCF,0x00,0x00,0x41,0xE1,0x00,0x00,0x3F,0xFF,0x9B,0x00,
0x00,0x01,0x83,0x80,0x1F,0xC7,0x80,0x00,0x21,0xE2,0x00,0x00,0x1F,0xFD,0xB6,0x00,
0x00,0x01,0xC3,0x00,0x0E,0xE6,0x80,0x00,0x19,0xEC,0x00,0x00,0x07,0xFE,0x20,0x00,
0x00,0x00,0xC3,0x00,0x07,0x67,0x40,0x00,0x07,0xF0,0x00,0x00,0x03,0x3E,0x00,0x00,
0x00,0x00,0x02,0x00,0x03,0xE7,0xA0,0x00,0x00,0x00,0x00,0x00,0x02,0x8E,0x00,0x00,
0x00,0x00,0x06,0x00,0x03,0x83,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x03,0x00,0x00,0

```

```
x00,0x00,0x06,0x00,0x07,0x03,0x00,0x77,0x46,0x74,0x24,0x80,0x06,0x04,0x00,0x00,0x
00,0x00,0x1C,0x00,0x06,0x00,0x00,0x55,0x45,0x54,0x57,0x80,0x00,0x00,0x00,0x00,0x0
0,0x00,0x1C,0x00,0x0E,0x00,0x00,0x45,0x45,0x74,0x57,0x80,0x08,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x1C,0x00,0x00,0x55,0x45,0x44,0x74,0x80,0xF0,0x00,0x00,0x00,0x00,0
x00,0x00,0x00,0x38,0x00,0x00,0x77,0x76,0x47,0x54,0x80,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};
```

2.5 实验练习与习题

1. 熟悉示例 LED 工程架构及工程配置；修改源码，改变流水灯的点亮方向、点亮模式等。
2. 在 LED 示例工程中，改用位带方式控制 IO 输出实现 LED 亮灭控制，并记录和分析位带 IO 地址映射范围和实现方法。

提示：“位带”定义在“SYSTEM/sys/sys.h”中，并参考课本中关于“位带”映射的描述

3. 完成 LCD 工程配置及代码实现, 控制 LCD 显示不同字符及图像显示。

