**STM32F407 CORTEX-M4 ARM**

**实验指导手册**

ST STM32F407 CORTEX-M4是ST公司ARM Cortex-M4内核的MCU增强型处理器STM32F407ZGT6为核心单元，1MB的FLASH，配以独立的按键、开关、矩阵键盘等用户输入设备，和数码管、LED、LCD屏等显示设备，以及RS232/

RS422/I2C/SPI等通信总线，可在该平台上做各种STM32F407教学实验以及针对STM32F407的板载和外设实验。实验箱上主要由核心处理芯片、各功能模块处理芯片、下载电路模块、通信模块的接口、通信信号的测试点、用户输入拨码开关和按键开关、用户的矩阵键盘和SD卡接口、用户显示模块等组成。通过该实验箱，使得学习者可以在该实验箱上完整系统地学习STM32F407原理的理论和实践内容。

**硬件参数：**

* 处理器芯片：STM32F407ZGT6
* 下载和在线调试接口：
* 板载J-Link下载与在线调试电路
* 外置J-Link下载与在线调试JTAG接口
* 板载时钟：8MHz贴片晶体振荡器
* 应用电路：
* 光敏器光照ADC采集电路
* 变阻器电压ADC采集电路
* 2路独立SMA通用模拟信号输入电路
* 2路独立SMA通用模拟信号输出电路
* 基于I2C的24LC02存储电路
* 基于SPI的93LC46存储电路
* 基于SDIO的SD卡存储电路
* 基于SPI的DS1302日历时钟电路
* 基于SPI的VS1003音频电路
* 基于NEC的1838T红外接收电路
* 基于1-Wire的DS18B20数字温度测量电路
* 基于PWM的呼吸灯驱动电路
* 基于PWM的蜂鸣器驱动电路
* 通信总线接口：
* 1个DB9-FEMALE RS232串行通信总线接口
* 2个DB9-MALE RS232串行通信总线接口
* 五线接线端子式RS422串行通信总线接口
* 两线接线端子式CAN总线通讯接口
* 1个USB DEVICE接口
* 1个USB HOST接口
* 通用扩展接口
* 2组8个/组IO的EMOD电路扩展接口

# 集成开发环境的安装配置及ARM程序调试

## Keil uVision5应用软件的安装

Keil是德国知名软件公司Keil（现已并入ARM公司）开发的微控制器软件开发平台，是目前ARM内核单片机开发的主流工具。Keil提供了包括C编译器、连接器、库管理和一个功能强大的仿真调试器在内的完整开发方案，通过一个集成开发环境（μVision）将这些功能组合在一起。

2011年3月ARM公司发布最新集成开发环境RealView MDK开发工具中集成了最新版本的Keil μVision4，其编译器、调试工具实现与ARM器件的最完美匹配。

2013年10月，Keil正式发布了Keil μVision5 IDE。KEIL MDK5这一次重大改版，更加方便我们使用。与之前的版本相比，最大的区别在于器件(Software Packs)与编译器(MDK core)分离。也就是说，我们安装好编译器(mdk\_5xx.exe)以后，编译器里面没有任何器件。如果我们要对STM32进行开发，只需要再下载STM32的器件安装包(packs)即可

KEIL5完美兼容KEIL4，安装好以后不用任何设置即可使用以前用KEIL4做的工程。本实验教程所用的为Keil uVision5的升级版本MDK，安装过程略...。

**注：**程序安装完成后，还需要一个器件（pack installer）的安装过程，用来安装相应芯片的开发库和插件等，如图1.1.1所示：

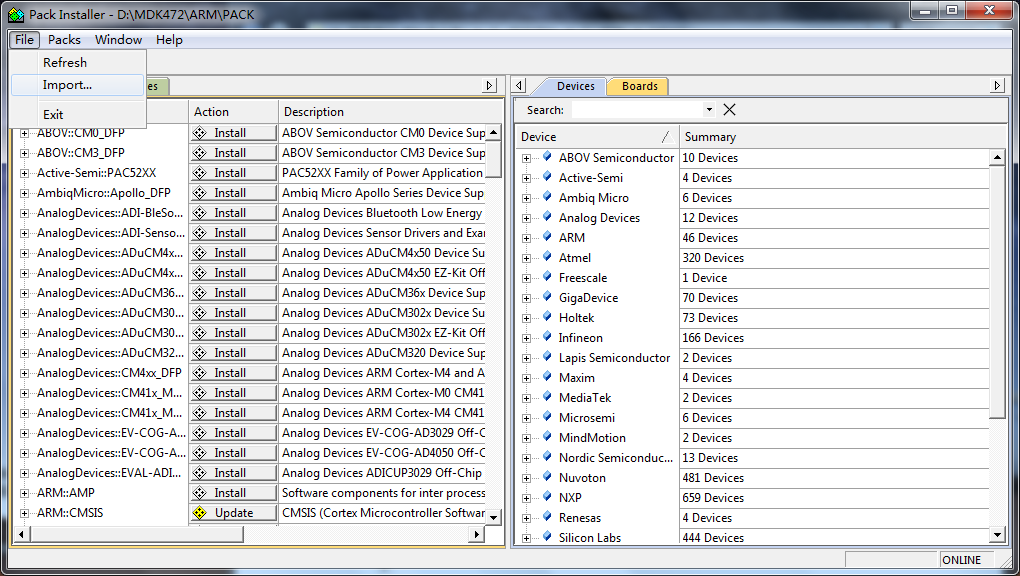


图1.1.1 Keil-MDK 器件库函数和插件安装界面

安装文件可以从网上下载也可以先下载到本地后再安装，然后点击File

-Import，在解压“解压Keil安装软件”后的文件夹中找到“Keil.STM32-

F4xx\_DFP.1.0.8”，如图1.1.2所示：

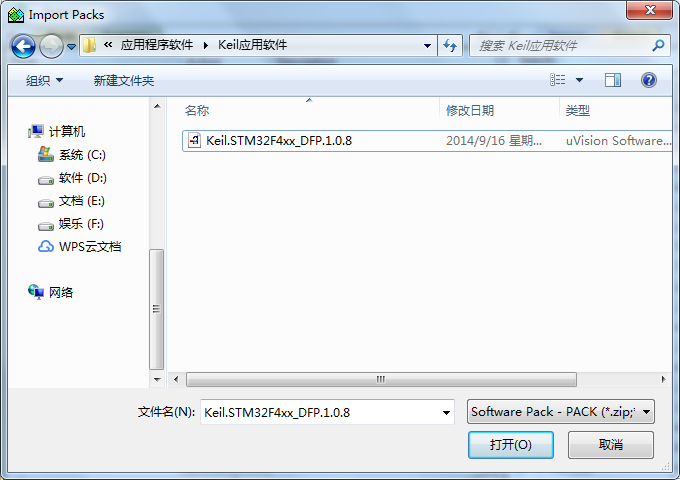


图1.1.2 Keil-MDK库文件导入界面

选择“Keil.STM32F4xx\_DFP.1.0.8”,点击“打开”，图1.1.3：

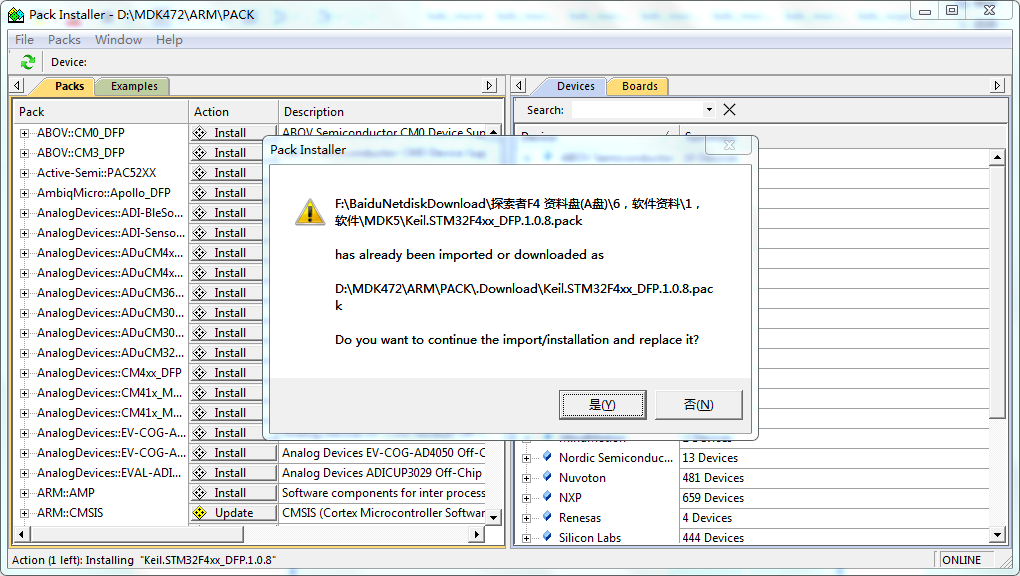


图1.1.3 Keil-MDK安装界面10

点击“是”，等待一段时间完成，至此集成开发环境安装完成。

## SSCOM V3.2应用软件的安装

SSCOM V3.2串口助手是一款通过电脑串口（包括USB口）收发数据并且显示的应用软件，一般用于电脑与嵌入式系统的通讯，借助于它来调试串口通讯或者系统的运行状态。也可以用于采集其他系统的数据，用于观察系统的运行情况。相对于Windows系统的超级终端，串口助手使用更方便和灵活，界面更友好。串口调试助手无需另行安装，直接打开就可以使用，找到实验箱光盘自带的软件安装文件夹，解压SSCOM V3.2压缩包，运行程序，界面如下：



图1.2.1串口助手界面

图中左下角部分是一个设置框，可以设置串口号，波特率、数据位、停止位，打开或关闭串口。上面文本框是串口接收区，PC机的RS232串口接收到的数据将会显示在这里，字符输入框是发送数据功能，输入所要发送的数据点发送，清除窗口为清除所显示的数据，显示方式选HEX显示则以16进制显示返回的数据。如果需要发送文件给设备，则可以点击【打开文件】，在弹出的选择框中选择对应的文件，然后再点击【发送文件】，完成发送。

使用时将串口线插入电脑，打开串口调试助手，这时候串口调试助手会自动检测已有的串口号，点击“打开串口”，就可以正常使用了，有数据来会显示在显示框。



## CH340G驱动软件的安装

找到实验箱光盘自带的软件安装文件夹，解压CH340G驱动软件压缩包，找到图标，双击图标进入安装界面，安装过程略...。

安装完成后，将 USB插入电脑中后，电脑会自动识别进行安装驱动，当提示安装完成后，进入设备管理器查看串口，如图，此时，驱动安装完成。

## USB驱动软件安装

找到实验箱光盘自带的软件安装文件夹，解压USB驱动软件压缩包，找到图标，双击图标进入安装界面，安装过程略...。

以上1.3和1.4安装后可以将USB口虚拟成串口用于和实验箱串口进行数据传输。

## 简单汇编新工程的建立及编译、链接、调试过程

MDK可以编写基于C和ARM 汇编的工程，下面我们先编写一个简单的汇编工程并进行编译、链接和调试（MDK为5.16版本）。

1. 打开Keil软件，进入软件，点击菜单栏上的“Project”→“New uVision Project…”。

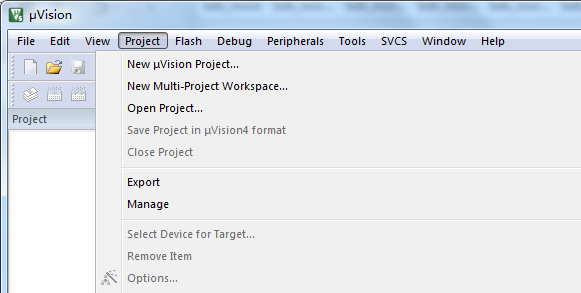


图1.6.1 Keil-MDK建立新工程

1. 设定保存工程路径和工程名（汇编工程比较简单，不需要设置库函数目录，可以直接将工程文件存在根目录下，如图1.6.2中即工程名为test2存在test目录中），扩展名为\*.uvproj或者\*.uvprojx。

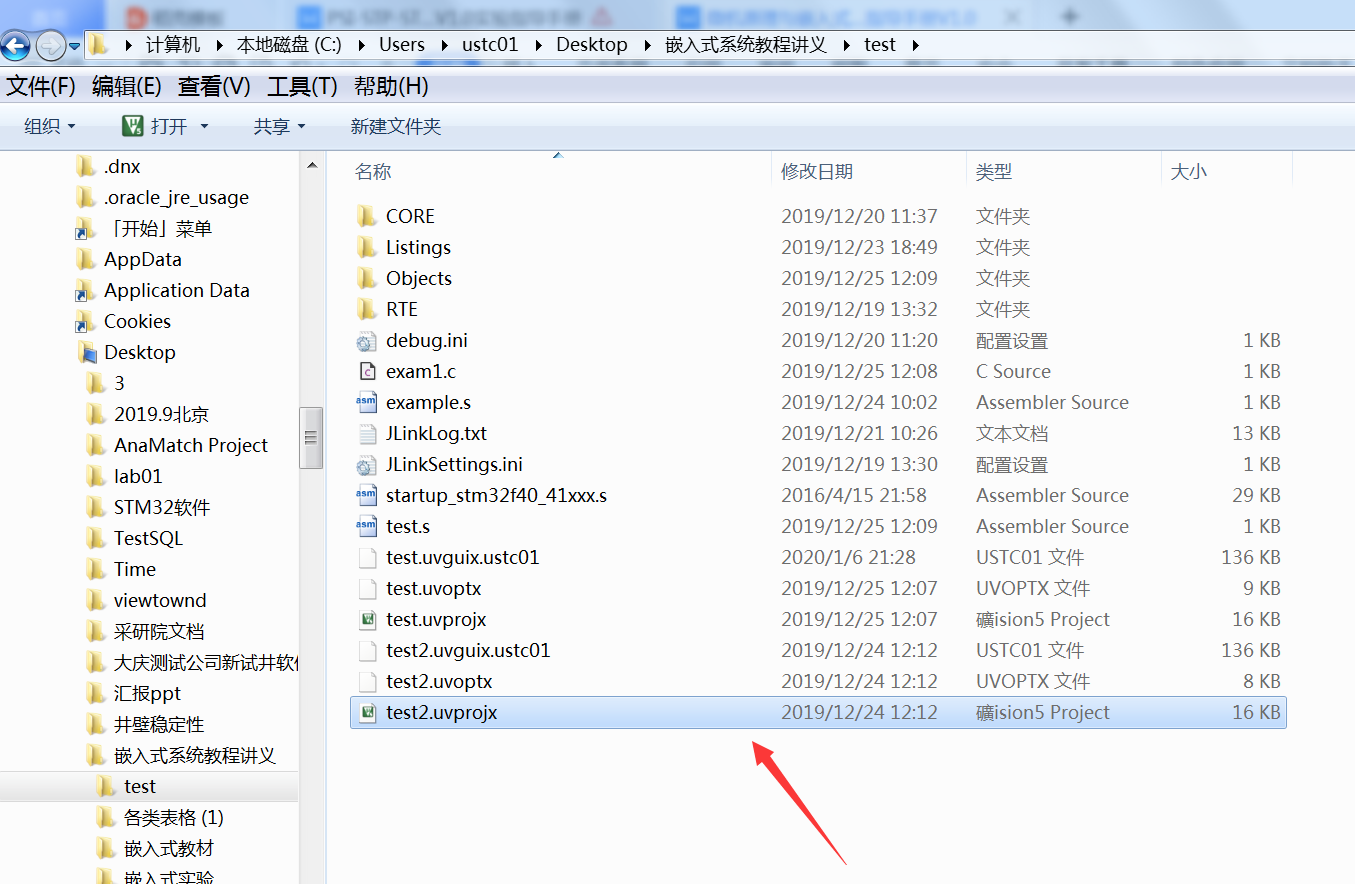


图1.6.2 新建工程保存文件夹和工程名设置

1. 选择所需要的单片机型号，本实验箱的核心处理器芯片是STM32F407ZGT6，在此我们选择STM公司的STM32F407ZG芯片。

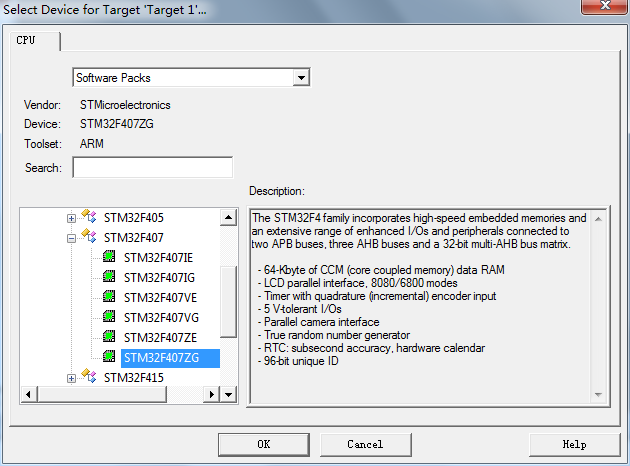


图1.6.3 选择处理器芯片型号

1. 点击 OK选定后，MDK会弹出Manage Run-Time Environment（管理运行环境）对话框，这是 MDK5 新增的一个功能，在这个界面，我们可以添加自己需要的组件，从而方便构建开发环境，其中最重要的一个功能是可以选择Device的Startup功能，勾选后系统会自动帮你把启动文件（后缀名为\*.s的汇编文件）拷贝到工程目录下，启动文件会做很多系统初始化的设置，包括指令指针初始化、堆栈初始化、各种变量的存放等待功能。

**注意：启动文件一般在最后会找C中的main函数做为入口点，所以汇编程序中一般不会选择系统启动，这也意味着这些初始化功能我们必须自己选择最简单模式实现。**

这里我们直接点击Cancel，这样一个新的工程就建立完成了。

1. 下一步需要为工程添加文件（因为工程文件\*.uvproj或者\*.uvprojx只包含工程结构信息，不包括源码，所以需要添加源码文件，汇编工程要添加汇编文件，C工程要添加main.c文件等），因为这个项目是一个简单的汇编工程，所以只需要添加一个汇编文件即可，File->New

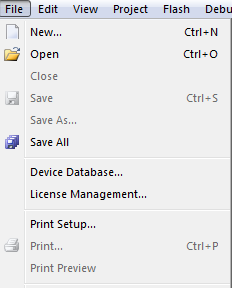


图1.6.4 工程添加新文件

添加完毕后将文件另存为相应文件名并添加扩展名，我们这里用example1.s，添加后如图1.6.5所示，图左边标注可以看出example.s自动加入工程结构中。

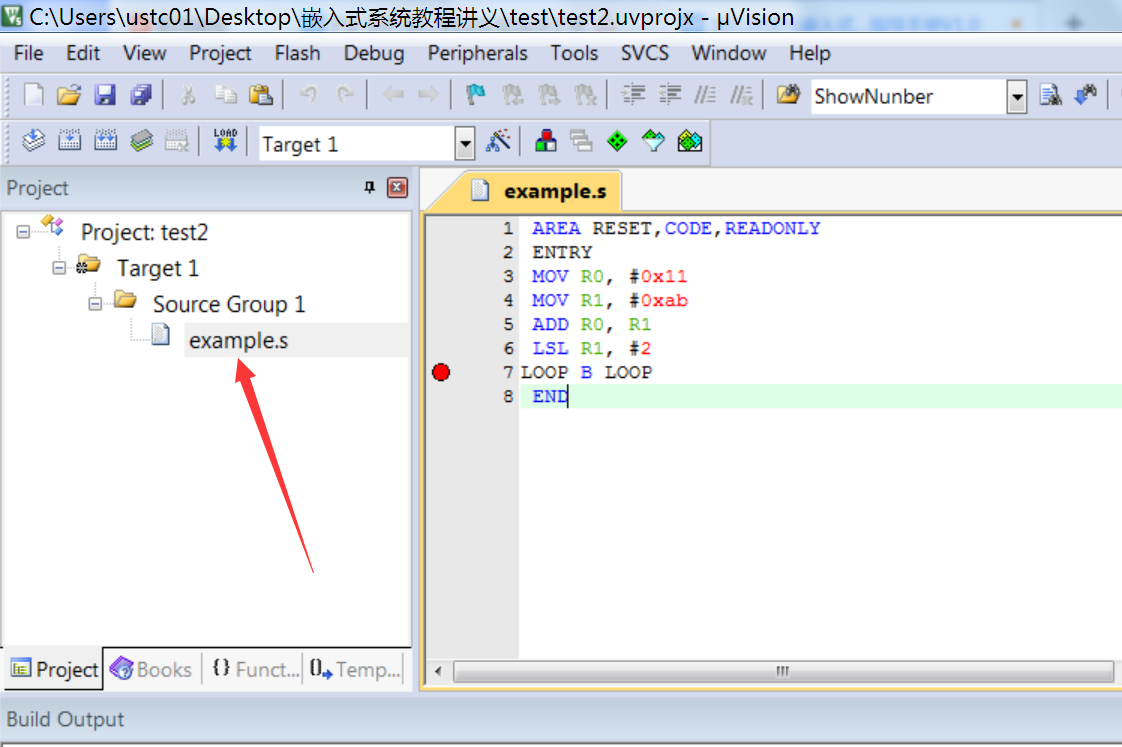


图1.6.5 添加的汇编源文件并添加到工程结构中

同时这里需要对工程进行一些设置，Options for Target重要的几个设置如下图。

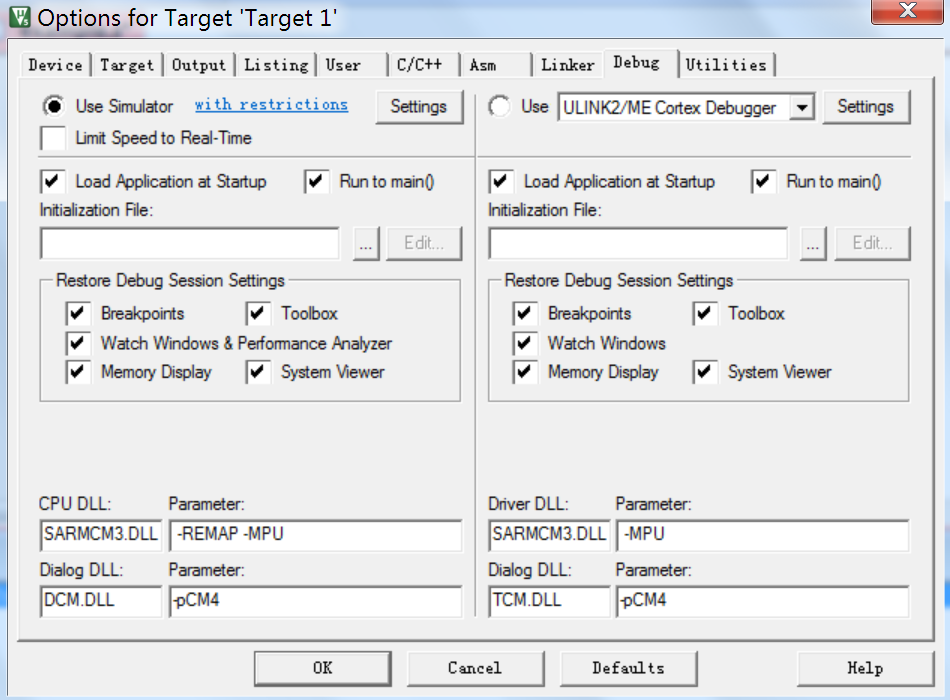
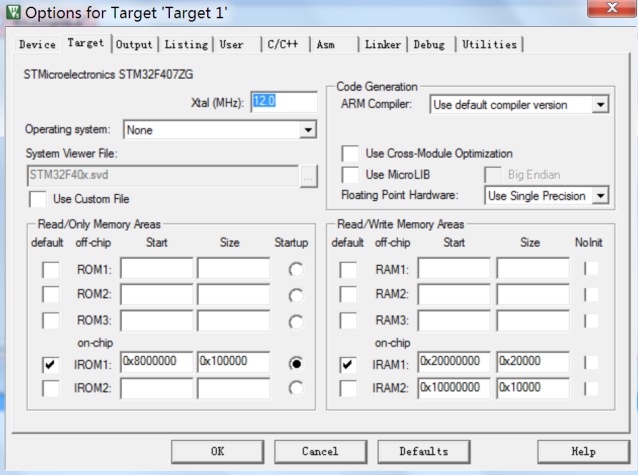


图1.6.6 汇编工程设置

这里设置的是IROM和IRAM的起始地址和大小，也就是系统的代码区和数据区，这和我们开始选择的芯片型号相关，本工程IROM从0x08000000开始，IRAM从0x20000000开始，一般选定了芯片起始地址和大小就是默认值。另一个设置是Debug选型里的仿真类型，我们用软件仿真Use Simulator，另一个选项是仿真器仿真。

1. 至此设置完毕，可以在example1.s中开始代码编写。

AREA RESET,CODE,READONLY

ENTRY

MOV R0, #0x11

MOV R1, #0xab

ADD R0, R1

LSL R1, #2

LOOP B LOOP

END

**注意：程序中伪指令AREA ENTRY END以及各条指令不能左边顶格书写，而标号LOOP必须顶格书写，否则编译会报错。**

这个程序代码实现简单的寄存器赋值功能，并对寄存器内容进行左移操作，最后进入死循环状态。代码编辑完毕后点击进行编译，通过后即可进行调试。

选择Debug->Start/Stop Debug Session，如图1.6.7所示。

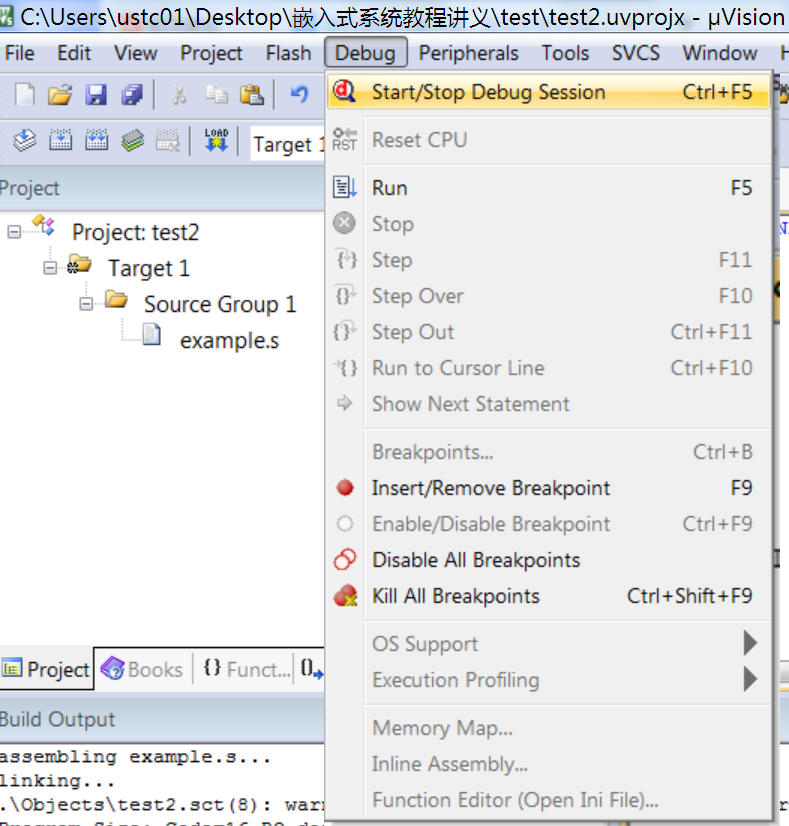


图1.6.7 程序进入调试模式菜单选项

进入调试状态后如图1.6.8所示，包括寄存器状态、反汇编、代码区、内存观察等窗口。

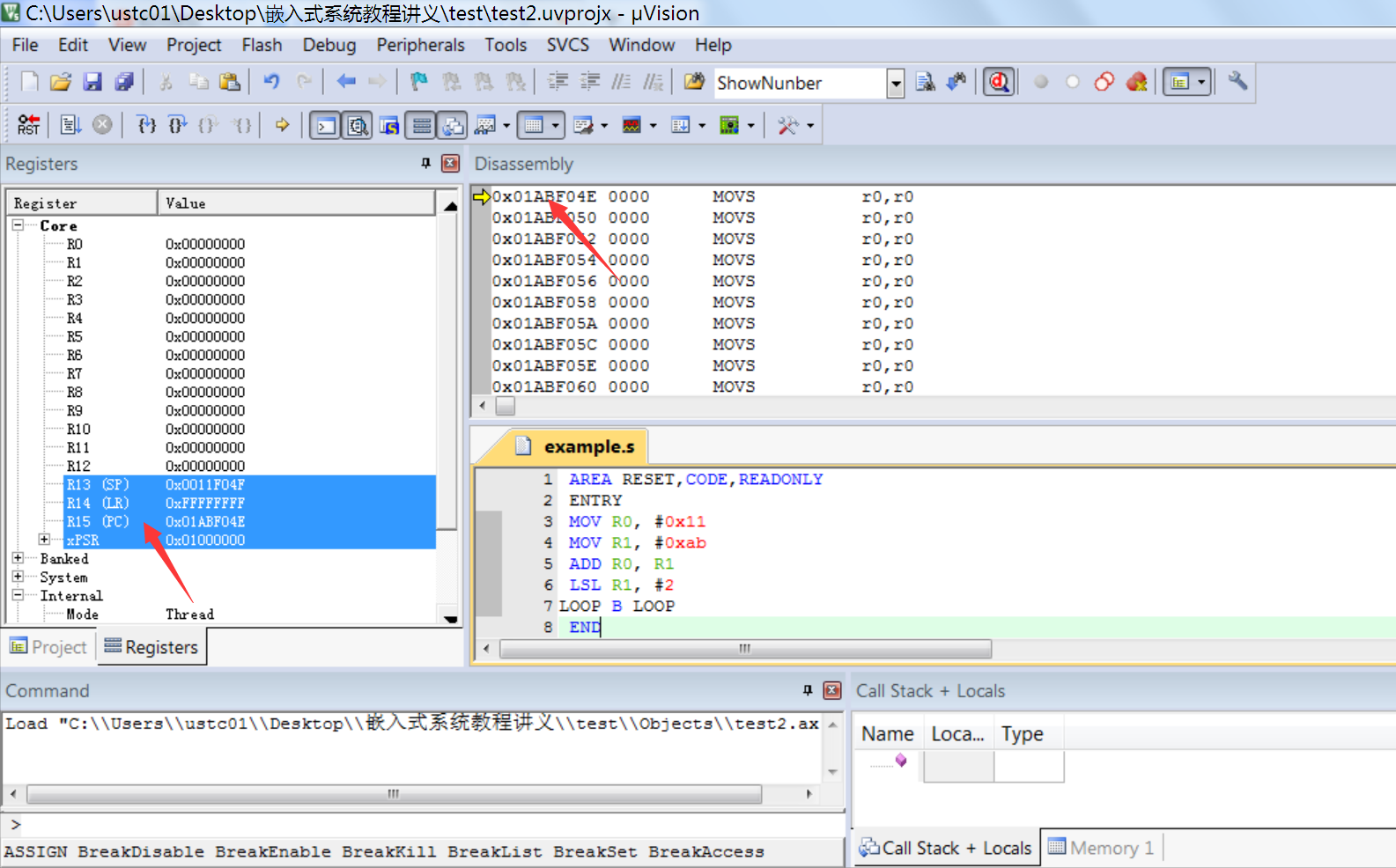


图1.6.8 程序进入调试模式

**注意：前面说过，汇编程序没有用系统自带的启动程序，所以系统初始化等功能都没有执行，R13R14R15内容为随机数，R15如图所示也就是PC并未指向我们的代码，我们必须通过手动修改的形式将PC指向我们需要执行的代码，根据我们所用芯片，应将R15赋值为0x08000000，修改后如图1.6.9所示。**

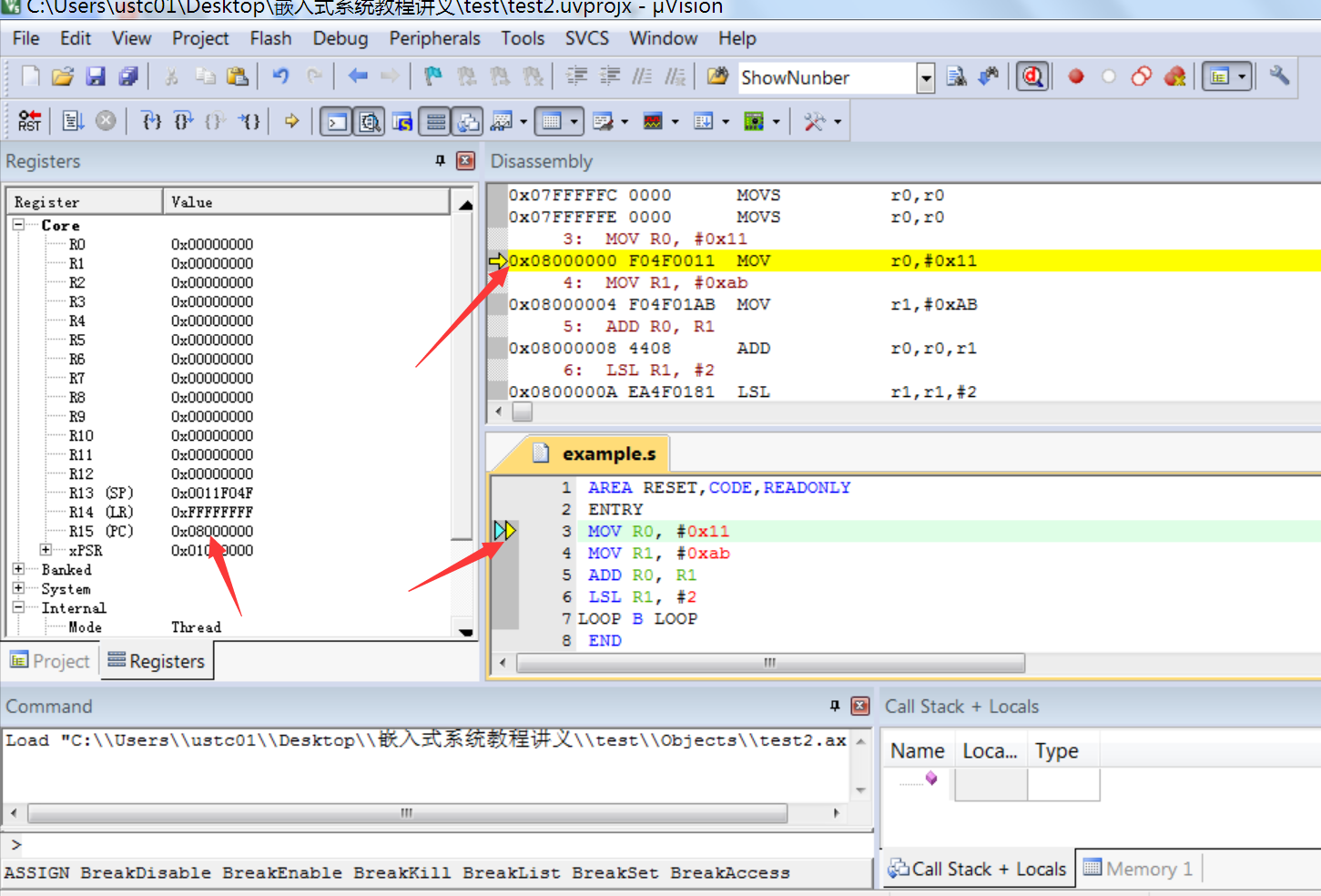


图1.6.9 修改PC寄存器后的调试状态

此时，PC指向我们刚刚写好的汇编代码，程序可以进入正常调试状态。调试主要有两种模式，单步执行和设断点，这两种模式结合使用效果更好。单步执行点击选择快捷键F10（step over）或F11（step in）进行单步跟踪并观察寄存器或者内存状态；设短点则在相应代码行设置断点，程序到此处会停止执行。本汇编代码中，我们可以将断点设在LOOP标号死循环处，并查看程序运行结果，如图1.6.10所示。

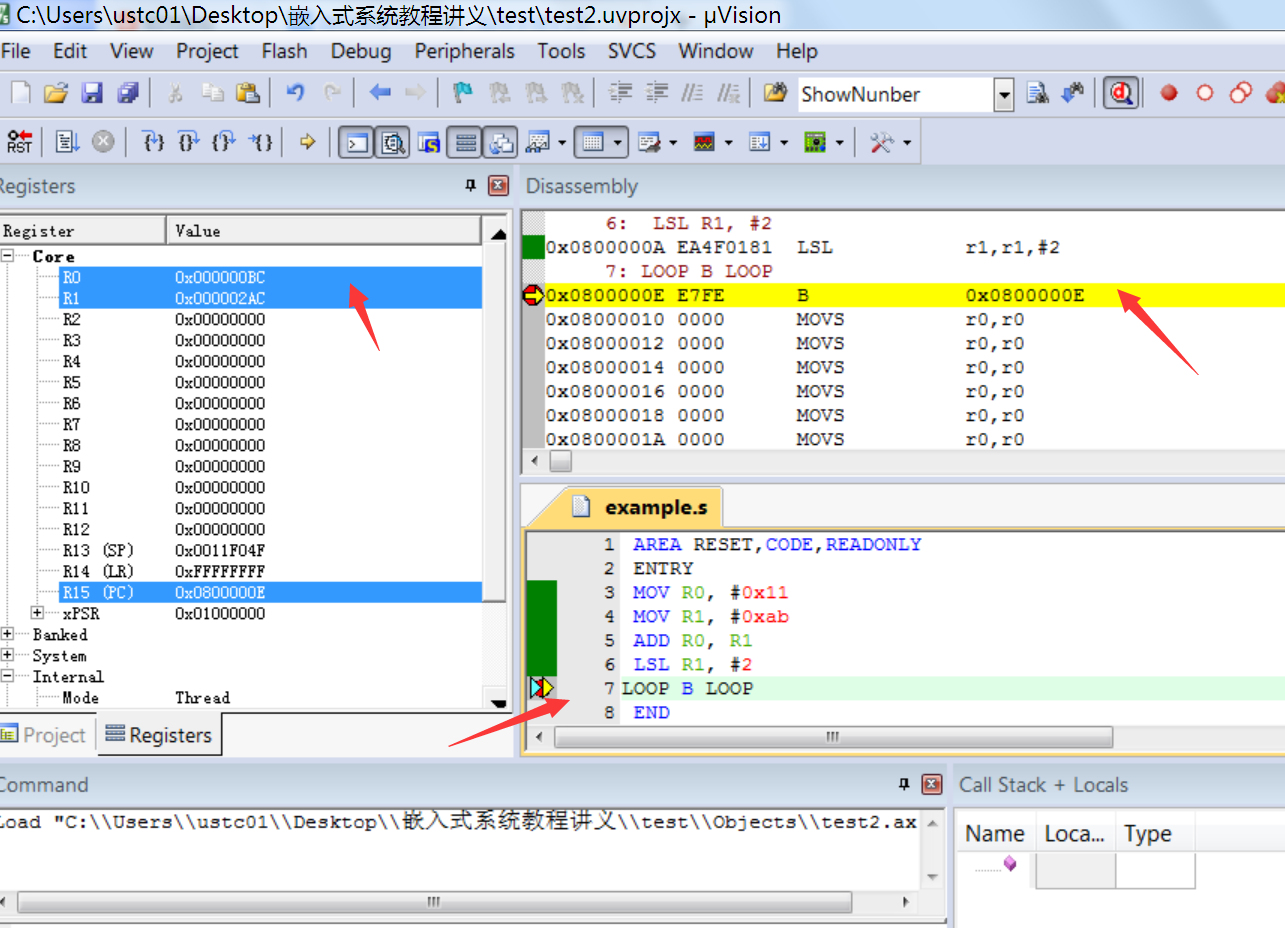


图1.6.10 设置断点后程序执行结果

我们可以看到程序执行结束后R0=0xab+0x11=0xbc,R1=0xab左移两位后为0x2ac。

汇编程序具有执行效率高、占用内存资源少等优点，但是对于新手来说上手门槛较高，同时因为不能利用系统自带初始化程序进行初始化，自己实现这些过程操作过于繁琐，所以对于绝大多数开发者来说都是利用C语言来开发项目，在必要时可以通过调用汇编子程序或者嵌入汇编代码的方式使用汇编，下面开始介绍利用C语言开发嵌入式应用程序的基本流程。

## J-Link V8驱动软件的安装

J-Link是SEGGER公司为支持仿真ARM内核芯片推出的JTAG仿真器。配合IAR EWAR，ADS，KEIL，WINARM，RealView等集成开发环境支持所有ARM7/ARM9 /ARM11,Cortex M0/M1/M3/M4, Cortex A5/A8/A9等内核芯片的仿真，与IAR，Keil等编译环境无缝连接，操作方便、连接方便、简单易学，是学习开发ARM最好最实用的开发工具。

本教学实验箱的底板集成J-Link V8在线下载程序和仿真功能，学习者无需另外购买仿真器就能实现程序的下载和在线调试，在这之前，必须安装J-LinkV8的驱动程序，否则无法使用。找到实验箱光盘自带的软件安装文件夹，解压J-Link V8驱动程序软件压缩包，找到Setup\_JLink\_V490安装程序，安装过程略...。

安装完毕后在开始－＞所有程序－＞SEGGER中查看到如下界面

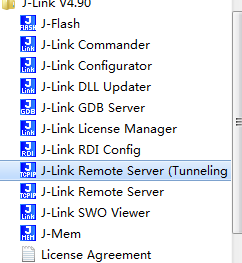


图1.5.1 J-Link V8安装后显示内容

这时插上USB\_J-Link下载线，打开实验箱上电源开关。在设备管理器中可以查看到J-Linkdrive,如下图。



图1.5.2 J-Link V8连接下载线后设备管理器显示

到这里已经成功安装J-Link V8下载调试驱动程序。

## 基于C新工程的建立及编译、链接、调试过程

之前步骤与建立汇编工程完全一样，在这里不再赘述。

**注意：在C工程中一般按目录进行工程文件管理，在工程目录下建立USER文件夹，并将工程文件\*.uvprojx保存在子目录USER下。**

### 库文件的添加

基于C的工程与汇编工程不同，需要用到大量的库函数和相应头文件，为了使工程层次清晰避免混乱，需要建立不同的子目录存放，具体操作流程如下。

1.工程目录结构配置

在工程文件夹中新建5个文件夹为CORE、OBJ、FWLIB、SYSTEM、USER（这个步骤可以在新建工程向导前实施），如图1.7.1所示，**下面我们对各文件夹存放的文件及各自的功能依次进行说明**。

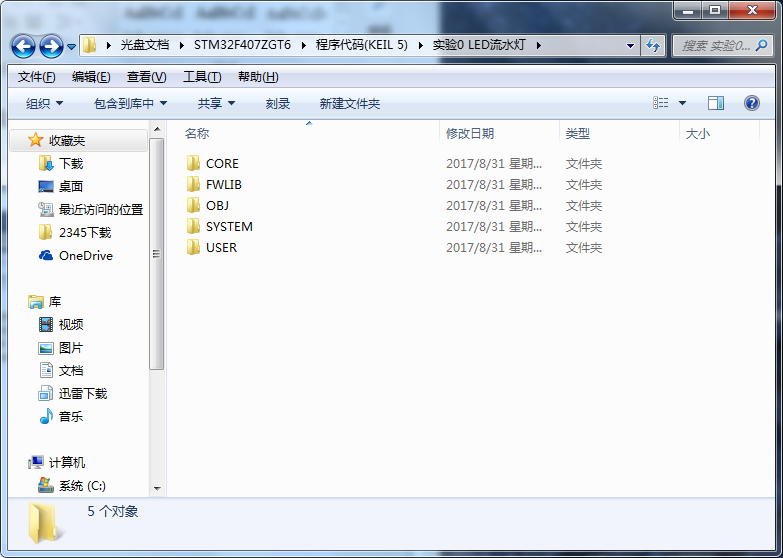


图1.7.1工程目录预览

（1）CORE文件夹下面存放的是固件库必须的核心文件和启动文件，这里面的文件用户不需要修改。大家可以根据自己的芯片型号选择对应的启动文件。

core\_cm4.h：内核功能的定义，比如NVIC相关寄存器的结构体和Systick配置；

core\_cm4\_simd.h：包含与编译器相关的处理；

core\_cmFunc.h：内核核心功能接口的头文件；

core\_cmInstr.h：包含一些内核核心专用指令；

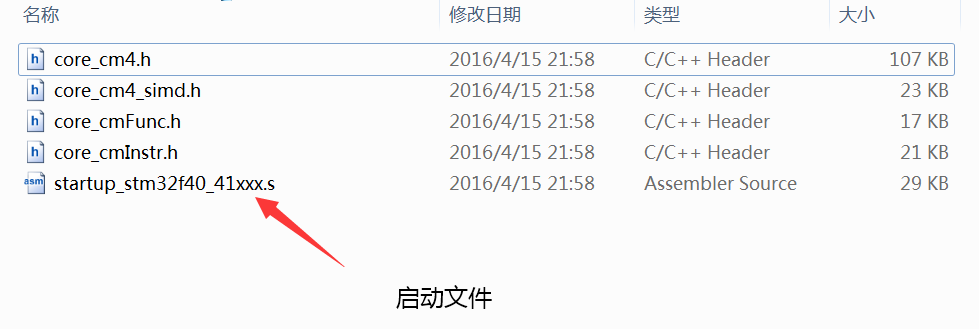


图1.7.2 CORE目录中文件

（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。

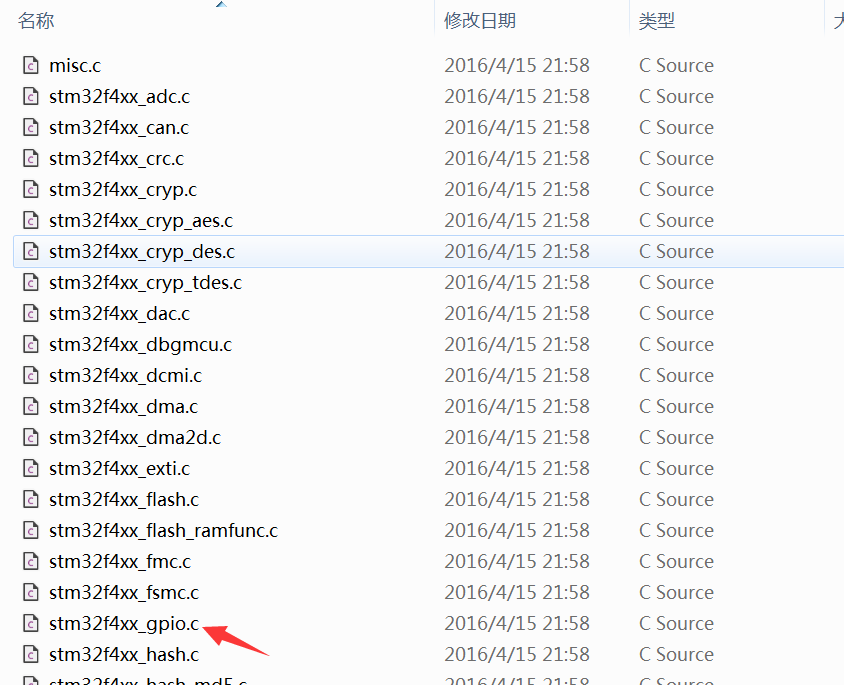
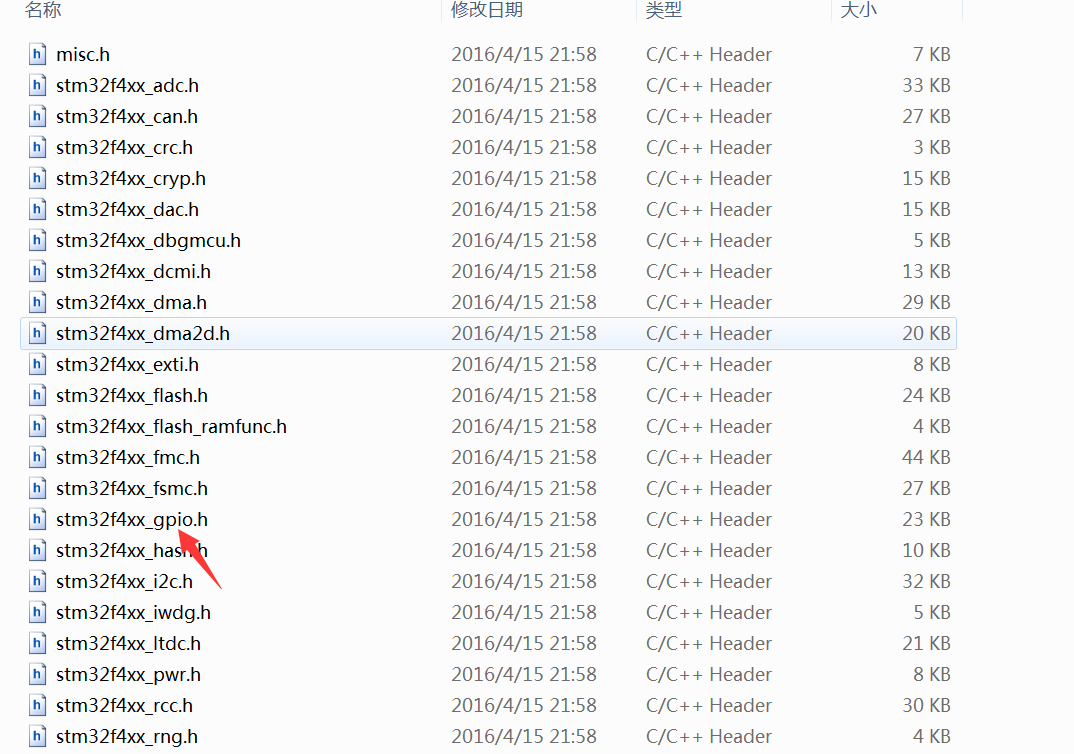


图1.7.3 FWLIB文件夹下inc和src中文件

（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。



图1.7.4 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**。**

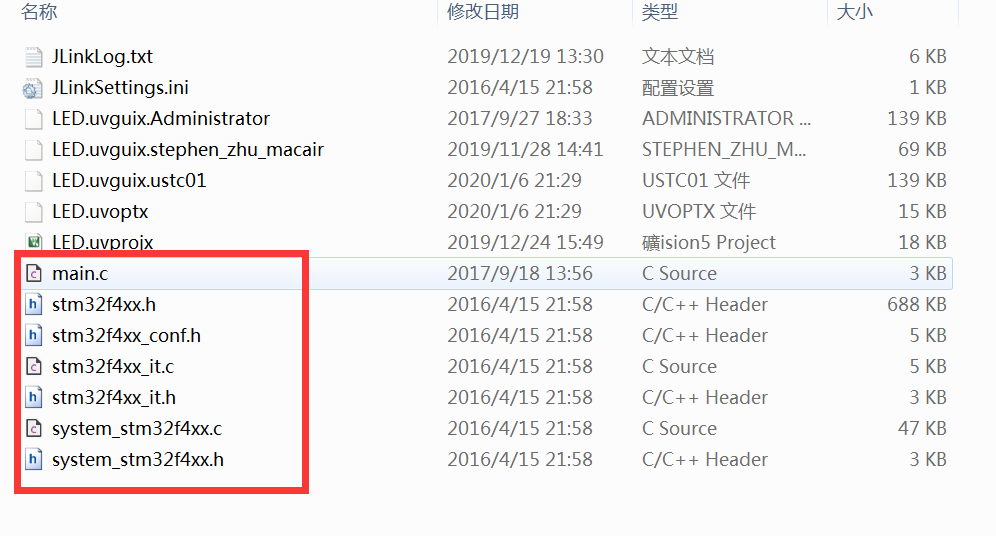


图1.7.5 USER文件夹下的主要文件

这些文件可以在Keil的安装目录或者在厂家提供的光盘相应目录中找到。光盘的目录\STM32F4xx\_DSP\_StdPeriph\_Lib\_V1.4.0\Libraries\STM32F4xx\_StdPeriph\_Driver 下面，将目录下面的 src,inc 文件夹copy到我们刚才建立的 FWLib文件夹下面；目录\STM32F4xx\_DSP\_StdPeriph\_Lib\_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm 下面，将文件startup\_stm32f40\_41xxx.s复制到 CORE 目录下面，这个.s汇编文件就是STM32F407的系统启动文件，完成系统初始化工作，系统启动时会自动寻找到该文件；目录\STM32F4xx\_DSP\_StdPeriph\_Lib\_V1.4.0\Libraries\CMSIS\Includ，将里面的头文件 core\_cm4.h和 和 core\_cm4\_simd.h 同样复制到 CORE 目录下面；目录：STM32F4xx\_DSP\_StdPeriph\_Lib\_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Include将里面的 2 个头文件 stm32f4xx.h 和 system\_stm32f4xx.h 复制到 USER 目录之下，这两个头文件是STM32F4 工程非常关键的两个头文件，然后进入目录\STM32F4xx\_DSP\_StdPeriph\_Lib\_V1.4.0\Project\STM32F4xx\_StdPeriph\_Templates ，将目录下面的5个文件main.c（主程序），stm32f4xx\_conf.h，stm32f4xx\_it.c（里面存放的是中断服务函数），stm32f4xx\_it.h ，system\_stm32f4xx.c复制到USER目录下；从实验代码中随便找到SYSTEM文件夹，把其中的文件复制到本实验中的SYSTEM文件夹中，如图1.7.6所示。

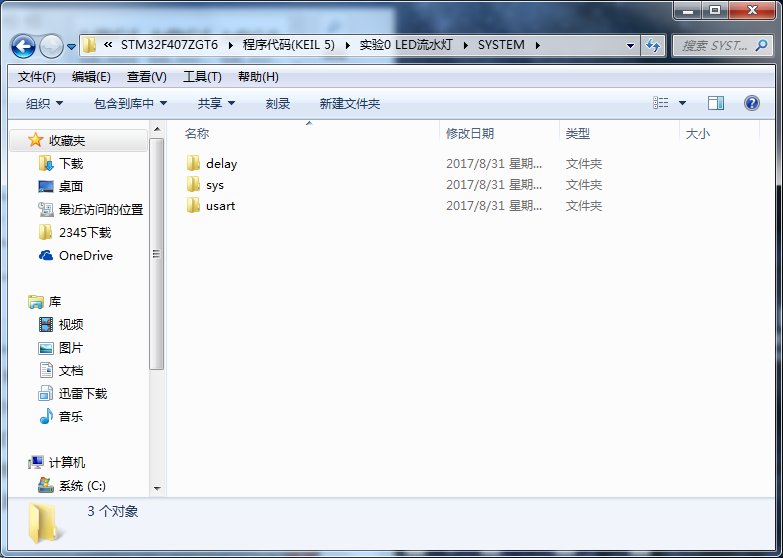


图1.7.6 SYSTEM目录文件浏览

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

2.工程文件管理

右键点击Target1，选择Manage Project Items，如图1.7.7所示。

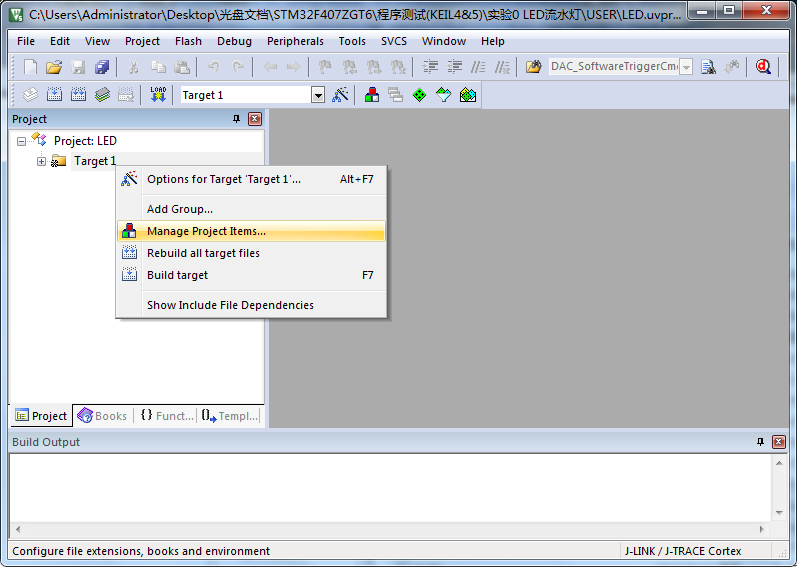


图1.7.7文件的添加设置

Project Targets一栏，我们在Groups一栏删掉一个Source Group1，建立4个 Groups：USER,CORE,FWLIB,SYSTEM。然后点击OK，如图1.7.8可以看到我们的Target名字以及 Groups情况。

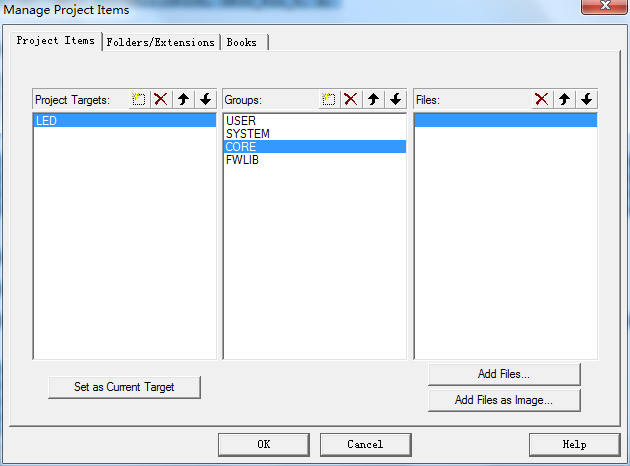


图1.7.8 新建组

下面我们往 Group 里面添加我们需要的文件（以LED流水灯为例）。这里第一步我们选择FWLIB，然后点击右边的Add Files, 定位到我们刚才建立的目录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我们添加的文件如图1.7.9。

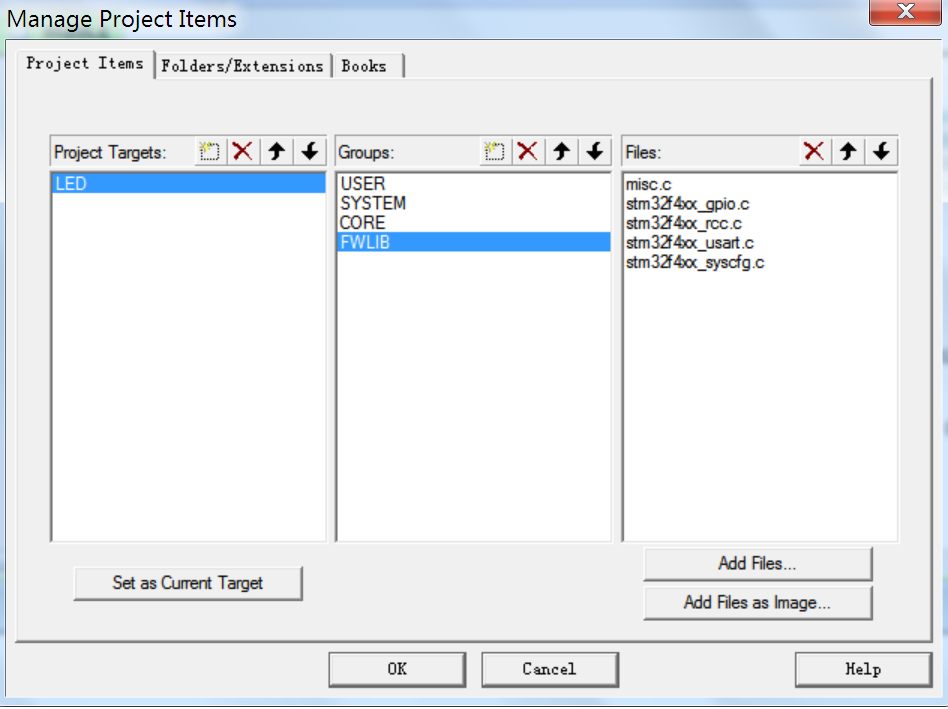


图1.7.9组FWLIB下添加的文件

用同样的方法，将Groups定位到CORE和USER下面，添加需要的文件。这里我们的CORE下面需要添加的文件为startup\_stm32f40\_41xxx.s ( 注意，默认添加的时候文件类型为.c, 也就是添加startup\_stm32f10x\_hd.s启动文件的时候，你需要选择文件类型为All files才能看得到这个文件)，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1.7.10和1.7.11所示。

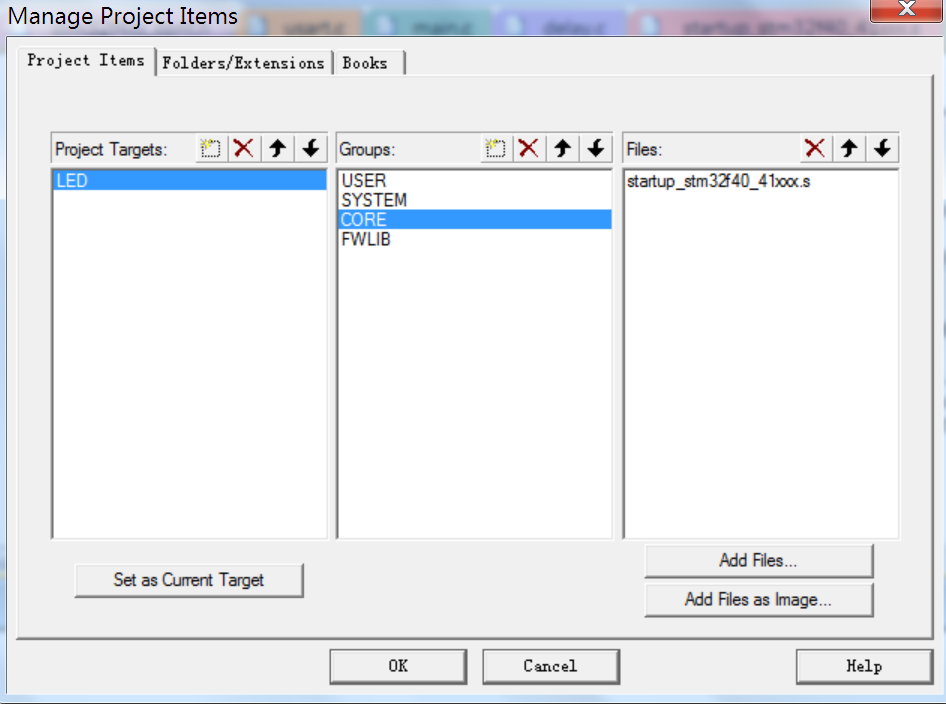


图1.7.10组CORE下添加的文件

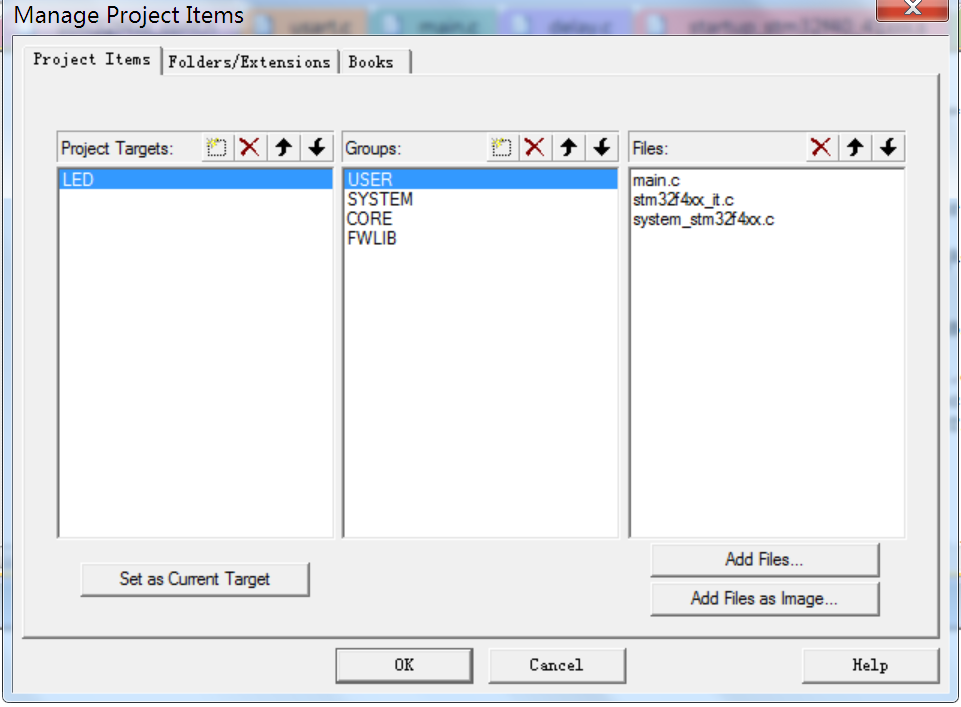


图1.7.11组USER下添加的文件

这样我们需要添加的文件已经添加到我们的工程中了，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。**

### 工程的配置

接下来我们要编译工程，在编译之前我们首先要选择编译中间文件编译后存放目录。方法是点击魔术棒，然后选择“Output”选项下面的“Select folder for objects…”,然后选择目录为我们上面新建的 OBJ 目录，如图1.7.12所示。这里大家注意，如果我们不设置 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。

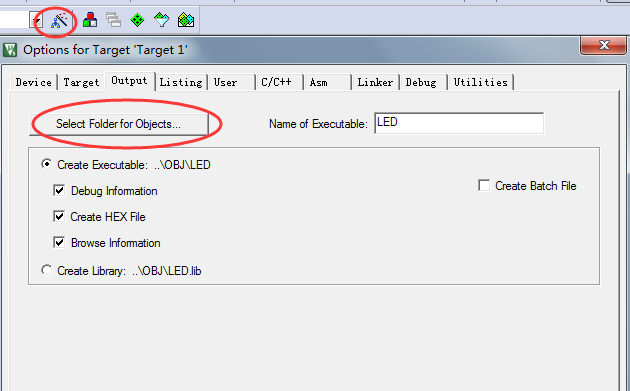


图1.7.12设置OBJ目录

下面我们要告诉 MDK，在哪些路径之下搜索需要的头文件，也就是头文件目录。这里大家要注意，对于任何一个工程，我们都需要把工程中引用到的所有头文件的路径都包含到进来。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1.7.13的路径添加进去，这些路径都是我们拷贝过.h头文件的路径。记住，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。

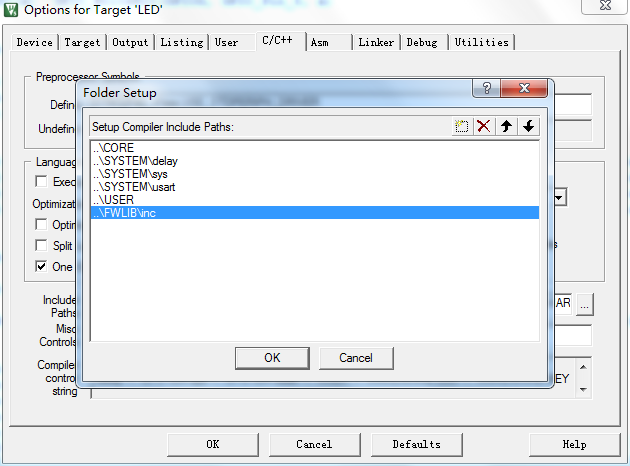
.

图1.7.13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，所以我们需要配置一个

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，他们之间是用逗号隔开的。

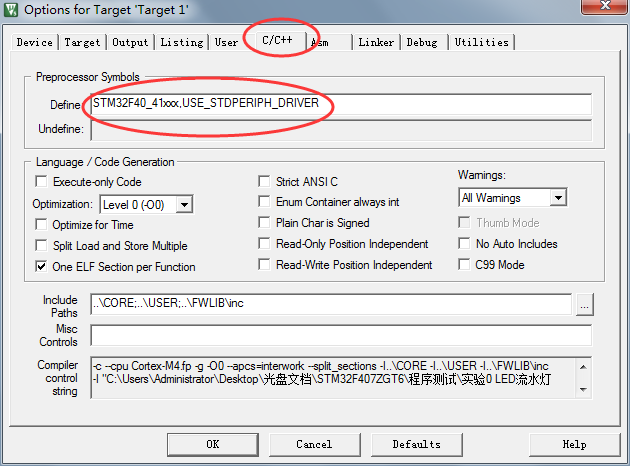
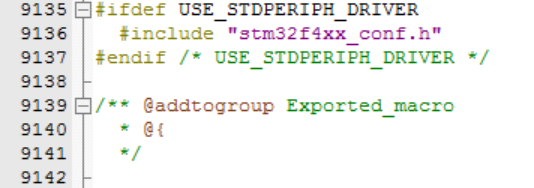


图1.7.14工程设置-宏定义的添加

**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：



点击‘Output’项勾选生产HEX文件,然后点击OK完成工程的配置。

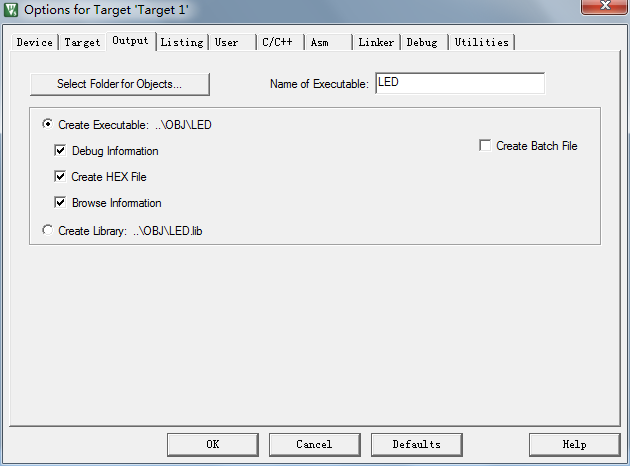


图1.7.15工程配置-文件输出

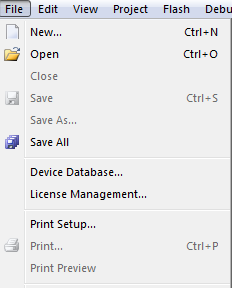
做完这些后我们就可以在main函数中编写我们的代码，在main函数中添加实验代码，然后进行下一步操作。

**配置好的工程可以做为后续所有实验的模板，可以保存后备用。**

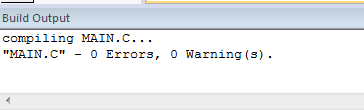
### 新文件的建立

前面中main已加入文件，如果需要添加新的代码文件和头文件，可新建文件，保存为.c或.h文件。

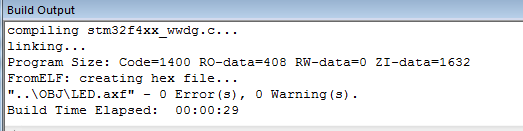
点击“File” →“New”



这时候编写程序，前面已经将代码写入main.c中，编写好后点击编译按钮，如果编译没出现错误如下：



点击编译整个工程按钮，如果没有错误，如下：



这时候就可以下载程序运行了！

### 在线下载调试

1.将USB线一端连接至计算机，一端连接至实验箱USB调试口。

2.打开Keil-MDK,设置好J-Link下载模式。FLASH－＞configure flash tools－＞Debug,选择J-Link/J-TRACE Cortex。

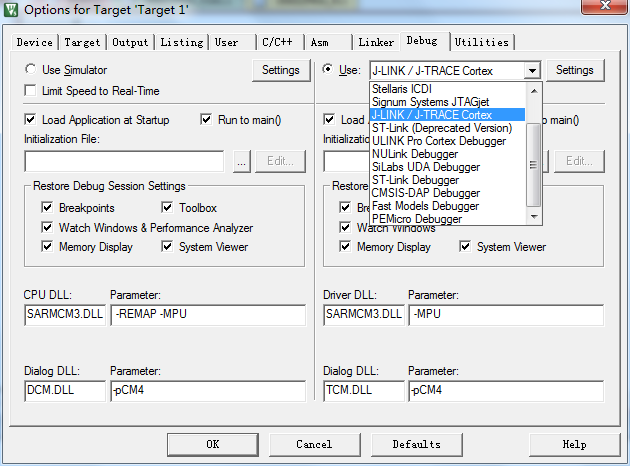


图1.7.16 Keil-MDK下载调试设置1

然后点击Settings进行如下设置

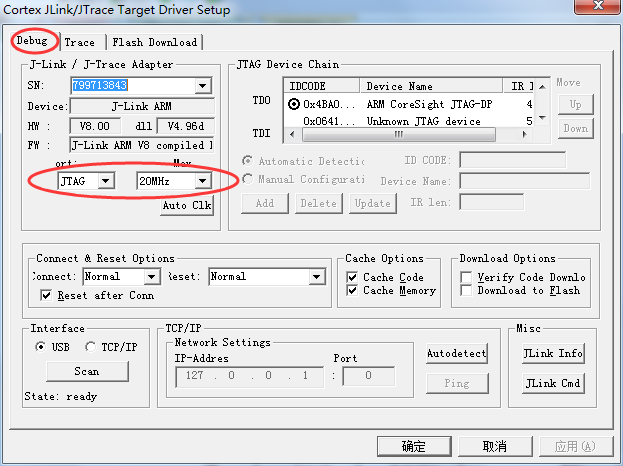


图1.7.17 Keil-MDK下载调试设置2

3.点击出现如下界面：选择J-Link/J-TRACE Cortex；

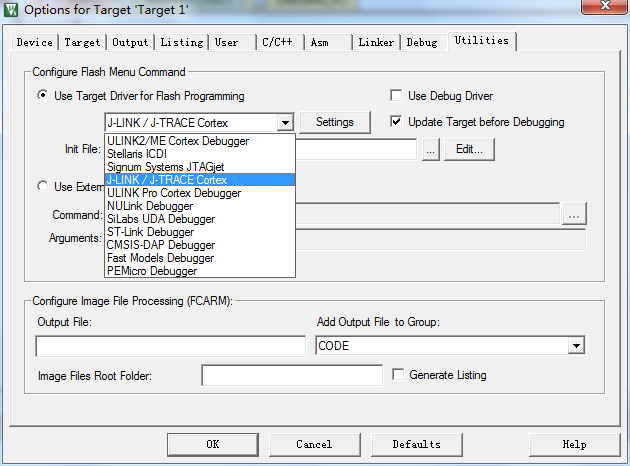


图1.7.18 Keil-MDK下载调试设置3

4.再设置FLASH DownLoad, FLASH－＞configure flash tools－＞Ulitities－＞ setting－＞flashdownload如下图所示：

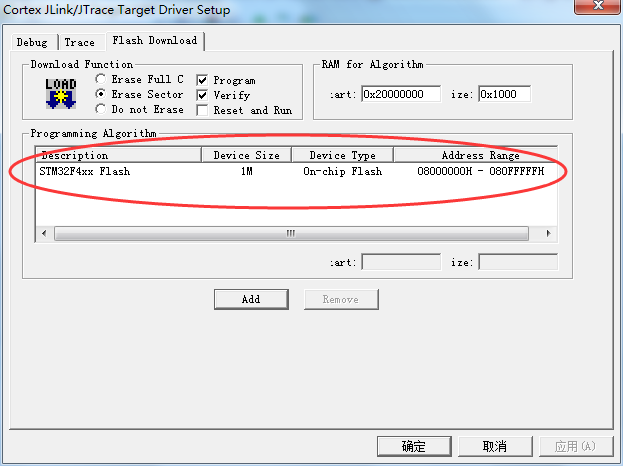


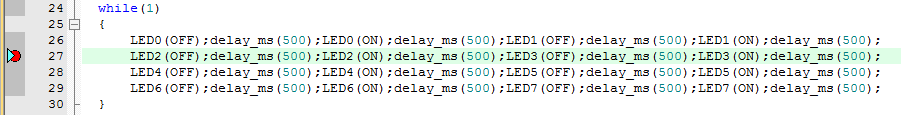
图1.7.19 Keil-MDK下载调试设置4

点击下载按钮下载程序下好程序后

点击仿真按钮Debug->Start/Stop Debug Session，然后点击全速运行按钮，观察实验现象。

点击单步运行按钮，程序每执行一步，LED灯会出现熄灭、点亮的动作

可以设置断点，双击程序该行，出现断点



这时程序就会执行到设置断点的那一步，调试流程与汇编调试基本一致，随着后面实验的进行更多问题的出现和解决，调试经验会慢慢积累。

## 练习

1. 利用MDK建立汇编工程并编写简单程序进行编译、调试。
2. 利用MDK建立C语言流水灯工程并进行配置、编译、调试等操作，并查看GPIO相应库函数的声明和定义。

附录：流水灯工程中main.c代码（本例子中GPIO设置也在main.c中实现）

#include "stm32f4xx.h"

#include "delay.h"

void LED\_Init(void);

#define ON Bit\_SET //可选中后F12查看定义

#define OFF Bit\_RESET

#define LED0(x) GPIO\_WriteBit(GPIOG, GPIO\_Pin\_11, x)

#define LED1(x) GPIO\_WriteBit(GPIOG, GPIO\_Pin\_10, x)

#define LED2(x) GPIO\_WriteBit(GPIOG, GPIO\_Pin\_9, x)

#define LED3(x) GPIO\_WriteBit(GPIOD, GPIO\_Pin\_7, x)

#define LED4(x) GPIO\_WriteBit(GPIOG, GPIO\_Pin\_3, x)

#define LED5(x) GPIO\_WriteBit(GPIOG, GPIO\_Pin\_2, x)

#define LED6(x) GPIO\_WriteBit(GPIOD, GPIO\_Pin\_13, x)

#define LED7(x) GPIO\_WriteBit(GPIOD, GPIO\_Pin\_12, x)

int main(void)

{

LED\_Init();

delay\_init(168);

while(1)

{

LED0(ON);delay\_ms(500);

LED0(OFF);delay\_ms(500);

LED1(ON);delay\_ms(500);

LED1(OFF);delay\_ms(500);

LED2(ON);delay\_ms(500);

LED2(OFF);delay\_ms(500);

LED3(ON);delay\_ms(500);

LED3(OFF);delay\_ms(500);

LED4(ON);delay\_ms(500);

LED4(OFF);delay\_ms(500);

LED5(ON);delay\_ms(500);

LED5(OFF);delay\_ms(500);

LED6(ON);delay\_ms(500);

LED6(OFF);delay\_ms(500);

LED7(ON);delay\_ms(500);

LED7(OFF);delay\_ms(500);

}

}

void LED\_Init(void)

{

GPIO\_InitTypeDef GPIO\_InitStructure;

RCC\_AHB1PeriphClockCmd(RCC\_AHB1Periph\_GPIOG|RCC\_AHB1Periph\_GPIOD, ENABLE);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_11 | GPIO\_Pin\_10 | GPIO\_Pin\_9 | GPIO\_Pin\_2 | GPIO\_Pin\_3;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_OUT;

GPIO\_InitStructure.GPIO\_OType = GPIO\_OType\_PP;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_100MHz;

GPIO\_InitStructure.GPIO\_PuPd = GPIO\_PuPd\_UP;

GPIO\_Init(GPIOG, &GPIO\_InitStructure);

GPIO\_ResetBits(GPIOG,GPIO\_Pin\_11 | GPIO\_Pin\_10 | GPIO\_Pin\_9 | GPIO\_Pin\_2 | GPIO\_Pin\_3);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_7 |GPIO\_Pin\_12 | GPIO\_Pin\_13;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_OUT;

GPIO\_InitStructure.GPIO\_OType = GPIO\_OType\_PP;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_100MHz;

GPIO\_InitStructure.GPIO\_PuPd = GPIO\_PuPd\_UP;

GPIO\_Init(GPIOD, &GPIO\_InitStructure);

GPIO\_ResetBits(GPIOD, GPIO\_Pin\_7 |GPIO\_Pin\_12|GPIO\_Pin\_13);

}

# 基本输入输出实验

实验目的：利用STM32F407 GPIO的输入输出功能控制实验箱上的LED、按键、数码管、LCD等外设，通过不同类型的实验熟悉并掌握GPIO的设置和使用方法。

## 流水灯功能实现

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STM32F4的GPIO口作为输出使用的方法，通过本节实验的学习，可以初步掌握STM32

F407基本IO口的使用。

### STM32F407 IO简介

STM32F4每组通用I/O端口包括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。这样，STM32F4每组IO有10个32位寄存器控制，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。如果在使用的时候，每次都直接用汇编操作寄存器配置IO，代码会比较多，也不容易记住，所以我们一般用库函数配置IO。

同STM32F1一样，STM32F4的IO可以由软件配置成如下8种模式中的任何一种：

1、输入浮空；2、输入上拉；3、输入下拉；4、模拟输入；5、开漏输出；6、推挽输出；

7、推挽式复用功能；8、开漏式复用功能。

如何配置常用的IO寄存器：MODER、OTYPER、OSPEEDR、PUPDR、ODR、IDR 、BSRR、AFRH和AFRL可以参考操作手册，这里不再赘述。下面讲解对应的库函数配置流程和方法。

### 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（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\_InitType

Def，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;

GPIOMode\_TypeDef GPIO\_Mode;

GPIOSpeed\_TypeDef GPIO\_Speed;

GPIOOType\_TypeDef GPIO\_OType;

GPIOPuPd\_TypeDef GPIO\_PuPd;

}GPIO\_InitTypeDef;

下面我们通过一个GPIO初始化实例来讲解这个结构体的成员变量的含义，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，我们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 \*/

}GPIOMode\_TypeDef;

GPIO\_Mode\_IN是用来设置为复位状态的输入，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

这里需要说明一下，实际我们的输入可以是GPIOSpeed\_TypeDef枚举类型中GPIO\_High\_Speed枚举类型值，也可以是GPIO\_Speed\_100MHz这样的值，实际上GPIO\_Speed\_100MHz就是通过define宏定义标识符定义出来的，它跟GPIO\_High\_Speed是等同的。

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，接下来看看如何利用库函数配置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)；而后者可以同时对多个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相关的函数我们先讲解到这里，这里我们做个概括性的总结，操作步骤为：

（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）操作IO，操作IO的方法就是上面我们讲解的方法。

### 实验步骤及练习

实验具体代码可以参考第一章附录内容，通读代码并用F12跟读到对应函数的声明和定义，掌握需要包含的头文件架构，运行程序并观察实验现象。

**练习:**

1. 通读源码，列出LED0-7用到的GPIO引脚。
2. 修改源码，改变流水灯的点亮方向、点亮模式等。

## 利用拨动开关DIP控制LED

实验内容：通过GPIO口读取DIP的拨动状态，同时将拨动状态通过GPIO口输出控制LED的亮灭，通过本节实验的学习，可以初步掌握STM32F407基本IO口的输入和输出功能。

### 工程目录结构

如2.1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2.2.1所示：

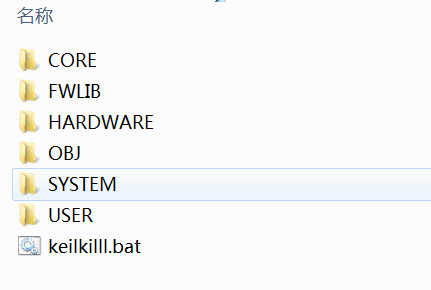


图2.2.1 通用工程目录结构

在子目录HARDWARE中可以按照不同外设再创建不同的子目录，本次实验中外设包括拨动开关和LED，所以在HARDWARE中再新建两个子目录DIP和LED如图2.2.2所示，DIP和LED子目录中文件结构如图2.2.3和2.2.4所示，分别存放拨动开关和LED功能实现的头文件和源文件。

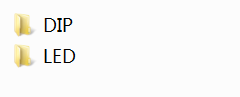


图2.2.2 外设子目录结构

图2.2.3 外设DIP子目录文件结构 图2.2.4 外设LED子目录文件结构

### 库函数配置GPIO流程

1. 拨动开关DIP配置流程

与拨动开关相连的GPIO引脚应设置为输入状态，GPIO相关寄存器中的IDR寄存器用于读取GPIOx的输入，该寄存器各位描述如图2.2.5所示：

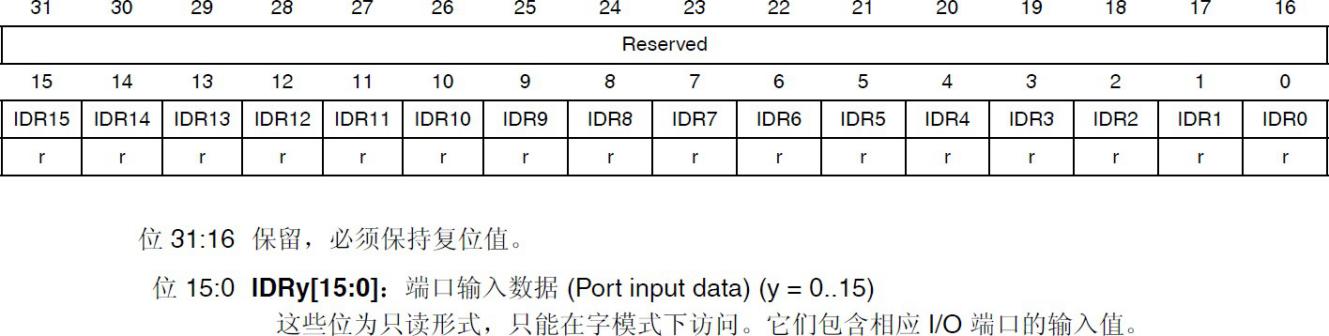


图2.2.5 GPIOx IDR寄存器各位描述

该寄存器用于读取某个IO的电平，如果对应的位为0(IDRy=0)，则说明该IO输入的是低电平，如果是1(IDRy=1)，则表示输入的是高电平。库函数相关函数为：

uint8\_t GPIO\_ReadInputDataBit(GPIO\_TypeDef\* GPIOx, uint16\_t GPIO\_Pin);

uint16\_t GPIO\_ReadInputData(GPIO\_TypeDef\* GPIOx);

前面的函数是用来读取一组IO口的一个或者几个IO口输入电平，后面的函数用来一次读取一组IO口所有IO口的输入电平。比如我们要读取GPIOF.5的输入电平，方法为：

GPIO\_ReadInputDataBit(GPIOF, GPIO\_Pin\_5);

dip.h中的函数声明和宏定义如下：

#ifndef \_\_DIP\_H

#define \_\_DIP\_H

#include "sys.h" //**sys.h中会#include "stm32f4xx.h" ，stm32f4xx.h中#include "stm32f4xx\_conf.h"，#include "stm32f4xx\_gpio.h"，相关GPIO库函数声明在stm32f4xx\_gpio.h中**

//LED端口定义

#define DIP0 GPIO\_ReadInputDataBit(GPIOE,GPIO\_Pin\_4)

#define DIP1 GPIO\_ReadInputDataBit(GPIOE,GPIO\_Pin\_5)

#define DIP2 GPIO\_ReadInputDataBit(GPIOC,GPIO\_Pin\_14)

#define DIP3 GPIO\_ReadInputDataBit(GPIOC,GPIO\_Pin\_15)

#define DIP4 GPIO\_ReadInputDataBit(GPIOF,GPIO\_Pin\_0)

#define DIP5 GPIO\_ReadInputDataBit(GPIOF,GPIO\_Pin\_1)

#define DIP6 GPIO\_ReadInputDataBit(GPIOF,GPIO\_Pin\_2)

#define DIP7 GPIO\_ReadInputDataBit(GPIOF,GPIO\_Pin\_3)

void DIP\_Init(void);//DIP初始化函数

#endif

dip.c中函数定义如下：

#include "DIP.h"

void DIP\_Init(void)

{

GPIO\_InitTypeDef GPIO\_InitStructure;

RCC\_AHB1PeriphClockCmd(RCC\_AHB1Periph\_GPIOE|RCC\_AHB1Periph\_GPIOF|RCC\_AHB1Periph\_GPIOC, ENABLE);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_4 | GPIO\_Pin\_5 ;

GPIO\_InitStructure.GPIO\_Mode =

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_100MHz;//100M

GPIO\_InitStructure.GPIO\_PuPd = GPIO\_PuPd\_UP;

GPIO\_Init(GPIOE, &GPIO\_InitStructure);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_14 | GPIO\_Pin\_15 ;

GPIO\_InitStructure.GPIO\_Mode =

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_100MHz;//100M

GPIO\_InitStructure.GPIO\_PuPd = GPIO\_PuPd\_UP;

GPIO\_Init(GPIOC, &GPIO\_InitStructure);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_0 | GPIO\_Pin\_1 | GPIO\_Pin\_2 | GPIO\_Pin\_3 ;

GPIO\_InitStructure.GPIO\_Mode =

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_100MHz;//100M

GPIO\_InitStructure.GPIO\_PuPd = GPIO\_PuPd\_UP;

GPIO\_Init(GPIOF, &GPIO\_InitStructure);

}

**输入模式下GPIO的配置流程与输出模式完全一样，只需要修改一个参数即可，同学们可以参考2.1的流程描述，修改该参数，并填入**DIP\_Init( )函数的空白部分**。**

1. LED配置流程

实验中涉及到GPIO的输入输出配置及相应库函数的调用，输出部分的配置和函数调用与2.1中流水灯实验基本完全一致，可以直接使用，其中有一点区别在于LED的输出控制在本实验中采用的是**位带**操作。

led.h中定义如下：

#ifndef \_\_LED\_H

#define \_\_LED\_H

#include "sys.h"

//LED端口定义

#define LED0 PGout(11)

#define LED1 PGout(10)

#define LED2 PGout(9)

#define LED3 PDout(7)

#define LED4 PGout(3)

#define LED5 PGout(2)

#define LED6 PDout(13)

#define LED7 PDout(12)

//其中PGout(11)可用F12跟踪在sys.h中定义

//#define PGout(n) BIT\_ADDR(GPIOG\_ODR\_Addr,n) //

//#define PGin(n) BIT\_ADDR(GPIOG\_IDR\_Addr,n) //

void LED\_Init(void);

#endif

操作IO口输出高低电平的3种方法。

1. 通过位带操作GPIO引脚输出高低电平

LED0=1

LED0=0 //LED0定义为PB5out

1. 使用固件库函数实现

GPIO\_SetBits(GPIOB,GPIO\_PIN\_5);

1. 直接操做寄存器

GPIOG->BSRRL=GPIO\_Pin\_5|GPIO\_Pin\_6;/\*置位\*/

GPIOG->BSRRH=GPIO\_Pin5|GPIO\_Pin\_6;/\*复位\*/

### 实验步骤及练习

拨动开关对应的GPIO引脚如表2.2.1所示：

表2.2.1 拨动开关与GPIO引脚对应关系表

|  |  |
| --- | --- |
| DIP0 | PE4 |
| DIP1 | PE5 |
| DIP2 | PC14 |
| DIP3 | PC15 |
| DIP4 | PF0 |
| DIP5 | PF1 |
| DIP6 | PF2 |
| DIP7 | PF3 |

通过读取拨动开关的输入状态，控制对应LED的亮灭，DIP0控制LED0，DIP1控制LED1，依此类推。

**练习：**

**在实验2.1的框架上添加DIP初始化代码，根据实验要求在循环体内修改主框架代码。**

# 外部中断及定时器中断实验

实验目的：利用STM32F407的外部输入中断和定时器中断控制LED的亮灭，通过不同类型的实验熟悉并掌握中断的配置流程和中断服务程序的编写。

## 外部中断设计

### STM32外部中断简介

ARM Cortex-M3/M4内核的中断系统很庞大，支持多达240个中断通道(不包含16个CM3的中断线)。但是STM32并没有使用CM3核的所有资源，目前STM32(非互联型)可支持60个中断通道，互联型支持68个。STM32外部中断：STM32有20个外部中断源，如下：16个GPIO输入中断；PVD输出；RTC闹钟事件；USB唤醒事件；以太网唤醒事件(只适用于互联型产品)。本节讨论的是16个GPIO输入中断。

GPIO输入中断虽然有16个输入通道，但是只占用了7个中断向量。EXTI0~EXTI4各占用一个中断向量，EXTI5~9共用一个，EXTI10~15共用一个。所以在编程的时候EXTI5~9将共用一个中断函数，EXTI10~15共用一个中断函数。这16个外部中断和GPIO的映射如图3.1.1所示。

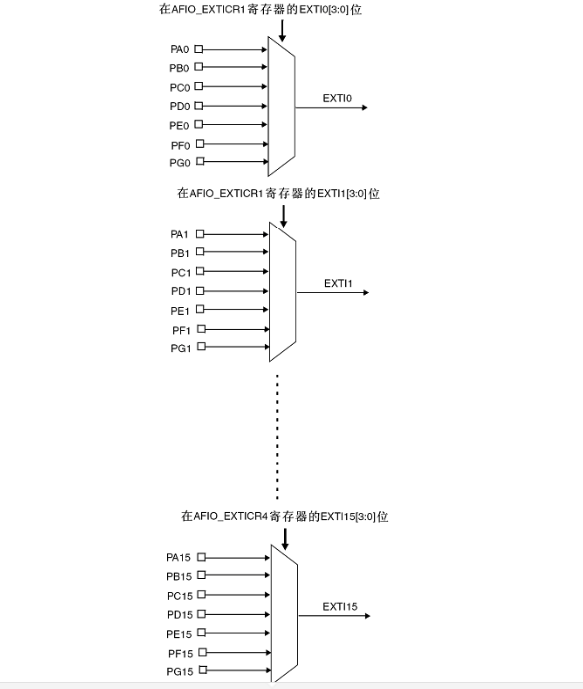


图3.1.1 GPIO和中断线的映射关系图

由上图可以看出，EXTI0通道，只能选择PA0，PB0，PC0，PD0，PE0，PF0，PG0这7个引脚中的一个；而EXTI1只能选择名称为Px1的7个引脚中的一个(x可取A~G中的一个)；依次类推，EXTI2只能选择Px2引脚；……。

外部中断EXTI和NVIC相关寄存器及各字段配置可以参考用户手册，这里不再赘述。可以总结为STM32外部中断要经过3个部分模块设置处理，然后才进入到中断服务程序的处理，其框图如下：

外部中断信号输入通道映射(选择)模块

外部中断控制器模块

NVIC中断控制器模块

中断服务程序

外部信号

### 库函数配置中断流程

通过上面的介绍和分析，我们对STM32的外部中断有了一定的了解，在编程的时候按照其信号路径，根据需要设置相关的寄存器即可。使用库函数配置外部中断的步骤如下：

（1）使能IO口时钟，初始化IO口为输入

首先，我们要使用IO口作为中断输入，所以我们要使能相应的IO口时钟，以及初始化相应的IO口为输入模式，具体的使用方法与第二章GPIO实验是一致的，这里就不做过多讲解。

（2）开启SYSCFG时钟，设置IO口与中断线的映射关系。

配置GPIO与中断线的映射关系，那么我们首先需要打开SYSCFG时钟。

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_SYSCFG, ENABLE);//使能SYSCFG时钟 ，**一定要注意，只要我们使用到外部中断，就必须打开SYSCFG时钟，而**STM32F4与STM32F1不同，STM32F1使用的是

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_AFIO,ENABLE); //使能复用时钟

接下来，我们配置GPIO与中断线的映射关系。在库函数中，配置GPIO与中断线的映射关系的函数SYSCFG\_EXTILineConfig( )来实现的：

void SYSCFG\_EXTILineConfig(uint8\_t EXTI\_PortSourceGPIOx, uint8\_t EXTI\_PinSourcex);

该函数将GPIO端口与中断线映射起来，使用范例是：

SYSCFG\_EXTILineConfig(EXTI\_PortSourceGPIOA, EXTI\_PinSource0);

//GPIO\_EXTILineConfig(GPIO\_PortSourceGPIOC, GPIO\_PinSource3);  //这是以前的F103所用的配置函数

将中断线0与GPIOA映射起来，那么很显然是GPIOA.0与EXTI1中断线连接了。设置好中断 线映射之后，那么到底来自这个IO 口的中断是通过什么方式触发的呢？接下来我们就要设置该中断线上中断的初始化参数了。

（3）初始化线上中断，设置触发条件等。

中断线上中断的初始化是通过函数EXTI\_Init()实现的。EXTI\_Init()函数的定义是：

void EXTI\_Init(EXTI\_InitTypeDef\* EXTI\_InitStruct); 下面我们用一个使用范例来说明这个函数的使用：

  EXTI\_InitTypeDef   EXTI\_InitStructure;

  EXTI\_InitStructure.EXTI\_Line=EXTI\_Line4;

  EXTI\_InitStructure.EXTI\_Mode = EXTI\_Mode\_Interrupt;

  EXTI\_InitStructure.EXTI\_Trigger = EXTI\_Trigger\_Falling; //下降沿触发

  EXTI\_InitStructure.EXTI\_LineCmd = ENABLE;

  EXTI\_Init(&EXTI\_InitStructure);     //初始化外设EXTI寄存器

上面的例子设置中断线4上的中断为下降沿触发。STM32的外设的初始化都是通过结构体来设置初始值的，这里就不再讲解结构体初始化的过程了。我们来看看结构体EXTI\_InitTypeDef的成员变量：

typedef struct

{

uint32\_t EXTI\_Line;

  EXTIMode\_TypeDef EXTI\_Mode;

  EXTITrigger\_TypeDef EXTI\_Trigger;

  FunctionalState EXTI\_LineCmd;

}EXTI\_InitTypeDef;

第一个参数是中断线的标号，对于我们的外部中断，取值范围为EXTI\_Line0~EXTI\_Line15。这个在上面已经讲过中断线的概念。也就是说，这个函数配置的是某个中断线上的中断参数。第二个参数是中断模式，可选值为中断EXTI\_Mode

\_Interrupt和事件EXTI\_Mode\_Event。第三个参数是触发方式，可以是下降沿触发EXTI

\_Trigger\_Falling，上升沿触发EXTI\_Trigger\_Rising，或者任意电平（上升沿和下降沿）触发EXTI\_Trigger\_Rising\_Falling，相信学过51的对这个不难理解。最后一个参数就是使能中断线了。

（4）配置中断分组（NVIC），并使能中断。

我们设置好中断线和GPIO映射关系，然后又设置好了中断的触发模式等初始化参数。既然是外部中断，涉及到中断我们当然还要设置NVIC中断优先级。这个在前面已经讲解过，这里我们就接着上面的范例，  设置中断线2的中断优先级。

NVIC\_InitTypeDef NVIC\_InitStructure;

NVIC\_InitStructure.NVIC\_IRQChannel = EXTI2\_IRQn;                  //使能按键外部中断通道

NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority = 0x02;  //抢占优先级2，

NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 0x02;    //响应优先级2

NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE;      //使能外部中断通道

NVIC\_Init(&NVIC\_InitStructure);                                                    //中断优先级分组初始化

（5）编写中断服务函数。

我们配置完中断优先级之后，接着我们要做的就是编写中断服务函数。中断服务函数的名字是在MDK中事先有定义的。这里需要说明一下，STM32F4的IO口外部中断函数只有7个，分别为：

EXPORT    EXTI0\_IRQHandler

EXPORT    EXTI1\_IRQHandler

EXPORT    EXTI2\_IRQHandler

EXPORT    EXTI3\_IRQHandler

EXPORT    EXTI4\_IRQHandler

EXPORT    EXTI9\_5\_IRQHandler

EXPORT    EXTI15\_10\_IRQHandler

**这些函数在系统启动文件中定义，名称不能有任何变化，我们使用时直接使用即可。**

中断线0-4每个中断线对应一个中断函数，中断线5-9共用中断函数EXTI9\_5

\_IRQHandler，中断线10-15共用中断函数EXTI15\_10\_IRQHandler。在编写中断服务函数的时候会经常使用到两个函数，第一个函数是判断某个中断线上的中断是否发生（标志位是否置位）： ITStatus EXTI\_GetITStatus(uint32\_t EXTI\_Line)； 这个函数一般使用在中断服务函数的开头判断中断是否发生。另一个函数是清除某个中断线上的中断标志位：

void EXTI\_ClearITPendingBit(uint32\_t EXTI\_Line)；

这个函数一般应用在中断服务函数结束之前，清除中断标志位。 常用的中断服务函数格式为：

void EXTI3\_IRQHandler(void)

{

if(EXTI\_GetITStatus(EXTI\_Line3)!=RESET)//判断某个线上的中断是否发生

  {

    EXTI\_ClearITPendingBit(EXTI\_Line3);    //清除LINE上的中断标志位

}

}

在这里需要说明一下，固件库还提供了两个函数用来判断外部中断状态以及清除外部状态标志位的函数EXTI\_GetFlagStatus和EXTI\_ClearFlag，他们的作用和前面两个函数的作用类似。只是在EXTI\_GetITStatus函数中会先判断这种中断是否使能，使能了才去判断中断标志位，而EXTI\_GetFlagStatus直接用来判断状态标志位。

使用IO口外部中断的一般步骤：

1）使能IO口时钟，初始化IO口为输入。

2）使能SYSCFG时钟，设置IO口与中断线的映射关系。

3）初始化线上中断，设置触发条件等。

4）配置中断分组（NVIC），并使能中断。

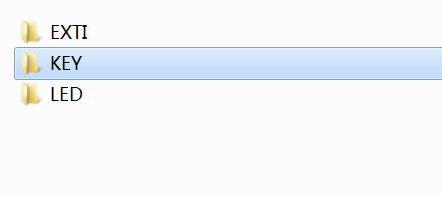
5）编写中断服务函数。

通过以上几个步骤的设置，我们就可以正常使用外部中断了。

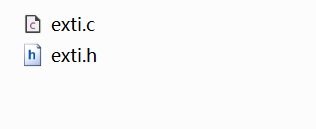
### 实验步骤及练习

实验中将利用SW0按键产生中断，并控制LED0状态的反转。

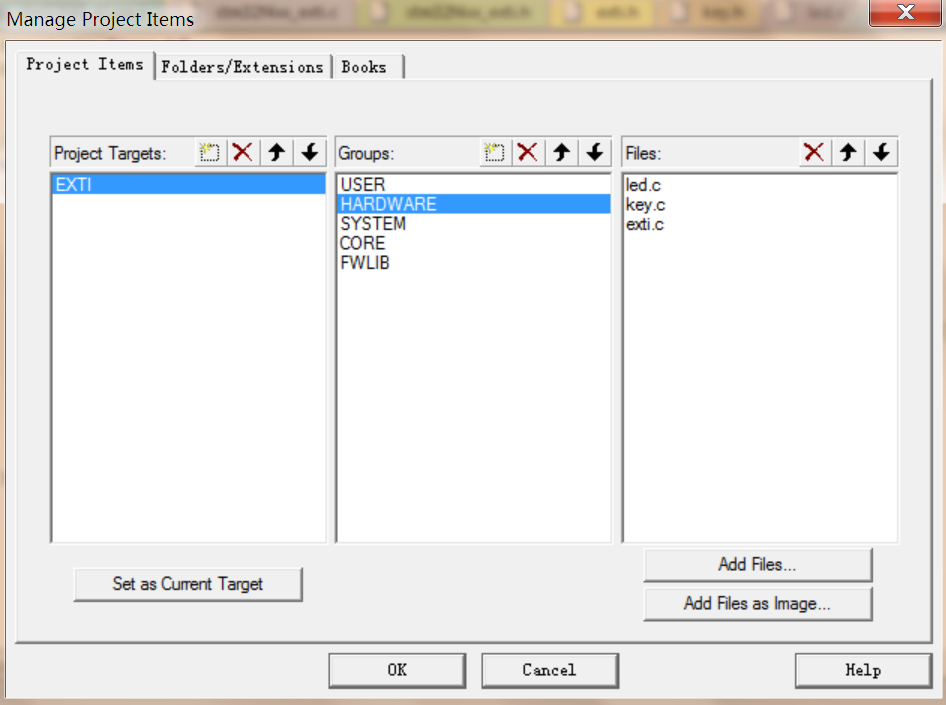
1. 工程目录HARDWARE中目录结构如图所示：



其中EXTI为中断相关目录，其中文件包含如下图所示：



LED与KEY目录中文件结构与前两章实验基本相同。将exti.c添加到工程中，如下图所示：



1. 中断配置

exti.h中声明如下：

#ifndef \_\_EXTI\_H

#define \_\_EXIT\_H

#include "sys.h"

void EXTIX\_Init(void);

#endif

exti.c如下所示：

#include "exti.h"

#include "delay.h"

#include "led.h"

#include "key.h"

//外部中断0服务程序

void EXTI0\_IRQHandler(void)

{

delay\_ms(10);

LED0=!LED0; //

EXTI\_ClearITPendingBit(EXTI\_Line0); //清除Line0上的中断标志位

}

void EXTIX\_Init(void)

{

NVIC\_InitTypeDef NVIC\_InitStructure;

EXTI\_InitTypeDef EXTI\_InitStructure;

KEY\_Init();

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_SYSCFG, ENABLE);

SYSCFG\_EXTILineConfig(EXTI\_PortSourceGPIOE, EXTI\_PinSource0);//PA0连接中断0

EXTI\_InitStructure.EXTI\_Line = EXTI\_Line0;//LINE0

EXTI\_InitStructure.EXTI\_Mode = EXTI\_Mode\_Interrupt;

EXTI\_InitStructure.EXTI\_Trigger = EXTI\_Trigger\_Rising;

EXTI\_InitStructure.EXTI\_LineCmd = ENABLE;

EXTI\_Init(&EXTI\_InitStructure);

NVIC\_InitStructure.NVIC\_IRQChannel = EXTI0\_IRQn;//外部中断0

NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority = 0x00;//抢占优先级0

NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 0x02;

NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE;

NVIC\_Init(&NVIC\_InitStructure);

}

1. 主程序main.c中代码如下：

#include "sys.h"

#include "delay.h"

#include "led.h"

#include "key.h"

#include "exti.h"

int main(void)

{

NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_2);

delay\_init(168);

LED\_Init();

KEY\_Init();

EXTIX\_Init();

LED0=0;//初始LED0关状态

while(1)

{

}

}

**练习：**

1. **编写LED与KEY目录中文件并实现功能要求。**
2. **修改中断配置文件实现不同中断通道控制。**

## 定时器中断设计

### STM32定时器简介

本节实验介绍如何使用 STM32F4 的通用定时器，STM32F4的定时器功能十分强大，有TIME1和TIME8 等高级定时器，也有TIME2~TIME5，TIM9~TIM14等通用定时器，还有TIME6和TIME7等基本定时器，总共达14个定时器之多。

STM32F4的通用定时器包含一个16位或32位自动重载计数器（CNT），该计数器由可编程预分频器（PSC）驱动。STM32F4的通用定时器可以被用于：测量输入信号的脉冲长度(输入捕获)或者产生输出波形(输出比较和 PWM)等。使用定时器预分频器和 RCC时钟控制器预分频器，脉冲长度和波形周期可以在几个微秒到几个毫秒间调整。STM32F4的每个通用定时器都是完全独立的，没有互相共享的任何资源。

STM32的通用TIMx（TIM2~TIM5和TIM9~TIM14）定时器功能包括：

1）16位/32 位（仅TIM2 和TIM5）向上、向下、向上/向下自动装载计数器（TIMx\_CNT），注意：TIM9~TIM14 只支持向上（递增）计数方式。

2）16位可编程（可以实时修改）预分频器（TIMx\_PSC）计数器时钟频率的分频系数为 1～65535之间的任意数值。

3）4 个独立通道（TIMx\_CH1~4，TIM9~TIM14 最多 2 个通道），这些通道可以用来作为：

A输入捕获

B输出比较

C PWM 生成(边缘或中间对齐模式) ，注意：TIM9~TIM14 不支持中间对齐模式

D单脉冲模式输出

可使用外部信号（TIMx\_ETR）控制定时器和定时器互连（可以用1个定时器控制另外一个定时器）的同步电路。如下事件发生时产生中断/DMA（TIM9~TIM14不支持DMA）：

更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)

触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)

由于STM32F4 通用定时器比较复杂，这里我们不再多介绍，请大家直接参考《STM32F4xx中文参考手册》。

### 库函数配置定时中断流程

接下来我们以通用定时器 TIM3 为实例，来说明要经过哪些步骤，才能达到这个要求，并产生中断。这里我们就对每个步骤通过库函数的实现方式来描述。首先要提到的是，定时器相关的库函数主要集中在固件库文件 stm32f4xx\_tim.h 和 stm32f4xx\_tim.c 文件中。定时器配置步骤如下：

1. TIM3时钟使能：TIM3 是挂载在APB1 之下，所以我们通过APB1 总线下的使能使能函数来使能 TIM3。调用的函数是：

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_TIM3,ENABLE); ///使能 TIM3 时钟

1. 初始化定时器参数,设置自动重装值，分频系数，计数方式等。

Void TIM\_TimeBaseInit(TIM\_TypeDef\*TIMx, TIM\_TimeBaseInitTypeDef\* TIM\_TimeBaseInitStruct);

在库函数中，定时器的初始化参数是通过初始化函数 TIM\_TimeBaseInit 实现的：

第一个参数是确定是哪个定时器，这个比较容易理解；第二个参数是定时器初始化参数结构体指针，结构体类型为TIM\_TimeBaseInitTypeDef，下面我们看看这个结构体的定义：

typedef struct

{

uint16\_t TIM\_Prescaler; //分频系数

uint16\_t TIM\_CounterMode; //计数方式

uint16\_t TIM\_Period; //自动重载计数值（即定时时间）

uint16\_t TIM\_ClockDivision;//时钟分频因子

uint8\_t TIM\_RepetitionCounter;

} TIM\_TimeBaseInitTypeDef;

这个结构体一共有5个成员变量，要说明的是，对于通用定时器只有前面四个参数有用，最后一个参数TIM\_RepetitionCounter 是高级定时器才有用的，这里不多解释。

第一个参数 TIM\_Prescaler 是用来设置分频系数的。第二个参数TIM\_CounterMode

是用来设置计数方式，可以设置为向上计数，向下计数方式还有中央对齐计数方式，比较常用的是向上计数模式 TIM\_CounterMode\_Up 和向下计数模式TIM\_CounterMode

\_Down。第三个参数是设置自动重载计数周期值。第四个参数是用来设置时钟分频因子。

针对TIM3 初始化范例代码格式：

TIM\_TimeBaseInitTypeDef TIM\_TimeBaseStructure; TIM\_TimeBaseStructure.TIM\_Period = 5000;

TIM\_TimeBaseStructure.TIM\_Prescaler =7199; TIM\_TimeBaseStructure.TIM\_ClockDivision = TIM\_CKD\_DIV1; TIM\_TimeBaseStructure.TIM\_CounterMode = TIM\_CounterMode\_Up;

TIM\_TimeBaseInit(TIM3, &TIM\_TimeBaseStructure);

1. 设置 TIM3\_DIER允许更新中断。

因为我们要使用 TIM3 的更新中断，寄存器的相应位便可使能更新中断。在库函数里面定时器中断使能是通过TIM\_ITConfig 函数来实现的：

void TIM\_ITConfig(TIM\_TypeDef\* TIMx, uint16\_t TIM\_IT, FunctionalState NewState)；

第一个参数是选择定时器号，这个容易理解，取值为 TIM1~TIM17。

第二个参数非常关键，是用来指明我们使能的定时器中断的类型，定时器中断的类型有很多种，包括更新中断TIM\_IT\_Update，触发中断TIM\_IT\_Trigger，以及输入捕获中断等等。

第三个参数就很简单了，就是失能还是使能。例如我们要使能TIM3 的更新中断，格式为：

TIM\_ITConfig(TIM3,TIM\_IT\_Update,ENABLE );

1. TIM3中断优先级设置。

在定时器中断使能之后，因为要产生中断，必不可少的要设置NVIC相关寄存器，设置中断优先级。之前多次讲解到用NVIC\_Init函数实现中断优先级的设置，这里就不重复讲解。

1. 允许TIM3工作，也就是使能TIM3。

光配置好定时器还不行，没有开启定时器，照样不能用。我们在配置完后要开启定时器，通过TIM3\_CR1 的CEN 位来设置。在固件库里面使能定时器的函数是通过 TIM\_Cmd 函数来实现的：

void TIM\_Cmd(TIM\_TypeDef\* TIMx, FunctionalState NewState)

这个函数非常简单，比如我们要使能定时器3，方法为：

TIM\_Cmd(TIM3, ENABLE); //使能 TIMx 外设

1. 编写中断服务函数

通过状态寄存器的值来判断此次产生的中断属于什么类型。然后执行相关的操作， 我们这里使用的是更新（溢出）中断，所以在状态寄存器 SR 的最低位。在处理完中断之后应该向TIM3\_SR 的最低位写 0，来清除该中断标志。

在固件库函数里面，用来读取中断状态寄存器的值判断中断类型的函数是：

ITStatus TIM\_GetITStatus(TIM\_TypeDef\* TIMx, uint16\_t) 该函数的作用是，判断定时器 TIMx 的中断类型 TIM\_IT 是否发生中断。比如

我们要判断定时器3是否发生更新（溢出）中断，方法为：

if (TIM\_GetITStatus(TIM3, TIM\_IT\_Update) != RESET)

{

}

固件库中清除中断标志位的函数是：

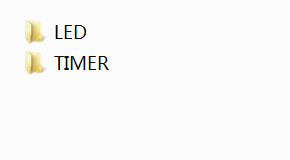
void TIM\_ClearITPendingBit(TIM\_TypeDef\* TIMx, uint16\_t TIM\_IT)

该函数的作用是，清除定时器 TIMx的中断TIM\_IT标志位。

### 实验步骤及练习

实验中将利用定时中断并控制LED0状态的反转。

1. 工程目录HARDWARE中目录结构如图所示：



工程中的 HARDWARE下面添加一个timer.c 文件（包括头文件 timer.h），这两个文件需要我们自己编写。同时还引入了定时器相关的固件库函数文件 stm32f4xx\_tim.c 和头文件stm32f4xx\_tim.h。time.c 文件。timer.c 文件代码如下：

//通用定时器 3 中断初始化

//arr：自动重装值。 psc：时钟预分频数

//定时器溢出时间计算方法:Tout=((arr+1)\*(psc+1))/Ft us.

//Ft=定时器工作频率,单位:Mhz

//这里使用的是定时器 3!

#include "timer.h"

#include "led.h"

void TIM3\_Int\_Init(u16 arr,u16 psc)

{

TIM\_TimeBaseInitTypeDef TIM\_TimeBaseInitStructure;

NVIC\_InitTypeDef NVIC\_InitStructure;

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_TIM3,ENABLE);

TIM\_TimeBaseInitStructure.TIM\_Period = arr;

TIM\_TimeBaseInitStructure.TIM\_Prescaler=psc;

TIM\_TimeBaseInitStructure.TIM\_CounterMode=TIM\_CounterMode\_Up;

TIM\_TimeBaseInitStructure.TIM\_ClockDivision=TIM\_CKD\_DIV1;

TIM\_TimeBaseInit(TIM3,&TIM\_TimeBaseInitStructure);

TIM\_ITConfig(TIM3,TIM\_IT\_Update,ENABLE);

TIM\_Cmd(TIM3,ENABLE);

NVIC\_InitStructure.NVIC\_IRQChannel=TIM3\_IRQn; NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority=0x01; NVIC\_InitStructure.NVIC\_IRQChannelSubPriority=0x03; NVIC\_InitStructure.NVIC\_IRQChannelCmd=ENABLE;

NVIC\_Init(&NVIC\_InitStructure);

}

void TIM3\_IRQHandler(void)

{

if(TIM\_GetITStatus(TIM3,TIM\_IT\_Update)==SET)

{

LED1=!LED1;

}

TIM\_ClearITPendingBit(TIM3,TIM\_IT\_Update); //Çå³ýÖÐ¶Ï±êÖ¾Î»

}

该文件下包含一个中断服务函数和一个定时器3中断初始化函数，中断服务函数比较简单，在每次中断后，判断TIM3的中断类型，如果中断类型正确，则执行LED1（DS1）的翻转。

TIM3\_Int\_Init 函数就是执行我们上面介绍的那 5 个步骤，使得TIM3开始工作，并开启中断。这里我们分别用标号①~⑤来标注定时器初始化的五个步骤。该函数的 2 个参数用来设置TIM3 的溢出时间。因为系统初始化 SystemInit 函数里面已经初始化 APB1 的时钟为 4 分频，所以 APB1 的时钟为 42M，而从STM32F4得知：当 APB1的时钟分频数为1的时候，TIM2~7 以及TIM12~14 的时钟为 APB1 的时钟，而如果APB1的时钟分频数不为1，那么 TIM2~7 以及 TIM12~14 的时钟频率将为APB1 时钟的两倍。因此，TIM3 的时钟为 84M，再根据我们设计的arr和psc的值，就可以计算中断时间了。计算公式如下：

Tout= ((arr+1)\*(psc+1))/Tclk；

其中：Tclk：TIM3 的输入时钟频率（单位为 Mhz）。Tout：TIM3 溢出时间（单位为 us）。

timer.h 头文件内容比较简单，这里我们就不做讲解。

最后，我们看看主函数代码如下：

#include "sys.h"

#include "delay.h"

#include "usart.h"

#include "led.h"

#include "timer.h"

int main(void)

{

NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_2);

delay\_init(168);

LED\_Init();

TIM3\_Int\_Init(5000-1,8400-1);

while(1)

{

};

}

这里的代码和之前大同小异，此段代码对TIM3进行初始化之后，进入死循环等待 TIM3 溢出中断，当 TIM3\_CNT 的值等于TIM3\_ARR 的值的时候，就会产生 TIM3 的更新中断，然后在中断里面取反LED1，TIM3\_CNT再从0开始计数。这里定时器定时时长500ms 是这样计算出来的，定时器的时钟为 84Mhz，分频系数为 8400, 所以分频后的计数频率为 84Mhz/8400=10KHz,然后计数到 5000，所以时长为 5000/10000=0.5s，也就是 500ms。

**练习：**

1. **编写LED目录中文件并实现功能要求。**
2. **修改配置参数实现不同定时时间间隔。**

# LCD的控制及字符显示

## LCD液晶显示原理

LCD是指液晶显示器，是Liquid Crystal Display的简称。LCD的构造是再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。

### LCD12864引脚功能与接口时序

LCD12864液晶显示屏是分辨率为128（列）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）及 64X256 点阵显示 RAM（GDRAM）。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，本实验箱的硬件连接为串行连接。

LCD12864硬件连接如图4.1.1所示：

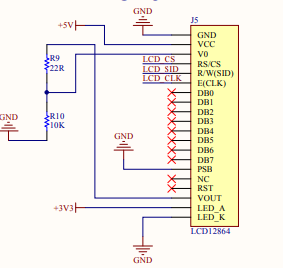
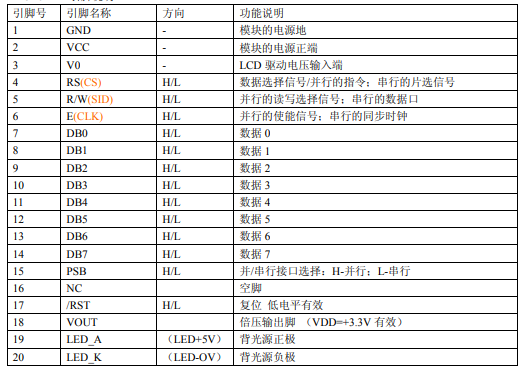


图4.1.1 LCD12864硬件连线图

LCD12864各引脚功能图如下表所示：



LCD12864串行接口时序如图4.1.2所示：

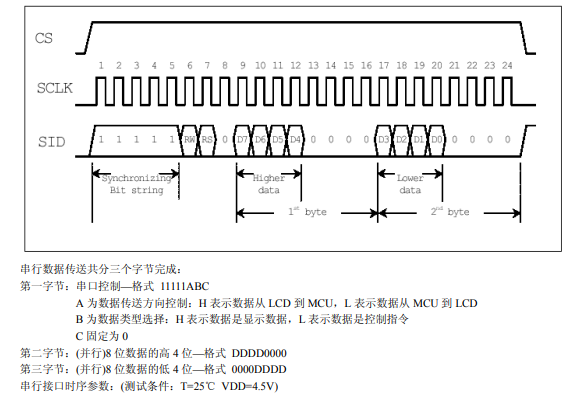


图4.1.2 LCD12864串行接口时序

### 实验步骤及练习

LCD液晶模块与GPIO引脚对应关系表：

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

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，显示班级，姓名学号。

指令集见《J12864中文液晶使用说明》

**练习:**

1. 通读源码，与指令表，结合按键显示不同字符信息。