STC32 库函数使用说明

2024.09.10

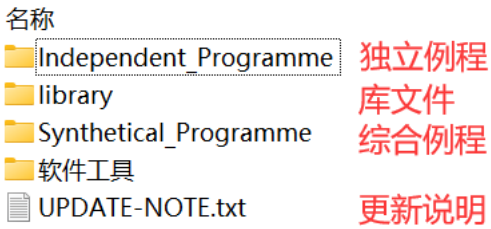
1 简介

为了方便初学者使用，快速的上手使用单片机进行开发，我们制作了这份库函数例程包，将单片机各个模块的寄存器配置通过函数封装起来，用户只要传递参数给函数并进行调用，就可以完成寄存器的配置，不用花太多的时间精力去研究单片机寄存器的功能和用法，极大的提升了开发速度。

2 例程包下载

使用浏览器输入官网链接：https://www.stcai.com/
通过导航栏的“软件工具”->“库函数”页面下载“32G 库函数”例程包：

下载例程包解压后根目录内容如下：

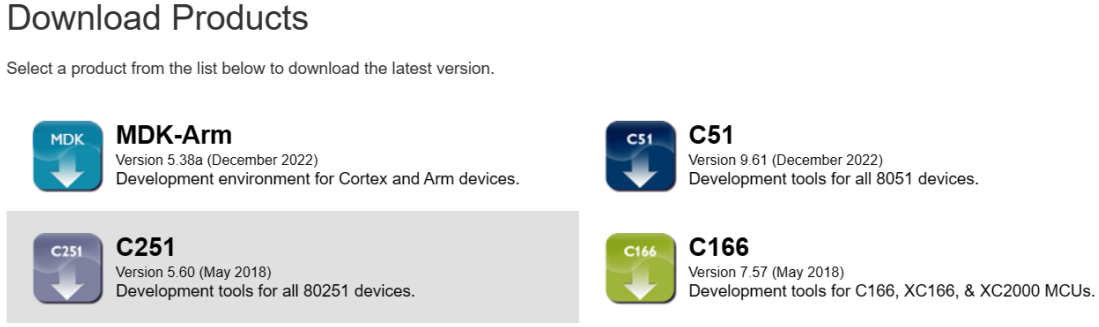


3 环境搭建

3.1 编译器安装

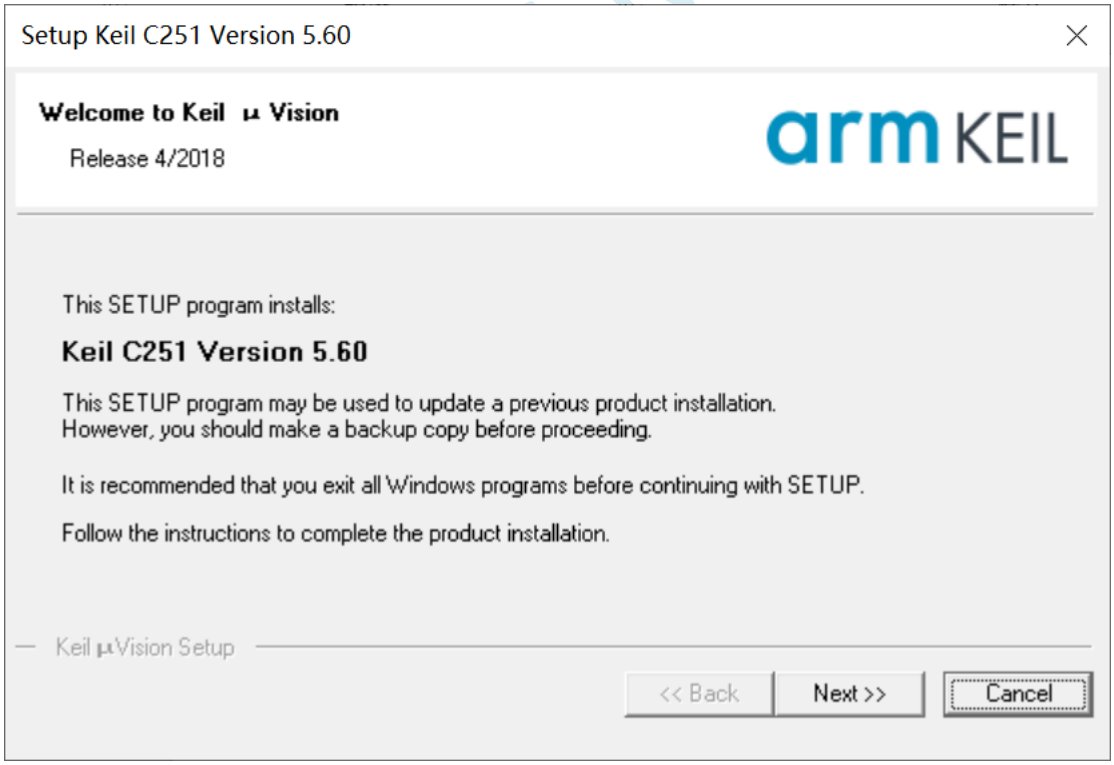
首先登录 Keil 官网，下载最新版的 C251 安装包，下载链接如下：

Keil Product Downloads

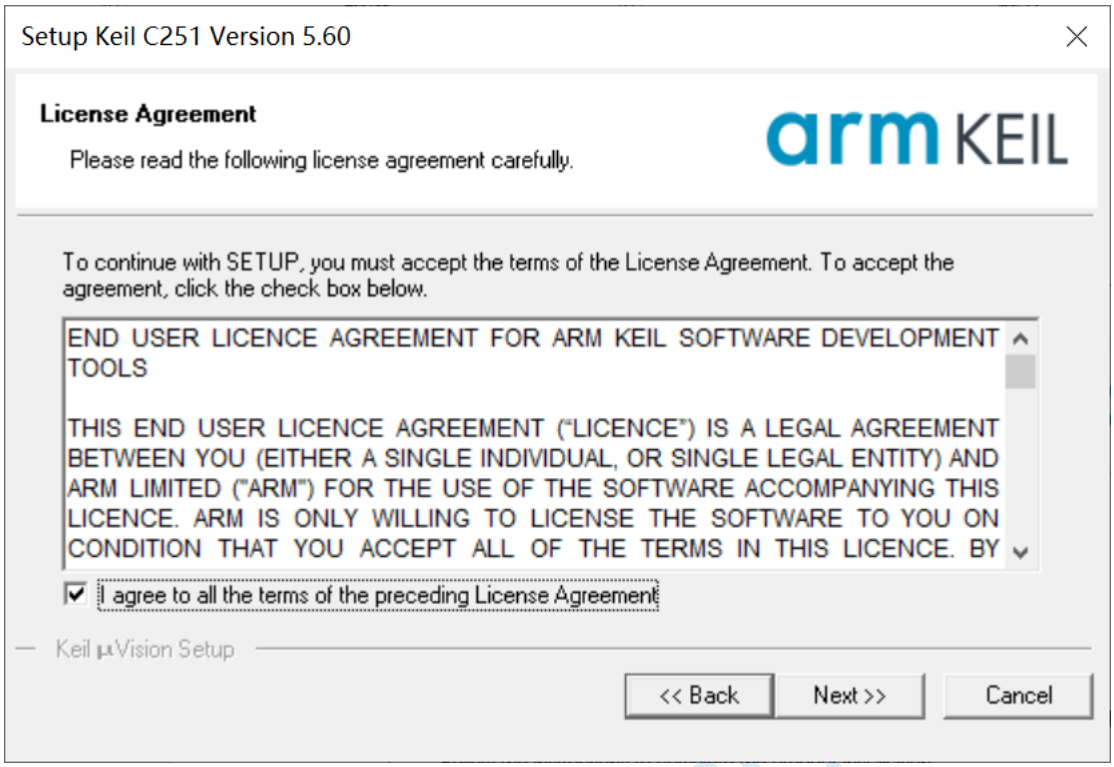


填写信息后，点确定进入下载页面进行下载。

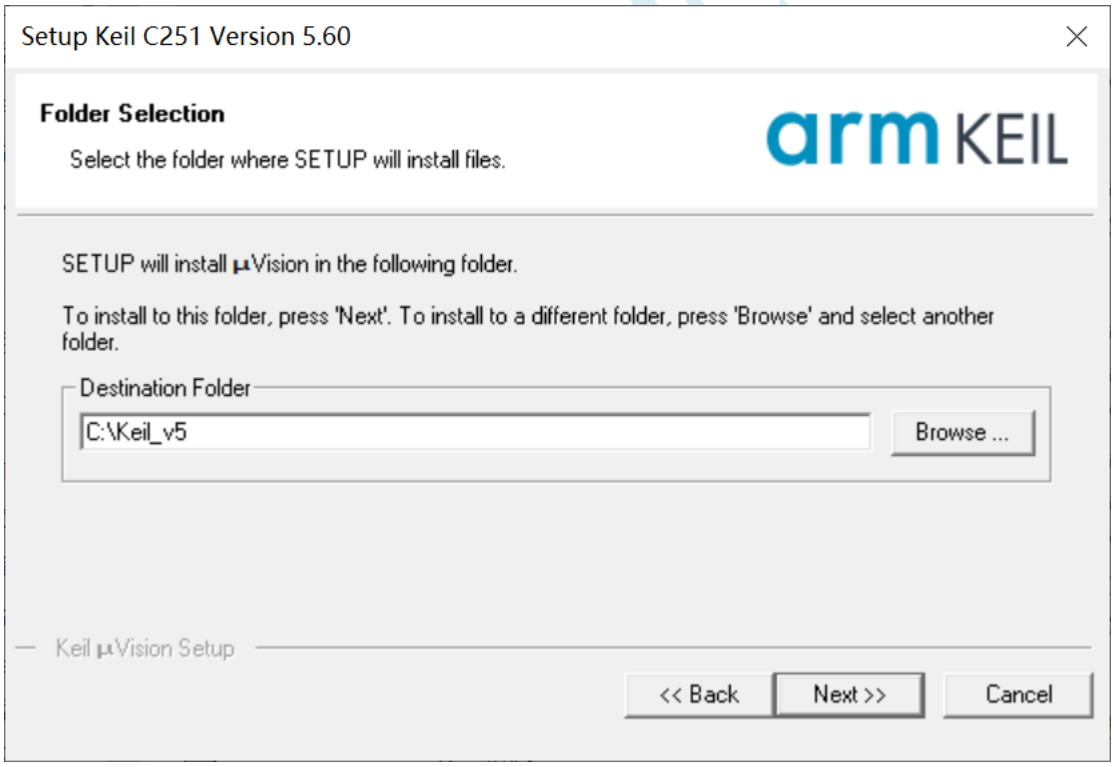
下载完成后，双击安装包开始安装，点击“Next”:



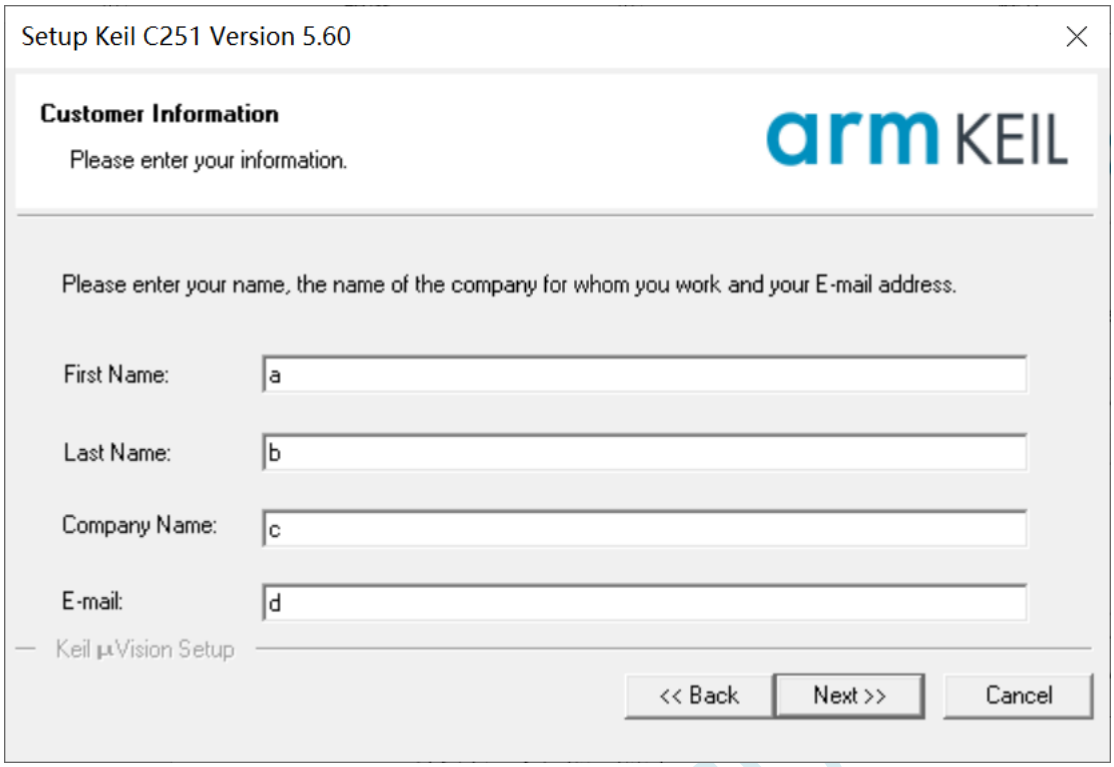
勾选“I agree to all the terms of the preceding License Agreement”,然后点击“Next”:



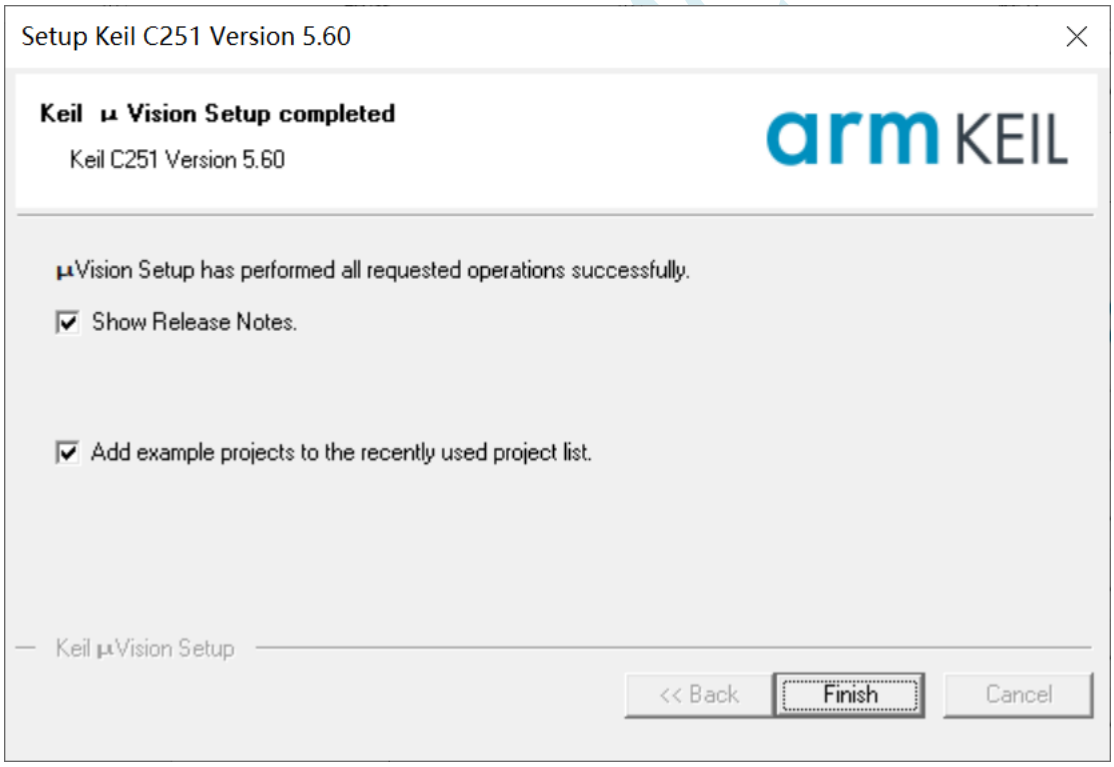
选择安装目录，然后点击“Next”:



填写个人信息，然后点击“Next”:



安装完成，点击“Finish”结束。

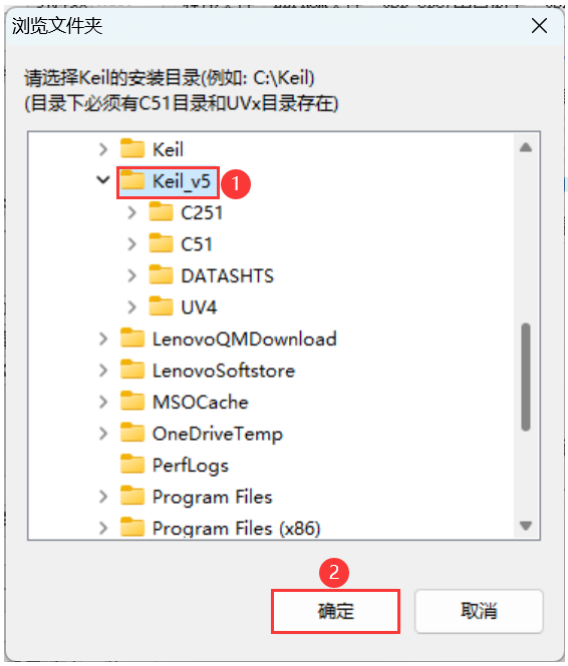


3.2 添加型号和头文件到 Keil

使用 Keil 之前需要先安装 STC 的仿真驱动。STC 的仿真驱动的安装步骤如下：

首先开 STC 的 ISP 下载软件，然后在软件右边功能区的“Keil 仿真设置”页面中点击“添加型号和头文件到 Keil 中 添加 STC 仿真器驱动到 Keil 中”按钮：

按下后会出现如下画面:



将目录定位到 Keil 软件的安装目录，然后确定。安装成功后会弹出如下的提示框:

即表示驱动正确安装了

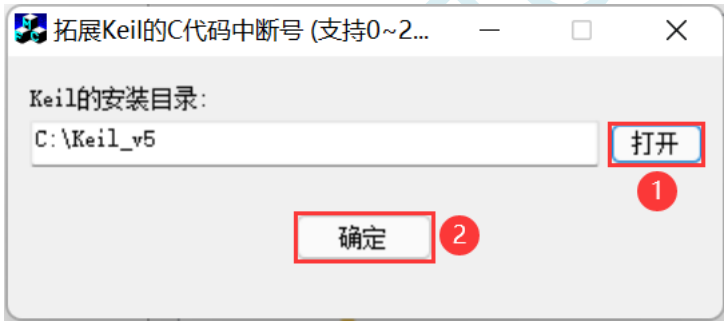
头文件默认复制到 Keil 安装目录下的“C251\INC\STC”目录中
在 C 代码中使用“#include <STC32G.H>”或者“#include "STC32G.H""进行包含均可正确使用

3.3 Keil 中断向量号拓展插件使用说明

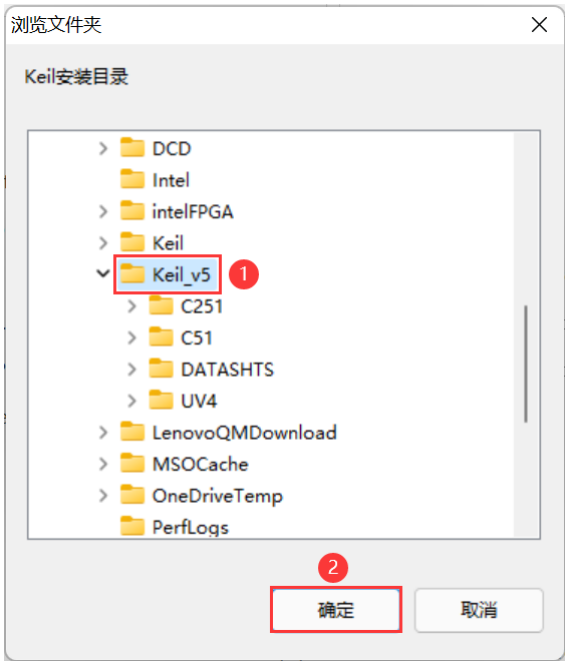
在 Keil 目前的 C51、C251 编译环境下，中断向量号只支持 0~31，即中断地址必须小于 0100H。程序里使用的中断向量号超过 31，编译时就会报错。随着芯片的功能越来越多、越来越强，STC 单片机的部分中断向量号已经超出 31。此前临时解决的方法是借用 13 号中断向量地址，通过汇编代码从当前中断地址跳转到 13 号中断地址执行。

此外，有网友提供了 keil 中断号拓展插件，安装到 C51、C251 目录下也可以解决中断向量号超 31 时编译报错的问题。

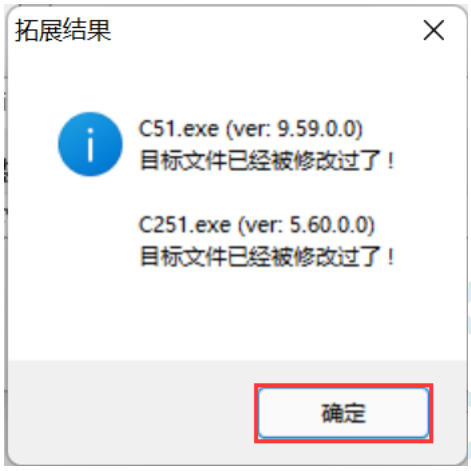
使用方法如下，打开例程包的“软件工具”文件夹，双击文件夹里面的可执行文件:



点击“打开”按钮选择 keil 的 C51/C251 编译器安装目录，然后点击确定按钮:



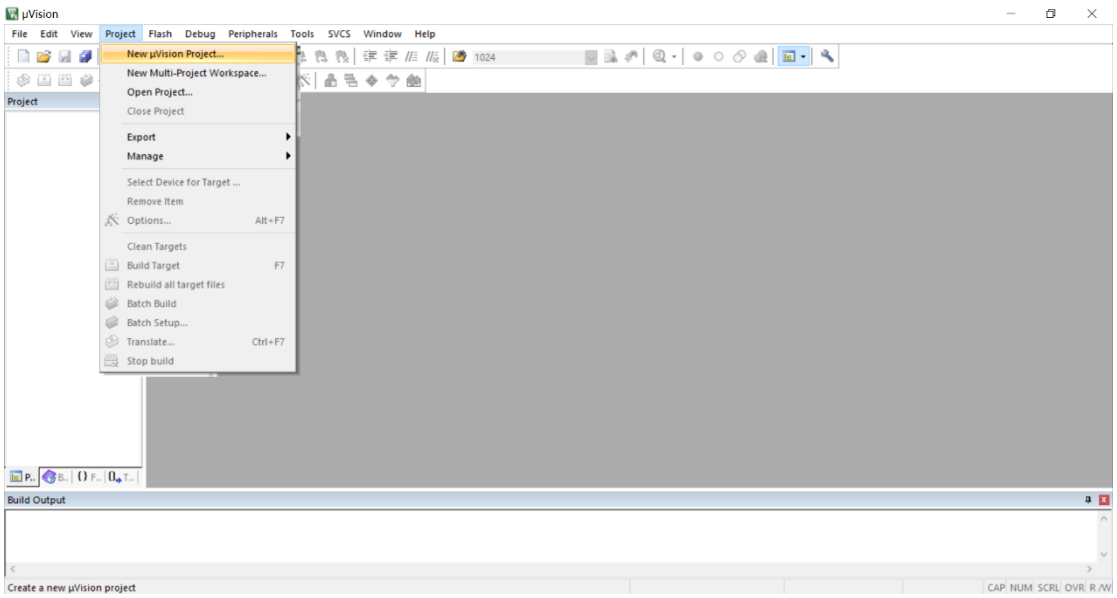
安装成功后显示如下提示信息：



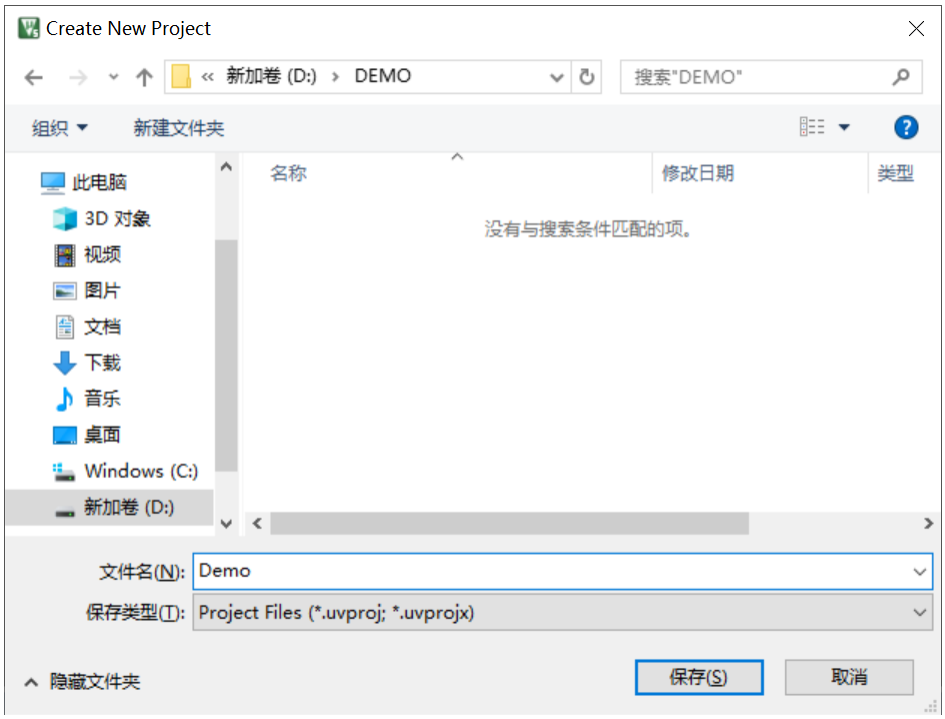
注意：部分旧版本 keil 编译器可能不支持此插件。

3.4 新建 Keil 项目

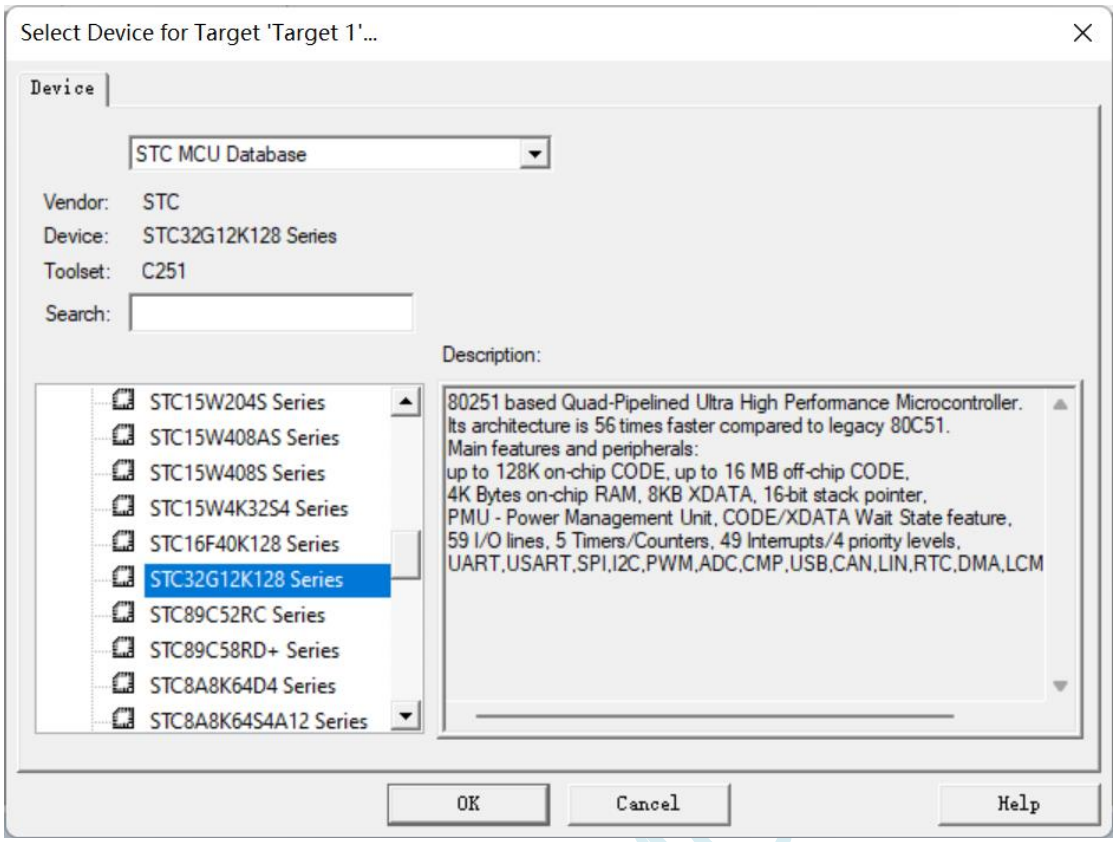
首先打开 Keil 软件，并打开“Project”菜单中的“New uVersion Project …”项



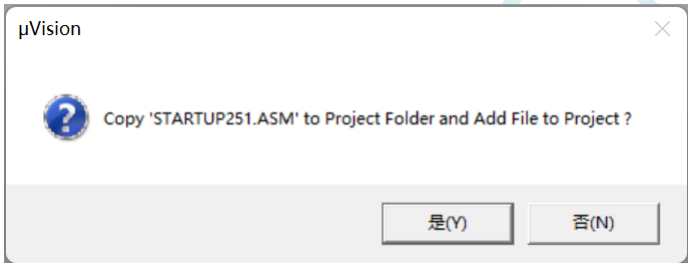
在下面的对话框中输入新建的项目名称，然后保存



接下来需要在如下的对话框内选择芯片型号（STC MCU Database 里如果找不到对应的芯片型号，可选用同系列的芯片）

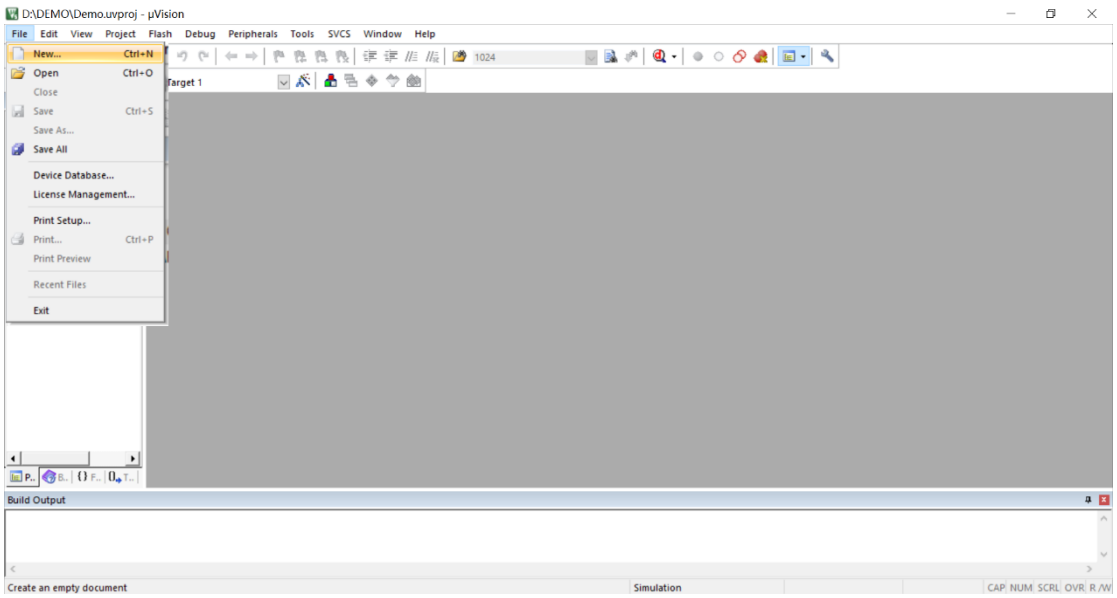


型号确定后，Keil 会弹出下面的对话框，问是否需要将启动代码文件添加到项目中。
可以选择“否”，也可选择“是”

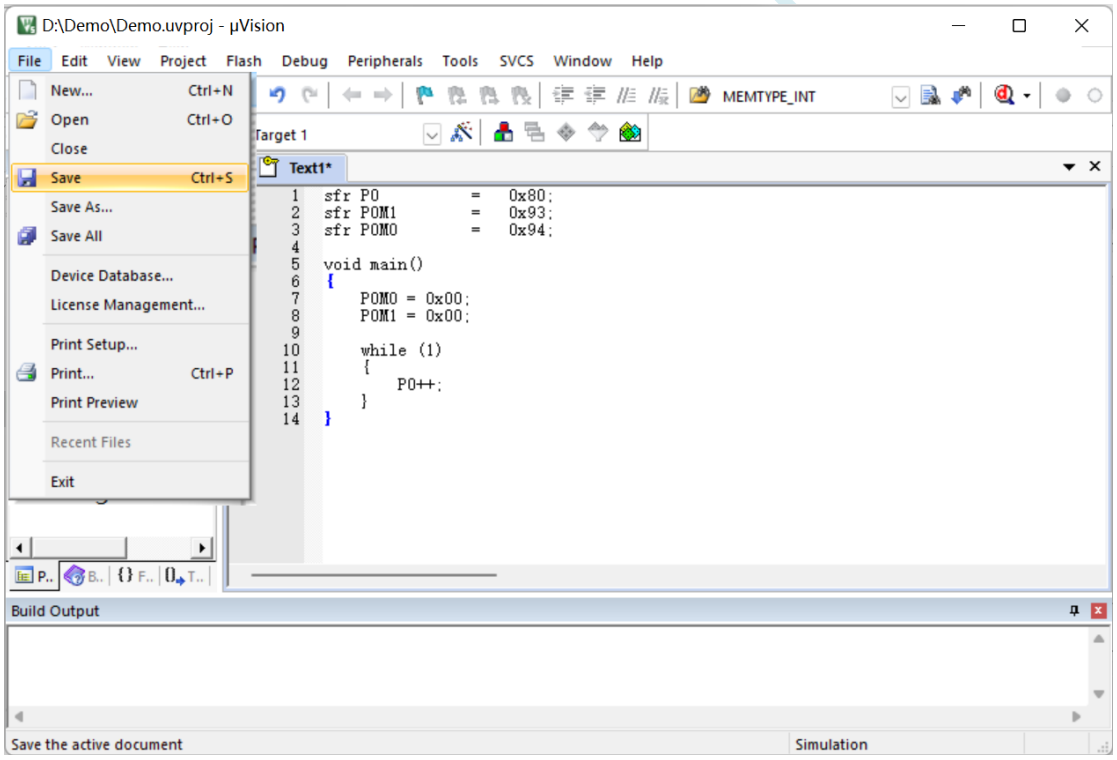


至此，基本的项目文件已基本建立。

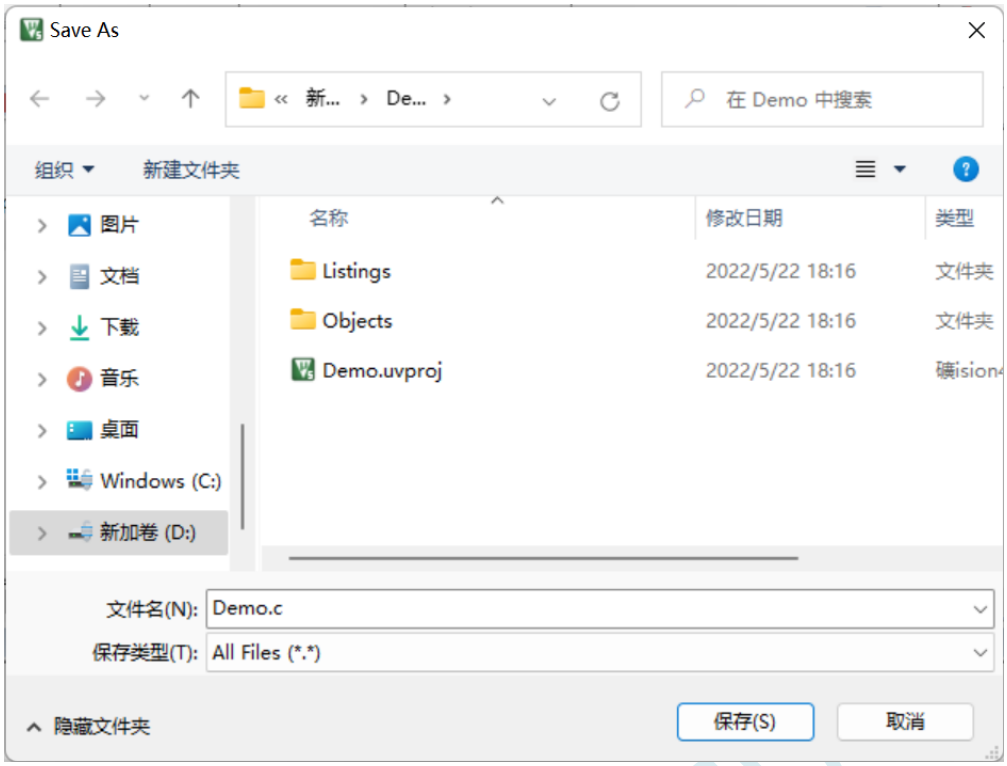
接下来需要新建源代码文件，打开“File”菜单中的“New …”项



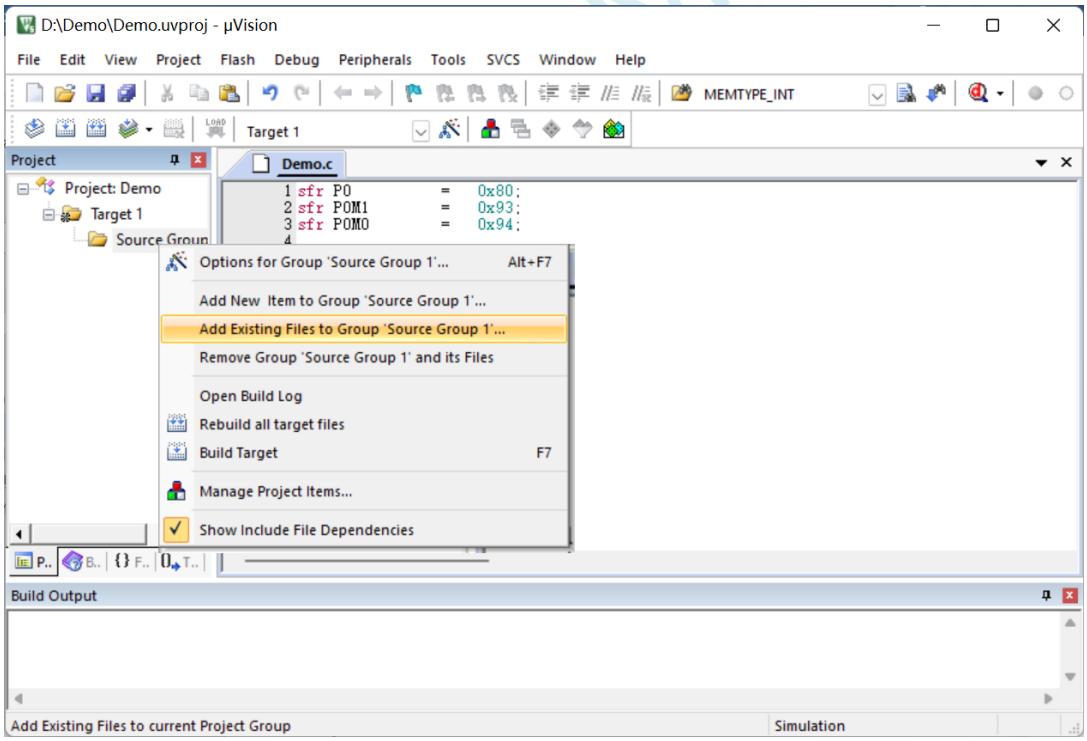
在新建的文件中输入相应的源代码，然后选择“File”菜单中的“Save”项对文件进行保存



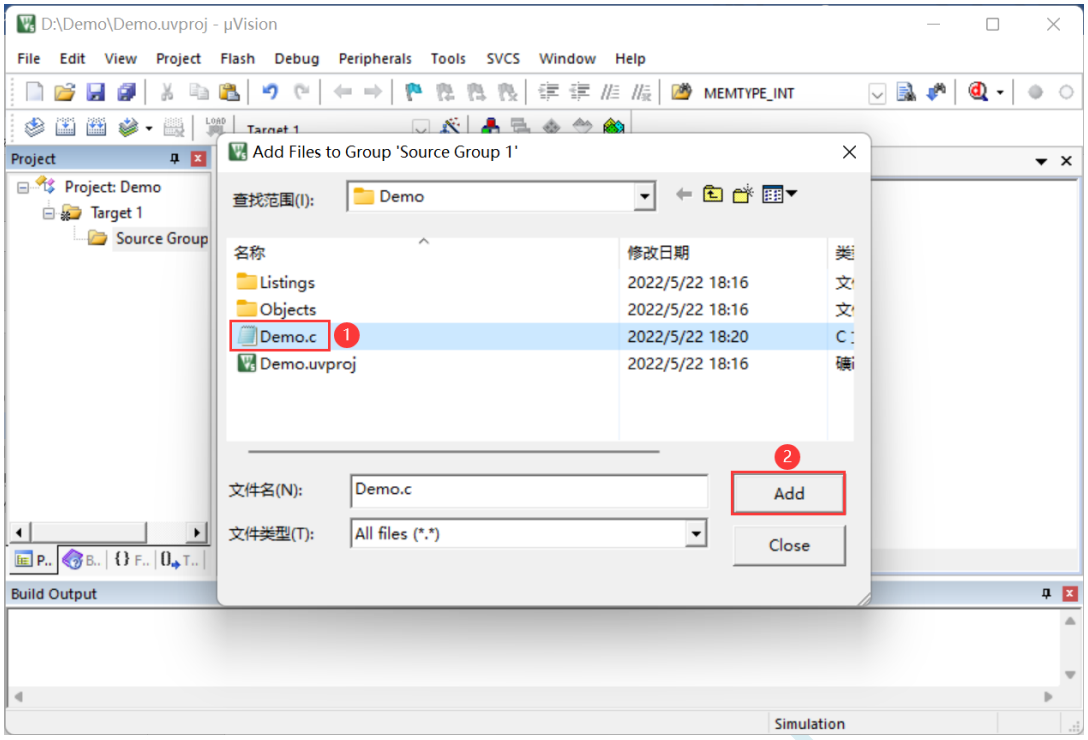
如下图



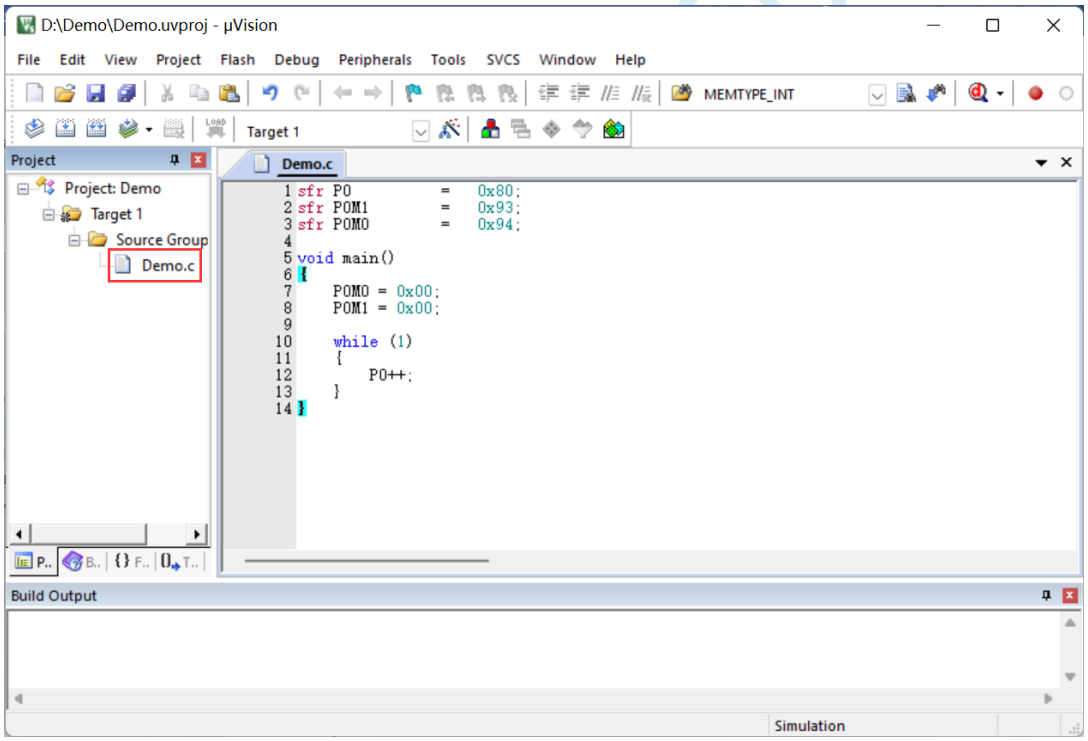
文件保存完成后需要使用下面的操作将源代码文件添加到项目中来，具体的操作方法是：使用鼠标右键单击“Project”列表中的“Source Group 1”项，在出现的右键菜单中选择“Add Existing Files to Group ‘Source Group 1’”项目



在下面的对话框中选择我们刚才保存的文件，并点击“Add”按钮即可将文件添加到项目中，完成后按下“Close”按钮关闭对话框

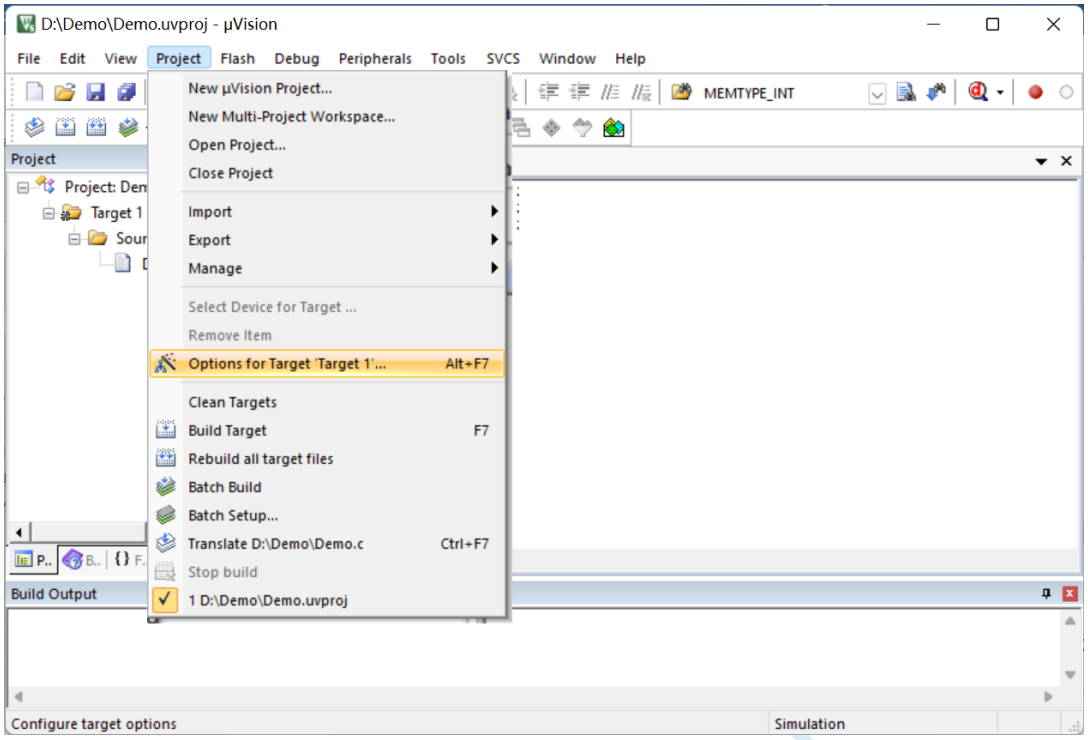


此时我们可以看到在项目中已经多了我们刚才添加的代码文件

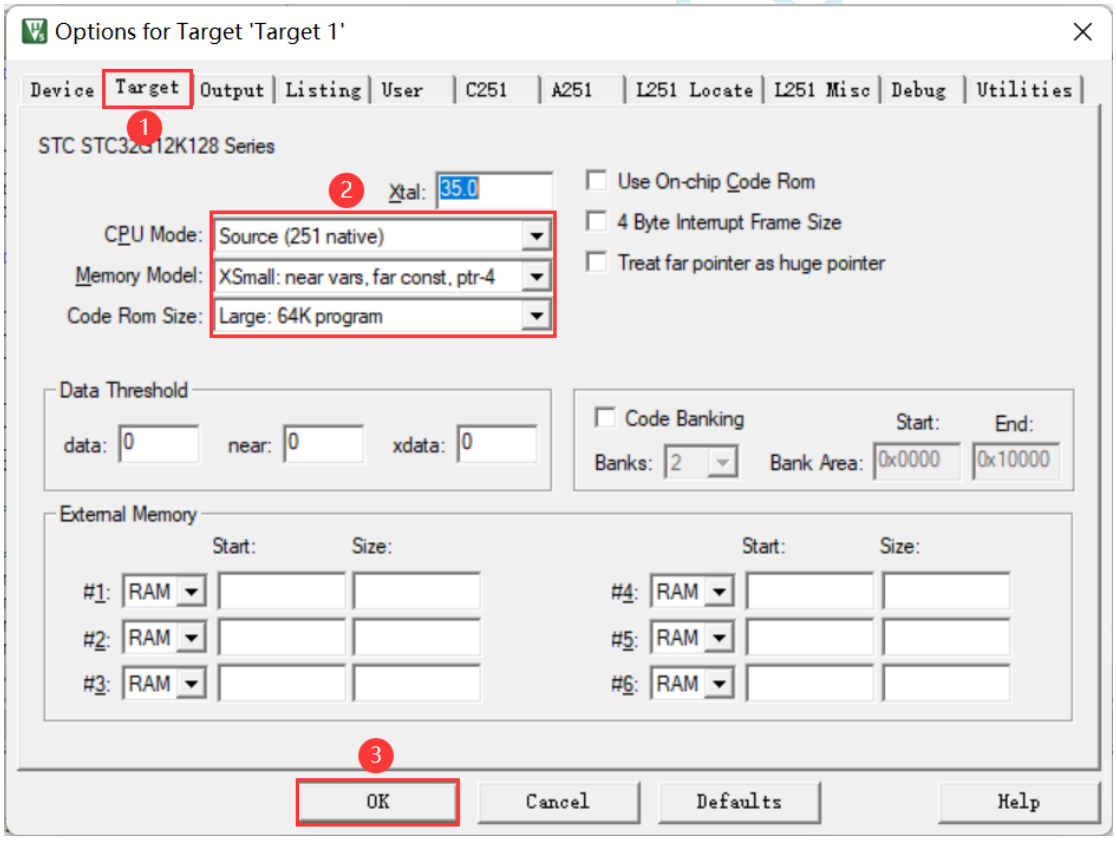


3.5 项目设置

通过 keil 工具栏的“Options for target...”按钮（快捷键“Alt+F7”）或者选择菜单“Project”中的“Option for Target ‘Target1’”，进入设置界面：



在如下的对话框中设置 CPU 模式、存储器模式、程序空间大小:



在“Memory Model”的下拉选项中选择“XSmall: ...”模式。80251 的存储器模式，在 Keil 环境下各种模式对比如下表:

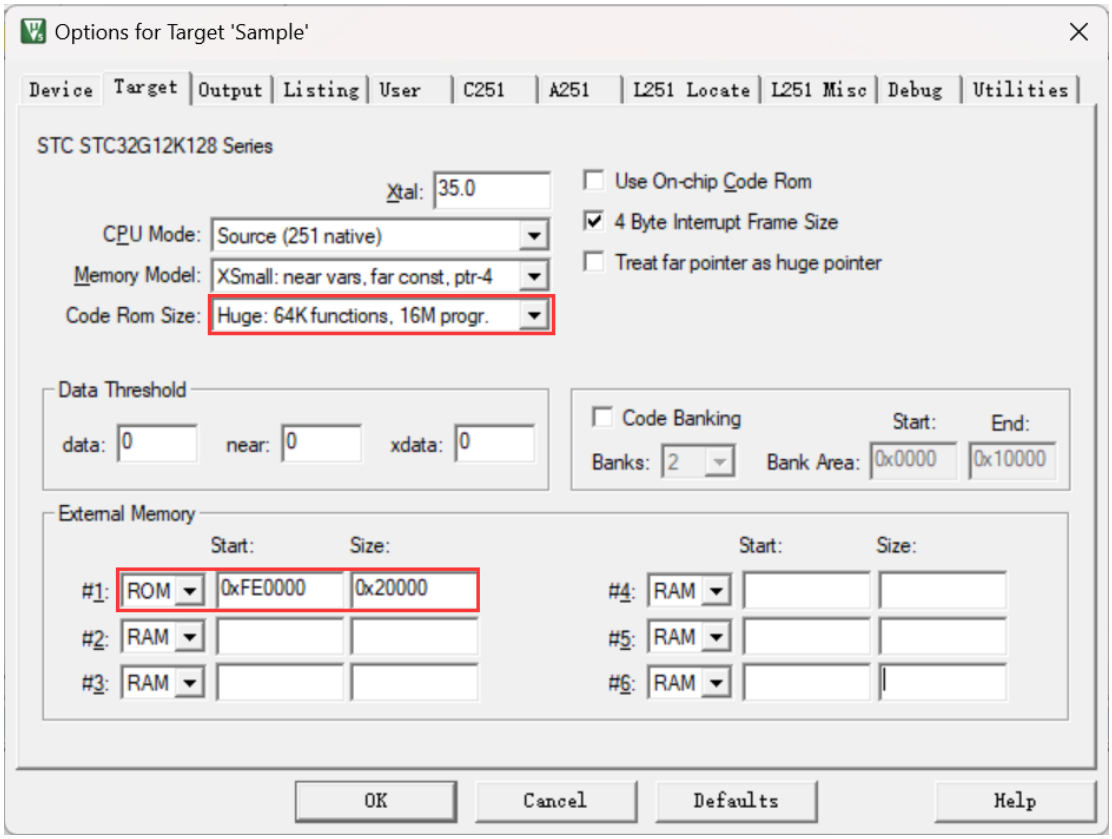
Memory Model	默认变量类型（数据存储器）	默认常量类型（程序存储器）	默认指针变量	指针访问范围
Tiny 模式	data	near	2 字节	00:0000 ~ 00:FFFF

XTiny 模式	edata	near	2 字节	00:0000 ~ 00:FFFF
Small 模式	data	far	4 字节	00:0000 ~ FF:FFFF
XSmall 模式	**edata**	**far**	**4 字节**	**00:0000 ~ FF:FFFF**
Large 模式	xdata	far	4 字节	00:0000 ~ FF:FFFF

由于 STC32G 的程序逻辑地址为 FE:0000H ~ FF:FFFFH，需要使用 24 位地址线才能正确访问，默认的常量类型（程序存储器类型）必须使用“far”类型，默认指针变量必须为 4 字节。

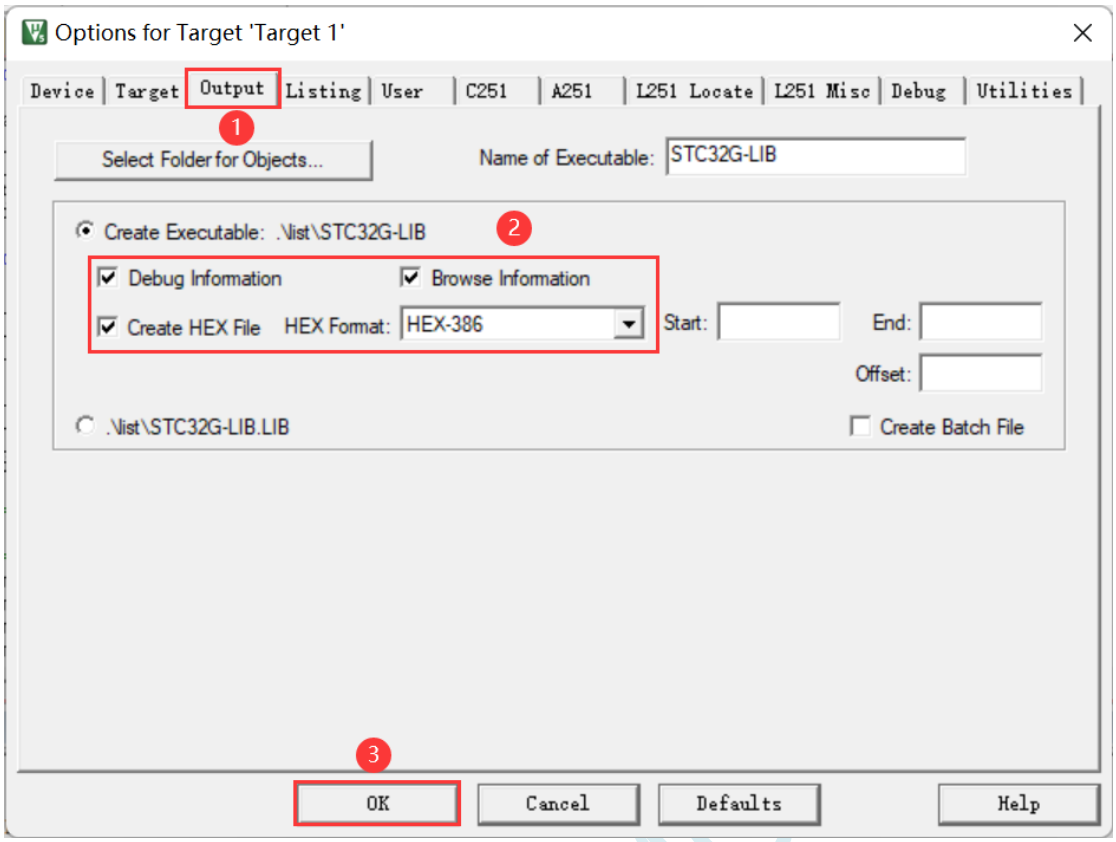
所以不建议使用“Small”“Tiny”和“XTiny”模式，推荐使用“XSmall”模式，这种模式默认将变量定义在内部 RAM(edata)，单时钟存取，访问速度快，且 STC32G12K128 系列芯片有 12K 的 edata 可以使用；使用“Small”模式时，默认将变量定义在内部 RAM(data)，单时钟存取，访问速度快（data 默认只有 128 字节，当用户对 RAM 需求超过 128 字节时，Keil 编译器会报错，此时用户需要将存储模式切换为 XSmall）数量有限，容易报错，所以不建议使用；不推荐使用“Large”模式，虽然该模式也能正确访问 STC32G 的全部 16M 寻址空间，但“Large”模式默认将变量定义在内部扩展 RAM(xdata)里面，存取需要 2~3 个时钟，访问速度慢

如果代码长度超过 64K，需选择“Huge”模式，参考以下配置方法：

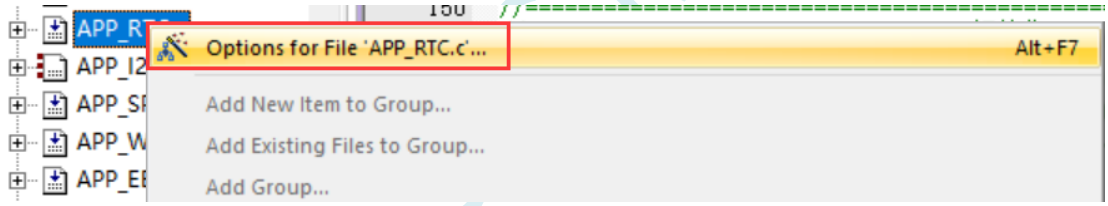


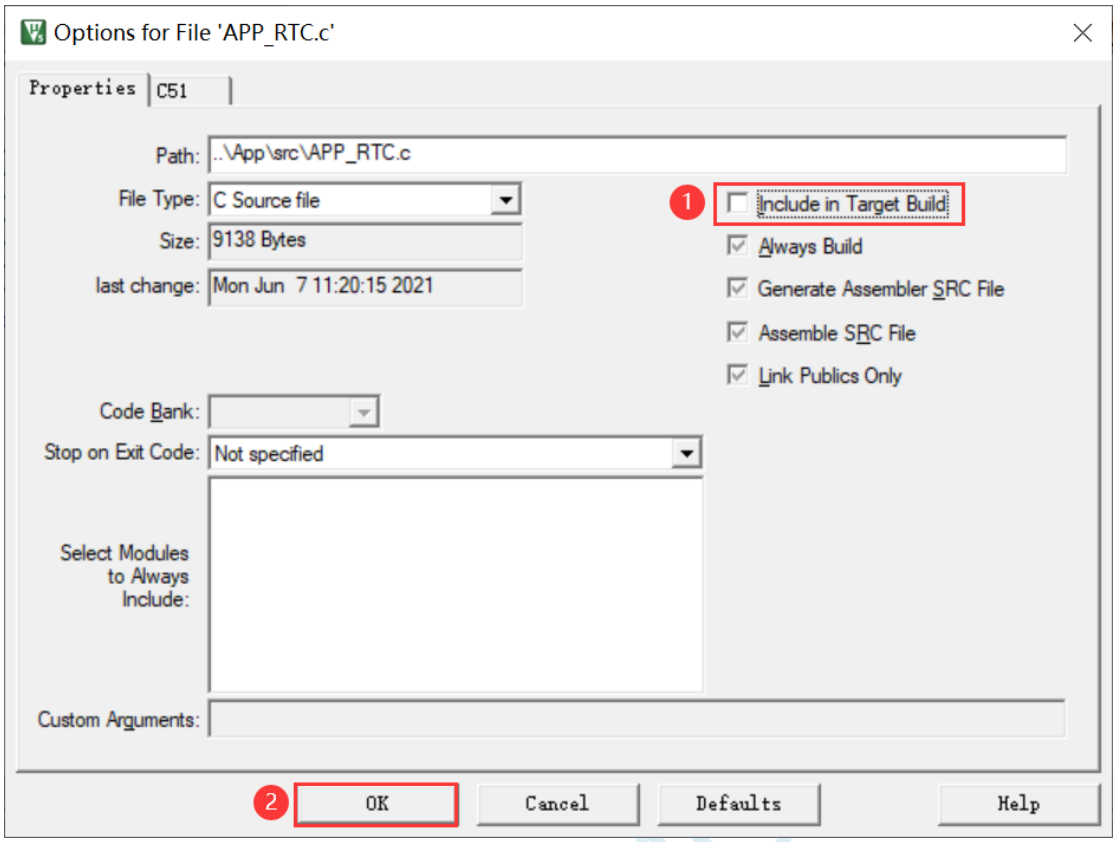
注意：设置的是 ROM 区域

在“Output”属性页中，将“Create HEX File”选项打上勾，即可在项目编译完成后自动生成 HEX 格式的目标文件，按“OK”保存。



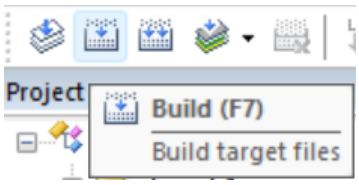
此外，还可以手动将一些没用到的文件设置为不参与编译，进一步降低资源消耗：

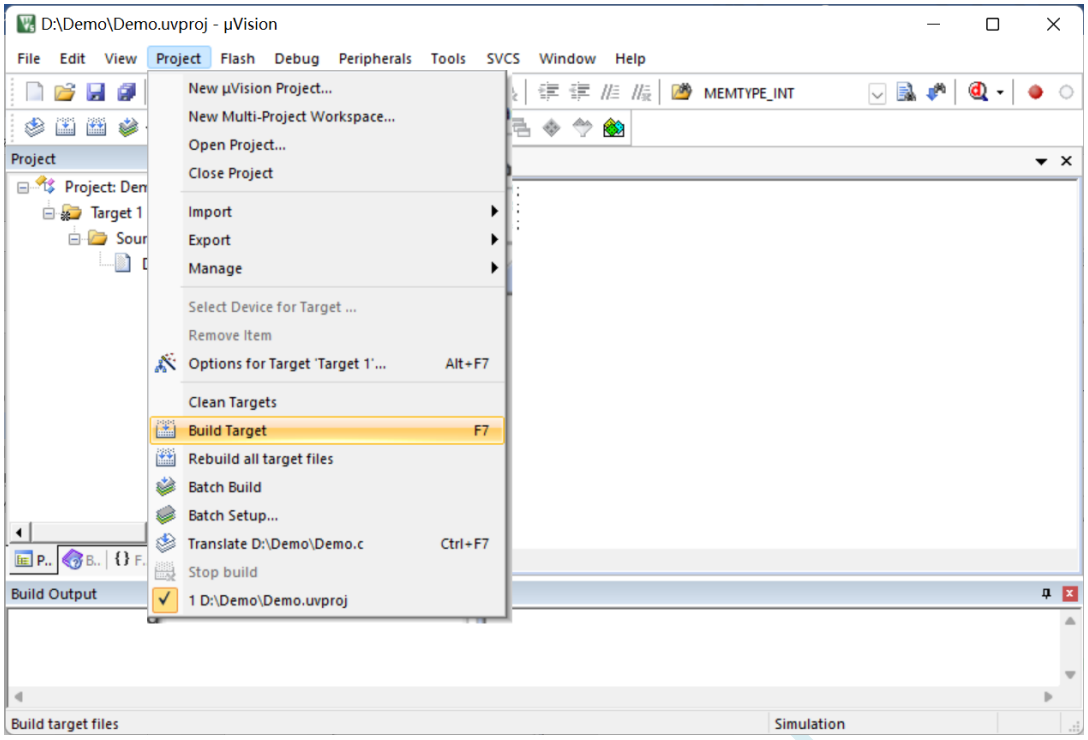




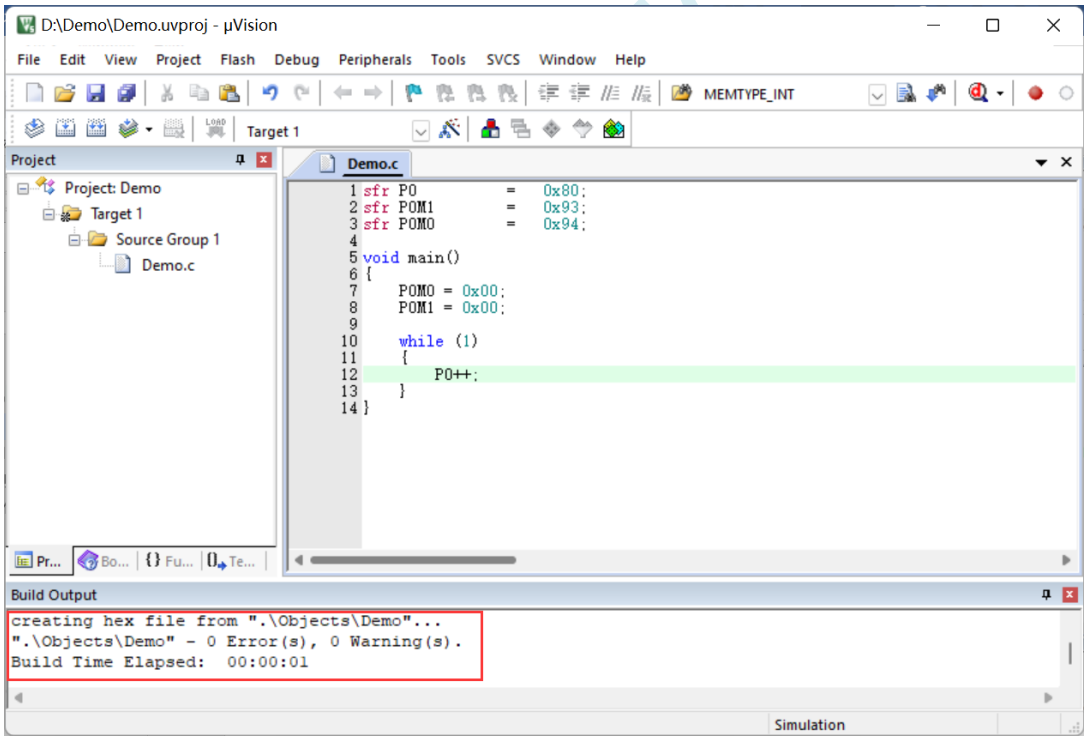
3.6 项目编译

按下快捷键“F7”或者选择菜单“Project”中的“Build Target”项对当前项目进行编译





若代码中没有错误，编译完成后则会在“Build Output”的信息输出框中显示“0 Error(s), 0 Warning(s)”,同时也会生成 HEX 的执行文件。到此创建项目完成。



4 库函数源文件介绍

例程包的“library”文件夹下存放着芯片各模块已实现的库文件，用户编程过程中可以从这个文件夹下复制程序所需的模块文件添加到项目里。

4.1 STC32G_GPIO

4.1.1 相关文件

“STC32G_GPIO.c”，主要用于 GPIO 功能的配置。
“STC32G_GPIO.h”，GPIO 所需的变量申明与宏定义。

4.1.2 IO 口初始化函数

函数名	u8 GPIO_Inilize(u8 GPIO, GPIO_InitTypeDef *GPIOx)
功能描述	初始化 IO 口
参数 1	GPIO: IO 口组号，取值 GPIO_P0~GPIO_P7
参数 2	GPIOx: 结构参数
返回	成功返回 SUCCESS，错误返回 FAIL

GPIOx: 结构参数定义：

```
typedef struct
{
    u8  Mode;
    u8  Pin;
} GPIO_InitTypeDef;
```

Mode: IO 模式设置

参数	功能描述
GPIO_PullUp	准双向口，内部弱上拉，可输入/输出，当输入时要先写 1
GPIO_HighZ	高阻输入，只能做输入
GPIO_OUT_OD	开漏输出，可输入/输出，输入/输出 1 时需要接上拉电阻
GPIO_OUT_PP	推挽输出，只能做输出，根据需要串接限流电阻

Pin: 要设置的端口

参数	功能描述
GPIO_Pin_0	IO 引脚 Px.0
GPIO_Pin_1	IO 引脚 Px.1
GPIO_Pin_2	IO 引脚 Px.2
GPIO_Pin_3	IO 引脚 Px.3
GPIO_Pin_4	IO 引脚 Px.4
GPIO_Pin_5	IO 引脚 Px.5
GPIO_Pin_6	IO 引脚 Px.6
GPIO_Pin_7	IO 引脚 Px.7

GPIO_Pin_LOW	Px 整组 IO 低 4 位引脚
GPIO_Pin_HIGH	Px 整组 IO 高 4 位引脚
GPIO_Pin_All	Px 整组 IO 8 位引脚

以上参数可以使用或运算，比如同时设置 Px.0，Px.1，Px.7：

GPIO_InitStructure.Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_7;

4.1.3 宏定义方式

参数参考上表：

1. 准双向口设置

```
P0_MODE_IO_PU（Pin）;    //设置 P0.x 口为准双向口
P1_MODE_IO_PU（Pin）;    //设置 P1.x 口为准双向口
P2_MODE_IO_PU（Pin）;    //设置 P2.x 口为准双向口
P3_MODE_IO_PU（Pin）;    //设置 P3.x 口为准双向口
P4_MODE_IO_PU（Pin）;    //设置 P4.x 口为准双向口
P5_MODE_IO_PU（Pin）;    //设置 P5.x 口为准双向口
P6_MODE_IO_PU（Pin）;    //设置 P6.x 口为准双向口
P7_MODE_IO_PU（Pin）;    //设置 P7.x 口为准双向口
```

2. 高阻输入设置

```
P0_MODE_IN_HIZ（Pin）;    //设置 P0.x 口为高阻输入
P1_MODE_IN_HIZ（Pin）;    //设置 P1.x 口为高阻输入
P2_MODE_IN_HIZ（Pin）;    //设置 P2.x 口为高阻输入
P3_MODE_IN_HIZ（Pin）;    //设置 P3.x 口为高阻输入
P4_MODE_IN_HIZ（Pin）;    //设置 P4.x 口为高阻输入
P5_MODE_IN_HIZ（Pin）;    //设置 P5.x 口为高阻输入
P6_MODE_IN_HIZ（Pin）;    //设置 P6.x 口为高阻输入
P7_MODE_IN_HIZ（Pin）;    //设置 P7.x 口为高阻输入
```

3. 开漏输出设置

```
P0_MODE_OUT_OD（Pin）; //设置 P0.x 口为开漏输出
P1_MODE_OUT_OD（Pin）; //设置 P1.x 口为开漏输出
P2_MODE_OUT_OD（Pin）; //设置 P2.x 口为开漏输出
P3_MODE_OUT_OD（Pin）; //设置 P3.x 口为开漏输出
P4_MODE_OUT_OD（Pin）; //设置 P4.x 口为开漏输出
P5_MODE_OUT_OD（Pin）; //设置 P5.x 口为开漏输出
P6_MODE_OUT_OD（Pin）; //设置 P6.x 口为开漏输出
P7_MODE_OUT_OD（Pin）; //设置 P7.x 口为开漏输出
```

4. 推挽输出设置

```
P0_MODE_OUT_PP（Pin）; //设置 P0.x 口为推挽输出
P1_MODE_OUT_PP（Pin）; //设置 P1.x 口为推挽输出
P2_MODE_OUT_PP（Pin）; //设置 P2.x 口为推挽输出
P3_MODE_OUT_PP（Pin）; //设置 P3.x 口为推挽输出
P4_MODE_OUT_PP（Pin）; //设置 P4.x 口为推挽输出
P5_MODE_OUT_PP（Pin）; //设置 P5.x 口为推挽输出
P6_MODE_OUT_PP（Pin）; //设置 P6.x 口为推挽输出
P7_MODE_OUT_PP（Pin）; //设置 P7.x 口为推挽输出
```

5. 内部 4.1K 上拉设置

```
P0_PULL_UP_ENABLE（Pin）; //使能 P0.x 内部 4.1K 上拉
P1_PULL_UP_ENABLE（Pin）; //使能 P1.x 内部 4.1K 上拉
P2_PULL_UP_ENABLE（Pin）; //使能 P2.x 内部 4.1K 上拉
P3_PULL_UP_ENABLE（Pin）; //使能 P3.x 内部 4.1K 上拉
P4_PULL_UP_ENABLE（Pin）; //使能 P4.x 内部 4.1K 上拉
P5_PULL_UP_ENABLE（Pin）; //使能 P5.x 内部 4.1K 上拉
P6_PULL_UP_ENABLE（Pin）; //使能 P6.x 内部 4.1K 上拉
P7_PULL_UP_ENABLE（Pin）; //使能 P7.x 内部 4.1K 上拉

P0_PULL_UP_DISABLE（Pin）; //禁止 P0.x 内部 4.1K 上拉
P1_PULL_UP_DISABLE（Pin）; //禁止 P1.x 内部 4.1K 上拉
P2_PULL_UP_DISABLE（Pin）; //禁止 P2.x 内部 4.1K 上拉
P3_PULL_UP_DISABLE（Pin）; //禁止 P3.x 内部 4.1K 上拉
P4_PULL_UP_DISABLE（Pin）; //禁止 P4.x 内部 4.1K 上拉
P5_PULL_UP_DISABLE（Pin）; //禁止 P5.x 内部 4.1K 上拉
P6_PULL_UP_DISABLE（Pin）; //禁止 P6.x 内部 4.1K 上拉
P7_PULL_UP_DISABLE（Pin）; //禁止 P7.x 内部 4.1K 上拉
```

6. 施密特触发设置

```
P0_ST_ENABLE（Pin）;      //使能 P0.x 施密特触发
P1_ST_ENABLE（Pin）;      //使能 P1.x 施密特触发
P2_ST_ENABLE（Pin）;      //使能 P2.x 施密特触发
P3_ST_ENABLE（Pin）;      //使能 P3.x 施密特触发
P4_ST_ENABLE（Pin）;      //使能 P4.x 施密特触发
P5_ST_ENABLE（Pin）;      //使能 P5.x 施密特触发
P6_ST_ENABLE（Pin）;      //使能 P6.x 施密特触发
P7_ST_ENABLE（Pin）;      //使能 P7.x 施密特触发

P0_ST_DISABLE（Pin）;     //禁止 P0.x 施密特触发
P1_ST_DISABLE（Pin）;     //禁止 P1.x 施密特触发
P2_ST_DISABLE（Pin）;     //禁止 P2.x 施密特触发
P3_ST_DISABLE（Pin）;     //禁止 P3.x 施密特触发
P4_ST_DISABLE（Pin）;     //禁止 P4.x 施密特触发
P5_ST_DISABLE（Pin）;     //禁止 P5.x 施密特触发
P6_ST_DISABLE（Pin）;     //禁止 P6.x 施密特触发
P7_ST_DISABLE（Pin）;     //禁止 P7.x 施密特触发
```

7. 端口电平转换速度设置

```
P0_SPEED_LOW（Pin）;      // P0.x 电平转换慢速，相应的上下冲比较小
P1_SPEED_LOW（Pin）;      // P1.x 电平转换慢速，相应的上下冲比较小
P2_SPEED_LOW（Pin）;      // P2.x 电平转换慢速，相应的上下冲比较小
P3_SPEED_LOW（Pin）;      // P3.x 电平转换慢速，相应的上下冲比较小
P4_SPEED_LOW（Pin）;      // P4.x 电平转换慢速，相应的上下冲比较小
P5_SPEED_LOW（Pin）;      // P5.x 电平转换慢速，相应的上下冲比较小
P6_SPEED_LOW（Pin）;      // P6.x 电平转换慢速，相应的上下冲比较小
P7_SPEED_LOW（Pin）;      // P7.x 电平转换慢速，相应的上下冲比较小
```

```
P0_SPEED_HIGH（Pin）;     // P0.x 电平转换快速，相应的上下冲比较大
P1_SPEED_HIGH（Pin）;     // P1.x 电平转换快速，相应的上下冲比较大
P2_SPEED_HIGH（Pin）;     // P2.x 电平转换快速，相应的上下冲比较大
P3_SPEED_HIGH（Pin）;     // P3.x 电平转换快速，相应的上下冲比较大
P4_SPEED_HIGH（Pin）;     // P4.x 电平转换快速，相应的上下冲比较大
P5_SPEED_HIGH（Pin）;     // P5.x 电平转换快速，相应的上下冲比较大
P6_SPEED_HIGH（Pin）;     // P6.x 电平转换快速，相应的上下冲比较大
P7_SPEED_HIGH（Pin）;     // P7.x 电平转换快速，相应的上下冲比较大
```

8. 端口驱动电流控制设置

```
P0_DRIVE_MEDIUM（Pin）; // 设置 P0.x 一般驱动能力
P1_DRIVE_MEDIUM（Pin）; // 设置 P1.x 一般驱动能力
P2_DRIVE_MEDIUM（Pin）; // 设置 P2.x 一般驱动能力
P3_DRIVE_MEDIUM（Pin）; // 设置 P3.x 一般驱动能力
P4_DRIVE_MEDIUM（Pin）; // 设置 P4.x 一般驱动能力
P5_DRIVE_MEDIUM（Pin）; // 设置 P5.x 一般驱动能力
P6_DRIVE_MEDIUM（Pin）; // 设置 P6.x 一般驱动能力
P7_DRIVE_MEDIUM（Pin）; // 设置 P7.x 一般驱动能力

P0_DRIVE_HIGH（Pin）;     // 设置 P0.x 增强驱动能力
P1_DRIVE_HIGH（Pin）;     // 设置 P1.x 增强驱动能力
P2_DRIVE_HIGH（Pin）;     // 设置 P2.x 增强驱动能力
P3_DRIVE_HIGH（Pin）;     // 设置 P3.x 增强驱动能力
P4_DRIVE_HIGH（Pin）;     // 设置 P4.x 增强驱动能力
P5_DRIVE_HIGH（Pin）;     // 设置 P5.x 增强驱动能力
P6_DRIVE_HIGH（Pin）;     // 设置 P6.x 增强驱动能力
P7_DRIVE_HIGH（Pin）;     // 设置 P7.x 增强驱动能力
```

9. 端口数字信号输入使能

```
P0_DIGIT_IN_ENABLE（Pin）;     // 使能 P0.x 数字信号输入
P1_DIGIT_IN_ENABLE（Pin）;     // 使能 P1.x 数字信号输入
P2_DIGIT_IN_ENABLE（Pin）;     // 使能 P2.x 数字信号输入
P3_DIGIT_IN_ENABLE（Pin）;     // 使能 P3.x 数字信号输入
P4_DIGIT_IN_ENABLE（Pin）;     // 使能 P4.x 数字信号输入
P5_DIGIT_IN_ENABLE（Pin）;     // 使能 P5.x 数字信号输入
P6_DIGIT_IN_ENABLE（Pin）;     // 使能 P6.x 数字信号输入
P7_DIGIT_IN_ENABLE（Pin）;     // 使能 P7.x 数字信号输入

P0_DIGIT_IN_DISABLE（Pin）;    // 禁止 P0.x 数字信号输入
P1_DIGIT_IN_DISABLE（Pin）;    // 禁止 P1.x 数字信号输入
P2_DIGIT_IN_DISABLE（Pin）;    // 禁止 P2.x 数字信号输入
P3_DIGIT_IN_DISABLE（Pin）;    // 禁止 P3.x 数字信号输入
P4_DIGIT_IN_DISABLE（Pin）;    // 禁止 P4.x 数字信号输入
P5_DIGIT_IN_DISABLE（Pin）;    // 禁止 P5.x 数字信号输入
P6_DIGIT_IN_DISABLE（Pin）;    // 禁止 P6.x 数字信号输入
P7_DIGIT_IN_DISABLE（Pin）;    // 禁止 P7.x 数字信号输入
```

4.2 STC32G_NVIC 中断系统

4.2.1 相关文件

“STC32G_NVIC.c”， 主要用于 NVIC 功能的配置。
“STC32G_NVIC.h”， NVIC 所需的变量申明与宏定义。

宏定义

```
#define  FALLING_EDGE      1      //产生下降沿中断
#define  RISING_EDGE       2      //产生上升沿中断
```

State: 中断使能状态

参数	功能描述
ENABLE	使能中断
DISABLE	禁止中断

Priority: 中断优先级

参数	功能描述
Priority_0	中断优先级为 0 级（最低级）
Priority_1	中断优先级为 1 级（较低级）
Priority_2	中断优先级为 2 级（较高级）
Priority_3	中断优先级为 3 级（最高级）

4.2.2 Timer0 嵌套向量中断

函数名	u8 NVIC_Timer0_Init(u8 State, u8 Priority)
功能描述	Timer0 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS，错误返回 FAIL

4.2.3 Timer1 嵌套向量中断

函数名	u8 NVIC_Timer1_Init(u8 State, u8 Priority)
功能描述	Timer1 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS，错误返回 FAIL

4.2.4 Timer2 嵌套向量中断

函数名	u8 NVIC_Timer2_Init(u8 State, u8 Priority)
功能描述	Timer2 嵌套向量中断控制器初始化

参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.5 Timer3 嵌套向量中断

函数名	u8 NVIC_Timer3_Init(u8 State, u8 Priority)
功能描述	Timer3 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.6 Timer4 嵌套向量中断

函数名	u8 NVIC_Timer4_Init(u8 State, u8 Priority)
功能描述	Timer4 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.7 INT0 嵌套向量中断

函数名	u8 NVIC_INT0_Init(u8 State, u8 Priority)
功能描述	INT0 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.8 INT1 嵌套向量中断

函数名	u8 NVIC_INT1_Init(u8 State, u8 Priority)
功能描述	INT1 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.9 INT2 嵌套向量中断

函数名	u8 NVIC_INT2_Init(u8 State, u8 Priority)
功能描述	INT2 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.10 INT3 嵌套向量中断

函数名	u8 NVIC_INT3_Init(u8 State, u8 Priority)
功能描述	INT3 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.11 INT4 嵌套向量中断

函数名	u8 NVIC_INT4_Init(u8 State, u8 Priority)
功能描述	INT4 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.12 ADC 嵌套向量中断

函数名	u8 NVIC_ADC_Init(u8 State, u8 Priority)
功能描述	ADC 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.13 CMP 嵌套向量中断

函数名	u8 NVIC_CMP_Init(u8 State, u8 Priority)
功能描述	比较器嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, RISING_EDGE/FALLING_EDGE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

State: 中断使能状态

参数	功能描述
DISABLE	禁止中断
RISING_EDGE	使能上升沿中断
FALLING_EDGE	使能下降沿中断

4.2.14 I2C 嵌套向量中断

函数名	u8 NVIC_I2C_Init(u8 State, u8 Priority)
功能描述	I2C 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, I2C_Mode_Master: ENABLE/DISABLE I2C_Mode_Slave: I2C_ESTAI/I2C_ERXI/I2C_ETXI/I2C_ESTOI/DISABLE

参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

State: 中断使能状态

参数		功能描述
主机模式	ENABLE	使能中断
	DISABLE	禁止中断
从机模式	I2C_ESTAI	从机接收 START 信号中断
	I2C_ERXI	从机接收 1 字节数据中断
	I2C_ETXI	从机发送 1 字节数据中断
	I2C_ESTOI	从机接收 STOP 信号中断
	DISABLE	禁止中断

4.2.15 UART1 嵌套向量中断

函数名	u8 NVIC_UART1_Init(u8 State, u8 Priority)
功能描述	UART1 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.16 UART2 嵌套向量中断

函数名	u8 NVIC_UART2_Init(u8 State, u8 Priority)
功能描述	UART2 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.17 UART3 嵌套向量中断

函数名	u8 NVIC_UART3_Init(u8 State, u8 Priority)
功能描述	UART3 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.18 UART4 嵌套向量中断

函数名	u8 NVIC_UART4_Init(u8 State, u8 Priority)
功能描述	UART4 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.19 SPI 嵌套向量中断

函数名	u8 NVIC_SPI_Init(u8 State, u8 Priority)
功能描述	SPI 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.20 DMA ADC 嵌套向量中断

函数名	u8 NVIC_DMA_ADC_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA ADC 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.21 DMA M2M 嵌套向量中断

函数名	u8 NVIC_DMA_M2M_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA M2M 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.22 DMA SPI 嵌套向量中断

函数名	u8 NVIC_DMA_SPI_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA SPI 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.23 DMA UART1 Tx 嵌套向量中断

函数名	u8 NVIC_DMA_UART1_Tx_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA UART1 Tx 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.24 DMA UART1 Rx 嵌套向量中断

函数名	u8 NVIC_DMA_UART1_Rx_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA UART1 Rx 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.25 DMA UART2 Tx 嵌套向量中断

函数名	u8 NVIC_DMA_UART2_Tx_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA UART2 Tx 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.26 DMA UART2 Rx 嵌套向量中断

函数名	u8 NVIC_DMA_UART2_Rx_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA UART2 Rx 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.27 DMA UART3 Tx 嵌套向量中断

函数名	u8 NVIC_DMA_UART3_Tx_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA UART3 Tx 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.28 DMA UART3 Rx 嵌套向量中断

函数名	u8 NVIC_DMA_UART3_Rx_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA UART3 Rx 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3

返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.29 DMA UART4 Tx 嵌套向量中断

函数名	u8 NVIC_DMA_UART4_Tx_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA UART4 Tx 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.30 DMA UART4 Rx 嵌套向量中断

函数名	u8 NVIC_DMA_UART4_Rx_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA UART4 Rx 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.31 DMA LCM 嵌套向量中断

函数名	u8 NVIC_DMA_LCM_Init(u8 State, u8 Priority, u8 Bus_Priority)
功能描述	DMA LCM 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
参数 3	Bus_Priority: 数据总线访问优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.2.32 LCM 嵌套向量中断

函数名	u8 NVIC_LCM_Init(u8 State, u8 Priority)
功能描述	LCM 嵌套向量中断控制器初始化
参数 1	State: 中断使能状态, ENABLE/DISABLE
参数 2	Priority: 中断优先级, Priority_0,Priority_1,Priority_2,Priority_3
返回	成功返回 SUCCESS, 错误返回 FAIL

4.3 STC32G_Exti 外部中断

4.3.1 相关文件

“STC32G_Exti.c”， 主要用于外部中断功能的配置。
“STC32G_Exti_Isr.c”, 主要包含外部中断的中断函数。
“STC32G_Exti.h”， 外部中断所需的变量申明与宏定义。

4.3.2 外部中断初始化函数

函数名	u8 Ext_Inilize(u8 EXT, EXTI_InitTypeDef *INTx)
功能描述	外部中断初始化程序
参数 1	EXT: 外部中断号。只有 INT0，INT1 可设置中断模式，其它默认下降沿中断。
参数 2	INTx: 结构参数
返回	成功返回 SUCCESS，错误返回 FAIL

INTx: 结构参数定义：

```
typedef struct
{
    u8  EXTI_Mode;
} EXTI_InitTypeDef;
```

EXTI_Mode: 中断模式设置

参数	功能描述
EXT_MODE_RiseFall	上升沿+下降沿（边沿）中断
EXT_MODE_Fall	下降沿中断

4.3.3 外部中断的中断函数

函数名	void INTx_ISR_Handler (void) interrupt INTx_VECTOR
功能描述	外部中断 x 的中断程序
参数	无
返回	无

程序框架:

```
void INT0_ISR_Handler (void) interrupt INT0_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码

}
void INT1_ISR_Handler (void) interrupt INT1_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码

}
void INT2_ISR_Handler (void) interrupt INT2_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码
```

```
}
void INT3_ISR_Handler (void) interrupt INT3_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码

}
void INT4_ISR_Handler (void) interrupt INT4_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码

}
```

4.4 STC32G_Timer

4.4.1 相关文件

“STC32G_Timer.c”， 主要用于 Timer 功能的配置。
“STC32G_Timer_Isr.c”，主要包含 Timer 的中断函数。
“STC32G_Timer.h”， Timer 所需的变量申明与宏定义。

4.4.2 定时器初始化函数

函数名	u8 Timer_Inilize(u8 TIM, TIM_InitTypeDef *TIMx)
功能描述	定时器初始化程序
参数 1	TIM: 定时器通道，取值 Timer0, Timer1, Timer2, Timer3, Timer4
参数 2	TIMx: 结构参数
返回	成功返回 SUCCESS，错误返回 FAIL

TIMx: 结构参数定义：

```
typedef struct
{
    u8  TIM_Mode;
    u8  TIM_ClkSource;
    u8  TIM_ClkOut;
    u16 TIM_Value;
    u8  TIM_Run;
} TIM_InitTypeDef;
```

TIM_Mode: 工作模式设置

参数	功能描述
TIM_16BitAutoReload	配置成 16 位自动重载模式
TIM_16Bit	配置成 16 位（手动重载）模式

TIM_8BitAutoReload	配置成 8 位自动重载模式
TIM_16BitAutoReloadNoMask	配置成 16 位自动重载模式，中断自动打开，不可屏蔽

TIM_ClkSource: 时钟源设置

参数	功能描述
TIM_CLOCK_1T	配置成 1T 模式
TIM_CLOCK_12T	配置成 12T 模式
TIM_CLOCK_Ext	配置成外部信号计数器模式

TIM_ClkOut: 可编程时钟输出设置

参数	功能描述
ENABLE	使能可编程时钟输出
DISABLE	禁止可编程时钟输出

TIM_Value: 装载定时/计数器初值。

TIM_Run: 是否运行设置

参数	功能描述
ENABLE	使能定时器
DISABLE	停止定时器

4.4.3 定时器中断函数

函数名	void Timerx_ISR_Handler (void) interrupt TMRx_VECTOR
功能描述	定时器 x 的中断程序
参数	无
返回	无

程序框架:

```
void Timer0_ISR_Handler (void) interrupt TMR0_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码

}
void Timer1_ISR_Handler (void) interrupt TMR1_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码

}
void Timer2_ISR_Handler (void) interrupt TMR2_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码
```

```
}
void Timer3_ISR_Handler (void) interrupt TMR3_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码

}
void Timer4_ISR_Handler (void) interrupt TMR4_VECTOR
{
    //进中断时已经清除标志
    // TODO: 在此处添加用户代码

}
```

4.5 STC32G_WDT 看门狗

4.5.1 相关文件

“STC32G_WDT.c”，主要用于看门狗的配置。
“STC32G_WDT.h”，WDT 所需的变量申明与宏定义。

4.5.2 看门狗初始化函数

函数名	void WDT_Inilize(WDT_InitTypeDef *WDT)
功能描述	看门狗初始化程序
参数	WDT: 结构参数
返回	无

WDT: 结构参数定义：

```
typedef struct
{
    u8   WDT_Enable;
    u8   WDT_IDLE_Mode;
    u8   WDT_PS;
} WDT_InitTypeDef;
```

WDT_Enable: 看门狗使能设置

参数	功能描述
ENABLE	使能看门狗
DISABLE	禁止看门狗

WDT_IDLE_Mode: IDLE 模式停止计数设置

参数	功能描述

WDT_IDLE_STOP	IDLE 模式停止计数
WDT_IDLE_RUN	IDLE 模式继续计数

WDT_PS: 看门狗定时器时钟分频系数

参数	功能描述
WDT_SCALE_2	系统时钟 2 分频
WDT_SCALE_4	系统时钟 4 分频
WDT_SCALE_8	系统时钟 8 分频
WDT_SCALE_16	系统时钟 16 分频
WDT_SCALE_32	系统时钟 32 分频
WDT_SCALE_64	系统时钟 64 分频
WDT_SCALE_128	系统时钟 128 分频
WDT_SCALE_256	系统时钟 256 分频

4.5.3 清看门狗

函数名	void WDT_Clear (void)
功能描述	看门狗喂狗程序
参数	无
返回	无

4.6 STC32G_ADC

4.6.1 相关文件

“STC32G_ADC.c”， 主要用于 ADC 功能的配置。
“STC32G_ADC_Isr.c”， 主要包含 ADC 中断函数。
“STC32G_ADC.h”， ADC 所需的变量申明与宏定义。

4.6.2 ADC 初始化函数

函数名	void ADC_Inilize(ADC_InitTypeDef *ADCx)
功能描述	ADC 初始化程序
参数	ADCx: 结构参数
返回	无

ADCx: 结构参数定义：

```
typedef struct
{
    u8   ADC_SMPduty;
    u8   ADC_Speed;
    u8   ADC_AdjResult;
    u8   ADC_CsSetup;
    u8   ADC_CsHold;
```

} ADC_InitTypeDef;

ADC_SMPduty: ADC 模拟信号采样时间控制，设置值 0~31（注意：SMPDUTY 一定不能设置小于 10）。

ADC_Speed: 设置 ADC 工作时钟频率

参数	功能描述
ADC_SPEED_2X1T	SYSclk/2/1
ADC_SPEED_2X2T	SYSclk/2/2
ADC_SPEED_2X3T	SYSclk/2/3
ADC_SPEED_2X4T	SYSclk/2/4
ADC_SPEED_2X5T	SYSclk/2/5
ADC_SPEED_2X6T	SYSclk/2/6
ADC_SPEED_2X7T	SYSclk/2/7
ADC_SPEED_2X8T	SYSclk/2/8
ADC_SPEED_2X9T	SYSclk/2/9
ADC_SPEED_2X10T	SYSclk/2/10
ADC_SPEED_2X11T	SYSclk/2/11
ADC_SPEED_2X12T	SYSclk/2/12
ADC_SPEED_2X13T	SYSclk/2/13
ADC_SPEED_2X14T	SYSclk/2/14
ADC_SPEED_2X15T	SYSclk/2/15
ADC_SPEED_2X16T	SYSclk/2/16

ADC_AdjResult: ADC 转换结果调整

参数	功能描述
ADC_LEFT_JUSTIFIED	转换结果左对齐
ADC_RIGHT_JUSTIFIED	转换结果右对齐

ADC_CsSetup: ADC 通道选择时间控制，取值 0(默认), 1

ADC_CsHold: ADC 通道选择保持时间控制，取值 0, 1(默认), 2, 3

4.6.3 ADC 电源控制

函数名	void ADC_PowerControl(u8 pwr)
功能描述	ADC 电源控制程序
参数	pwr: 电源控制,ENABLE 或 DISABLE.
返回	无

pwr: 电源控制

参数	功能描述
ENABLE	开启 ADC 模块电源
DISABLE	关闭 ADC 模块电源

4.6.4 查询法读取 ADC 转换结果

函数名	u16 Get_ADCResult(u8 channel)
功能描述	查询法读一次 ADC 结果
参数	channel: 选择要转换的 ADC 通道
返回	ADC 转换结果。返回值如果等于 4096，表示发生错误。

channel: 设置 0~15，分别表示 ADC0~ADC15.

4.6.5 ADC 中断函数

函数名	void ADC_ISR_Handler (void) interrupt ADC_VECTOR
功能描述	ADC 中断程序
参数	无
返回	无

程序框架:

```
void ADC_ISR_Handler (void) interrupt ADC_VECTOR
{
    ADC_FLAG = 0;               //清除中断标志

    // TODO: 在此处添加用户代码

}
```

4.7 STC32G_Compare 比较器

4.7.1 相关文件

“STC32G_Compare.c”，主要用于比较器功能的配置。
“STC32G_Compare_Isr.c”，主要包含比较器中断函数。
“STC32G_Compare.h”，比较器所需的变量申明与宏定义。

4.7.2 比较器初始化函数

函数名	void CMP_Inilize(CMP_InitDefine *CMPx)
功能描述	比较器初始化程序
参数	CMPx: 结构参数
返回	无

CMPx: 结构参数定义：

```
typedef struct
{
    u8   CMP_EN;
```

```
    u8   CMP_P_Select;
    u8   CMP_N_Select;
    u8   CMP_Outpt_En;
    u8   CMP_InvCMPO;
    u8   CMP_100nsFilter;
    u8   CMP_OutDelayDuty;
} CMP_InitDefine;
```

CMP_EN: 比较器使能设置

参数	功能描述
ENABLE	比较器使能
DISABLE	比较器禁止

CMP_P_Select: 比较器输入正极性选择

参数	功能描述
CMP_P_P37	选择外部端口 P3.7 做比较器正极输入源
CMP_P_P50	选择外部端口 P5.0 做比较器正极输入源
CMP_P_P51	选择外部端口 P5.1 做比较器正极输入源
CMP_P_ADC	由 ADC_CHS 所选择的 ADC 输入端做正极输入源

CMP_N_Select: 比较器输入负极性选择

参数	功能描述
CMP_N_P36	选择外部端口 P3.6 做比较器负极输入源
CMP_N_GAP	选择内部 BandGap 经过 OP 后的电压做负极输入源

CMP_Outpt_En: 比较结果输出设置

参数	功能描述
ENABLE	使能比较器结果输出，比较器结果输出到 P3.4 或者 P4.1
DISABLE	禁止比较器结果输出

CMP_InvCMPO: 比较器输出取反设置

参数	功能描述
ENABLE	使能比较器输出取反
DISABLE	禁止比较器输出取反

CMP_100nsFilter: 比较器内部 0.1us 滤波设置

参数	功能描述
ENABLE	使能内部 0.1us 滤波
DISABLE	禁止内部 0.1us 滤波

CMP_OutDelayDuty: 比较结果变化延时周期数，取值 0~63。

4.7.3 比较器中断函数

函数名	void CMP_ISR_Handler (void) interrupt CMP_VECTOR
功能描述	比较器中断程序
参数	无
返回	无

程序框架:

```
void CMP_ISR_Handler (void) interrupt CMP_VECTOR
{
    CMPIF = 0;              //清除中断标志

    // TODO: 在此处添加用户代码

}
```

4.8 STC32G_UART

4.8.1 相关文件

“STC32G_UART.c”，主要用于 UART 功能的配置。
“STC32G_UART_Isr.c”，主要包含 UART 的中断函数。
“STC32G_UART.h”，UART 所需的变量申明与宏定义。

宏定义

```
#define UART1    1
#define UART2    2
#define UART3    3
#define UART4    4
```

启用对应的 UART 通道，如果不使用该通道的 UART，就屏蔽对应的定义，减少系统开销。

```
#define COM_TX1_Lenth    128      //设置串口 1 数据发送缓冲区大小。
#define COM_RX1_Lenth    128      //设置串口 1 数据接收缓冲区大小。
#define COM_TX2_Lenth    16       //设置串口 2 数据发送缓冲区大小。
#define COM_RX2_Lenth    16       //设置串口 2 数据接收缓冲区大小。
#define COM_TX3_Lenth    64       //设置串口 3 数据发送缓冲区大小。
#define COM_RX3_Lenth    64       //设置串口 3 数据接收缓冲区大小。
#define COM_TX4_Lenth    32       //设置串口 4 数据发送缓冲区大小。
#define COM_RX4_Lenth    32       //设置串口 4 数据接收缓冲区大小。

#define  TimeOutSet1     5        //设置串口 1 数据接收超时时间。
#define  TimeOutSet2     5        //设置串口 2 数据接收超时时间。
#define  TimeOutSet3     5        //设置串口 3 数据接收超时时间。
#define  TimeOutSet4     5        //设置串口 4 数据接收超时时间。
```

TimeOutSet 毫秒超时没收到新的数据说明一段数据接收完成。

4.8.2 UART 初始化函数

函数名	u8 UART_Configuration(u8 UARTx, COMx_InitDefine *COMx)
功能描述	UART 初始化程序
参数 1	UARTx: UART 设置通道，取值 UART1, UART2, UART3, UART4
参数 2	COMx: 结构参数
返回	无

COMx: 结构参数定义：

```
typedef struct
{
    u8   UART_Mode;
    u8   UART_BRT_Use;
    u32  UART_BaudRate;
    u8   Morecommunicate;
    u8   UART_RxEnable;
    u8   BaudRateDouble;
} COMx_InitDefine;
```

UART_Mode: 模式设置

参数	功能描述
UART_ShiftRight	串口工作于同步输出方式，仅用于 UART1
UART_8bit_BRTx	串口工作于 8 位数据，可变波特率
UART_9bit	串口工作于 9 位数据，固定波特率
UART_9bit_BRTx	串口工作于 9 位数据，可变波特率

UART_BRT_Use: 波特率发生器设置

参数	功能描述
BRT_Timer1	使用 Timer1 作为波特率发生器，适用于 UART1
BRT_Timer2	使用 Timer2 作为波特率发生器，适用于 UART1, UART2, UART3, UART4
BRT_Timer3	使用 Timer3 作为波特率发生器，适用于 UART3
BRT_Timer4	使用 Timer4 作为波特率发生器，适用于 UART4

UART_BaudRate: 波特率设置，一般设为 110 ~ 115200。

Morecommunicate: 多机通讯设置

参数	功能描述
ENABLE	使能多机通讯
DISABLE	禁止多机通讯

UART_RxEnable: 允许接收设置

参数	功能描述
ENABLE	使能接收
DISABLE	禁止接收

BaudRateDouble: 波特率加倍设置(仅用于 UART1)

参数	功能描述
ENABLE	使能波特率加倍
DISABLE	禁止波特率加倍

4.8.3 UART 发送字节函数

函数名	void TX1_write2buff(u8 dat) void TX2_write2buff(u8 dat) void TX3_write2buff(u8 dat) void TX4_write2buff(u8 dat)
功能描述	UART 发送一个字节数据
参数	dat: 待发送数据
返回	无

4.8.4 UART 发送字符串函数

函数名	void PrintString1(u8 *puts) void PrintString2(u8 *puts) void PrintString3(u8 *puts) void PrintString4(u8 *puts)
功能描述	UART 发送一串数据，遇到停止符 0 结束
参数	*puts: 待发送数据缓冲区指针
返回	无

4.8.5 UART 中断函数

函数名	void UARTx_ISR_Handler (void) interrupt UARTx_VECTOR
功能描述	串口 x 的中断程序
参数	无
返回	无

程序框架:

```
void UART1_ISR_Handler (void) interrupt UART1_VECTOR
{
    if(RI)
    {
        RI = 0;
        // TODO: 在此处添加用户代码
    }
    if(TI)
    {
        TI = 0;
        // TODO: 在此处添加用户代码
```

```
    }
}
void UART2_ISR_Handler (void) interrupt UART2_VECTOR
{
    if(S2RI)
    {
        CLR_RI2();
        // TODO: 在此处添加用户代码
    }
    if(S2TI)
    {
        CLR_TI2();
        // TODO: 在此处添加用户代码
    }
}
void UART3_ISR_Handler (void) interrupt UART3_VECTOR
{
    if(S3RI)
    {
        CLR_ RI3();
        // TODO: 在此处添加用户代码
    }
    if(S3TI)
    {
        CLR_TI3();
        // TODO: 在此处添加用户代码
    }
}
void UART4_ISR_Handler (void) interrupt UART4_VECTOR
{
    if(S4RI)
    {
        CLR_ RI4();
        // TODO: 在此处添加用户代码
    }
    if(S4TI)
    {
        CLR_TI4();
        // TODO: 在此处添加用户代码
    }
}
```

4.9 STC32G_SPI

4.9.1 相关文件

“STC32G_SPI.c”，主要用于 SPI 功能的配置。
“STC32G_SPI_Isr.c”，主要包含 SPI 的中断函数。
“STC32G_SPI.h”，SPI 所需的变量申明与宏定义。

宏定义

```
#define SPI_BUF_LENTH    128      //设置 SPI 数据缓冲区大小。
```

4.9.2 SPI 初始化函数

函数名	void SPI_Init(SPI_InitTypeDef *SPIx)
功能描述	SPI 初始化程序
参数	SPIx: 结构参数
返回	无

COMx: 结构参数定义：

```
typedef struct
{
    u8  SPI_Enable;
    u8  SPI_SSIG;
    u8  SPI_FirstBit;
    u8  SPI_Mode;
    u8  SPI_CPOL;
    u8  SPI_CPHA;
    u8  SPI_Speed;
} SPI_InitTypeDef;
```

SPI_Enable: 功能使能设置

参数	功能描述
ENABLE	使能 SPI 功能
DISABLE	禁用 SPI 功能

SPI_SSIG: 片选位设置

参数	功能描述
ENABLE	忽略 SS 引脚功能，使用 MSTR 确定器件是主机还是从机
DISABLE	SS 引脚确定器件是主机还是从机

SPI_FirstBit: 数据发送/接收顺序设置

参数	功能描述
SPI_MSB	先发送/接收数据的高位（MSB）
SPI_LSB	先发送/接收数据的低位（LSB）

SPI_Mode: 主从模式设置

参数	功能描述
SPI_Mode_Master	设置为主机模式
SPI_Mode_Slave	设置为从机模式

SPI_CPOL: SPI 时钟极性设置

参数	功能描述
SPI_CPOL_Low	SCLK 空闲时为低电平
SPI_CPOL_High	SCLK 空闲时为高电平

SPI_CPHA: SPI 时钟相位设置

参数	功能描述
SPI_CPHA_1Edge	数据在 SCLK 的后时钟沿驱动，前时钟沿采样（必须 SSIG=0）
SPI_CPHA_2Edge	数据在 SCLK 的前时钟沿驱动，后时钟沿采样

SPI_Speed: SPI 时钟频率设置

参数	功能描述
SPI_Speed_4	SCLK 频率=SPI 输入时钟/4
SPI_Speed_8	SCLK 频率=SPI 输入时钟/8
SPI_Speed_16	SCLK 频率=SPI 输入时钟/16
SPI_Speed_2	SCLK 频率=SPI 输入时钟/2

4.9.3 SPI 模式设置

函数名	void SPI_SetMode(u8 mode)
功能描述	SPI 设置主从模式函数
参数	mode: 指定模式，取值 SPI_Mode_Master 或 SPI_Mode_Slave
返回	无

4.9.4 SPI 发送一个字节数据

函数名	void SPI_WriteByte(u8 dat)
功能描述	SPI 发送一个字节数据函数
参数	dat: 要发送的数据
返回	无

4.9.5 SPI 中断函数

函数名	void SPI_ISR_Handler() interrupt SPI_VECTOR
功能描述	SPI 中断程序
参数	无
返回	无

程序框架:

```
void SPI_ISR_Handler() interrupt SPI_VECTOR
{
    if(MSTR) //主机模式
    {
        // TODO: 在此处添加用户代码
    }
    else     //从机模式
    {
        // TODO: 在此处添加用户代码
    }
    SPI_ClearFlag();    //清除 SPIF 和 WCOL 标志
}
```

4.10 STC32G_I2C

4.10.1 相关文件

“STC32G_I2C.c”，主要用于 I2C 功能的配置。
“STC32G_I2C_Isr.c”，主要包含 I2C 的中断函数。
“STC32G_I2C.h”，I2C 所需的变量申明与宏定义。

宏定义

```
#define I2C_BUF_LENTH    8    //设置 I2C 数据缓冲区大小。
#define SLAW      0xA2        //设置 I2C 设备写地址
#define SLAR      0xA3        //设置 I2C 设备读地址
```

4.10.2 I2C 初始化函数

函数名	void I2C_Init(I2C_InitTypeDef *I2Cx)
功能描述	I2C 初始化程序
参数	I2Cx: 结构参数
返回	无

I2Cx: 结构参数定义：

```
typedef struct
{
    u8  I2C_Speed;
    u8  I2C_Enable
    u8  I2C_Mode;
    u8  I2C_MS_WDTA;
    u8  I2C_SL_ADR;
    u8  I2C_SL_MA;
} I2C_InitTypeDef;
```

I2C_Speed: 总线速度设置，取值 0~63。总线速度=Fosc/2/(Speed*2+4)。

I2C_Enable: 功能使能

参数	功能描述
ENABLE	使能 I2C 功能
DISABLE	禁止 I2C 功能

I2C_Mode: 主从模式选择

参数	功能描述
I2C_Mode_Master	设置为主机模式
I2C_Mode_Slave	设置为从机模式

I2C_MS_WDTA: 主机自动发送设置

参数	功能描述
ENABLE	使能主机自动发送
DISABLE	禁止主机自动发送

I2C_SL_ADR: 从机设备地址，取值 0~127。

I2C_SL_MA: 从机设备地址比较设置

参数	功能描述
ENABLE	使能从机设备地址比较
DISABLE	禁止从机设备地址比较

4.10.3 I2C 写入数据函数

函数名	void I2C_WriteNbyte(u8 dev_addr, u8 mem_addr, u8 *p, u8 number)
功能描述	I2C 写入若干数据程序
参数 1	dev_addr: 设备地址
参数 2	mem_addr: 存储地址
参数 3	*p: 写入数据存储位置
参数 4	number: 写入数据个数
返回	无

4.10.4 I2C 读取数据函数

函数名	void I2C_ReadNbyte(u8 dev_addr, u8 mem_addr, u8 *p, u8 number)
功能描述	I2C 读取若干数据程序
参数 1	dev_addr: 设备地址
参数 2	mem_addr: 存储地址
参数 3	*p: 读取数据存储位置
参数 4	number: 读取数据个数
返回	无

4.10.5 I2C 中断函数

函数名	void I2C_ISR_Handler() interrupt I2C_VECTOR
功能描述	I2C 中断程序
参数	无
返回	无

程序框架:

```
void I2C_ISR_Handler() interrupt I2C_VECTOR
{
    // TODO: 在此处添加用户代码
    // 主机模式
    I2CMSST &= ~0x40;         //I2C 指令发送完成状态清除

    if(DMA_I2C_CR & 0x04)     //ACKERR
    {
        DMA_I2C_CR &= ~0x04;  //发数据后收到 NAK
    }

    // 从机模式
    if (I2CSLST & 0x40)
    {
        I2CSLST &= ~0x40;                       //处理 START 事件
    }
    else if (I2CSLST & 0x20)
    {
        I2CSLST &= ~0x20;                       //处理 RECV 事件，SLACKO 设置为 0
        if (I2CIsr.isda)
        {
            I2CIsr.isda = 0;                    //处理 RECV 事件（RECV DEVICE ADDR）
        }
        else if (I2CIsr.isma)
        {
            I2CIsr.isma = 0;                    //处理 RECV 事件（RECV MEMORY ADDR）
            I2CIsr.addr = I2CRXD;
            I2CTXD = I2C_Buffer[I2CIsr.addr];
        }
        else
        {
            I2C_Buffer[I2CIsr.addr++] = I2CRXD;     //处理 RECV 事件（RECV DATA）
        }
    }
    else if (I2CSLST & 0x10)
    {
        I2CSLST &= ~0x10;                       //处理 SEND 事件
        if (I2CSLST & 0x02)
```

```
        {
            I2CTXD = 0xff;
        }
        else
        {
            I2CTXD = I2C_Buffer[++I2Clsr.addr];
        }
    }
    else if (I2CSLST & 0x08)
    {
        I2CSLST &= ~0x08;              //处理 STOP 事件
        I2Clsr.isda = 1;               //重置状态标志位
        I2Clsr.isma = 1;               //重置状态标志位
        DisplayFlag = 1;               //设置接收完成标志
    }
}
```

4.11 STC32G_PWM

4.11.1 相关文件

“STC32G_PWM.c”，主要用于 PWM 功能的配置。
“STC32G_PWM_Isr.c”，主要包含 PWM 的中断函数。
“STC32G_PWM.h”，PWM 所需的变量申明与宏定义。

4.11.2 PWM 初始化

函数名	u8 PWM_Configuration(u8 PWM, PWMx_InitDefine *PWMx)
功能描述	16 位高级 PWM 初始化程序
参数 1	PWM: PWM 通道，取值 PWM1~PWM8,PWMA,PWMB
参数 2	PWMx: 结构参数
返回	成功返回 SUCCESS，错误返回 FAIL

PWMx: 结构参数定义：

```
typedef struct
{
    u8   PWM_Mode;
    u16  PWM_Period;
    u16  PWM_Duty;
    u8   PWM_DeadTime;
    u8   PWM_EnoSelect;
    u8   PWM_CEN_Enable;
    u8   PWM_MainOutEnable;
} PWMx_InitDefine;
```

PWM_Mode: PWM 模式设置

参数	功能描述
CCMRn_FREEZE	冻结
CCMRn_MATCH_VALID	匹配时设置通道 n 的输出为有效电平
CCMRn_MATCH_INVALID	匹配时设置通道 n 的输出为无效电平
CCMRn_ROLLOVER	翻转
CCMRn_FORCE_INVALID	强制为无效电平
CCMRn_FORCE_VALID	强制为有效电平
CCMRn_PWM_MODE1	PWM 模式 1
CCMRn_PWM_MODE2	PWM 模式 2

PWM_Period: 周期时间，取值 0~65535。

PWM_Duty: 占空比时间，取值 0~ PWM_Period。

PWM_DeadTime: 死区发生器设置，取值 0~ 255。

PWM_EnoSelect: PWM 输出通道选择

参数		功能描述
PWMA	ENO1P	选择 PWM1P 输出
	ENO1N	选择 PWM1N 输出
	ENO2P	选择 PWM2P 输出
	ENO2N	选择 PWM2N 输出
	ENO3P	选择 PWM3P 输出
	ENO3N	选择 PWM3N 输出
	ENO4P	选择 PWM4P 输出
	ENO4N	选择 PWM4N 输出
PWMB	ENO5P	选择 PWM5P 输出
	ENO6P	选择 PWM6P 输出
	ENO7P	选择 PWM7P 输出
	ENO8P	选择 PWM8P 输出

以上参数同组可以使用或运算，比如：

```
PWMx_InitStructure.PWM_EnoSelect    = ENO1P | ENO1N;
```

PWM_CEN_Enable: PWM 计数器使能设置

参数	功能描述
ENABLE	使能计数器
DISABLE	禁止计数器

PWM_MainOutEnable: PWM 主输出使能设置

参数	功能描述
ENABLE	使能主输出
DISABLE	禁止主输出

4.11.3 更新 PWM 占空比值

函数名	void UpdatePwm(u8 PWM, PWMx_Duty *PWMx)
功能描述	更新 PWM 占空比值
参数 1	PWM: PWM 通道，取值 PWM1~PWM8,PWMA,PWMB
参数 2	PWMx: 结构参数
返回	无

PWMx_Duty: 结构参数定义：

```
typedef struct
{
    u16 PWM1_Duty;          //PWM1 占空比时间, 0~Period
    u16 PWM2_Duty;          //PWM2 占空比时间, 0~Period
    u16 PWM3_Duty;          //PWM3 占空比时间, 0~Period
    u16 PWM4_Duty;          //PWM4 占空比时间, 0~Period
    u16 PWM5_Duty;          //PWM5 占空比时间, 0~Period
    u16 PWM6_Duty;          //PWM6 占空比时间, 0~Period
    u16 PWM7_Duty;          //PWM7 占空比时间, 0~Period
    u16 PWM8_Duty;          //PWM8 占空比时间, 0~Period
} PWMx_Duty;
```

4.11.4 高速 PWM 初始化

函数名	void HSPWM_Configuration(u8 PWM, HSPWMx_InitDefine *PWMx, PWMx_Duty *DUTYx)
功能描述	16 位高速高级 PWM 初始化程序
参数 1	PWM: PWM 通道，取值 PWM1~PWM8,PWMA,PWMB
参数 2	PWMx: 结构参数
参数 3	DUTYx: 结构参数
返回	成功返回 SUCCESS，错误返回 FAIL

PWMx: 结构参数定义：

```
typedef struct
{
    u16 PWM_Period;
    u8  PWM_DeadTime;
    u8  PWM_EnoSelect;
    u8  PWM_CEN_Enable;
    u8  PWM_MainOutEnable;
} HSPWMx_InitDefine;
```

PWM_Period: 周期时间，取值 0~65535。

PWM_DeadTime: 死区发生器设置，取值 0~ 255。

PWM_EnoSelect: PWM 输出通道选择

参数		功能描述
PWMA	ENO1P	选择 PWM1P 输出
	ENO1N	选择 PWM1N 输出
	ENO2P	选择 PWM2P 输出
	ENO2N	选择 PWM2N 输出
	ENO3P	选择 PWM3P 输出
	ENO3N	选择 PWM3N 输出
	ENO4P	选择 PWM4P 输出
	ENO4N	选择 PWM4N 输出
PWMB	ENO5P	选择 PWM5P 输出
	ENO6P	选择 PWM6P 输出
	ENO7P	选择 PWM7P 输出
	ENO8P	选择 PWM8P 输出

以上参数同组可以使用或运算，比如：

PWMx_InitStructure.PWM_EnoSelect = ENO1P | ENO1N;

PWM_CEN_Enable: PWM 计数器使能设置

参数	功能描述
ENABLE	使能计数器
DISABLE	禁止计数器

PWM_MainOutEnable: PWM 主输出使能设置

参数	功能描述
ENABLE	使能主输出
DISABLE	禁止主输出

DUTYx: 结构参数定义：

```
typedef struct
{
    u16 PWM1_Duty;          //PWM1 占空比时间, 0~Period
    u16 PWM2_Duty;          //PWM2 占空比时间, 0~Period
    u16 PWM3_Duty;          //PWM3 占空比时间, 0~Period
    u16 PWM4_Duty;          //PWM4 占空比时间, 0~Period
    u16 PWM5_Duty;          //PWM5 占空比时间, 0~Period
    u16 PWM6_Duty;          //PWM6 占空比时间, 0~Period
    u16 PWM7_Duty;          //PWM7 占空比时间, 0~Period
    u16 PWM8_Duty;          //PWM8 占空比时间, 0~Period
} PWMx_Duty;
```

PWMn_Duty: PWMn 占空比时间，取值 0~Period。

4.11.5 更新高速 PWM 占空比值

函数名	void UpdateHSPwm(u8 PWM, PWMx_Duty *PWMx)
功能描述	更新高速 PWM 占空比值
参数 1	PWM: PWM 通道，取值 PWM1~PWM8,PWMA,PWMB
参数 2	PWMx: 结构参数
返回	无

PWMx_Duty: 结构参数定义：

```
typedef struct
{
    u16 PWM1_Duty;          //PWM1 占空比时间, 0~Period
    u16 PWM2_Duty;          //PWM2 占空比时间, 0~Period
    u16 PWM3_Duty;          //PWM3 占空比时间, 0~Period
    u16 PWM4_Duty;          //PWM4 占空比时间, 0~Period
    u16 PWM5_Duty;          //PWM5 占空比时间, 0~Period
    u16 PWM6_Duty;          //PWM6 占空比时间, 0~Period
    u16 PWM7_Duty;          //PWM7 占空比时间, 0~Period
    u16 PWM8_Duty;          //PWM8 占空比时间, 0~Period
} PWMx_Duty;
```

4.11.6 PWMA 中断函数

函数名	void PWMA_ISR_Handler (void) interrupt PWMA_VECTOR
功能描述	PWMA 的中断程序
参数	无
返回	无

程序框架:

```
void PWMA_ISR_Handler (void) interrupt PWMA_VECTOR
{
    // TODO: 在此处添加用户代码
    if (PWMA_SR1 & 0x01)      //UIFA 更新中断
    {
        PWMA_SR1 &= ~0x01;

    }
    if (PWMA_SR1 & 0x02)      //CC1IF 捕获/比较中断
    {
        PWMA_SR1 &= ~0x02;

    }
    if (PWMA_SR1 & 0x04)      //CC2IF 捕获/比较中断
    {
        PWMA_SR1 &= ~0x04;
```

```
    }
    if (PWMA_SR1 & 0x08)      //CC3IF 捕获/比较中断
    {
        PWMA_SR1 &= ~0x08;

    }
    if (PWMA_SR1 & 0x10)      //CC4IF 捕获/比较中断
    {
        PWMA_SR1 &= ~0x10;

    }
    if (PWMA_SR1 & 0x20)      //COMIFA 中断
    {
        PWMA_SR1 &= ~0x20;

    }
    if (PWMA_SR1 & 0x40)      //TIFA 触发器中断
    {
        PWMA_SR1 &= ~0x40;

    }
    if (PWMA_SR1 & 0x80)      //BIFA 刹车中断
    {
        PWMA_SR1 &= ~0x80;

    }
}
```

4.11.7 PWMB 中断函数

函数名	void PWMB_ISR_Handler (void) interrupt PWMB_VECTOR
功能描述	PWMB 的中断程序
参数	无
返回	无

程序框架:

```
void PWMB_ISR_Handler (void) interrupt PWMB_VECTOR
{
    // TODO: 在此处添加用户代码
    if (PWMB_SR1 & 0x01)      //UIFB 更新中断
    {
        PWMB_SR1 &= ~0x01;

    }
    if (PWMB_SR1 & 0x02)      //CC5IF 捕获/比较中断
    {
```

```
        PWMB_SR1 &= ~0x02;

    }
    if (PWMB_SR1 & 0x04)      //CC6IF 捕获/比较中断
    {
        PWMB_SR1 &= ~0x04;

    }
    if (PWMB_SR1 & 0x08)      //CC7IF 捕获/比较中断
    {
        PWMB_SR1 &= ~0x08;

    }
    if (PWMB_SR1 & 0x10)      //CC8IF 捕获/比较中断
    {
        PWMB_SR1 &= ~0x10;

    }
    if (PWMB_SR1 & 0x20)      //COMIFB 中断
    {
        PWMB_SR1 &= ~0x20;

    }
    if (PWMB_SR1 & 0x40)      //TIFB 触发器中断
    {
        PWMB_SR1 &= ~0x40;

    }
    if (PWMB_SR1 & 0x80)      //BIFB 刹车中断
    {
        PWMB_SR1 &= ~0x80;

    }
}
```

4.12 STC32G_PWM (网友提供)

4.12.1 相关文件

“stc32g_pwma.c”， 主要包含用于 PWMA 的功能配置函数。
“stc32g_pwmb.c”，主要包含用于 PWMB 的功能配置函数。
“stc32g_pwma.h”， PWMA 所需的变量申明与宏定义。
“stc32g_pwmb.h”， PWMB 所需的变量申明与宏定义。
“Type_def.h”， PWM 所需的变量申明与宏定义。

4.12.2 PWMA 反初始化

函数名	void PWMA_DeInit(void)
功能描述	将 PWMA 外围寄存器取消初始化为其默认重置值
参数 1	无
返回	无

4.12.3 初始化 PWMA 时基单元值

函数名	void PWMA_TimeBaseInit(uint16_t PWMA_Prescaler, PWMA_CounterMode_TypeDef PWMA_CounterMode, uint16_t PWMA_Period, uint8_t PWMA_RepetitionCounter)
功能描述	根据指定的参数初始化 PWMA 时基单位
参数 1	PWMA_Pescaler 指定预缩放器值
参数 2	PWMA_CounterMode 从@ref PWMA_CoounterMode_TypeDef 指定计数器模式
参数 3	PWMA_Period 指定 Period 值
参数 4	PWMA_ReditionCounter 指定重复计数器值
返回	无

其中:

```
typedef enum
{
  PWMA_COUNTERMODE_UP                = ((uint8_t)0x00),
  PWMA_COUNTERMODE_DOWN              = ((uint8_t)0x10),
  PWMA_COUNTERMODE_CENTERALIGNED1    = ((uint8_t)0x20),
  PWMA_COUNTERMODE_CENTERALIGNED2    = ((uint8_t)0x40),
  PWMA_COUNTERMODE_CENTERALIGNED3    = ((uint8_t)0x60)
}PWMA_CounterMode_TypeDef;
```

4.12.4 初始化 PWMA 通道 1

函数名	void PWMA_OC1Init(PWMA_OCMode_TypeDef PWMA_OCMode, PWMA_OutputState_TypeDef PWMA_OutputState, PWMA_OutputNState_TypeDef PWMA_OutputNState, uint16_t PWMA_Pulse, PWMA_OCPolarity_TypeDef PWMA_OCPolarity, PWMA_OCNPolarity_TypeDef

	PWMA_OCNPolarity, PWMA_OCIdleState_TypeDef PWMA_OCIdleState, PWMA_OCNIdleState_TypeDef PWMA_OCNIdleState)
功能描述	根据指定的参数初始化 PWMA 通道 1
参数 1	PWMA_OCMode 指定输出比较模式
参数 2	PWMA_OutputState 指定输出状态
参数 3	PWMA_OutputNState 指定互补输出状态
参数 4	PWMA_Pulse 指定脉冲宽度值。
参数 5	PWMA_OCPolarity 指定输出比较极性
参数 6	PWMA_OCNPolarity 指定互补输出比较极性
参数 7	PWMA_OCIdleState 指定输出比较空闲状态
参数 8	PWMA_OCNIdleState 指定互补输出比较空闲
返回	无

PWMA 输出比较和 PWM 模式:

```
typedef enum
{
    PWMA_OCMODE_TIMING      = ((uint8_t)0x00),
    PWMA_OCMODE_ACTIVE      = ((uint8_t)0x10),
    PWMA_OCMODE_INACTIVE    = ((uint8_t)0x20),
    PWMA_OCMODE_TOGGLE      = ((uint8_t)0x30),
    PWMA_OCMODE_PWM1        = ((uint8_t)0x60),
    PWMA_OCMODE_PWM2        = ((uint8_t)0x70)
}PWMA_OCMode_TypeDef;
```

4.12.5 初始化 PWMA 通道 2

函数名	void PWMA_OC2Init(PWMA_OCMode_TypeDef PWMA_OCMode, PWMA_OutputState_TypeDef PWMA_OutputState, PWMA_OutputNState_TypeDef PWMA_OutputNState, uint16_t PWMA_Pulse, PWMA_OCPolarity_TypeDef PWMA_OCPolarity, PWMA_OCNPolarity_TypeDef PWMA_OCNPolarity, PWMA_OCIdleState_TypeDef PWMA_OCIdleState, PWMA_OCNIdleState_TypeDef PWMA_OCNIdleState)
功能描述	根据指定的参数初始化 PWMA 通道 2

参数 1	PWMA_OCMode 指定输出比较模式
参数 2	PWMA_OutputState 指定输出状态
参数 3	PWMA_OutputNState 指定互补输出状态
参数 4	PWMA_Pulse 指定脉冲宽度值。
参数 5	PWMA_OCPolarity 指定输出比较极性
参数 6	PWMA_OCNPolarity 指定互补输出比较极性
参数 7	PWMA_OCIdleState 指定输出比较空闲状态
参数 8	PWMA_OCNIdleState 指定互补输出比较空闲
返回	无

4.12.6 初始化 PWMA 通道 3

函数名	void PWMA_OC3Init(PWMA_OCMode_TypeDef PWMA_OCMode, PWMA_OutputState_TypeDef PWMA_OutputState, PWMA_OutputNState_TypeDef PWMA_OutputNState, uint16_t PWMA_Pulse, PWMA_OCPolarity_TypeDef PWMA_OCPolarity, PWMA_OCNPolarity_TypeDef PWMA_OCNPolarity, PWMA_OCIdleState_TypeDef PWMA_OCIdleState, PWMA_OCNIdleState_TypeDef PWMA_OCNIdleState)
功能描述	根据指定的参数初始化 PWMA 通道 3
参数 1	PWMA_OCMode 指定输出比较模式
参数 2	PWMA_OutputState 指定输出状态
参数 3	PWMA_OutputNState 指定互补输出状态
参数 4	PWMA_Pulse 指定脉冲宽度值。
参数 5	PWMA_OCPolarity 指定输出比较极性
参数 6	PWMA_OCNPolarity 指定互补输出比较极性
参数 7	PWMA_OCIdleState 指定输出比较空闲状态
参数 8	PWMA_OCNIdleState 指定互补输出比较空闲
返回	无

4.12.7 初始化 PWMA 通道 4

函数名	void PWMA_OC4Init(PWMA_OCMode_TypeDef PWMA_OCMode,

	PWMA_OutputState_TypeDef PWMA_OutputState, PWMA_OutputNState_TypeDef PWMA_OutputNState, uint16_t PWMA_Pulse, PWMA_OCPolarity_TypeDef PWMA_OCPolarity, PWMA_OCNPolarity_TypeDef PWMA_OCNPolarity, PWMA_OCIdleState_TypeDef PWMA_OCIdleState, PWMA_OCNIdleState_TypeDef PWMA_OCNIdleState)
功能描述	根据指定的参数初始化 PWMA 通道 4
参数 1	PWMA_OCMode 指定输出比较模式
参数 2	PWMA_OutputState 指定输出状态
参数 3	PWMA_OutputNState 指定互补输出状态
参数 4	PWMA_Pulse 指定脉冲宽度值。
参数 5	PWMA_OCPolarity 指定输出比较极性
参数 6	PWMA_OCNPolarity 指定互补输出比较极性
参数 7	PWMA_OCIdleState 指定输出比较空闲状态
参数 8	PWMA_OCNIdleState 指定互补输出比较空闲
返回	无

4.12.8 PWMA 刹车死区功能配置

函数名	void PWMA_BDTRConfig(PWMA_OSSIState_TypeDef PWMA_OSSIState, PWMA_LockLevel_TypeDef PWMA_LockLevel, uint8_t PWMA_DeadTime, PWMA_BreakState_TypeDef PWMA_Break, PWMA_BreakPolarity_TypeDefPWMA_BreakPolarity, PWMA_AutomaticOutput_TypeDef PWMA_AutomaticOutput)
功能描述	配置刹车、死区、锁定级别、OSSI，以及 AOE（自动输出启用）
参数 1	PWMA_OSSIState 从@ref PWMA_OSSIState_TypeDef 指定 OSSIS 状态
参数 2	PWMA_LockLevel 从@ref PWMA_lock level_TypeDef 指定锁定级别
参数 3	PWMA_DeadTime 指定停滞时间值
参数 4	PWMA_Break 指定 Break 状态@ref PWMA_BBreakState_TypeDef
参数 5	PWMA_BreakPolarity 指定来自*@ref PWMA_BreakPolarity_TypeDef。
参数 6	PWMA_AutomaticOutput 指定自动输出配置
返回	无

4.12.9 PWMA 输入捕获初始化

函数名	void PWMA_ICInit(PWMA_Channel_TypeDef PWMA_Channel, PWMA_ICPolarity_TypeDef PWMA_ICPolarity, PWMA_ICSelection_TypeDef PWMA_ICSelection, PWMA_ICPSC_TypeDef PWMA_ICPrescaler, uint8_t PWMA_ICFilter)
功能描述	根据指定的参数初始化 PWMA 外围设备
参数 1	PWMA_Channel 指定来自 PWMA_Cannel_TypeDef 的输入捕获通道
参数 2	PWMA_ICPolarity 指定输入捕获极性
参数 3	PWMA_ICSelection 指定输入捕获源选择
参数 4	PWMA_ICP 缩放器指定输入捕获预缩放器
参数 5	PWMA_ICFilter 指定输入捕获滤波器值
返回	无

```
PWMA 输入捕获预分频器
{
  PWMA_ICPSC_DIV1                = ((uint8_t)0x00),
  PWMA_ICPSC_DIV2                = ((uint8_t)0x04),
  PWMA_ICPSC_DIV4                = ((uint8_t)0x08),
  PWMA_ICPSC_DIV8                = ((uint8_t)0x0C)
}PWMA_ICPSC_TypeDef;
```

4.12.10 PWMA 输入捕获配置

函数名	void PWMA_PWMIConfig(PWMA_Channel_TypeDef PWMA_Channel, PWMA_ICPolarity_TypeDef PWMA_ICPolarity, PWMA_ICSelection_TypeDef PWMA_ICSelection, PWMA_ICPSC_TypeDef PWMA_ICPrescaler, uint8_t PWMA_ICFilter)
功能描述	根据指定参数在 PWM 输入模式下配置 PWMA 外围设备
参数 1	配置通道 PWMA_Channel
参数 2	配置极性 PWMA_ICPolarity
参数 3	配置选择源 PWMA_ICSelection
参数 4	配置预缩放器
参数 5	PWMA_ICFilter 指定输入捕获滤波器值
返回	无

4.12.11 启用或禁用 PWMA

函数名	void PWMA_Cmd(FunctionalState NewState)
功能描述	启用或禁用 PWMA 外围设备
参数 1	NewState PWMA 外备的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.12 启用或禁用 PWMA 设备主输出

函数名	void PWMA_CtrlPWMOutputs(FunctionalState NewState)
功能描述	启用或禁用 PWMA 设备主输出
参数 1	NewState PWMA 外备的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.13 启用或禁用指定的 PWMA 中断

函数名	void PWMA_ITConfig(PWMA_IT_TypeDef PWMA_IT, FunctionalState NewState)
功能描述	启用或禁用指定的 PWMA 中断
参数 1	PWMA_IT 指定要启用或禁用的 PWMA 中断源。 *此参数可以是以下值的任意组合: *-PWMA_IT_UPDATE: PWMA 更新中断源 *-PWMA_IT_CC1:PWMA 捕获比较 1 中断源 *-PWMA_IT_CC2:PWMA 捕获比较 2 中断源 *-PWMA_IT_CC3:PWMA 捕获比较 3 中断源 *-PWMA_IT_CC4:PWMA 捕获比较 4 中断源 *-PWMA_IT_CU 更新: PWMA 捕获比较更新中断源 *-PWMA_IT_TRIGGER:PWMA 触发中断源 *-PWMA_IT_BEAK:PWMA 中断源
参数 2	NewState PWMA 外围设备的新状态
返回	无

PWMA 中断源

```
typedef enum
{
  PWMA_IT_UPDATE                  = ((uint8_t)0x01),
  PWMA_IT_CC1                     = ((uint8_t)0x02),
  PWMA_IT_CC2                     = ((uint8_t)0x04),
```

```
    PWMA_IT_CC3                     = ((uint8_t)0x08),
    PWMA_IT_CC4                     = ((uint8_t)0x10),
    PWMA_IT_COM                      = ((uint8_t)0x20),
    PWMA_IT_TRIGGER                 = ((uint8_t)0x40),
    PWMA_IT_BREAK                   = ((uint8_t)0x80)
}PWMA_IT_TypeDef;
```

4.12.14 配置 PWMA 内部时钟

函数名	void PWMA_InternalClockConfig(void)
功能描述	启用配置 PWMA 内部时钟
参数 1	无
返回	无

4.12.15 配置 PWMA 外部时钟模式 1

函数名	void PWMA_ETRClockMode1Config(PWMA_ExtTRGPSC_TypeDef PWMA_ExtTRGPrescaler, PWMA_ExtTRGPolarity_TypeDef PWMA_ExtTRGPolarity, uint8_t ExtTRGFilter)
功能描述	配置 PWMA 外部时钟模式 1
参数 1	PWMA_ExtTRGPrescaler 指定外部触发器预缩放器。 *此参数可以是以下值之一: *-PWMA_EXTTRGPSC_OFF *-PWMA_EXTTRGPSC_DIV2 *-PWMA_EXTTRGPSC_DIV4 *-PWMA_EXTTRGPSC_DIV8。
参数 2	PWMA_ExtTRGPolarity 指定外部触发极性。 *此参数可以是以下值之一: *-PWMA_EXTTRGPOLARITY_INVERTED *-PWMA_EXTTRGPOLARITY_NONINVERTED
参数 3	ExtTRGFilter 指定外部触发器滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.16 配置 PWMA 外部时钟模式 2

函数名	void PWMA_ETRClockMode2Config(PWMA_ExtTRGPSC_TypeDef PWMA_ExtTRGPrescaler,

	PWMA_ExtTRGPolarity_TypeDef PWMA_ExtTRGPolarity, uint8_t ExtTRGFilter)
功能描述	配置 PWMA 外部时钟模式 2
参数 1	PWMA_ExtTRGPrescaler 指定外部触发器预缩放器。 *此参数可以是以下值之一: *-PWMA_EXTTRGPSC_OFF *-PWMA_EXTTRGPSC_DIV2 *-PWMA_EXTTRGPSC_DIV4 *-PWMA_EXTTRGPSC_DIV8。
参数 2	PWMA_ExtTRGPolarity 指定外部触发极性。 *此参数可以是以下值之一: *-PWMA_EXTTRGPOLARITY_INVERTED *-PWMA_EXTTRGPOLARITY_NONINVERTED
参数 3	ExtTRGFilter 指定外部触发器滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.17 配置 PWMA 外部触发器

函数名	void PWMA_ETRConfig(PWMA_ExtTRGPSC_TypeDef PWMA_ExtTRGPrescaler, PWMA_ExtTRGPolarity_TypeDef PWMA_ExtTRGPolarity, uint8_t ExtTRGFilter)
功能描述	配置 PWMA 外部时钟模式 2
参数 1	PWMA_ExtTRGPrescaler 指定外部触发器预缩放器。 *此参数可以是以下值之一: *-PWMA_EXTTRGPSC_OFF *-PWMA_EXTTRGPSC_DIV2 *-PWMA_EXTTRGPSC_DIV4 *-PWMA_EXTTRGPSC_DIV8。
参数 2	PWMA_ExtTRGPolarity 指定外部触发极性。 *此参数可以是以下值之一: *-PWMA_EXTTRGPOLARITY_INVERTED *-PWMA_EXTTRGPOLARITY_NONINVERTED
参数 3	ExtTRGFilter 指定外部触发器滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.18 将 PWMA 触发器配置为外部时钟

函数名	void PWMA_TIxExternalClockConfig(

	PWMA_TIxExternalCLK1Source_TypeDef PWMA_TIxExternalCLKSource, PWMA_ICPolarity_TypeDef PWMA_ICPolarity, uint8_t ICFilter)
功能描述	将 PWMA 触发器配置为外部时钟
参数 1	PWMA_TIxExternalCLKSource 指定触发器源。 *此参数可以是以下值之一: *-PWMA_TIXEXTERNALCLK1SOURCE_TI1:TI1 边缘检测器 *-PWMA_TIXEXTERNALCLK1SOURCE_TI2：过滤的 PWMA 输入 1 *-PWMA_TIXEXTERNALCLK1SOURCE_TI1ED：过滤的 PWMA 输入 2
参数 2	PWMA_ICP 极性指定 TIx 极性。 *此参数可以是: *-PWMA_ICPOLARITY_RISING *-PWMA_ICPOLARITY_FALLING
参数 3	ICFilter 指定滤波器值。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.19 配置 PWMA 输入触发源

函数名	void PWMA_SelectInputTrigger(PWMA_TS_TypeDef PWMA_InputTriggerSource)
功能描述	选择 PWMA 输入触发源
参数 1	PWMA_InputTriggerSource 指定输入触发源。 *此参数可以是以下值之一: *-PWMA_TS_TI1F_ED:TI1 边缘检测器 *-PWMA_TS_TI1FP1：过滤定时器输入 1 *-PWMA_TS_TI2FP2：过滤定时器输入 2 *PWMA_TS_ETRF：外部触发输入
返回	无

4.12.20 启用或禁用 PWMA Update 事件

函数名	void PWMA_UpdateDisableConfig(FunctionalState NewState)
功能描述	启用或禁用 PWMA Update 事件
参数 1	NewState PWMA 外围预加载寄存器的新状态。此参数可以 *为 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.21 配置 PWMA 更新请求中断源

函数名	void PWMA_UpdateRequestConfig(PWMA_UpdateSource_TypeDef PWMA_UpdateSource)
功能描述	选择 PWMA 更新请求中断源
参数 1	PWMA_UpdateSource 指定更新源。 *此参数可以是以下值之一 - PWMA_UPDATESOURCE_REGULAR - PWMA_UPDATESOURCE_GLOBAL
参数 2	
参数 3	
返回	无

4.12.22 启用或禁用 PWMA 霍尔传感器

函数名	void PWMA_SelectHallSensor(FunctionalState NewState)
功能描述	启用或禁用 PWMA 霍尔传感器
参数 1	NewState PWMA 霍尔传感器接口的新状态。此参数可以 *为 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.23 选择 PWMA 单脉冲模式

函数名	void PWMA_SelectOnePulseMode(PWMA_OPMode_TypeDef PWMA_OPMode)
功能描述	选择 PWMA 单脉冲模式
参数 1	PWMA_OPMode 指定要使用的 OPM 模式。 *此参数可以是以下值之一 *-PWMA_OPMODE_SINGLE *-PWMA_OPMODE_repetive
返回	无

4.12.24 选择 PWMA 触发器输出模式

函数名	void PWMA_SelectOutputTrigger(PWMA_TRGOSource_TypeDef PWMA_TRGOSource)
功能描述	选择 PWMA 触发器输出模式
参数 1	PWMA_TRGOSource 指定触发输出源。

	*此参数可以是以下值之一 * PWMA_TRGOSOURCE_RESET * - PWMA_TRGOSOURCE_ENABLE * - PWMA_TRGOSOURCE_UPDATE * - PWMA_TRGOSource_OC1 * - PWMA_TRGOSOURCE_OC1REF * - PWMA_TRGOSOURCE_OC2REF * - PWMA_TRGOSOURCE_OC3REF
返回	无

4.12.25 选择 PWMA 从模式

函数名	void PWMA_SelectSlaveMode(PWMA_SlaveMode_TypeDef PWMA_SlaveMode)
功能描述	选择 PWMA 从模式
参数 1	PWMA_SlaveMode 指定 PWMA 从模式。 *此参数可以是以下值之一 *-PWMA_SLAVEMODE_RESET *-PWMA_SLAVEMODE_GATED *- PWMA_SLAVEMODE_TRIGGER *-PWMA_SLAVEMODE_EXTERNAL1
返回	无

4.12.26 设置 PWMA 主/从模式

函数名	void PWMA_SelectMasterSlaveMode(FunctionalState NewState)
功能描述	设置或重置 PWMA 主/从模式
参数 1	NewState PWMA 及其从属设备之间同步的新状态 *（通过 TRGO）。此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.27 配置 PWMA 编码器接口

函数名	void PWMA_EncoderInterfaceConfig(PWMA_EncoderMode_TypeDef PWMA_EncoderMode, PWMA_ICPolarity_TypeDef PWMA_IC1Polarity, PWMA_ICPolarity_TypeDef PWMA_IC2Polarity)

功能描述	配置 PWMA 编码器接口
参数 1	PWMA_EncoderMode 指定 PWMA 编码器模式。 *此参数可以是以下值之一 *-PWMA_ENCODERMODE_TI1：TI1FP1 边缘上的计数器计数 *取决于 TI2FP2 水平。 *-PWMA_ENCODERMODE_TI2：TI2FP2 边缘上的计数器计数 *取决于 TI1FP1 水平。 *-PWMA_ENCODERMODE_TI12：TI1FP1 和 *TI2FP2 边缘取决于其他输入的电平。
参数 2	PWMA_IC1 极性指定 IC1 极性。 *此参数可以是以下值之一 *-PWMA_ICPOLARITY_FALLING *-PWMA_ICPOLARITY_RISING
参数 3	PWMA_IC2 极性指定 IC2 极性。 *此参数可以是以下值之一 *-PWMA_ICPOLARITY_FALLING *-PWMA_ICPOLARITY_RISING
返回	无

4.12.28 配置 PWMA 预分频器

函数名	void PWMA_PrescalerConfig(uint16_t Prescaler, PWMA_PSCReloadMode_TypeDef PWMA_PSCReloadMode)
功能描述	配置 PWMA 预分频器
参数 1	Prescaler 指定 Prescaler 寄存器值 *此参数的值必须介于 0x0000 和 0xFFFF 之间
参数 2	PWMA_PSCReloadMode 指定 PWMA 预缩放器重新加载模式。 *此参数可以是以下值之一 *-PWMA_PSCRELOADMODE_IMMEDIATE：预缩放器立即加载。 *-PWMA_PSCREELOADMODE_UPDATE：在更新事件时加载预缩放器
参数 3	
返回	无

4.12.29 指定要使用的 PWMA 计数器模式

函数名	void PWMA_CounterModeConfig(PWMA_CounterMode_TypeDef PWMA_CounterMode)
功能描述	指定要使用的 PWMA 计数器模式
参数 1	PWMA_CounterMode 指定要使用的计数器模式 *此参数可以是以下值之一:

	*-PWMA_COUNTERMODE_UP:PWMA 递增计数模式 *-PWMA_COUNTERMODE_DOWN:PWMA 递减计数模式 *-PWMA_COUNTERMODE_CENTERALIGNED1:PWMA 中心对齐模式 1 *-PWMA_CounterMode_CenterAligned2:PWMA 中心对齐模式 2 *-PWMA_COUNTERMODE_CENTERALIGNED3:PWMA 中心对齐模式 3
返回	无

4.12.30 强制 PWMA 通道 1 输出高低电平

函数名	void PWMA_ForcedOC1Config(PWMA_ForcedAction_TypeDef PWMA_ForcedAction)
功能描述	强制 PWMA 通道 1 输出波形为高低电平
参数 1	PWMA_ForcedAction 指定要设置为输出波形的强制动作。 *此参数可以是以下值之一: *-PWMA_FORCEDACTION_ACTIVE：强制 OC1REF 上的高电平 *-PWMA_FORCEDACTION_INACTIVE：强制 OC1REF 处于低电平。
返回	无

4.12.31 强制 PWMA 通道 2 输出高低电平

函数名	void PWMA_ForcedOC2Config(PWMA_ForcedAction_TypeDef PWMA_ForcedAction)
功能描述	强制 PWMA 通道 2 输出波形为高低电平
参数 1	PWMA_ForcedAction 指定要设置为输出波形的强制动作。 *此参数可以是以下值之一: *-PWMA_FORCEDACTION_ACTIVE：强制 OC2REF 上的高电平 *-PWMA_FORCEDACTION_INACTIVE：强制 OC2REF 处于低电平。
返回	无

4.12.32 强制 PWMA 通道 3 输出高低电平

函数名	void PWMA_ForcedOC3Config(PWMA_ForcedAction_TypeDef PWMA_ForcedAction)
功能描述	强制 PWMA 通道 3 输出波形为高低电平
参数 1	PWMA_ForcedAction 指定要设置为输出波形的强制动作。 *此参数可以是以下值之一: *-PWMA_FORCEDACTION_ACTIVE：强制 OC3REF 上的高电平 *-PWMA_FORCEDACTION_INACTIVE：强制 OC3REF 处于低电平。
返回	无

4.12.33 强制 PWMA 通道 4 输出高低电平

函数名	void PWMA_ForcedOC4Config(PWMA_ForcedAction_TypeDef PWMA_ForcedAction)
功能描述	强制 PWMA 通道 4 输出波形为高低电平
参数 1	PWMA_ForcedAction 指定要设置为输出波形的强制动作。 *此参数可以是以下值之一: *-PWMA_FORCEDACTION_ACTIVE：强制 OC4REF 上的高电平 *-PWMA_FORCEDACTION_INACTIVE：强制 OC4REF 处于低电平。
返回	无

4.12.34 启用或禁用 ARR 上的 PWMA 预加载寄存器

函数名	void PWMA_ARRPreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 ARR 上的 PWMA 外围预加载寄存器
参数 1	NewState PWMA 外围预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.35 选择 PWMA com 事件

函数名	void PWMA_SelectCOM(FunctionalState NewState)
功能描述	选择 PWMA 外围 事件
参数 1	NewState 通勤事件的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.36 设置 PWMA 外围设备捕获比较预加载控制位

函数名	void PWMA_CCPreloadControl(FunctionalState NewState)
功能描述	设置或重置 PWMA 外围设备捕获比较预加载控制位
参数 1	NewState 捕获比较预加载控制位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.37 设置 CCR1 上的 PWMA 预加载寄存器

函数名	void PWMA_OC1PreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 CCR1 上的 PWMA 预加载寄存器
参数 1	NewState 捕获比较预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.38 设置 CCR2 上的 PWMA 预加载寄存器

函数名	void PWMA_OC2PreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 CCR2 上的 PWMA 预加载寄存器
参数 1	NewState 捕获比较预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.39 设置 CCR3 上的 PWMA 预加载寄存器

函数名	void PWMA_OC3PreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 CCR3 上的 PWMA 预加载寄存器
参数 1	NewState 捕获比较预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.40 设置 CCR4 上的 PWMA 预加载寄存器

函数名	void PWMA_OC4PreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 CCR4 上的 PWMA 预加载寄存器
参数 1	NewState 捕获比较预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.41 配置 PWMA 捕获比较 1 快速使能

函数名	void PWMA_OC1FastConfig(FunctionalState NewState)
功能描述	配置 PWMA 捕获比较 1 快速功能
参数 1	NewState 输出比较快速启用位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.42 配置 PWMA 捕获比较 2 快速使能

函数名	void PWMA_OC2FastConfig(FunctionalState NewState)
功能描述	配置 PWMA 捕获比较 1 快速功能
参数 1	NewState 输出比较快速启用位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.43 配置 PWMA 捕获比较 3 快速使能

函数名	void PWMA_OC3FastConfig(FunctionalState NewState)
功能描述	配置 PWMA 捕获比较 1 快速功能
参数 1	NewState 输出比较快速启用位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.44 配置 PWMA 捕获比较 4 快速使能

函数名	void PWMA_OC4FastConfig(FunctionalState NewState)
功能描述	配置 PWMA 捕获比较 1 快速功能
参数 1	NewState 输出比较快速启用位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.45 配置要由软件生成的 PWMA 事件

函数名	void PWMA_GenerateEvent(PWMA_EventSource_TypeDef PWMA_EventSource)
功能描述	配置要由软件生成的 PWMA 事件
参数 1	PWMA_EventSource 指定事件源。 *此参数可以是以下值之一: *-PWMA_EVENTSOURCE_UPDATE:PWMA 更新事件源 *-PWMA_EVENTSOURCE_CC1:PWMA 捕获比较 1 事件源 *-PWMA_EVENTSOURCE_CC2:PWMA 捕获比较 2 事件源 *-PWMA_EVENTSOURCE_CC3:PWMA 捕获比较 3 事件源 *-PWMA_EVENTSOURCE_CC4:PWMA 捕获比较 4 事件源 *-PWMA_EVENTSOURCE_COM:PWMA COM 事件源 *-PWMA_EVENTSOURCE_TRIGGER:PWMA 触发事件源 *-PWMA_EventSourceBreak:PWMA Break 事件源
返回	无

4.12.46 配置 PWMA 通道 1 极性

函数名	void PWMA_OC1PolarityConfig(PWMA_OCPolarity_TypeDef PWMA_OCPolarity)
功能描述	配置 PWMA 通道 1 极性
参数 1	PWMA_OCP 极性指定 OC1 极性。 *此参数可以是以下值之一: *-PWMA_OCPOLARITY_LOW: 输出比较激活低 *-PWMA_OPOLARITY_HIGH: 输出比较激活高
返回	无

4.12.47 配置 PWMA 通道 2 极性

函数名	void PWMA_OC2PolarityConfig(PWMA_OCPolarity_TypeDef PWMA_OCPolarity)
功能描述	配置 PWMA 通道 2 极性
参数 1	PWMA_OCP 极性指定 OC2 极性。 *此参数可以是以下值之一: *-PWMA_OCPOLARITY_LOW: 输出比较激活低 *-PWMA_OPOLARITY_HIGH: 输出比较激活高
返回	无

4.12.48 配置 PWMA 通道 3 极性

函数名	void PWMA_OC3PolarityConfig(PWMA_OCPolarity_TypeDef PWMA_OCPolarity)
功能描述	配置 PWMA 通道 3 极性
参数 1	PWMA_OCP 极性指定 OC3 极性。 *此参数可以是以下值之一: *-PWMA_OCPOLARITY_LOW: 输出比较激活低 *-PWMA_OPOLARITY_HIGH: 输出比较激活高
返回	无

4.12.49 配置 PWMA 通道 4 极性

函数名	void PWMA_OC4PolarityConfig(PWMA_OCPolarity_TypeDef PWMA_OCPolarity)
功能描述	配置 PWMA 通道 4 极性
参数 1	PWMA_OCP 极性指定 OC4 极性。

	*此参数可以是以下值之一: *-PWMA_OCPOLARITY_LOW: 输出比较激活低 *-PWMA_OPOLARITY_HIGH: 输出比较激活高
返回	无

4.12.50 配置 PWMA 通道 1N 极性

函数名	void PWMA_OC1NPolarityConfig(PWMA_OCNPolarity_TypeDef PWMA_OCNPolarity)
功能描述	配置 PWMA 通道 1N 极性
参数 1	PWMA_OCNPolarity 指定 OC1N 极性。 *此参数可以是以下值之一: *-PWMA_OCNPOLARITY_LOW: 输出比较激活低 *-PWMA_OCNPOLARITY_HHIGH: 输出比较激活高
返回	无

4.12.51 配置 PWMA 通道 2N 极性

函数名	void PWMA_OC2NPolarityConfig(PWMA_OCNPolarity_TypeDef PWMA_OCNPolarity)
功能描述	配置 PWMA 通道 2N 极性
参数 1	PWMA_OCNPolarity 指定 OC2N 极性。 *此参数可以是以下值之一: *-PWMA_OCNPOLARITY_LOW: 输出比较激活低 *-PWMA_OCNPOLARITY_HHIGH: 输出比较激活高
返回	无

4.12.52 配置 PWMA 通道 3N 极性

函数名	void PWMA_OC3NPolarityConfig(PWMA_OCNPolarity_TypeDef PWMA_OCNPolarity)
功能描述	配置 PWMA 通道 3N 极性
参数 1	PWMA_OCNPolarity 指定 OC3N 极性。 *此参数可以是以下值之一: *-PWMA_OCNPOLARITY_LOW: 输出比较激活低 *-PWMA_OCNPOLARITY_HHIGH: 输出比较激活高
返回	无

4.12.53 配置 PWMA 通道 4N 极性

函数名	void PWMA_OC4NPolarityConfig(PWMA_OCNPolarity_TypeDef PWMA_OCNPolarity)
功能描述	配置 PWMA 通道 4N 极性
参数 1	PWMA_OCNPolarity 指定 OC4N 极性。 *此参数可以是以下值之一: *-PWMA_OCNPOLARITY_LOW: 输出比较激活低 *-PWMA_OCNPOLARITY_HHIGH: 输出比较激活高
返回	无

4.12.54 启用或禁用 PWMA 捕获比较通道 P

函数名	void PWMA_CCxCmd(PWMA_Channel_TypeDef PWMA_Channel, FunctionalState NewState)
功能描述	启用或禁用 PWMA 捕获比较通道 x（x=1，..，4）
参数 1	PWMA_Channel 指定 PWMA 通道。 *此参数可以是以下值之一: *-PWMA_CHANNEL_1:PWMA 通道 1 *-PWMA_CHANNEL_2:PWMA 通道 2 *-PWMA_CHANNEL_3:PWMA 通道 3 *-PWMA_CHANNEL_4:PWMA 通道 4
参数 2	NewState 指定 PWMA 信道 CCxE 位的新状态。 *此参数可以是: ENABLE 或 DISABLE
返回	无

4.12.55 启用或禁用 PWMA 捕获比较通道 N

函数名	void PWMA_CCxNCmd(PWMA_Channel_TypeDef PWMA_Channel, FunctionalState NewState)
功能描述	启用或禁用 PWMA 捕获比较通道 xN（x=1，..，4）
参数 1	PWMA_Channel 指定 PWMA 通道。 *此参数可以是以下值之一: *-PWMA_CHANNEL_1:PWMA 通道 1 *-PWMA_CHANNEL_2:PWMA 通道 2 *-PWMA_CHANNEL_3:PWMA 通道 3 *-PWMA_CHANNEL_4:PWMA 通道 4
参数 2	NewState 指定 PWMA 信道 CCxNE 位的新状态。 *此参数可以是: ENABLE 或 DISABLE
返回	无

4.12.56 配置 PWMA 输出比较模式

函数名	void PWMA_SelectOCxM(PWMA_Channel_TypeDef PWMA_Channel, PWMA_OCMode_TypeDef PWMA_OCMode)
功能描述	选择 PWMA 输出比较模式。此功能禁用在更改"输出比较模式"之前选择的频道。用户必须*使用 PWMA_CCxCmd 和 PWMA_CCxNCmd 函数启用此通道
参数 1	PWMA_Channel 指定 PWMA 通道。 *此参数可以是以下值之一: *-PWMA_CHANNEL_1:PWMA 通道 1 *-PWMA_CHANNEL_2:PWMA 通道 2 *-PWMA_CHANNEL_3:PWMA 通道 3 *-PWMA_CHANNEL_4:PWMA 通道 4
参数 2	PWMA_OCMode 指定 PWMA 输出比较模式。 *此参数可以是以下值之一: *-PWMA_OCMODE_TIMING *-PWMA_OCMODE_ACTIVE *-PWMA_OCMODE_TOGGLE *-PWMA_OCMODE_PWM1 *-PWMA_OCMODE_PWM2 *-PWMA_frcedaction_ACTIVE *-PWMA_FORCEDACTION_INACTIVE
返回	无

4.12.57 设置 PWMA 计数器寄存器值

函数名	void PWMA_SetCounter(uint16_t Counter)
功能描述	设置 PWMA 计数器寄存器值
参数 1	Counter 指定 Counter 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.58 设置 PWMA 自动重新加载寄存器值

函数名	void PWMA_SetAutoreload(uint16_t Autoreload)
功能描述	设置 PWMA 自动重新加载寄存器值
参数 1	Autoreload 指定自动重新加载寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.59 设置 PWMA 捕获比较器 1 寄存器值

函数名	void PWMA_SetCompare1(uint16_t Compare1)
功能描述	设置 PWMA 捕获比较 1 寄存器值
参数 1	Compare1 指定 Capture Compare1 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.60 设置 PWMA 捕获比较器 2 寄存器值

函数名	void PWMA_SetCompare2(uint16_t Compare1)
功能描述	设置 PWMA 捕获比较 2 寄存器值
参数 1	Compare2 指定 Capture Compare2 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.61 设置 PWMA 捕获比较器 3 寄存器值

函数名	void PWMA_SetCompare3(uint16_t Compare1)
功能描述	设置 PWMA 捕获比较 3 寄存器值
参数 1	Compare3 指定 Capture Compare3 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.62 设置 PWMA 捕获比较器 4 寄存器值

函数名	void PWMA_SetCompare4(uint16_t Compare1)
功能描述	设置 PWMA 捕获比较 4 寄存器值
参数 1	Compare4 指定 Capture Compare4 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.63 设置 PWMA 输入捕获 1 预分频器

函数名	void PWMA_SetIC1Prescaler(PWMA_ICPSC_TypeDef PWMA_IC1Prescaler)
功能描述	设置 PWMA 输入捕获 1 预分频器
参数 1	PWMA_IC1 预分频器指定输入捕获预分频器的新值 *此参数可以是以下值之一: *-PWMA_ICPSC_DIV1: 无预分频器 *-PWMA_ICPSC_DIV2: 每 2 个事件捕获一次

	*-PWMA_ICPSC_DIV4：每 4 个事件捕获一次 *-PWMA_ICPSC_DIV8：每 8 个事件捕获一次
返回	无

4.12.64 设置 PWMA 输入捕获 2 预分频器

函数名	void PWMA_SetIC2Prescaler(PWMA_ICPSC_TypeDef PWMA_IC1Prescaler)
功能描述	设置 PWMA 输入捕获 2 预分频器
参数 1	PWMA_IC2 预分频器指定输入捕获预分频器的新值 *此参数可以是以下值之一: *-PWMA_ICPSC_DIV1：无预分频器 *-PWMA_ICPSC_DIV2：每 2 个事件捕获一次 *-PWMA_ICPSC_DIV4：每 4 个事件捕获一次 *-PWMA_ICPSC_DIV8：每 8 个事件捕获一次
返回	无

4.12.65 设置 PWMA 输入捕获 3 预分频器

函数名	void PWMA_SetIC3Prescaler(PWMA_ICPSC_TypeDef PWMA_IC1Prescaler)
功能描述	设置 PWMA 输入捕获 3 预分频器
参数 1	PWMA_IC3 预分频器指定输入捕获预分频器的新值 *此参数可以是以下值之一: *-PWMA_ICPSC_DIV1：无预分频器 *-PWMA_ICPSC_DIV2：每 2 个事件捕获一次 *-PWMA_ICPSC_DIV4：每 4 个事件捕获一次 *-PWMA_ICPSC_DIV8：每 8 个事件捕获一次
返回	无

4.12.66 设置 PWMA 输入捕获 4 预分频器

函数名	void PWMA_SetIC4Prescaler(PWMA_ICPSC_TypeDef PWMA_IC1Prescaler)
功能描述	设置 PWMA 输入捕获 4 预分频器
参数 1	PWMA_IC4 预分频器指定输入捕获预分频器的新值 *此参数可以是以下值之一: *-PWMA_ICPSC_DIV1：无预分频器 *-PWMA_ICPSC_DIV2：每 2 个事件捕获一次 *-PWMA_ICPSC_DIV4：每 4 个事件捕获一次 *-PWMA_ICPSC_DIV8：每 8 个事件捕获一次
返回	无

4.12.67 获取 PWMA 输入捕获 1 的值

函数名	uint16_t PWMA_GetCapture1(void)
功能描述	获取 PWMA 输入捕获 1 的值
参数	无
返回	Capture 比较 1 的寄存器值

4.12.68 获取 PWMA 输入捕获 2 的值

函数名	uint16_t PWMA_GetCapture2(void)
功能描述	获取 PWMA 输入捕获 2 的值
参数	无
返回	Capture 比较 2 的寄存器值

4.12.69 获取 PWMA 输入捕获 3 的值

函数名	uint16_t PWMA_GetCapture3(void)
功能描述	获取 PWMA 输入捕获 3 的值
参数	无
返回	Capture 比较 3 的寄存器值

4.12.70 获取 PWMA 输入捕获 4 的值

函数名	uint16_t PWMA_GetCapture4(void)
功能描述	获取 PWMA 输入捕获 4 的值
参数	无
返回	Capture 比较 4 的寄存器值

4.12.71 获取 PWMA 计数器值

函数名	uint16_t PWMA_GetCounter(void)
功能描述	获取 PWMA 计数器值
参数	无
返回	计数器寄存器值

4.12.72 获取 PWMA 预分频器值

函数名	uint16_t PWMA_GetPrescaler(void)
功能描述	获取 PWMA 预分频器值

参数	无
返回	预分频器寄存器值

4.12.73 获取指定的 PWMA 标志

函数名	FlagStatus PWMA_GetFlagStatus(PWMA_FLAG_TypeDef PWMA_FLAG)
功能描述	检查是否设置了指定的 PWMA 标志
参数 1	PWMA_FLAG 指定要检查的标志。 *此参数可以是以下值之一: *-PWMA_FLAG_UPDATE:PWMA 更新标志 *-PWMA_FLAG_CC1:PWMA 捕获比较 1 标志 *-PWMA_FLAG_CC2:PWMA 捕获比较 2 标志 *-PWMA_FLAG_CC3:PWMA 捕获比较 3 标志 *-PWMA_FLAG_CC4:PWMA 捕获比较 4 标志 *-PWMA_FLAG_COMM:PWMA 交换标志 *-PWMA_FLAG_TRIGGER:PWMA 触发标志 *-PWMA_FLAG_BREAK:PWMA 中断标志 *-PWMA_FLAG_CC1F:PWMA 捕获比较 1 过捕获标志 *-PWMA_FLAG_CC2OF:PWMA 捕获比较 2 过捕获标志 *-PWMA_FLAG_CC3OF: PWMA 捕获比较 3 过捕获标志 *-PWMA_FLAG_CC4OF:PWMA 捕获比较 4 过捕获标志
返回	FlagStatus PWMA_FLAG 的新状态（SET 或 RESET）

4.12.74 清除 PWMA 挂起标志

函数名	void PWMA_ClearFlag(PWMA_FLAG_TypeDef PWMA_FLAG)
功能描述	清除 PWMA 挂起标志
参数 1	PWMA_FLAG 指定要清除的标志。 *此参数可以是以下值之一: *-PWMA_FLAG_UPDATE:PWMA 更新标志 *-PWMA_FLAG_CC1:PWMA 捕获比较 1 标志 *-PWMA_FLAG_CC2:PWMA 捕获比较 2 标志 *-PWMA_FLAG_CC3:PWMA 捕获比较 3 标志 *-PWMA_FLAG_CC4:PWMA 捕获比较 4 标志 *-PWMA_FLAG_COMM:PWMA 交换标志 *-PWMA_FLAG_TRIGGER:PWMA 触发标志 *-PWMA_FLAG_BREAK:PWMA 中断标志 *-PWMA_FLAG_CC1F:PWMA 捕获比较 1 过捕获标志 *-PWMA_FLAG_CC2OF:PWMA 捕获比较 2 过捕获标志 *-PWMA_FLAG_CC3OF: PWMA 捕获比较 3 过捕获标志 *-PWMA_FLAG_CC4OF:PWMA 捕获比较 4 过捕获标志
返回	无

4.12.75 检查 PWMA 中断是否发生

函数名	ITStatus PWMA_GetITStatus(PWMA_IT_TypeDef PWMA_IT)
功能描述	检查 PWMA 中断是否发生
参数 1	PWMA_IT 指定要检查的 PWMA 中断源。 *此参数可以是以下值之一: *-PWMA_IT_UPDATE: PWMA 更新中断源 *-PWMA_IT_CC1:PWMA 捕获比较 1 中断源 *-PWMA_IT_CC2:PWMA 捕获比较 2 中断源 *-PWMA_IT_CC3:PWMA 捕获比较 3 中断源 *-PWMA_IT_CC4:PWMA 捕获比较 4 中断源 *-PWMA_IT_COM:PWMA 换向中断源 *-PWMA_IT_TRIGGER:PWMA 触发中断源 *-PWMA_IT_BEAK:PWMA 中断源
返回	ITStatus PWMA_IT 的新状态（SET 或 RESET）

4.12.76 清除 PWMA 的中断挂起位

函数名	void PWMA_ClearITPendingBit(PWMA_IT_TypeDef PWMA_IT)
功能描述	清除 PWMA 的中断挂起位
参数 1	PWMA_IT 指定要清除的挂起位。 *此参数可以是以下值之一: *-PWMA_IT_UPDATE: PWMA 更新中断源 *-PWMA_IT_CC1:PWMA 捕获比较 1 中断源 *-PWMA_IT_CC2:PWMA 捕获比较 2 中断源 *-PWMA_IT_CC3:PWMA 捕获比较 3 中断源 *-PWMA_IT_CC4:PWMA 捕获比较 4 中断源 *-PWMA_IT_COM:PWMA 换向中断源 *-PWMA_IT_TRIGGER:PWMA 触发中断源 *-PWMA_IT_BEAK:PWMA 中断源
返回	无

4.12.77 将 TI1 配置为输入

函数名	static void TI1_Config(uint8_t PWMA_ICPolarity, uint8_t PWMA_ICSelection, uint8_t PWMA_ICFilter)
功能描述	将 TI1 配置为输入
参数 1	PWMA_ICP 极性输入极性。

	*此参数可以是以下值之一: *-PWMA_ICPOLARITY_FALLING *-PWMA_ICPOLARITY_RISING
参数 2	PWMA_ICSelection 指定要使用的输入。 *此参数可以是以下值之一: *-PWMA_ICSELECTION_DIRECTTI:PWMA 输入 1 选择为 *连接到 IC1。 *-PWMA_ICSELECTION_INDIRECTTI:PWMA 输入 1 选择为 *连接到 IC2。
参数 3	PWMA_ICFilter 指定输入捕获滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.78 将 TI2 配置为输入

函数名	static void TI2_Config(uint8_t PWMA_ICPolarity, uint8_t PWMA_ICSelection, uint8_t PWMA_ICFilter)
功能描述	将 TI2 配置为输入
参数 1	PWMA_ICP 极性输入极性。 *此参数可以是以下值之一: *-PWMA_ICPOLARITY_FALLING *-PWMA_ICPOLARITY_RISING
参数 2	PWMA_ICSelection 指定要使用的输入。 *此参数可以是以下值之一: *-PWMA_ICSELECTION_DIRECTTI:PWMA 输入 2 选择为 *连接到 IC2。 *-PWMA_ICSELECTION_INDIRECTTI:PWMA 输入 2 选择为 *连接到 IC1。
参数 3	PWMA_ICFilter 指定输入捕获滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.79 将 TI3 配置为输入

函数名	static void TI3_Config(uint8_t PWMA_ICPolarity, uint8_t PWMA_ICSelection, uint8_t PWMA_ICFilter)
功能描述	将 TI3 配置为输入
参数 1	PWMA_ICP 极性输入极性。 *此参数可以是以下值之一: *-PWMA_ICPOLARITY_FALLING

	*-PWMA_ICPOLARITY_RISING
参数 2	PWMA_ICSelection 指定要使用的输入。 *此参数可以是以下值之一: *-PWMA_ICSELECTION_DIRECTTI:PWMA 输入 3 选择为 *连接到 IC3。 *-PWMA_ICSELECTION_INDIRECTTI:PWMA 输入 3 选择为 *连接到 IC4。
参数 3	PWMA_ICFilter 指定输入捕获滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.80 将 TI4 配置为输入

函数名	static void TI4_Config(uint8_t PWMA_ICPolarity, uint8_t PWMA_ICSelection, uint8_t PWMA_ICFilter)
功能描述	将 TI4 配置为输入
参数 1	PWMA_ICP 极性输入极性。 *此参数可以是以下值之一: *-PWMA_ICPOLARITY_FALLING *-PWMA_ICPOLARITY_RISING
参数 2	PWMA_ICSelection 指定要使用的输入。 *此参数可以是以下值之一: *-PWMA_ICSELECTION_DIRECTTI:PWMA 输入 4 选择为 *连接到 IC4。 *-PWMA_ICSELECTION_INDIRECTTI:PWMA 输入 4 选择为 *连接到 IC3。
参数 3	PWMA_ICFilter 指定输入捕获滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.81 PWMB 反初始化

函数名	void PWMB_DeInit(void)
功能描述	将 PWMB 外围寄存器取消初始化为其默认重置值
参数 1	无
返回	无

4.12.82 初始化 PWMB 时基单元值

函数名	void PWMB_TimeBaseInit(uint16_t PWMB_Prescaler, PWMB_CounterMode_TypeDef PWMB_CounterMode,

	uint16_t PWMB_Period, uint8_t PWMB_RepetitionCounter)
功能描述	根据指定的参数初始化 PWMB 时基单位
参数 1	PWMB_Pescaler 指定预缩放器值
参数 2	PWMB_CounterMode 从@ref PWMB_CoounterMode_TypeDef 指定计数器模式
参数 3	PWMB_Period 指定 Period 值
参数 4	PWMB_ReditionCounter 指定重复计数器值
返回	无

其中:

```
typedef enum
{
  PWMB_COUNTERMODE_UP                 = ((uint8_t)0x00),
  PWMB_COUNTERMODE_DOWN               = ((uint8_t)0x10),
  PWMB_COUNTERMODE_CENTERALIGNED1     = ((uint8_t)0x20),
  PWMB_COUNTERMODE_CENTERALIGNED2     = ((uint8_t)0x40),
  PWMB_COUNTERMODE_CENTERALIGNED3     = ((uint8_t)0x60)
}PWMB_CounterMode_TypeDef;
```

4.12.83 初始化 PWMB 通道 5

函数名	void PWMB_OC5Init(PWMB_OCMode_TypeDef PWMB_OCMode, PWMB_OutputState_TypeDef PWMB_OutputState, uint16_t PWMB_Pulse, PWMB_OCPolarity_TypeDef PWMB_OCPolarity, PWMB_OCIdleState_TypeDef PWMB_OCIdleState,)
功能描述	根据指定的参数初始化 PWMB 通道 5
参数 1	PWMB_OCMode 指定输出比较模式
参数 2	PWMB_OutputState 指定输出状态
参数 3	PWMB_Pulse 指定脉冲宽度值。
参数 4	PWMB_OCPolarity 指定输出比较极性
参数 5	PWMB_OCIdleState 指定输出比较空闲状态
返回	无

PWMB 输出比较和 PWM 模式:

```
typedef enum
{
```

```
    PWMB_OCMODE_TIMING      = ((uint8_t)0x00),
    PWMB_OCMODE_ACTIVE      = ((uint8_t)0x10),
    PWMB_OCMODE_INACTIVE    = ((uint8_t)0x20),
    PWMB_OCMODE_TOGGLE      = ((uint8_t)0x30),
    PWMB_OCMODE_PWM1        = ((uint8_t)0x60),
    PWMB_OCMODE_PWM2        = ((uint8_t)0x70)
}PWMB_OCMode_TypeDef;
```

4.12.84 初始化 PWMB 通道 6

函数名	void PWMB_OC6Init(PWMB_OCMode_TypeDef PWMB_OCMode, PWMB_OutputState_TypeDef PWMB_OutputState, uint16_t PWMB_Pulse, PWMB_OCPolarity_TypeDef PWMB_OCPolarity, PWMB_OCIdleState_TypeDef PWMB_OCIdleState,)
功能描述	根据指定的参数初始化 PWMB 通道 6
参数 1	PWMB_OCMode 指定输出比较模式
参数 2	PWMB_OutputState 指定输出状态
参数 3	PWMB_Pulse 指定脉冲宽度值。
参数 4	PWMB_OCPolarity 指定输出比较极性
参数 5	PWMB_OCIdleState 指定输出比较空闲状态
返回	无

PWMB 输出比较和 PWM 模式:

```
typedef enum
{
    PWMB_OCMODE_TIMING      = ((uint8_t)0x00),
    PWMB_OCMODE_ACTIVE      = ((uint8_t)0x10),
    PWMB_OCMODE_INACTIVE    = ((uint8_t)0x20),
    PWMB_OCMODE_TOGGLE      = ((uint8_t)0x30),
    PWMB_OCMODE_PWM1        = ((uint8_t)0x60),
    PWMB_OCMODE_PWM2        = ((uint8_t)0x70)
}PWMB_OCMode_TypeDef;
```

4.12.85 初始化 PWMB 通道 7

函数名	void PWMB_OC7Init(PWMB_OCMode_TypeDef PWMB_OCMode, PWMB_OutputState_TypeDef

	PWMB_OutputState, uint16_t PWMB_Pulse, PWMB_OCPolarity_TypeDef PWMB_OCPolarity, PWMB_OCIdleState_TypeDef PWMB_OCIdleState,)
功能描述	根据指定的参数初始化 PWMB 通道 7
参数 1	PWMB_OCMode 指定输出比较模式
参数 2	PWMB_OutputState 指定输出状态
参数 3	PWMB_Pulse 指定脉冲宽度值。
参数 4	PWMB_OCPolarity 指定输出比较极性
参数 5	PWMB_OCIdleState 指定输出比较空闲状态
返回	无

PWMB 输出比较和 PWM 模式:

```
typedef enum
{
    PWMB_OCMODE_TIMING      = ((uint8_t)0x00),
    PWMB_OCMODE_ACTIVE      = ((uint8_t)0x10),
    PWMB_OCMODE_INACTIVE    = ((uint8_t)0x20),
    PWMB_OCMODE_TOGGLE      = ((uint8_t)0x30),
    PWMB_OCMODE_PWM1        = ((uint8_t)0x60),
    PWMB_OCMODE_PWM2        = ((uint8_t)0x70)
}PWMB_OCMode_TypeDef;
```

4.12.86 初始化 PWMB 通道 8

函数名	void PWMB_OC8Init(PWMB_OCMode_TypeDef PWMB_OCMode, PWMB_OutputState_TypeDef PWMB_OutputState, uint16_t PWMB_Pulse, PWMB_OCPolarity_TypeDef PWMB_OCPolarity, PWMB_OCIdleState_TypeDef PWMB_OCIdleState,)
功能描述	根据指定的参数初始化 PWMB 通道 8
参数 1	PWMB_OCMode 指定输出比较模式
参数 2	PWMB_OutputState 指定输出状态
参数 3	PWMB_Pulse 指定脉冲宽度值。
参数 4	PWMB_OCPolarity 指定输出比较极性
参数 5	PWMB_OCIdleState 指定输出比较空闲状态

返回	无

PWMB 输出比较和 PWM 模式:

```
typedef enum
{
    PWMB_OCMODE_TIMING      = ((uint8_t)0x00),
    PWMB_OCMODE_ACTIVE      = ((uint8_t)0x10),
    PWMB_OCMODE_INACTIVE    = ((uint8_t)0x20),
    PWMB_OCMODE_TOGGLE      = ((uint8_t)0x30),
    PWMB_OCMODE_PWM1        = ((uint8_t)0x60),
    PWMB_OCMODE_PWM2        = ((uint8_t)0x70)
}PWMB_OCMode_TypeDef;
```

4.12.87 PWMB 输入捕获初始化

函数名	void PWMB_ICInit(PWMB_Channel_TypeDef PWMB_Channel, PWMB_ICPolarity_TypeDef PWMB_ICPolarity, PWMB_ICSelection_TypeDef PWMB_ICSelection, PWMB_ICPSC_TypeDef PWMB_ICPrescaler, uint8_t PWMB_ICFilter)
功能描述	根据指定的参数初始化 PWMB 外围设备
参数 1	PWMB_Channel 指定来自 PWMB_Cannel_TypeDef 的输入捕获通道
参数 2	PWMB_ICPolarity 指定输入捕获极性
参数 3	PWMB_ICSelection 指定输入捕获源选择
参数 4	PWMB_ICP 缩放器指定输入捕获预缩放器
参数 5	PWMB_ICFilter 指定输入捕获滤波器值
返回	无

PWMB 输入捕获预分频器

```
{
    PWMB_ICPSC_DIV1                 = ((uint8_t)0x00),
    PWMB_ICPSC_DIV2                 = ((uint8_t)0x04),
    PWMB_ICPSC_DIV4                 = ((uint8_t)0x08),
    PWMB_ICPSC_DIV8                 = ((uint8_t)0x0C)
}PWMB_ICPSC_TypeDef;
```

4.12.88 PWMB 输入捕获配置

函数名	void PWMB_PWMIConfig(PWMB_Channel_TypeDef PWMB_Channel, PWMB_ICPolarity_TypeDef PWMB_ICPolarity, PWMB_ICSelection_TypeDef PWMB_ICSelection, PWMB_ICPSC_TypeDef PWMB_ICPrescaler, uint8_t PWMB_ICFilter)

功能描述	根据指定参数在 PWM 输入模式下配置 PWMB 外围设备
参数 1	配置通道 PWMB_Channel
参数 2	配置极性 PWMB_ICPolarity
参数 3	配置选择源 PWMB_ICSelection
参数 4	配置预缩放器
参数 5	PWMB_ICFilter 指定输入捕获滤波器值
返回	无

4.12.89 启用或禁用 PWMB

函数名	void PWMB_Cmd(FunctionalState NewState)
功能描述	启用或禁用 PWMB 外围设备
参数 1	NewState PWMB 外备的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.90 启用或禁用 PWMB 设备主输出

函数名	void PWMB_CtrlPWMOutputs(FunctionalState NewState)
功能描述	启用或禁用 PWMB 设备主输出
参数 1	NewState PWMB 外备的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.91 启用或禁用指定的 PWMB 中断

函数名	void PWMB_ITConfig(PWMB_IT_TypeDef PWMB_IT, FunctionalState NewState)
功能描述	启用或禁用指定的 PWMB 中断
参数 1	PWMB_IT 指定要启用或禁用的 PWMB 中断源。 *此参数可以是以下值的任意组合: *-PWMB_IT_UPDATE：PWMB 更新中断源 *-PWMB_IT_CC1:PWMB 捕获比较 5 中断源 *-PWMB_IT_CC2:PWMB 捕获比较 6 中断源 *-PWMB_IT_CC3:PWMB 捕获比较 7 中断源 *-PWMB_IT_CC4:PWMB 捕获比较 8 中断源 *-PWMB_IT_CU 更新：PWMB 捕获比较更新中断源

	*-PWMB_IT_TRIGGER:PWMB 触发中断源 *-PWMB_IT_BEAK:PWMB 中断源
参数 2	NewState PWMB 外围设备的新状态
返回	无

PWMB 中断源

```
typedef enum
{
  PWMB_IT_UPDATE                  = ((uint8_t)0x01),
  PWMB_IT_CC1                     = ((uint8_t)0x02),
  PWMB_IT_CC2                     = ((uint8_t)0x04),
  PWMB_IT_CC3                     = ((uint8_t)0x08),
  PWMB_IT_CC4                     = ((uint8_t)0x10),
  PWMB_IT_COM                     = ((uint8_t)0x20),
  PWMB_IT_TRIGGER                 = ((uint8_t)0x40),
  PWMB_IT_BREAK                   = ((uint8_t)0x80)
}PWMB_IT_TypeDef;
```

4.12.92 配置 PWMB 内部时钟

函数名	void PWMB_InternalClockConfig(void)
功能描述	启用配置 PWMB 内部时钟
参数 1	无
返回	无

4.12.93 配置 PWMB 外部时钟模式 1

函数名	void PWMB_ETRClockMode1Config(PWMB_ExtTRGPSC_TypeDef PWMB_ExtTRGPrescaler, PWMB_ExtTRGPolarity_TypeDef PWMB_ExtTRGPolarity, uint8_t ExtTRGFilter)
功能描述	配置 PWMB 外部时钟模式 1
参数 1	PWMB_ExtTRGPrescaler 指定外部触发器预缩放器。 *此参数可以是以下值之一: *-PWMB_EXTTRGPSC_OFF *-PWMB_EXTTRGPSC_DIV2 *-PWMB_EXTTRGPSC_DIV4 *-PWMB_EXTTRGPSC_DIV8。
参数 2	PWMB_ExtTRGPolarity 指定外部触发极性。 *此参数可以是以下值之一: *-PWMB_EXTTRGPOLARITY_INVERTED

	*-PWMB_EXTTRGPOLARITY_NONINVERTED
参数 3	ExtTRGFilter 指定外部触发器滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.94 配置 PWMB 外部时钟模式 2

函数名	void PWMB_ETRClockMode2Config(PWMB_ExtTRGPSC_TypeDef PWMB_ExtTRGPrescaler, PWMB_ExtTRGPolarity_TypeDef PWMB_ExtTRGPolarity, uint8_t ExtTRGFilter)
功能描述	配置 PWMB 外部时钟模式 2
参数 1	PWMB_ExtTRGPrescaler 指定外部触发器预缩放器。 *此参数可以是以下值之一: *-PWMB_EXTTRGPSC_OFF *-PWMB_EXTTRGPSC_DIV2 *-PWMB_EXTTRGPSC_DIV4 *-PWMB_EXTTRGPSC_DIV8。
参数 2	PWMB_ExtTRGPolarity 指定外部触发极性。 *此参数可以是以下值之一: *-PWMB_EXTTRGPOLARITY_INVERTED *-PWMB_EXTTRGPOLARITY_NONINVERTED
参数 3	ExtTRGFilter 指定外部触发器滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.95 配置 PWMB 外部触发器

函数名	void PWMB_ETRConfig(PWMB_ExtTRGPSC_TypeDef PWMB_ExtTRGPrescaler, PWMB_ExtTRGPolarity_TypeDef PWMB_ExtTRGPolarity, uint8_t ExtTRGFilter)
功能描述	配置 PWMB 外部时钟模式 2
参数 1	PWMB_ExtTRGPrescaler 指定外部触发器预缩放器。 *此参数可以是以下值之一: *-PWMB_EXTTRGPSC_OFF *-PWMB_EXTTRGPSC_DIV2 *-PWMB_EXTTRGPSC_DIV4 *-PWMB_EXTTRGPSC_DIV8。
参数 2	PWMB_ExtTRGPolarity 指定外部触发极性。 *此参数可以是以下值之一:

	*-PWMB_EXTTRGPOLARITY_INVERTED *-PWMB_EXTTRGPOLARITY_NONINVERTED
参数 3	ExtTRGFilter 指定外部触发器滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.96 将 PWMB 触发器配置为外部时钟

函数名	void PWMB_TIxExternalClockConfig(PWMB_TIxExternalCLK1Source_TypeDef PWMB_TIxExternalCLKSource, PWMB_ICPolarity_TypeDef PWMB_ICPolarity, uint8_t ICFilter)
功能描述	将 PWMB 触发器配置为外部时钟
参数 1	PWMB_TIxExternalCLKSource 指定触发器源。 *此参数可以是以下值之一: *-PWMB_TIXEXTERNALCLK1SOURCE_TI1:TI1 边缘检测器 *-PWMB_TIXEXTERNALCLK1SOURCE_TI2: 过滤的 PWMB 输入 1 *-PWMB_TIXEXTERNALCLK1SOURCE_TI1ED: 过滤的 PWMB 输入 2
参数 2	PWMB_ICP 极性指定 TIx 极性。 *此参数可以是: *-PWMB_ICPOLARITY_RISING *-PWMB_ICPOLARITY_FALLING
参数 3	ICFilter 指定滤波器值。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.97 配置 PWMB 输入触发源

函数名	void PWMB_SelectInputTrigger(PWMB_TS_TypeDef PWMB_InputTriggerSource)
功能描述	选择 PWMB 输入触发源
参数 1	PWMB_InputTriggerSource 指定输入触发源。 *此参数可以是以下值之一: *-PWMB_TS_TI1F_ED:TI1 边缘检测器 *-PWMB_TS_TI1FP1: 过滤定时器输入 1 *-PWMB_TS_TI2FP2: 过滤定时器输入 2 *PWMB_TS_ETRF: 外部触发输入
返回	无

4.12.98 启用或禁用 PWMB Update 事件

函数名	void PWMB_UpdateDisableConfig(FunctionalState NewState)
功能描述	启用或禁用 PWMB Update 事件
参数 1	NewState PWMB 外围预加载寄存器的新状态。此参数可以 *为 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.99 配置 PWMB 更新请求中断源

函数名	void PWMB_UpdateRequestConfig(PWMB_UpdateSource_TypeDef PWMB_UpdateSource)
功能描述	选择 PWMB 更新请求中断源
参数 1	PWMB_UpdateSource 指定更新源。 *此参数可以是以下值之一 - PWMB_UPDATESOURCE_REGULAR - PWMB_UPDATESOURCE_GLOBAL
参数 2	
参数 3	
返回	无

4.12.100 启用或禁用 PWMB 霍尔传感器

函数名	void PWMB_SelectHallSensor(FunctionalState NewState)
功能描述	启用或禁用 PWMB 霍尔传感器
参数 1	NewState PWMB 霍尔传感器接口的新状态。此参数可以 *为 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.101 选择 PWMB 单脉冲模式

函数名	void PWMB_SelectOnePulseMode(PWMB_OPMode_TypeDef PWMB_OPMode)
功能描述	选择 PWMB 单脉冲模式
参数 1	PWMB_OPMode 指定要使用的 OPM 模式。 *此参数可以是以下值之一 *-PWMB_OPMODE_SINGLE

	*-PWMB_OPMODE_repetive
返回	无

4.12.102 选择 PWMB 触发器输出模式

函数名	void PWMB_SelectOutputTrigger(PWMB_TRGOSource_TypeDef PWMB_TRGOSource)
功能描述	选择 PWMB 触发器输出模式
参数 1	PWMB_TRGOSource 指定触发输出源。 *此参数可以是以下值之一 * PWMB_TRGOSOURCE_RESET * - PWMB_TRGOSOURCE_ENABLE * - PWMB_TRGOSOURCE_UPDATE * - PWMB_TRGOSource_OC5 * - PWMB_TRGOSOURCE_OC5REF * - PWMB_TRGOSOURCE_OC6REF * - PWMB_TRGOSOURCE_OC7REF
返回	无

4.12.103 选择 PWMB 从模式

函数名	void PWMB_SelectSlaveMode(PWMB_SlaveMode_TypeDef PWMB_SlaveMode)
功能描述	选择 PWMB 从模式
参数 1	PWMB_SlaveMode 指定 PWMB 从模式。 *此参数可以是以下值之一 *-PWMB_SLAVEMODE_RESET *-PWMB_SLAVEMODE_GATED *- PWMB_SLAVEMODE_TRIGGER *-PWMB_SLAVEMODE_EXTERNAL1
返回	无

4.12.104 设置 PWMB 主/从模式

函数名	void PWMB_SelectMasterSlaveMode(FunctionalState NewState)
功能描述	设置或重置 PWMB 主/从模式
参数 1	NewState PWMB 及其从属设备之间同步的新状态

	*（通过 TRGO）。此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.105 配置 PWMB 编码器接口

函数名	void PWMB_EncoderInterfaceConfig(PWMB_EncoderMode_TypeDef PWMB_EncoderMode, PWMB_ICPolarity_TypeDef PWMB_IC1Polarity, PWMB_ICPolarity_TypeDef PWMB_IC2Polarity)
功能描述	配置 PWMB 编码器接口
参数 1	PWMB_EncoderMode 指定 PWMB 编码器模式。 *此参数可以是以下值之一 *-PWMB_ENCODERMODE_TI1：TI1FP1 边缘上的计数器计数 *取决于 TI2FP2 水平。 *-PWMB_ENCODERMODE_TI2：TI2FP2 边缘上的计数器计数 *取决于 TI1FP1 水平。 *-PWMB_ENCODERMODE_TI12：TI1FP1 和 *TI2FP2 边缘取决于其他输入的电平。
参数 2	PWMB_IC1 极性指定 IC1 极性。 *此参数可以是以下值之一 *-PWMB_ICPOLARITY_FALLING *-PWMB_ICPOLARITY_RISING
参数 3	PWMB_IC2 极性指定 IC2 极性。 *此参数可以是以下值之一 *-PWMB_ICPOLARITY_FALLING *-PWMB_ICPOLARITY_RISING
返回	无

4.12.106 配置 PWMB 预分频器

函数名	void PWMB_PrescalerConfig(uint16_t Prescaler, PWMB_PSCReloadMode_TypeDef PWMB_PSCReloadMode)
功能描述	配置 PWMB 预分频器
参数 1	Prescaler 指定 Prescaler 寄存器值 *此参数的值必须介于 0x0000 和 0xFFFF 之间
参数 2	PWMB_PSCReloadMode 指定 PWMB 预缩放器重新加载模式。 *此参数可以是以下值之一 *-PWMB_PSCRELOADMODE_IMMEDIATE：预缩放器立即加载。 *-PWMB_PSCREELOADMODE_UPDATE：在更新事件时加载预缩放器
参数 3	

返回	无

4.12.107 指定要使用的 PWMB 计数器模式

函数名	void PWMB_CounterModeConfig(PWMB_CounterMode_TypeDef PWMB_CounterMode)
功能描述	指定要使用的 PWMB 计数器模式
参数 1	PWMB_CounterMode 指定要使用的计数器模式 *此参数可以是以下值之一: *-PWMB_COUNTERMODE_UP:PWMB 递增计数模式 *-PWMB_COUNTERMODE_DOWN:PWMB 递减计数模式 *-PWMB_COUNTERMODE_CENTERALIGNED1:PWMB 中心对齐模式 1 *-PWMB_CounterMode_CenterAligned2:PWMB 中心对齐模式 2 *-PWMB_COUNTERMODE_CENTERALIGNED3:PWMB 中心对齐模式 3
返回	无

4.12.108 强制 PWMB 通道 5 输出高低电平

函数名	void PWMB_ForcedOC5Config(PWMB_ForcedAction_TypeDef PWMB_ForcedAction)
功能描述	强制 PWMB 通道 5 输出波形为高低电平
参数 1	PWMB_ForcedAction 指定要设置为输出波形的强制动作。 *此参数可以是以下值之一: *-PWMB_FORCEDACTION_ACTIVE：强制 OC5REF 上的高电平 *-PWMB_FORCEDACTION_INACTIVE：强制 OC5REF 处于低电平。
返回	无

4.12.109 强制 PWMB 通道 6 输出高低电平

函数名	void PWMB_ForcedOC6Config(PWMB_ForcedAction_TypeDef PWMB_ForcedAction)
功能描述	强制 PWMB 通道 6 输出波形为高低电平
参数 1	PWMB_ForcedAction 指定要设置为输出波形的强制动作。 *此参数可以是以下值之一: *-PWMB_FORCEDACTION_ACTIVE：强制 OC6REF 上的高电平 *-PWMB_FORCEDACTION_INACTIVE：强制 OC6REF 处于低电平。
返回	无

4.12.110 强制 PWMB 通道 7 输出高低电平

函数名	void PWMB_ForcedOC7Config(PWMB_ForcedAction_TypeDef PWMB_ForcedAction)
功能描述	强制 PWMB 通道 7 输出波形为高低电平
参数 1	PWMB_ForcedAction 指定要设置为输出波形的强制动作。 *此参数可以是以下值之一: *-PWMB_FORCEDACTION_ACTIVE：强制 OC7REF 上的高电平 *-PWMB_FORCEDACTION_INACTIVE：强制 OC7REF 处于低电平。
返回	无

4.12.111 强制 PWMB 通道 8 输出高低电平

函数名	void PWMB_ForcedOC8Config(PWMB_ForcedAction_TypeDef PWMB_ForcedAction)
功能描述	强制 PWMB 通道 8 输出波形为高低电平
参数 1	PWMB_ForcedAction 指定要设置为输出波形的强制动作。 *此参数可以是以下值之一: *-PWMB_FORCEDACTION_ACTIVE：强制 OC8REF 上的高电平 *-PWMB_FORCEDACTION_INACTIVE：强制 OC8REF 处于低电平。
返回	无

4.12.112 启用或禁用 ARR 上的 PWMB 预加载寄存器

函数名	void PWMB_ARRPreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 ARR 上的 PWMB 外围预加载寄存器
参数 1	NewState PWMB 外围预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.113 选择 PWMB com 事件

函数名	void PWMB_SelectCOM(FunctionalState NewState)
功能描述	选择 PWMB 外围 事件
参数 1	NewState 通勤事件的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.114 设置 PWMB 外围设备捕获比较预加载控制位

函数名	void PWMB_CCPreloadControl(FunctionalState NewState)
功能描述	设置或重置 PWMB 外围设备捕获比较预加载控制位
参数 1	NewState 捕获比较预加载控制位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.115 设置 CCR5 上的 PWMB 预加载寄存器

函数名	void PWMB_OC5PreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 CCR5 上的 PWMB 预加载寄存器
参数 1	NewState 捕获比较预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.116 设置 CCR6 上的 PWMB 预加载寄存器

函数名	void PWMB_OC6PreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 CCR6 上的 PWMB 预加载寄存器
参数 1	NewState 捕获比较预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.117 设置 CCR7 上的 PWMB 预加载寄存器

函数名	void PWMB_OC7PreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 CCR7 上的 PWMB 预加载寄存器
参数 1	NewState 捕获比较预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.118 设置 CCR8 上的 PWMB 预加载寄存器

函数名	void PWMB_OC8PreloadConfig(FunctionalState NewState)
功能描述	启用或禁用 CCR8 上的 PWMB 预加载寄存器
参数 1	NewState 捕获比较预加载寄存器的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.119 配置 PWMB 捕获比较 1 快速使能

函数名	void PWMB_OC5FastConfig(FunctionalState NewState)
功能描述	配置 PWMB 捕获比较 1 快速功能
参数 1	NewState 输出比较快速启用位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.120 配置 PWMB 捕获比较 2 快速使能

函数名	void PWMB_OC6FastConfig(FunctionalState NewState)
功能描述	配置 PWMB 捕获比较 2 快速功能
参数 1	NewState 输出比较快速启用位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.121 配置 PWMB 捕获比较 3 快速使能

函数名	void PWMB_OC7FastConfig(FunctionalState NewState)
功能描述	配置 PWMB 捕获比较 3 快速功能
参数 1	NewState 输出比较快速启用位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.122 配置 PWMB 捕获比较 4 快速使能

函数名	void PWMB_OC8FastConfig(FunctionalState NewState)
功能描述	配置 PWMB 捕获比较 4 快速功能
参数 1	NewState 输出比较快速启用位的新状态。 *此参数可以是 ENABLE（启用）或 DISABLE（禁用）
返回	无

4.12.123 配置要由软件生成的 PWMB 事件

函数名	void PWMB_GenerateEvent(PWMB_EventSource_TypeDef PWMB_EventSource)
功能描述	配置要由软件生成的 PWMB 事件
参数 1	PWMB_EventSource 指定事件源。 *此参数可以是以下值之一:

	*-PWMB_EVENTSOURCE_UPDATE:PWMB 更新事件源 *-PWMB_EVENTSOURCE_CC1:PWMB 捕获比较 1 事件源 *-PWMB_EVENTSOURCE_CC2:PWMB 捕获比较 2 事件源 *-PWMB_EVENTSOURCE_CC3:PWMB 捕获比较 3 事件源 *-PWMB_EVENTSOURCE_CC4:PWMB 捕获比较 4 事件源 *-PWMB_EVENTSOURCE_COM:PWMB COM 事件源 *-PWMB_EVENTSOURCE_TRIGGER:PWMB 触发事件源 *-PWMB_EventSourceBreak:PWMB Break 事件源
返回	无

4.12.124 配置 PWMB 通道 5 极性

函数名	void PWMB_OC5PolarityConfig(PWMB_OCPolarity_TypeDef PWMB_OCPolarity)
功能描述	配置 PWMB 通道 5 极性
参数 1	PWMB_OCP 极性指定 OC5 极性。 *此参数可以是以下值之一: *-PWMB_OCPOLARITY_LOW：输出比较激活低 *-PWMB_OPOLARITY_HIGH：输出比较激活高
返回	无

4.12.125 配置 PWMB 通道 6 极性

函数名	void PWMB_OC6PolarityConfig(PWMB_OCPolarity_TypeDef PWMB_OCPolarity)
功能描述	配置 PWMB 通道 6 极性
参数 1	PWMB_OCP 极性指定 OC6 极性。 *此参数可以是以下值之一: *-PWMB_OCPOLARITY_LOW：输出比较激活低 *-PWMB_OPOLARITY_HIGH：输出比较激活高
返回	无

4.12.126 配置 PWMB 通道 7 极性

函数名	void PWMB_OC7PolarityConfig(PWMB_OCPolarity_TypeDef PWMB_OCPolarity)
功能描述	配置 PWMB 通道 7 极性
参数 1	PWMB_OCP 极性指定 OC7 极性。 *此参数可以是以下值之一: *-PWMB_OCPOLARITY_LOW：输出比较激活低 *-PWMB_OPOLARITY_HIGH：输出比较激活高

返回	无

4.12.127 配置 PWMB 通道 8 极性

函数名	void PWMB_OC8PolarityConfig(PWMB_OCPolarity_TypeDef PWMB_OCPolarity)
功能描述	配置 PWMB 通道 8 极性
参数 1	PWMB_OCP 极性指定 OC8 极性。 *此参数可以是以下值之一: *-PWMB_OCPOLARITY_LOW: 输出比较激活低 *-PWMB_OPOLARITY_HIGH: 输出比较激活高
返回	无

4.12.128 配置 PWMB 通道 5N 极性

函数名	void PWMB_OC5NPolarityConfig(PWMB_OCNPolarity_TypeDef PWMB_OCNPolarity)
功能描述	配置 PWMB 通道 5N 极性
参数 1	PWMB_OCNPolarity 指定 OC5N 极性。 *此参数可以是以下值之一: *-PWMB_OCNPOLARITY_LOW: 输出比较激活低 *-PWMB_OCNPOLARITY_HHIGH: 输出比较激活高
返回	无

4.12.129 配置 PWMB 通道 6N 极性

函数名	void PWMB_OC6NPolarityConfig(PWMB_OCNPolarity_TypeDef PWMB_OCNPolarity)
功能描述	配置 PWMB 通道 6N 极性
参数 1	PWMB_OCNPolarity 指定 OC6N 极性。 *此参数可以是以下值之一: *-PWMB_OCNPOLARITY_LOW: 输出比较激活低 *-PWMB_OCNPOLARITY_HHIGH: 输出比较激活高
返回	无

4.12.130 配置 PWMB 通道 7N 极性

函数名	void PWMB_OC7NPolarityConfig(PWMB_OCNPolarity_TypeDef PWMB_OCNPolarity)
功能描述	配置 PWMB 通道 7N 极性

参数 1	PWMB_OCNPolarity 指定 OC7N 极性。 *此参数可以是以下值之一: *-PWMB_OCNPOLARITY_LOW: 输出比较激活低 *-PWMB_OCNPOLARITY_HHIGH: 输出比较激活高
返回	无

4.12.131 配置 PWMB 通道 8N 极性

函数名	void PWMB_OC8NPolarityConfig(PWMB_OCNPolarity_TypeDef PWMB_OCNPolarity)
功能描述	配置 PWMB 通道 8N 极性
参数 1	PWMB_OCNPolarity 指定 OC8N 极性。 *此参数可以是以下值之一: *-PWMB_OCNPOLARITY_LOW: 输出比较激活低 *-PWMB_OCNPOLARITY_HHIGH: 输出比较激活高
返回	无

4.12.132 启用或禁用 PWMB 捕获比较通道 P

函数名	void PWMB_CCxCmd(PWMB_Channel_TypeDef PWMB_Channel, FunctionalState NewState)
功能描述	启用或禁用 PWMB 捕获比较通道 x（x=1，..，4）
参数 1	PWMB_Channel 指定 PWMB 通道。 *此参数可以是以下值之一: *-PWMB_CHANNEL_1:PWMB 通道 5 *-PWMB_CHANNEL_2:PWMB 通道 6 *-PWMB_CHANNEL_3:PWMB 通道 7 *-PWMB_CHANNEL_4:PWMB 通道 8
参数 2	NewState 指定 PWMB 信道 CCxE 位的新状态。 *此参数可以是: ENABLE 或 DISABLE
返回	无

4.12.133 启用或禁用 PWMB 捕获比较通道 N

函数名	void PWMB_CCxNCmd(PWMB_Channel_TypeDef PWMB_Channel, FunctionalState NewState)
功能描述	启用或禁用 PWMB 捕获比较通道 xN（x=1，..，4）
参数 1	PWMB_Channel 指定 PWMB 通道。 *此参数可以是以下值之一: *-PWMB_CHANNEL_1:PWMB 通道 5 *-PWMB_CHANNEL_2:PWMB 通道 6

	*-PWMB_CHANNEL_3:PWMB 通道 7 *-PWMB_CHANNEL_4:PWMB 通道 8
参数 2	NewState 指定 PWMB 信道 CCxNE 位的新状态。 *此参数可以是：ENABLE 或 DISABLE
返回	无

4.12.134 配置 PWMB 输出比较模式

函数名	void PWMB_SelectOCxM(PWMB_Channel_TypeDef PWMB_Channel, PWMB_OCMode_TypeDef PWMB_OCMode)
功能描述	选择 PWMB 输出比较模式。此功能禁用在更改“输出比较模式”之前选择的频道。用户必须*使用 PWMB_CCxCmd 和 PWMB_CCxNCmd 函数启用此通道
参数 1	PWMB_Channel 指定 PWMB 通道。 *此参数可以是以下值之一: *-PWMB_CHANNEL_1:PWMB 通道 5 *-PWMB_CHANNEL_2:PWMB 通道 6 *-PWMB_CHANNEL_3:PWMB 通道 7 *-PWMB_CHANNEL_4:PWMB 通道 8
参数 2	PWMB_OCMode 指定 PWMB 输出比较模式。 *此参数可以是以下值之一: *-PWMB_OCMODE_TIMING *-PWMB_OCMODE_ACTIVE *-PWMB_OCMODE_TOGGLE *-PWMB_OCMODE_PWM1 *-PWMB_OCMODE_PWM2 *-PWMB_frcedaction_ACTIVE *-PWMB_FORCEDACTION_INACTIVE
返回	无

4.12.135 设置 PWMB 计数器寄存器值

函数名	void PWMB_SetCounter(uint16_t Counter)
功能描述	设置 PWMB 计数器寄存器值
参数 1	Counter 指定 Counter 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.136 设置 PWMB 自动重新加载寄存器值

函数名	void PWMB_SetAutoreload(uint16_t Autoreload)

功能描述	设置 PWMB 自动重新加载寄存器值
参数 1	Autoreload 指定自动重新加载寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.137 设置 PWMB 捕获比较器 5 寄存器值

函数名	void PWMB_SetCompare5(uint16_t Compare5)
功能描述	设置 PWMB 捕获比较 5 寄存器值
参数 1	Compare5 指定 Capture Compare5 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.138 设置 PWMB 捕获比较器 6 寄存器值

函数名	void PWMB_SetCompare6(uint16_t Compare6)
功能描述	设置 PWMB 捕获比较 6 寄存器值
参数 1	Compare6 指定 Capture Compare6 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.139 设置 PWMB 捕获比较器 7 寄存器值

函数名	void PWMB_SetCompare7(uint16_t Compare7)
功能描述	设置 PWMB 捕获比较 7 寄存器值
参数 1	Compare7 指定 Capture Compare7 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.140 设置 PWMB 捕获比较器 8 寄存器值

函数名	void PWMB_SetCompare8(uint16_t Compare8)
功能描述	设置 PWMB 捕获比较 8 寄存器值
参数 1	Compare8 指定 Capture Compare8 寄存器的新值。 *此参数介于 0x0000 和 0xFFFF 之间
返回	无

4.12.141 设置 PWMB 输入捕获 5 预分频器

函数名	void PWMB_SetIC5Prescaler(PWMB_ICPSC_TypeDef PWMB_IC5Prescaler)
功能描述	设置 PWMB 输入捕获 5 预分频器
参数 1	PWMB_IC5 预分频器指定输入捕获预分频器的新值 *此参数可以是以下值之一: *-PWMB_ICPSC_DIV1: 无预分频器 *-PWMB_ICPSC_DIV2: 每 2 个事件捕获一次 *-PWMB_ICPSC_DIV4: 每 4 个事件捕获一次 *-PWMB_ICPSC_DIV8: 每 8 个事件捕获一次
返回	无

4.12.142 设置 PWMB 输入捕获 6 预分频器

函数名	void PWMB_SetIC6Prescaler(PWMB_ICPSC_TypeDef PWMB_IC6Prescaler)
功能描述	设置 PWMB 输入捕获 6 预分频器
参数 1	PWMB_IC6 预分频器指定输入捕获预分频器的新值 *此参数可以是以下值之一: *-PWMB_ICPSC_DIV1: 无预分频器 *-PWMB_ICPSC_DIV2: 每 2 个事件捕获一次 *-PWMB_ICPSC_DIV4: 每 4 个事件捕获一次 *-PWMB_ICPSC_DIV8: 每 8 个事件捕获一次
返回	无

4.12.143 设置 PWMB 输入捕获 7 预分频器

函数名	void PWMB_SetIC7Prescaler(PWMB_ICPSC_TypeDef PWMB_IC7Prescaler)
功能描述	设置 PWMB 输入捕获 7 预分频器
参数 1	PWMB_IC7 预分频器指定输入捕获预分频器的新值 *此参数可以是以下值之一: *-PWMB_ICPSC_DIV1: 无预分频器 *-PWMB_ICPSC_DIV2: 每 2 个事件捕获一次 *-PWMB_ICPSC_DIV4: 每 4 个事件捕获一次 *-PWMB_ICPSC_DIV8: 每 8 个事件捕获一次
返回	无

4.12.144 设置 PWMB 输入捕获 8 预分频器

函数名	void PWMB_SetIC8Prescaler(PWMB_ICPSC_TypeDef PWMB_IC8Prescaler)
功能描述	设置 PWMB 输入捕获 8 预分频器
参数 1	PWMB_IC8 预分频器指定输入捕获预分频器的新值

	*此参数可以是以下值之一: *-PWMB_ICPSC_DIV1: 无预分频器 *-PWMB_ICPSC_DIV2: 每 2 个事件捕获一次 *-PWMB_ICPSC_DIV4: 每 4 个事件捕获一次 *-PWMB_ICPSC_DIV8: 每 8 个事件捕获一次
返回	无

4.12.145 获取 PWMB 输入捕获 5 的值

函数名	uint16_t PWMB_GetCapture5(void)
功能描述	获取 PWMB 输入捕获 5 的值
参数	无
返回	Capture 比较 5 的寄存器值

4.12.146 获取 PWMB 输入捕获 6 的值

函数名	uint16_t PWMB_GetCapture6(void)
功能描述	获取 PWMB 输入捕获 6 的值
参数	无
返回	Capture 比较 6 的寄存器值

4.12.147 获取 PWMB 输入捕获 7 的值

函数名	uint16_t PWMB_GetCapture7(void)
功能描述	获取 PWMB 输入捕获 7 的值
参数	无
返回	Capture 比较 7 的寄存器值

4.12.148 获取 PWMB 输入捕获 8 的值

函数名	uint16_t PWMB_GetCapture8(void)
功能描述	获取 PWMB 输入捕获 8 的值
参数	无
返回	Capture 比较 8 的寄存器值

4.12.149 获取 PWMB 计数器值

函数名	uint16_t PWMB_GetCounter(void)
功能描述	获取 PWMB 计数器值

参数	无
返回	计数器寄存器值

4.12.150 获取 PWMB 预分频器值

函数名	uint16_t PWMB_GetPrescaler(void)
功能描述	获取 PWMB 预分频器值
参数	无
返回	预分频器寄存器值

4.12.151 获取指定的 PWMB 标志

函数名	FlagStatus PWMB_GetFlagStatus(PWMB_FLAG_TypeDef PWMB_FLAG)
功能描述	检查是否设置了指定的 PWMB 标志
参数 1	PWMB_FLAG 指定要检查的标志。 *此参数可以是以下值之一: *-PWMB_FLAG_UPDATE:PWMB 更新标志 *-PWMB_FLAG_CC1:PWMB 捕获比较 5 标志 *-PWMB_FLAG_CC2:PWMB 捕获比较 6 标志 *-PWMB_FLAG_CC3:PWMB 捕获比较 7 标志 *-PWMB_FLAG_CC4:PWMB 捕获比较 8 标志 *-PWMB_FLAG_COMM:PWMB 交换标志 *-PWMB_FLAG_TRIGGER:PWMB 触发标志 *-PWMB_FLAG_BREAK:PWMB 中断标志 *-PWMB_FLAG_CC1F:PWMB 捕获比较 5 过捕获标志 *-PWMB_FLAG_CC2OF:PWMB 捕获比较 6 过捕获标志 *-PWMB_FLAG_CC3OF: PWMB 捕获比较 7 过捕获标志 *-PWMB_FLAG_CC4OF:PWMB 捕获比较 8 过捕获标志
返回	FlagStatus PWMB_FLAG 的新状态（SET 或 RESET）

4.12.152 清除 PWMB 挂起标志

函数名	void PWMB_ClearFlag(PWMB_FLAG_TypeDef PWMB_FLAG)
功能描述	清除 PWMB 挂起标志
参数 1	PWMB_FLAG 指定要清除的标志。 *此参数可以是以下值之一: *-PWMB_FLAG_UPDATE:PWMB 更新标志 *-PWMB_FLAG_CC1:PWMB 捕获比较 5 标志 *-PWMB_FLAG_CC2:PWMB 捕获比较 6 标志 *-PWMB_FLAG_CC3:PWMB 捕获比较 7 标志 *-PWMB_FLAG_CC4:PWMB 捕获比较 8 标志

	*-PWMB_FLAG_COMM:PWMB 交换标志 *-PWMB_FLAG_TRIGGER:PWMB 触发标志 *-PWMB_FLAG_BREAK:PWMB 中断标志 *-PWMB_FLAG_CC1F:PWMB 捕获比较 5 过捕获标志 *-PWMB_FLAG_CC2OF:PWMB 捕获比较 6 过捕获标志 *-PWMB_FLAG_CC3OF：PWMB 捕获比较 7 过捕获标志 *-PWMB_FLAG_CC4OF:PWMB 捕获比较 8 过捕获标志
返回	无

4.12.153 检查 PWMB 中断是否发生

函数名	ITStatus PWMB_GetITStatus(PWMB_IT_TypeDef PWMB_IT)
功能描述	检查 PWMB 中断是否发生
参数 1	PWMB_IT 指定要检查的 PWMB 中断源。 *此参数可以是以下值之一: *-PWMB_IT_UPDATE：PWMB 更新中断源 *-PWMB_IT_CC1:PWMB 捕获比较 5 中断源 *-PWMB_IT_CC2:PWMB 捕获比较 6 中断源 *-PWMB_IT_CC3:PWMB 捕获比较 7 中断源 *-PWMB_IT_CC4:PWMB 捕获比较 8 中断源 *-PWMB_IT_COM:PWMB 换向中断源 *-PWMB_IT_TRIGGER:PWMB 触发中断源 *-PWMB_IT_BEAK:PWMB 中断源
返回	ITStatus PWMB_IT 的新状态（SET 或 RESET）

4.12.154 清除 PWMB 的中断挂起位

函数名	void PWMB_ClearITPendingBit(PWMB_IT_TypeDef PWMB_IT)
功能描述	清除 PWMB 的中断挂起位
参数 1	PWMB_IT 指定要清除的挂起位。 *此参数可以是以下值之一: *-PWMB_IT_UPDATE：PWMB 更新中断源 *-PWMB_IT_CC1:PWMB 捕获比较 5 中断源 *-PWMB_IT_CC2:PWMB 捕获比较 6 中断源 *-PWMB_IT_CC3:PWMB 捕获比较 7 中断源 *-PWMB_IT_CC4:PWMB 捕获比较 8 中断源 *-PWMB_IT_COM:PWMB 换向中断源 *-PWMB_IT_TRIGGER:PWMB 触发中断源 *-PWMB_IT_BEAK:PWMB 中断源
返回	无

4.12.155 将 TI5 配置为输入

函数名	static void TI5_Config(uint8_t PWMB_ICPolarity, uint8_t PWMB_ICSelection, uint8_t PWMB_ICFilter)
功能描述	将 TI5 配置为输入
参数 1	PWMB_ICP 极性输入极性。 *此参数可以是以下值之一: *-PWMB_ICPOLARITY_FALLING *-PWMB_ICPOLARITY_RISING
参数 2	PWMB_ICSelection 指定要使用的输入。 *此参数可以是以下值之一: *-PWMB_ICSELECTION_DIRECTTI:PWMB 输入 5 选择为 *连接到 IC1。 *-PWMB_ICSELECTION_INDIRECTTI:PWMB 输入 5 选择为 *连接到 IC2。
参数 3	PWMB_ICFilter 指定输入捕获滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.156 将 TI6 配置为输入

函数名	static void TI6_Config(uint8_t PWMB_ICPolarity, uint8_t PWMB_ICSelection, uint8_t PWMB_ICFilter)
功能描述	将 TI6 配置为输入
参数 1	PWMB_ICP 极性输入极性。 *此参数可以是以下值之一: *-PWMB_ICPOLARITY_FALLING *-PWMB_ICPOLARITY_RISING
参数 2	PWMB_ICSelection 指定要使用的输入。 *此参数可以是以下值之一: *-PWMB_ICSELECTION_DIRECTTI:PWMB 输入 6 选择为 *连接到 IC2。 *-PWMB_ICSELECTION_INDIRECTTI:PWMB 输入 6 选择为 *连接到 IC1。
参数 3	PWMB_ICFilter 指定输入捕获滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.157 将 TI7 配置为输入

函数名	static void TI7_Config(uint8_t PWMB_ICPolarity, uint8_t PWMB_ICSelection, uint8_t PWMB_ICFilter)
功能描述	将 TI7 配置为输入
参数 1	PWMB_ICP 极性输入极性。 *此参数可以是以下值之一: *-PWMB_ICPOLARITY_FALLING *-PWMB_ICPOLARITY_RISING
参数 2	PWMB_ICSelection 指定要使用的输入。 *此参数可以是以下值之一: *-PWMB_ICSELECTION_DIRECTTI:PWMB 输入 7 选择为 *连接到 IC3。 *-PWMB_ICSELECTION_INDIRECTTI:PWMB 输入 7 选择为 *连接到 IC4。
参数 3	PWMB_ICFilter 指定输入捕获滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.12.158 将 TI8 配置为输入

函数名	static void TI8_Config(uint8_t PWMB_ICPolarity, uint8_t PWMB_ICSelection, uint8_t PWMB_ICFilter)
功能描述	将 TI8 配置为输入
参数 1	PWMB_ICP 极性输入极性。 *此参数可以是以下值之一: *-PWMB_ICPOLARITY_FALLING *-PWMB_ICPOLARITY_RISING
参数 2	PWMB_ICSelection 指定要使用的输入。 *此参数可以是以下值之一: *-PWMB_ICSELECTION_DIRECTTI:PWMB 输入 8 选择为 *连接到 IC4。 *-PWMB_ICSELECTION_INDIRECTTI:PWMB 输入 8 选择为 *连接到 IC3。
参数 3	PWMB_ICFilter 指定输入捕获滤波器。 *此参数的值必须介于 0x00 和 0x0F 之间
返回	无

4.13 STC32G_EEPROM

4.13.1 相关文件

“STC32G_EEPROM.c”， 主要包含 EEPROM 操作函数。
“STC32G_EEPROM.h”， 包含 EEPROM 函数所需的头文件、宏定义以及函数申明。

4.13.2 EEPROM 读取函数

函数名	void EEPROM_read_n(u32 EE_address,u8 *DataAddress,u16 number)
功能描述	从指定 EEPROM 首地址读取若干个字节放指定的缓冲
参数 1	EE_address: 读取 EEPROM 的首地址
参数 2	DataAddress: 读取数据存放缓冲区的首地址
参数 3	number: 读取的字节长度
返回	无

4.13.3 EEPROM 写入函数

函数名	Void EEPROM_write_n(u32 EE_address,u8 *DataAddress,u16 number)
功能描述	把缓冲区的若干个字节写入指定首地址的 EEPROM
参数 1	EE_address: 写入 EEPROM 的首地址
参数 2	DataAddress: 写入数据源缓冲区的首地址
参数 3	number: 写入的字节长度
返回	无

4.13.4 EEPROM 擦除函数

函数名	void EEPROM_SectorErase(u32 EE_address)
功能描述	把指定地址的 EEPROM 扇区擦除
参数	EE_address: 要擦除的扇区 EEPROM 的地址
返回	无

4.14 STC32G_LCM TFT-I8080/M6800

4.14.1 相关文件

“STC32G_LCM.c”， 主要用于 LCM 功能的配置。
“STC32G_LCM_Isr.c”，主要包含 LCM 的中断函数。
“STC32G_LCM.h”， LCM 所需的变量申明与宏定义。

4.14.2 LCM 初始化

函数名	void LCM_Inilize(LCM_InitTypeDef *LCM)
功能描述	DMA LCM 初始化程序

参数	DMA: 结构参数
返回	无

DMAx: 结构参数定义：

```
typedef struct
{
    u8   LCM_Enable;
    u8   LCM_Mode;
    u8   LCM_Bit_Wide;
    u8   LCM_Setup_Time;
    u8   LCM_Hold_Time;
} LCM_InitTypeDef;
```

LCM_Enable: LCM 接口使能设置

参数	功能描述
ENABLE	使能 LCM 接口
DISABLE	禁止 LCM 接口

LCM_Mode: LCM 接口模式设置

参数	功能描述
MODE_I8080	LCM 接口设置为 I8080 模式
MODE_M6800	LCM 接口设置为 M6800 模式

LCM_Bit_Wide: LCM 数据宽度设置

参数	功能描述
BIT_WIDE_8	LCM 接口设置为 8 位数据宽度
BIT_WIDE_16	LCM 接口设置为 16 位数据宽度

LCM_Setup_Time: LCM 通信数据建立时间，设置范围 0~7。

LCM_Hold_Time: LCM 通信数据保持时间，设置范围 0~3。

4.14.3 LCM 中断函数

函数名	void LCM_ISR_Handler (void) interrupt LCM_VECTOR
功能描述	LCM 中断程序
参数	无
返回	无

程序框架:

```
void LCM_ISR_Handler (void) interrupt LCM_VECTOR
{
    if(LCMIFSTA & 0x01)
    {
        LCMIFSTA = 0x00;       //清除中断标志
```

```
        // TODO: 在此处添加用户代码
    }
}
```

4.15 STC32G_DMA

4.15.1 相关文件

“STC32G_DMA.c”，主要用于 DMA 功能的配置。
“STC32G_DMA_Isr.c”，主要包含 DMA 中断函数。
“STC32G_DMA.h”，DMA 所需的变量申明与宏定义。

4.15.2 ADC DMA 初始化

函数名	void DMA_ADC_Inilize(DMA_ADC_InitTypeDef *DMA)
功能描述	DMA ADC 初始化程序
参数	DMA: 结构参数
返回	无

DMAx: 结构参数定义：

```
typedef struct
{
    u8   DMA_Enable;
    u16  DMA_Channel;
    u16  DMA_Buffer;
    u8   DMA_Times;
} DMA_ADC_InitTypeDef;
```

DMA_Enable: DMA 使能设置

参数	功能描述
ENABLE	使能 ADC DMA
DISABLE	禁止 ADC DMA

DMA_Channel: ADC 通道使能寄存器, bit15~bit0 对应 ADC15~ADC0，置 1 使能对应通道。

DMA_Buffer: ADC 转换数据存储地址。

DMA_Times: 每个通道转换次数

参数	数值	功能描述
ADC_1_Times	0xxx	转换 1 次
ADC_2_Times	1000	转换 2 次
ADC_4_Times	1001	转换 4 次
ADC_8_Times	1010	转换 8 次
ADC_16_Times	1011	转换 16 次

ADC_32_Times	1100	转换 32 次
ADC_64_Times	1101	转换 64 次
ADC_128_Times	1110	转换 128 次
ADC_256_Times	1111	转换 256 次

4.15.3 M2M DMA 初始化

函数名	void DMA_M2M_Inilize(DMA_M2M_InitTypeDef *DMA)
功能描述	DMA M2M 初始化程序
参数	DMA: 结构参数
返回	无

DMAx: 结构参数定义：

```
typedef struct
{
    u8   DMA_Enable;
    u16  DMA_Rx_Buffer;
    u16  DMA_Tx_Buffer;
    u16  DMA_Length;
    u8   DMA_SRC_Dir;
    u8   DMA_DEST_Dir;
} DMA_M2M_InitTypeDef;
```

DMA_Enable: DMA 使能设置

参数	功能描述
ENABLE	使能 M2M DMA
DISABLE	禁止 M2M DMA

DMA_Rx_Buffer: 接收数据存储地址。

DMA_Tx_Buffer: 发送数据存储地址。

DMA_Length: DMA 传输总字节数，设置范围(0~65535)，实际传输字节数为设置值 + 1，不要超过芯片 xdata 空间上限。

DMA_SRC_Dir: 数据源地址改变方向

参数	功能描述
M2M_ADDR_INC	数据读取完成后源地址自动递增
M2M_ADDR_DEC	数据读取完成后源地址自动递减

DMA_DEST_Dir: 数据目标地址改变方向

参数	功能描述
M2M_ADDR_INC	数据写入完成后目标地址自动递增
M2M_ADDR_DEC	数据写入完成后目标地址自动递减

4.15.4 UART DMA 初始化

函数名	void DMA_UART_Inilize(u8 UARTx, DMA_UART_InitTypeDef *DMA)
功能描述	DMA UART 初始化程序
参数 1	UARTx: UART 设置通道，取值 UART1, UART2, UART3, UART4
参数 2	DMA: 结构参数
返回	无

DMAx: 结构参数定义：

```
typedef struct
{
    u8   DMA_TX_Enable;
    u16  DMA_TX_Length;
    u16  DMA_TX_Buffer;

    u8   DMA_RX_Enable;
    u16  DMA_RX_Length;
    u16  DMA_RX_Buffer;
} DMA_UART_InitTypeDef;
```

DMA_TX_Enable: DMA 使能设置

参数	功能描述
ENABLE	使能串口发送 DMA
DISABLE	禁止串口发送 DMA

DMA_TX_Length: DMA 发送总字节数，设置范围(0~65535)，实际传输字节数为设置值 + 1，不要超过芯片 xdata 空间上限。

DMA_Tx_Buffer: 发送数据存储地址。

DMA_RX_Enable: DMA 使能设置

参数	功能描述
ENABLE	使能串口接收 DMA
DISABLE	禁止串口接收 DMA

DMA_RX_Length: DMA 接收总字节数，设置范围(0~65535)，实际传输字节数为设置值 + 1，不要超过芯片 xdata 空间上限。

DMA_RX_Buffer: 接收数据存储地址。

4.15.5 SPI DMA 初始化

函数名	void DMA_SPI_Inilize(DMA_SPI_InitTypeDef *DMA)

功能描述	DMA SPI 初始化程序
参数	DMA: 结构参数
返回	无

DMAx: 结构参数定义：

```
typedef struct
{
    u8   DMA_Enable;
    u8   DMA_Tx_Enable;
    u8   DMA_Rx_Enable;
    u16  DMA_Rx_Buffer;
    u16  DMA_Tx_Buffer;
    u16  DMA_Length;
    u8   DMA_AUTO_SS;
    u8   DMA_SS_Sel;
} DMA_SPI_InitTypeDef;
```

DMA_Enable: DMA 使能设置

参数	功能描述
ENABLE	使能 SPI DMA
DISABLE	禁止 SPI DMA

DMA_Tx_Enable: DMA 发送数据使能设置

参数	功能描述
ENABLE	使能 SPI DMA 发送数据
DISABLE	禁止 SPI DMA 发送数据

DMA_Rx_Enable: DMA 接收数据使能设置

参数	功能描述
ENABLE	使能 SPI DMA 接收数据
DISABLE	禁止 SPI DMA 接收数据

DMA_Rx_Buffer: 接收数据存储地址。

DMA_Tx_Buffer: 发送数据存储地址。

DMA_Length: DMA 传输总字节数，设置范围(0~65535)，实际传输字节数为设置值 + 1，不要超过芯片 xdata 空间上限。

DMA_AUTO_SS: 自动控制 SS 脚使能设置

参数	功能描述
ENABLE	SPI DMA 传输过程中，自动拉低 SS 脚，传输完成后恢复原始状态
DISABLE	SPI DMA 传输过程中，不自动控制 SS 脚

DMA_SS_Sel: 自动控制 SS 脚选择

参数	功能描述
SPI_SS_P12	选择 P1.2 作为自动控制 SS 脚
SPI_SS_P22	选择 P2.2 作为自动控制 SS 脚
SPI_SS_P74	选择 P7.4 作为自动控制 SS 脚
SPI_SS_P35	选择 P3.5 作为自动控制 SS 脚

4.15.6 LCM DMA 初始化

函数名	void DMA_LCM_Inilize(DMA_LCM_InitTypeDef *DMA)
功能描述	DMA LCM 初始化程序
参数	DMA: 结构参数
返回	无

DMAx: 结构参数定义：

```
typedef struct
{
    u8   DMA_Enable;
    u16  DMA_Rx_Buffer;
    u16  DMA_Tx_Buffer;
    u16  DMA_Length;
} DMA_LCM_InitTypeDef;
```

DMA_Enable: DMA 使能设置

参数	功能描述
ENABLE	使能 LCM DMA
DISABLE	禁止 LCM DMA

DMA_Rx_Buffer: 接收数据存储地址。

DMA_Tx_Buffer: 发送数据存储地址。

DMA_Length: DMA 传输总字节数，设置范围(0~65535)，实际传输字节数为设置值 + 1，不要超过芯片 xdata 空间上限。

4.15.7 I2C DMA 初始化

函数名	void DMA_I2C_Inilize(DMA_I2C_InitTypeDef *DMA)
功能描述	DMA I2C 初始化程序
参数	DMA: 结构参数
返回	无

DMAx: 结构参数定义：

```
typedef struct
```

```
{
    u8   DMA_TX_Enable;
    u16  DMA_TX_Length;
    u16  DMA_TX_Buffer;

    u8   DMA_RX_Enable;
    u16  DMA_RX_Length;
    u16  DMA_RX_Buffer;
} DMA_I2C_InitTypeDef;
```

DMA_TX_Enable: DMA 使能设置

参数	功能描述
ENABLE	使能 I2C 发送 DMA
DISABLE	禁止 I2C 发送 DMA

DMA_TX_Length: DMA 发送总字节数，设置范围(0~65535)，实际传输字节数为设置值 + 1，不要超过芯片 xdata 空间上限。

DMA_Tx_Buffer: 发送数据存储地址。

DMA_RX_Enable: DMA 使能设置

参数	功能描述
ENABLE	使能 I2C 接收 DMA
DISABLE	禁止 I2C 接收 DMA

DMA_RX_Length: DMA 接收总字节数，设置范围(0~65535)，实际传输字节数为设置值 + 1，不要超过芯片 xdata 空间上限。

DMA_RX_Buffer: 接收数据存储地址。

4.15.8 ADC DMA 中断函数

函数名	void DMA_ADC_ISR_Handler (void) interrupt DMA_ADC_VECTOR
功能描述	ADC DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_ADC_ISR_Handler (void) interrupt DMA_ADC_VECTOR
{
    if(DMA_ADC_STA & 0x01)   //ADC DMA 转换完成
    {
        DMA_ADC_STA &= ~0x01; //清标志位
        // TODO: 在此处添加用户代码
    }
```

```
}
```

4.15.9 M2M DMA 中断函数

函数名	void DMA_M2M_ISR_Handler (void) interrupt DMA_M2M_VECTOR
功能描述	M2M DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_M2M_ISR_Handler (void) interrupt DMA_M2M_VECTOR
{
    if(DMA_M2M_STA & 0x01)  //M2M DMA 传输完成
    {
        DMA_M2M_STA &= ~0x01;       //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.10 UART1 Tx DMA 中断函数

函数名	void DMA_UART1TX_ISR_Handler (void) interrupt DMA_UR1T_VECTOR
功能描述	UART1 Tx DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_UART1TX_ISR_Handler (void) interrupt DMA_UR1T_VECTOR
{
    if (DMA_UR1T_STA & 0x01) //UART1 DMA 发送完成
    {
        DMA_UR1T_STA &= ~0x01;       //清标志位
        // TODO: 在此处添加用户代码
    }
    if (DMA_UR1T_STA & 0x04) //产生数据覆盖
    {
        DMA_UR1T_STA &= ~0x04;       //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.11 UART1 Rx DMA 中断函数

函数名	void DMA_UART1RX_ISR_Handler (void) interrupt DMA_UR1R_VECTOR
功能描述	UART1 Rx DMA 中断程序
参数	无

返回	无

程序框架:

```
void DMA_UART1RX_ISR_Handler (void) interrupt DMA_UR1R_VECTOR
{
    if (DMA_UR1R_STA & 0x01) //UART1 DMA 发送完成
    {
        DMA_UR1R_STA &= ~0x01;      //清标志位
        // TODO: 在此处添加用户代码
    }
    if (DMA_UR1R_STA & 0x02) //产生数据丢弃
    {
        DMA_UR1R_STA &= ~0x02;      //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.12 UART2 Tx DMA 中断函数

函数名	void DMA_UART2TX_ISR_Handler (void) interrupt DMA_UR2T_VECTOR
功能描述	UART2 Tx DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_UART2TX_ISR_Handler (void) interrupt DMA_UR2T_VECTOR
{
    if (DMA_UR2T_STA & 0x01) //UART2 DMA 发送完成
    {
        DMA_UR2T_STA &= ~0x01;      //清标志位
        // TODO: 在此处添加用户代码
    }
    if (DMA_UR2T_STA & 0x04) //产生数据覆盖
    {
        DMA_UR2T_STA &= ~0x04;      //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.13 UART2 Rx DMA 中断函数

函数名	void DMA_UART2RX_ISR_Handler (void) interrupt DMA_UR2R_VECTOR
功能描述	UART2 Rx DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_UART2RX_ISR_Handler (void) interrupt DMA_UR2R_VECTOR
{
    if (DMA_UR2R_STA & 0x01) //UART2 DMA 发送完成
    {
        DMA_UR2R_STA &= ~0x01;      //清标志位
        // TODO: 在此处添加用户代码
    }
    if (DMA_UR2R_STA & 0x02) //产生数据丢弃
    {
        DMA_UR2R_STA &= ~0x02;      //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.14 UART3 Tx DMA 中断函数

函数名	void DMA_UART3TX_ISR_Handler (void) interrupt DMA_UR3T_VECTOR
功能描述	UART3 Tx DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_UART3TX_ISR_Handler (void) interrupt DMA_UR3T_VECTOR
{
    if (DMA_UR3T_STA & 0x01) //UART3 DMA 发送完成
    {
        DMA_UR3T_STA &= ~0x01;      //清标志位
        // TODO: 在此处添加用户代码
    }
    if (DMA_UR3T_STA & 0x04) //产生数据覆盖
    {
        DMA_UR3T_STA &= ~0x04;      //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.15 UART3 Rx DMA 中断函数

函数名	void DMA_UART3RX_ISR_Handler (void) interrupt DMA_UR3R_VECTOR
功能描述	UART3 Rx DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_UART3RX_ISR_Handler (void) interrupt DMA_UR3R_VECTOR
{
    if (DMA_UR3R_STA & 0x01) //UART3 DMA 发送完成
```

```
    {
        DMA_UR3R_STA &= ~0x01;      //清标志位
        // TODO: 在此处添加用户代码
    }
    if (DMA_UR3R_STA & 0x02) //产生数据丢弃
    {
        DMA_UR3R_STA &= ~0x02;      //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.16 UART4 Tx DMA 中断函数

函数名	void DMA_UART4TX_ISR_Handler (void) interrupt DMA_UR4T_VECTOR
功能描述	UART4 Tx DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_UART4TX_ISR_Handler (void) interrupt DMA_UR4T_VECTOR
{
    if (DMA_UR4T_STA & 0x01) //UART4 DMA 发送完成
    {
        DMA_UR4T_STA &= ~0x01;      //清标志位
        // TODO: 在此处添加用户代码
    }
    if (DMA_UR4T_STA & 0x04) //产生数据覆盖
    {
        DMA_UR4T_STA &= ~0x04;      //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.17 UART4 Rx DMA 中断函数

函数名	void DMA_UART4RX_ISR_Handler (void) interrupt DMA_UR4R_VECTOR
功能描述	UART4 Rx DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_UART4RX_ISR_Handler (void) interrupt DMA_UR4R_VECTOR
{
    if (DMA_UR4R_STA & 0x01) //UART4 DMA 发送完成
    {
        DMA_UR4R_STA &= ~0x01;      //清标志位
        // TODO: 在此处添加用户代码
```

```
    }
    if (DMA_UR4R_STA & 0x02) //产生数据丢弃
    {
        DMA_UR4R_STA &= ~0x02;      //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.18 SPI DMA 中断函数

函数名	void DMA_SPI_ISR_Handler (void) interrupt DMA_SPI_VECTOR
功能描述	SPI DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_SPI_ISR_Handler (void) interrupt DMA_SPI_VECTOR
{
    if(DMA_SPI_STA & 0x01)    //SPI DMA 传输完成
    {
        DMA_SPI_STA &= ~0x01;       //清标志位
        // TODO: 在此处添加用户代码
    }
    if(DMA_SPI_STA & 0x02)    //数据丢弃
    {
        DMA_SPI_STA &= ~0x02;   //清标志位
        // TODO: 在此处添加用户代码
    }
    if(DMA_SPI_STA & 0x04)    //数据覆盖
    {
        DMA_SPI_STA &= ~0x04;   //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.19 I2C DMA 发送完成中断函数

函数名	void DMA_I2CT_ISR_Handler (void) interrupt DMA_I2CT_VECTOR
功能描述	I2C DMA 发送完成中断程序
参数	无
返回	无

程序框架:

```
void DMA_I2CT_ISR_Handler (void) interrupt DMA_I2CT_VECTOR
{
    if(DMA_I2CT_STA & 0x01)   //发送完成
    {
```

```
        DMA_I2CT_STA &= ~0x01;       //清标志位
        // TODO: 在此处添加用户代码
    }
    if(DMA_I2CT_STA & 0x04)  //数据覆盖
    {
        DMA_I2CT_STA &= ~0x04;       //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.20 I2C DMA 接收完成中断函数

函数名	void DMA_I2CR_ISR_Handler (void) interrupt DMA_I2CR_VECTOR
功能描述	I2C DMA 接收完成中断程序
参数	无
返回	无

程序框架:

```
void DMA_I2CR_ISR_Handler (void) interrupt DMA_I2CR_VECTOR
{
    if(DMA_I2CR_STA & 0x01)  //接收完成
    {
        DMA_I2CR_STA &= ~0x01;       //清标志位
        // TODO: 在此处添加用户代码
    }
    if(DMA_I2CR_STA & 0x02)  //数据丢弃
    {
        DMA_I2CR_STA &= ~0x02;       //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.15.21 LCM DMA 中断函数

函数名	void DMA_LCM_ISR_Handler (void) interrupt DMA_LCM_VECTOR
功能描述	LCM DMA 中断程序
参数	无
返回	无

程序框架:

```
void DMA_LCM_ISR_Handler (void) interrupt DMA_LCM_VECTOR
{
    if(DMA_LCM_STA & 0x01)  //LCM DMA 传输完成
    {
        DMA_LCM_STA &= ~0x01;       //清标志位
        // TODO: 在此处添加用户代码
    }
```

```
    if(DMA_LCM_STA & 0x02)   //数据覆盖
    {
        DMA_LCM_STA &= ~0x02; //清标志位
        // TODO: 在此处添加用户代码
    }
}
```

4.16 STC32G_Clock

4.16.1 相关文件

“STC32G_Clock.c”， 主要包含系统时钟、高速外设时钟的初始化函数。
“STC32G_Clock.h”， 包含时钟初始化函数所需的头文件以及函数申明。

4.16.2 高速 IRC 时钟初始化函数

函数名	void HIRCClkConfig(u8 div)
功能描述	高速 IRC 时钟初始化程序
参数	div: 时钟分频系数，取值范围 0~255。
返回	无

4.16.3 外部晶振时钟初始化函数

函数名	void XOSCClkConfig(u8 div)
功能描述	外部晶振时钟初始化程序
参数	div: 时钟分频系数，取值范围 0~255。
返回	无

4.16.4 低速 32K IRC 时钟初始化函数

函数名	void IRC32KClkConfig(u8 div)
功能描述	低速 32K IRC 时钟初始化程序
参数	div: 时钟分频系数，取值范围 0~255。
返回	无

4.16.5 高速 IO 时钟初始化函数

函数名	void HSPllClkConfig(u8 clksrc, u8 pllsel, u8 div)
功能描述	高速 IO 时钟初始化程序
参数 1	clksrc: 主时钟选择
参数 2	pllsel: PLL 时钟选择
参数 3	div: 高速 IO 时钟分频系数，取值范围 0~255
返回	无

clksrc: 主时钟选择

参数	功能描述
MCLKSEL_HIRC	主时钟选择内部高精度 IRC
MCLKSEL_XIRC	主时钟选择外部高速晶振
MCLKSEL_X32K	主时钟选择外部 32KHz 晶振
MCLKSEL_I32K	主时钟选择内部 32KHz 低速 IRC
MCLKSEL_PLL	主时钟选择内部 PLL
MCLKSEL_PLL2	主时钟选择内部 PLL/2
MCLKSEL_I48M	主时钟选择内部高速 48MHz 高速 IRC

pllsel: PLL 时钟选择

参数	功能描述
PLL_96M	PLL 输出 96MHz
PLL_144M	PLL 输出 144MHz

4.17 STC32G_CAN

4.17.1 相关文件

“STC32G_CAN.c”， 主要包含 CAN 总线模块的初始化函数。
“STC32G_CAN_Isr.c”，主要包含 CAN 总线模块的中断函数。
“STC32G_CAN.h”， 包含 CAN 总线模块函数所需的头文件以及函数申明。

4.17.2 CAN 功能寄存器读取函数

函数名	u8 CanReadReg(u8 addr)
功能描述	CAN 功能寄存器读取函数
参数	addr: CAN 功能寄存器地址
返回	CAN 功能寄存器数据

4.17.3 CAN 功能寄存器配置函数

函数名	void CanWriteReg(u8 addr, u8 dat)
功能描述	CAN 功能寄存器配置函数
参数 1	addr: CAN 功能寄存器地址
参数 2	dat: CAN 功能寄存器数据
返回	无

4.17.4 CAN 初始化函数

函数名	void CAN_Inilize(u8 CANx, CAN_InitTypeDef *CAN)
功能描述	CAN 初始化程序
参数 1	CANx: CAN 设置通道， 取值 CAN1, CAN2
参数 2	CAN: 结构参数

返回	无

CAN: 结构参数定义：

```
typedef struct
{
    u8   CAN_Enable;
    u8   CAN_IMR;
    u8   CAN_SJW;
    u8   CAN_BRP;
    u8   CAN_SAM;
    u8   CAN_TSG1;
    u8   CAN_TSG2;
    u8   CAN_ListenOnly;
    u8   CAN_Filter;

    u8   CAN_ACR0;
    u8   CAN_ACR1;
    u8   CAN_ACR2;
    u8   CAN_ACR3;
    u8   CAN_AMR0
    u8   CAN_AMR1;
    u8   CAN_AMR2;
    u8   CAN_AMR3;
} CAN_InitTypeDef;
```

CAN_Enable: 功能使能

参数	功能描述
ENABLE	使能 CAN 功能
DISABLE	禁止 CAN 功能

CAN_IMR: 中断寄存器

参数	功能描述
CAN_DOIM	接收溢出中断
CAN_BEIM	总线错位中断
CAN_TIM	发送中断
CAN_RIM	接收中断
CAN_EPIM	被动错位中断
CAN_EWIM	错位警告中断
CAN_ALIM	仲裁丢失中断
CAN_ALLIM	使能所有中断
DISABLE	禁止所有中断

CAN_SJW: 重新同步跳跃宽度，取值 0~3。

CAN_BRP: 波特率分频系数，取值 0~63。

CAN_SAM: 总线电平采样次数，0:采样 1 次； 1:采样 3 次。

CAN_TSG1: 同步采样段 1，取值 0~15。

CAN_TSG2: 同步采样段 2，取值 1~7 (TSG2 不能设置为 0)。

CAN_ListenOnly: 监听模式使能

参数	功能描述
ENABLE	使能 Listen Only 模式
DISABLE	禁止 Listen Only 模式

CAN_Filter: 滤波模式选择

参数	功能描述
DUAL_FILTER	使用双滤波模式
SINGLE_FILTER	使用单滤波模式

CAN_ACR0: 总线验收代码寄存器，取值 0x00~0xFF。
CAN_ACR1: 总线验收代码寄存器，取值 0x00~0xFF。
CAN_ACR2: 总线验收代码寄存器，取值 0x00~0xFF。
CAN_ACR3: 总线验收代码寄存器，取值 0x00~0xFF。

CAN_AMR0: 总线验收屏蔽寄存器，取值 0x00~0xFF。
CAN_AMR1: 总线验收屏蔽寄存器，取值 0x00~0xFF。
CAN_AMR2: 总线验收屏蔽寄存器，取值 0x00~0xFF。
CAN_AMR3: 总线验收屏蔽寄存器，取值 0x00~0xFF。

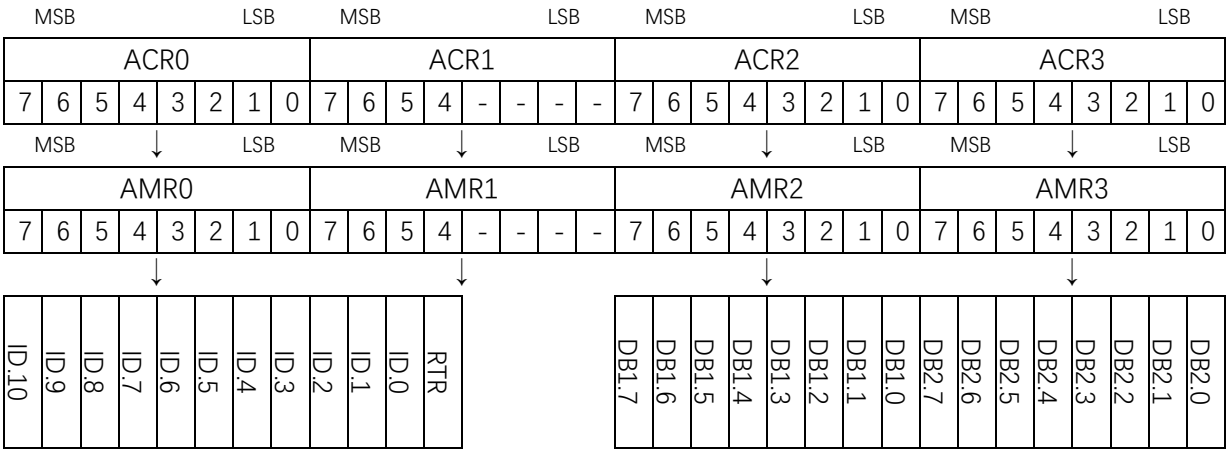
滤波的方式有两种，由模式寄存器中的 AFM（MR.0）位选择：单滤波器模式（AFM 位是 1）、双滤波器模式（AFM 位是 0）。

滤波的规则是：每一位验收屏蔽分别对应每一位验收代码，当该位验收屏蔽位为“1”的时候（即设为无关），接收的相应帧 ID 位无论是否和相应的验收代码位相同均会表示为接收；当验收屏蔽位为“0”的时候（即设为相关），只有相应的帧 ID 位和相应的验收代码位值相同的情况才会表示为接收。只有在所有的位都表示为接收的时候，CAN 控制器才会接收该报文。

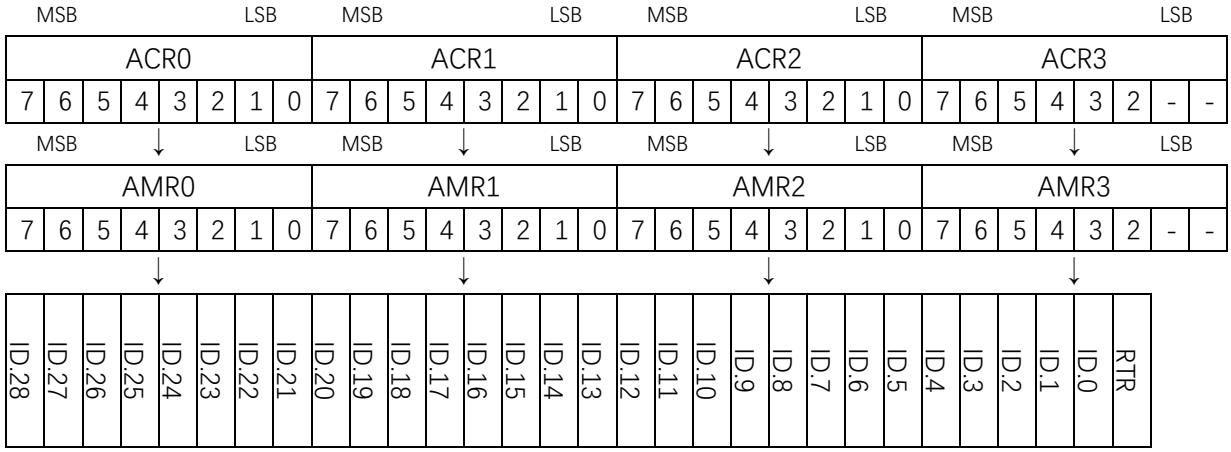
（1）单滤波器的配置

这种滤波器配置定义了一个长滤波器（4 字节、32 位），由 4 个验收码寄存器和 4 个验收屏蔽寄存器组成的验收滤波器，滤波器字节和信息字节之间位的对应关系取决于当前接收帧格式。接收 CAN 标准帧单滤波器配置：对于标准帧，11 位标识符、RTR 位、数据场前两个字节参与滤波；对于参与滤波的数据，所有 AMR 为 0 的位所对应的 ACR 位和参与滤波数据的对应位必须相同才算验收通过；如果由于置位 RTR=1 位而没有数据字节，或因为设置相应的数据长度代码而没有或只有一个数据字节信息，报文也会被接收。对于一个成功接收的报文，所有单个位在滤波器中的比较结果都必须为“接受”； 注意 AMR1 和 ACR1 的低四位是不用的，为了和将来的产品兼容，这些位可通过设置 AMR1.3、AMR1.2、AMR1.1 和 AMR1.0 为 1 而定为“不影响”。

接收 CAN 标准帧时单滤波器配置：


MSB LSB MSB LSB MSB LSB MSB LSB
ACR0 ACR1 ACR2 ACR3
7 6 5 4 3 2 1 0 7 6 5 4 - - - - 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
MSB LSB MSB LSB MSB LSB MSB LSB
AMR0 AMR1 AMR2 AMR3
7 6 5 4 3 2 1 0 7 6 5 4 - - - - 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
ID.10 ID.9 ID.8 ID.7 ID.6 ID.5 ID.4 ID.3 ID.2 ID.1 ID.0 RTR
DB1.7 DB1.6 DB1.5 DB1.4 DB1.3 DB1.2 DB1.1 DB1.0 DB2.7 DB2.6 DB2.5 DB2.4 DB2.3 DB2.2 DB2.1 DB2.0

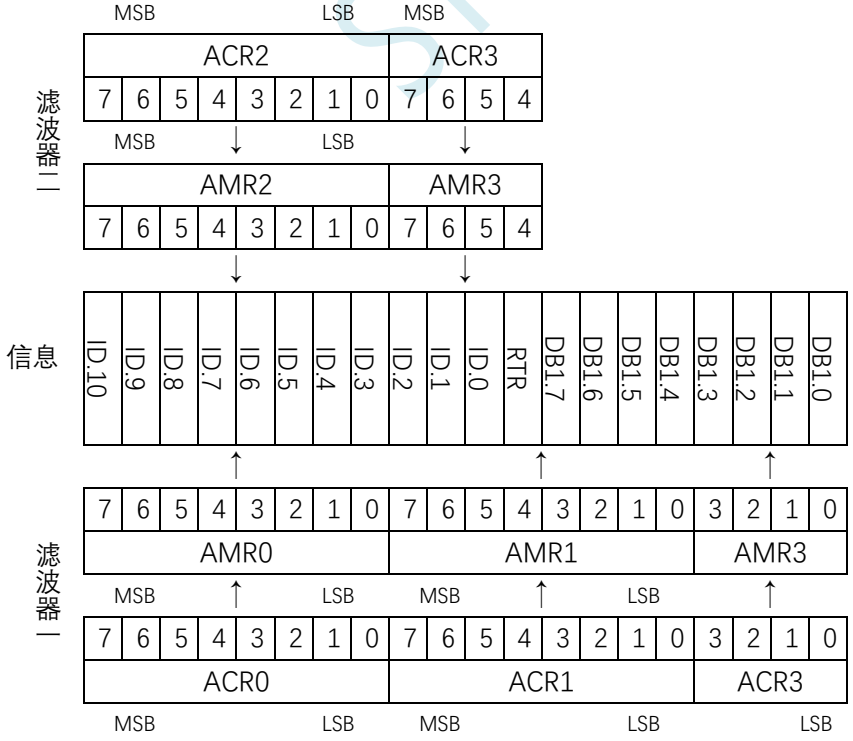
接收 CAN 扩展帧时单滤波器配置：



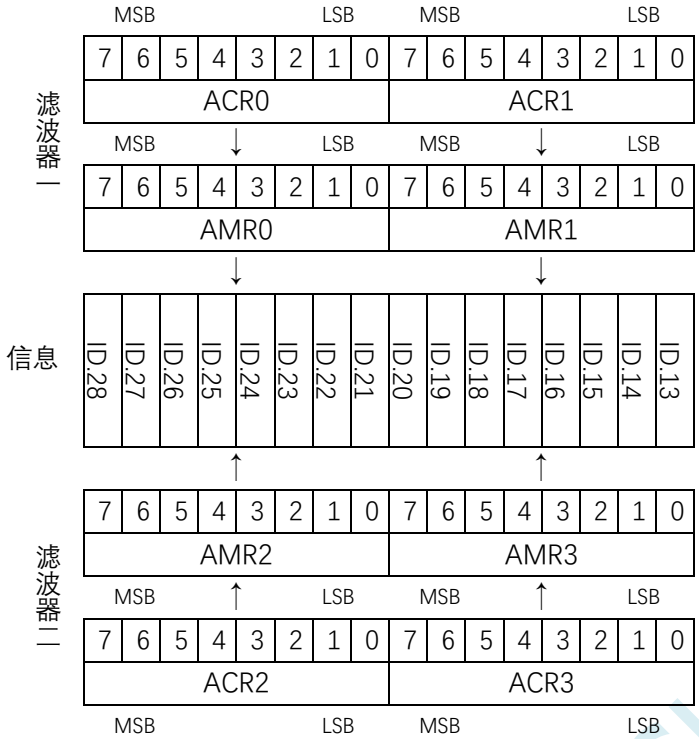
双滤波器的配置

这种配置可以定义两个短滤波器，由 4 个 ACR 和 4 个 AMR 构成两个短滤波器。总线上的信息只要通过任意一个滤波器就被接收。 滤波器字节和信息字节之间位的对应关系取决于当前接收的帧格式。接收 CAN 标准帧双滤波器的配置：如果接收的是标准帧信息，被定义的两个滤波器是不一样的。 第一个滤波器由 ACR0、ACR1、AMR0、AMR1 以及 ACR3、AMR3 低 4 位组成，11 位标识符、RTR 位和数据场第 1 字节参与滤波；第二个滤波器由 ACR2、AMR2 以及 ACR3、AMR3 高 4 位组成，11 位标识符和 RTR 位参与滤波。为了成功接收信息，在所有单个位的比较时，应至少有一个滤波器表示接受。RTR 位置为“1”或数据长度代码是“0 ”，表示没有数据字节存在；只要从开始到 RTR 位的部分都被表示接收，信息就可以通过滤波器 1。如果没有数据字节向滤波器请求过滤，AMR1 和 AMR3 的低 4 位必须被置为“1”，即“不影响”。此时，两个滤波器的识别工作都是验证包括 RTR 位在内的整个标准识别码。

接收 CAN 标准帧时双滤波器配置：



接收 CAN 扩展帧时双滤波器配置：



4.17.5 读取 CAN 缓冲区数据函数

函数名	void CanReadFifo(CAN_DataDef *CAN)
功能描述	读取 CAN 缓冲区数据函数
参数	CAN: 结构参数
返回	无

CAN: 结构参数定义：

```
typedef struct
{
    u8   DLC:4;              //数据长度, bit0~bit3
    u8   :2;                 //空数据, bit4~bit5
    u8   RTR:1;              //帧类型, bit6
    u8   FF:1;               //帧格式, bit7
    u32  ID;                 //CAN ID
    u8   DataBuffer[8];   //数据缓存
} CAN_DataDef;
```

DLC: 数据长度，位段定义于 bit0~bit3，取值 0~8。
: 空数据，位段定义于 bit4~bit5。
RTR: 帧类型，位段定义于 bit6，0: 标准帧；1: 扩展帧。
FF: 帧格式，位段定义于 bit7，0: 数据帧；1: 远程帧。

ID: CAN ID，标准帧取值 0x000~0x7ff；扩展帧取值 0x00000000~0x1fffffff。

DataBuffer[8]: CAN 数据缓冲区，上限 8 个字节。

4.17.6 CAN 接收数据函数

函数名	u8 CanReadMsg(CAN_DataDef *CAN)
功能描述	CAN 接收数据函数
参数	CAN: 结构参数，同上
返回	帧个数

4.17.7 CAN 发送标准帧函数

函数名	void CanSendMsg(CAN_DataDef *CAN)
功能描述	CAN 发送标准帧函数
参数	CAN: 结构参数，同上
返回	无

4.18 STC32G_LIN

4.18.1 相关文件

“STC32G_LIN.c”，主要包含 LIN 总线模块的初始化函数。
“STC32G_LIN_Isr.c”，主要包含 LIN 总线模块的中断函数。
“STC32G_LIN.h”，包含 LIN 总线模块函数所需的头文件以及函数申明。

4.18.2 Lin 功能寄存器读取函数

函数名	u8 LinReadReg(u8 addr)
功能描述	Lin 功能寄存器读取函数
参数	addr: LIN 功能寄存器地址
返回	LIN 功能寄存器数据

4.18.3 Lin 功能寄存器配置函数

函数名	void LinWriteReg(u8 addr, u8 dat)
功能描述	LIN 功能寄存器配置函数
参数 1	addr: LIN 功能寄存器地址
参数 2	dat: LIN 功能寄存器数据
返回	无

4.18.4 Lin 从机发送应答数据

函数名	void LinTxResponse(u8 *pdat)
功能描述	Lin 从机发送应答数据，跟主机发送的 Header 拼成一个完整的帧
参数	*pdat: 发送数据缓冲区

返回	无

4.18.5 Lin 从机接收数据帧函数

函数名	void LinReadFrame(u8 *pdat)
功能描述	Lin 从机接收数据帧函数
参数	*pdat: 接收数据缓冲区
返回	无

4.18.6 Lin 主机发送完整帧函数

函数名	void LinSendFrame(u8 lid, u8 *pdat)
功能描述	Lin 主机发送完整帧函数
参数 1	lid: Lin ID
参数 2	*pdat: 发送数据缓冲区
返回	无

4.18.7 Lin 主机发送帧头，接收从机应答数据

函数名	void LinSendHeaderRead(u8 lid, u8 *pdat)
功能描述	Lin 主机发送 Header，由从机发送应答数据，拼成一个完整的帧
参数 1	lid: 发送应答从机的总线 ID
参数 2	*pdat: 接收数据缓冲区
返回	无

4.18.8 Lin 总线波特率设置函数

函数名	void LinSetBaudrate(u16 brt)
功能描述	Lin 总线波特率设置函数
参数	brt: 波特率
返回	无

4.18.9 LIN 初始化程序

函数名	void LIN_Inilize(LIN_InitTypeDef *LIN)
功能描述	LIN 初始化程序
参数	LIN: 结构参数
返回	无

LIN: 结构参数定义：

```
typedef struct
{
    u8   LIN_Enable;
    u16  LIN_Baudrate;
```

```
    u8   LIN_IE;
    u8   LIN_HeadDelay;
    u8   LIN_HeadPrescaler;
} LIN_InitTypeDef;
```

LIN_Enable: 功能使能

参数	功能描述
ENABLE	使能 LIN 功能
DISABLE	禁止 LIN 功能

LIN_Baudrate: LIN 波特率(bit/s)，取值 1000~20000。

LIN_IE: LIN 中断使能

参数	功能描述
LIN_LIDE	使能 Head 中断
LIN_RDYE	使能 Ready 中断
LIN_ERRE	使能错误中断
LIN_ABORTE	使能终止中断
LIN_ALLIE	使能所有中断

LIN_HeadDelay: 帧头延时计数，取值 0~(65535*1000)/MAIN_Fosc。

LIN_HeadPrescaler: 帧头延时分频，取值 0~63。

4.19 STC32G_USART_LIN

4.19.1 相关文件

“STC32G_USART_LIN.c”， 主要包含 USART LIN 总线模块的初始化函数。
“STC32G_USART_LIN.h”， 包含 USART LIN 总线模块函数所需的头文件以及函数申明。

4.19.2 USART LIN 初始化程序

函数名	u8 UASRT_LIN_Configuration(u8 USARTx, USARTx_LIN_InitDefine *USART)
功能描述	USART LIN 初始化程序
参数 1	USARTx: UART 组号
参数 2	*USART: USART LIN 结构参数
返回	无

USART LIN: 结构参数定义：

```
typedef struct
{
    u8   LIN_Enable;
    u8   LIN_Mode;
```

```
    u8   LIN_AutoSync;
    u16  LIN_Baudrate;
} USARTx_LIN_InitDefine;
```

LIN_Enable: 功能使能

参数	功能描述
ENABLE	使能 LIN 功能
DISABLE	禁止 LIN 功能

LIN_Mode: 模式选择

参数	功能描述
LinMasterMode	设置主机模式
LinSlaveMode	设置从机模式

LIN_AutoSync: 自动同步使能

参数	功能描述
ENABLE	使能自动同步功能
DISABLE	禁止自动同步功能

LIN_Baudrate: LIN 波特率(bit/s)，取值 1000~20000。

4.19.3 USART LIN 发送数据函数

函数名	void UsartLinSendData(u8 USARTx, u8 *pdat, u8 len)
功能描述	USART Lin 发送数据函数
参数 1	USARTx: USART 组号，取值：USART1 或者 USART2
参数 2	*pdat: 发生数据缓冲区
参数 3	len: 数据长度
返回	无

4.19.4 USART LIN 计算校验码并发送函数

函数名	void UsartLinSendChecksum(u8 USARTx, u8 *dat, u8 len)
功能描述	USART Lin 计算校验码并发送
参数 1	USARTx: USART 组号，取值：USART1 或者 USART2
参数 2	*dat: 数据场传输的数据
参数 3	len: 数据长度
返回	无

4.19.5 USART LIN 主机发送帧头函数

函数名	void UsartLinSendHeader(u8 USARTx, u8 lid)
功能描述	USART Lin 主机发送帧头
参数 1	USARTx: USART 组号，取值：USART1 或者 USART2

参数 2	lid: LIN ID
返回	无

4.19.6 USART LIN 主机发送完整帧函数

函数名	void UsartLinSendFrame(u8 USARTx, u8 lid, u8 *pdat, u8 len)
功能描述	USART Lin 主机发送完整帧
参数 1	USARTx: USART 组号，取值：USART1 或者 USART2
参数 2	*pdat: 发送数据缓冲区
参数 3	len: 数据长度
返回	无

4.20 STC32G_Delay

4.20.1 相关文件

“STC32G_Delay.c”， 主要包含一个自适应主时钟的延时函数。
“STC32G_Delay.h”， 包含延时函数所需的头文件以及函数申明。

4.20.2 延时函数

函数名	void delay_ms(unsigned char ms)
功能描述	延时函数
参数	ms: 要延时的毫秒数，这里只支持 1~255ms。自动适应主时钟。
返回	无

4.21 STC32G_Soft_I2C

4.21.1 相关文件

“STC32G_Soft_I2C.c”，主要用于 IO 口模拟 I2C 功能的配置。
“STC32G_Soft_I2C.h”，IO 口模拟 I2C 所需的变量申明与宏定义。

宏定义

```
#define SLAW    0x5A      //设置模拟 I2C 设备写地址
#define SLAR    0x5B      //设置模拟 I2C 设备读地址

sbit    SDA = P0^1;       //定义模拟 I2C 的 SDA 脚
sbit    SCL = P0^0;       //定义模拟 I2C 的 SCL 脚
```

4.21.2 IO 口模拟模拟 I2C 发送一串数据

函数名	void SI2C_WriteNbyte(u8 dev_addr, u8 mem_addr, u8 *p, u8 number)
功能描述	IO 口模拟 I2C 发送一串数据函数

参数 1	dev_addr: 设备地址
参数 2	mem_addr: 存储地址
参数 3	*p: 发送数据存储位置
参数 4	number: 发送数据个数
返回	无

4.21.3 IO 口模拟 I2C 读取一串数据

函数名	void SI2C_ReadNbyte(u8 dev_addr, u8 mem_addr, u8 *p, u8 number)
功能描述	IO 口模拟 I2C 读取一串数据函数
参数 1	dev_addr: 设备地址
参数 2	mem_addr: 存储地址
参数 3	*p: 读取数据存储位置
参数 4	number: 读取数据个数
返回	无

4.22 STC32G_Soft_UART

4.22.1 相关文件

“STC32G_Soft_UART.c”，主要用于 IO 口模拟 UART 功能的配置。
“STC32G_Soft_UART.h”，IO 口模拟 UART 所需的变量申明与宏定义。

宏定义

```
sbit    P_TXD = P3^1;    //定义模拟串口发送端,可以是任意 IO
```

4.22.2 IO 口模拟 UART 发送一个字节数据

函数名	void TxSend(u8 dat)
功能描述	模拟串口发送程序，可作为测试监控用。固定串口参数: 9600,8,n,1。为避免中断影响，发送时关闭总中断。
参数	dat: 待发送的字节
返回	无

4.22.3 IO 口模拟 UART 发送一串数据

函数名	void PrintString(unsigned char code *puts)
功能描述	模拟串口发送一串字符串
参数	*puts: 要发送的字符指针
返回	无

5 独立例程使用说明

打开例程包的“Independent_Programme”文件夹，里面包含单片机各硬件模块的使用例程:

开发过程中需要用到哪个模块的功能，就可以选择对应的例程进行验证、修改、移植等。

5.1 IO 口使用-跑马灯

IO 口的输入/输出是单片机最基本的功能，STC 单片机的 IO 口有四种模式：准双向口、推挽输出、高阻输入、开漏输出。

其中 P3.0，P3.1 口上电默认为准双向口模式，其他脚位上电默认是高阻输入模式。所以在 IO 口使用之前，要根据功能需要对相应的 IO 口进行模式配置。

5.1.1 项目文件

打开例程“01-IO-跑马灯”，其中包含编译文件存放目录“list”文件夹，其它文件介绍如下。由于跑马灯点亮 LED 后需要延时一段时间再进行翻转，方便目测，所以需要延时函数相关文件“STC32G_Delay.c”和配套头文件“STC32G_Delay.h”。

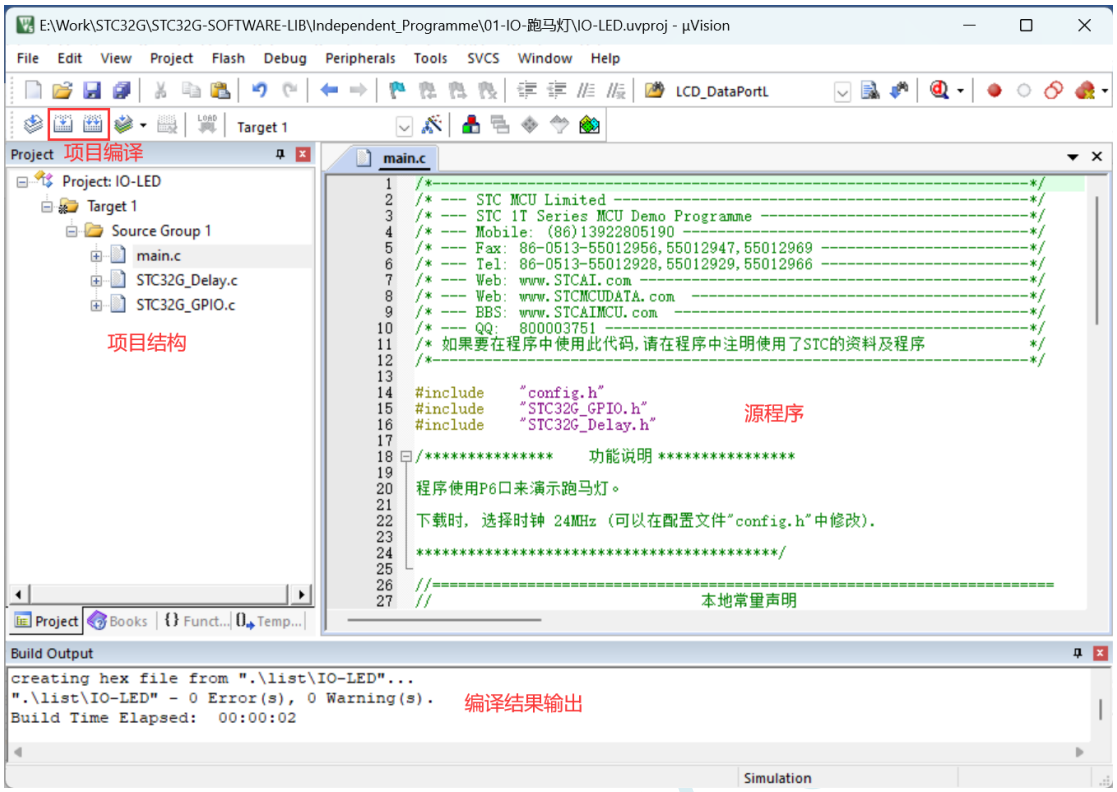
文件	描述
Config.h	用户配置文件，主要是主时钟定义
IO-LED(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_Delay (.h .c)	标准延时函数
Type_def.h	数据类型定义文件

5.1.2 程序框架

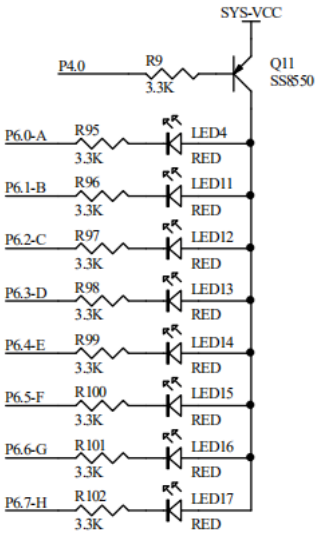
双击“IO-LED.uvproj”使用 Keil 编译器打开项目。

在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧

项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程是在 STC32G 实验箱上进行验证，实验箱流水灯连接原理图如下:



8 个独立 LED 指示灯实验

LED 电源通过 P4.0 脚控制，8 个 LED 灯通过 P6.0~P6.7 口分别控制。所以在使用前需要对这些脚位进行模式配置，在例程里设置为准双向口模式:

```
void GPIO_config(void)
{
    P4_MODE_IO_PU(GPIO_Pin_0);      //P4.0 设置为准双向口
    P6_MODE_IO_PU(GPIO_Pin_All);    //P6  设置为准双向口
}
```

每项设置可以选择的参数值都可以在第四章对应小节里面找到。

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
EAXSFR();   //扩展 SFR(XFR)访问使能
CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

```
GPIO_config();
P40 = 0;        //打开实验箱 LED 电源
```

主循环包含指令如下:

```
while(1)
{
    delay_ms(250);          //延时 250ms，用于保持当前 LED 状态，方便目测
    P6 = ~ledNum[ledIndex];  //设置 P6 口 LED 灯的驱动电平
    ledIndex++;             //LED 状态指针加 1
    if(ledIndex > 7)        //如果 LED 状态指针大于 7
    {
        ledIndex = 0;       //LED 状态指针清零，重新循环设置 LED 状态
    }
}
```

5.2 定时器使用-Timer0-Timer1-Timer2-Timer3-Timer4 测试程序

5.2.1 项目文件

打开例程“02-Timer0-Timer1-Timer2-Timer3-Timer4 测试程序”，其中包含编译文件存放目录“list”文件夹，其它文件介绍如下:

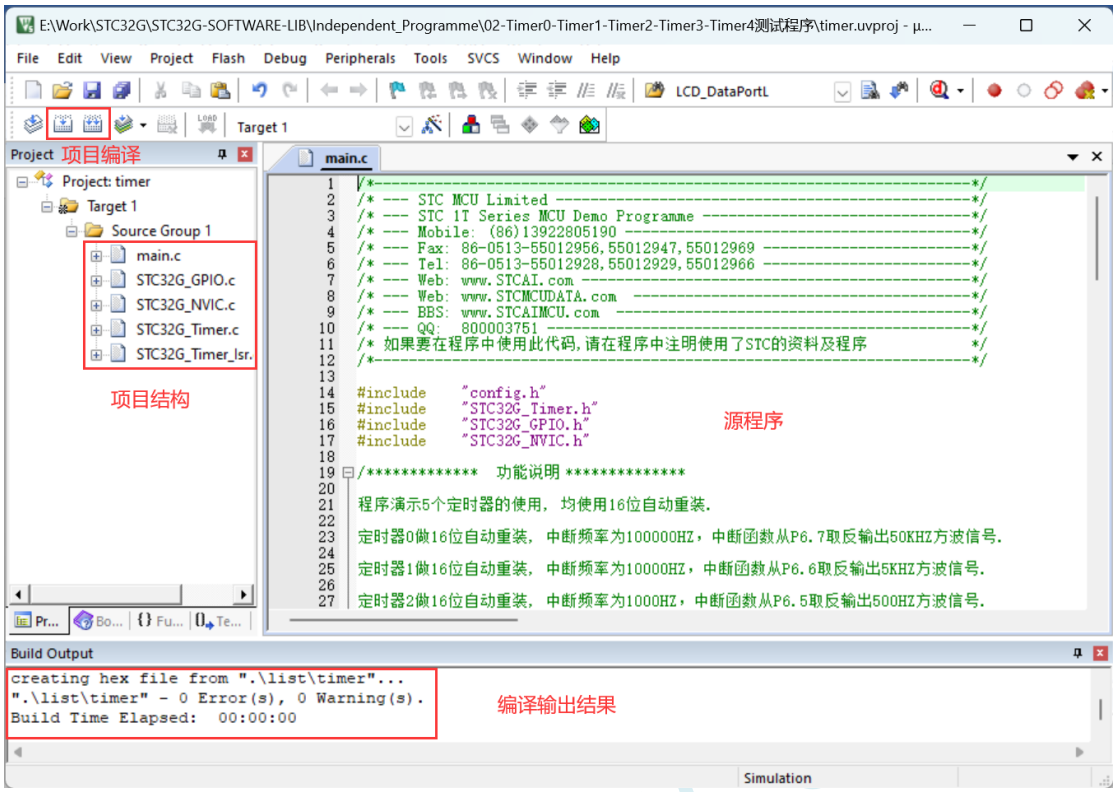
文件	描述
Config.h	用户配置文件，主要是主时钟定义
timer(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_Timer (.h .c)	定时器模块初始化及应用相关函数库
STC32G_Timer_Isr.c	定时器模块中断函数库
Type_def.h	数据类型定义文件

5.2.2 程序框架

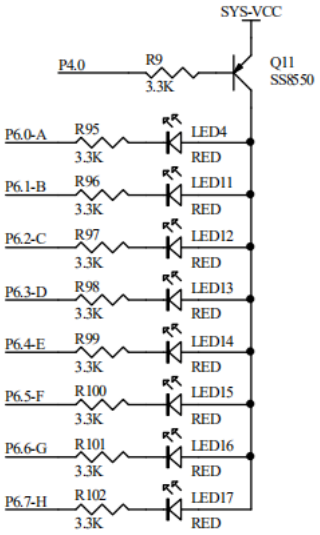
双击“timer.uvproj”使用 Keil 编译器打开项目。

在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧

项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程是在 STC32G 实验箱上进行验证，在定时器中断里面翻转指示灯控制 IO 口，通过示波器测量指示灯闪烁周期来验证定时器中断周期是否准确，实验箱指示灯连接原理图如下：



8 个独立 LED 指示灯实验

LED 电源通过 P4.0 脚控制，8 个 LED 灯通过 P6.0~P6.7 口分别控制。所以在使用前需要对这些脚位进行模式配置，在例程里设置为准双向口模式：

```
void GPIO_config(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;          //结构定义

    //指定要初始化的 IO, GPIO_Pin_0 ~ GPIO_Pin_7, 或操作
    GPIO_InitStructure.Pin    = GPIO_Pin_HIGH | GPIO_Pin_3;      //选择 Px.3 ~ Px.7
```

```
    //指定 IO 的输入或输出方式,GPIO_PullUp,GPIO_HighZ,GPIO_OUT_OD,GPIO_OUT_PP
    GPIO_InitStructure.Mode = GPIO_PullUp;      //选择准双向模式
    GPIO_Inilize(GPIO_P6,&GPIO_InitStructure);  //初始化

    //指定要初始化的 IO, GPIO_Pin_0 ~ GPIO_Pin_7, 或操作
    GPIO_InitStructure.Pin  = GPIO_Pin_0;       //选择 Px.0
    //指定 IO 的输入或输出方式,GPIO_PullUp,GPIO_HighZ,GPIO_OUT_OD,GPIO_OUT_PP
    GPIO_InitStructure.Mode = GPIO_PullUp;      //选择准双向模式
    GPIO_Inilize(GPIO_P4,&GPIO_InitStructure);  //初始化
}
```

定时器参数设置内容如下:

```
    TIM_InitTypeDef     TIM_InitStructure;      //结构定义
    TIM_InitStructure.TIM_Mode      = TIM_16BitAutoReload; //16 位自动重载模式
    TIM_InitStructure.TIM_ClkSource = TIM_CLOCK_1T;      //指定 1T 时钟源
    TIM_InitStructure.TIM_ClkOut    = DISABLE;           //不输出高速脉冲
    TIM_InitStructure.TIM_Value     = (u16)(65536UL - (MAIN_Fosc / 100000UL)); //周期值
    TIM_InitStructure.TIM_Run       = ENABLE;            //初始化后启动定时器
    Timer_Inilize(Timer0,&TIM_InitStructure);            //初始化 Timer0
    NVIC_Timer0_Init(ENABLE,Priority_0);      // Timer0 中断使能,优先级 0 级
```

每项设置可以选择的参数值都可以在第四章对应小节里面找到。

其中周期值: (65536UL - (MAIN_Fosc / 100000UL)) 表示 1 秒钟中断 100000 次, 中断频率为 100000Hz

同样的方式可以设置 Timer1~Timer4。

主函数起始位置推荐先执行以下几条指令, 提升芯片的性能, 设置扩展寄存器访问使能 (起始位置开启后可以不用再关闭):

```
    WTST = 0;   //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();   //扩展 SFR(XFR)访问使能
    CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

```
    GPIO_config();
    Timer_config();
    EA = 1;     //主中断使能, 程序中用到任意中断, 都要先使能主中断
    P40 = 0;    //打开实验箱 LED 电源
```

此例程主要验证定时器中断功能, 主循环为空。

```
    while (1);
```

5.3 外中断 INT0-INT1-INT2-INT3- INT4 测试

5.3.1 项目文件

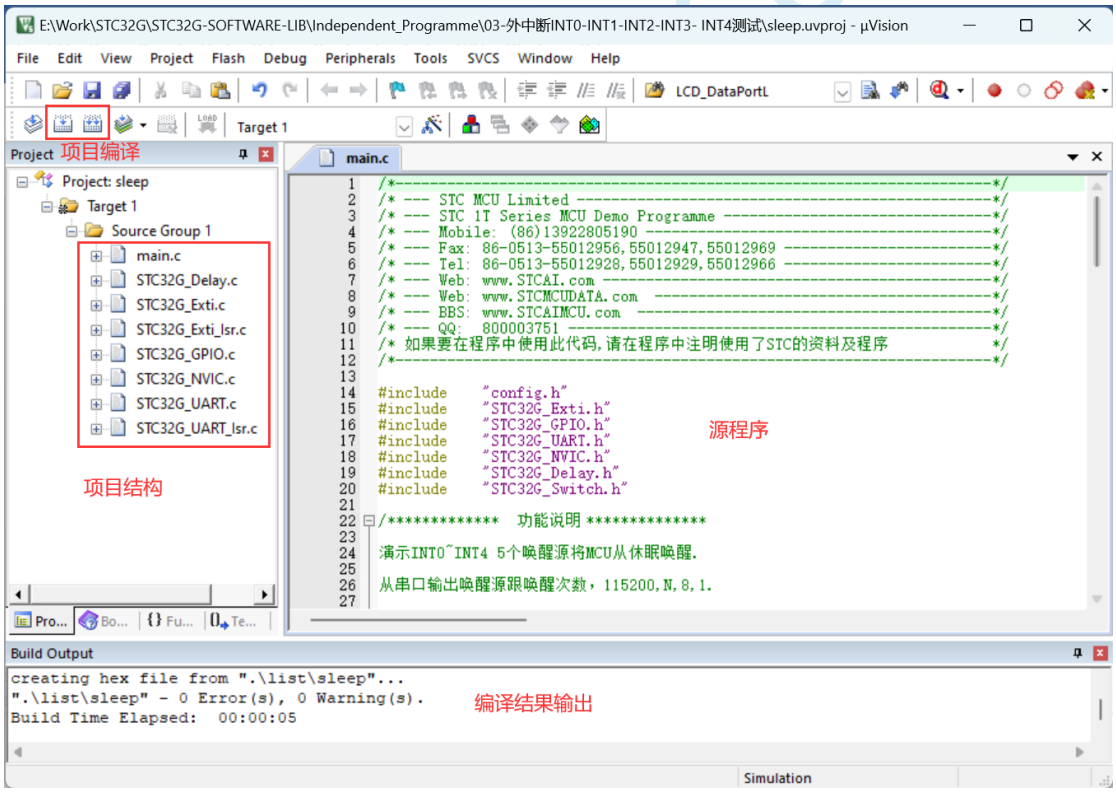
打开例程"03-外中断 INT0-INT1-INT2-INT3- INT4 测试", 其中包含编译文件存放目录"list"文件夹, 其它文件介绍如下:

文件	描述

Config.h	用户配置文件，主要是主时钟定义
sleep(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_Exti (.h .c)	外部中断模块初始化及应用相关函数库
STC32G_Exti_Isr.c	外部中断模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.3.2 程序框架

双击“sleep.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程演示 INT0~INT4，五个唤醒源将 MCU 从休眠唤醒，从串口 2 输出唤醒源跟唤醒次数，下载时，选择时钟 22.1184MHz (用户可在"config.h"修改频率)。

INT 脚位于 P3 口，INT0 : P3.2，INT1 : P3.3，INT2 : P3.6，INT3 : P3.7，INT4 : P3.0。串口 2 使用 P4.6, P4.7。
在使用前需要对这些脚位进行模式配置，在例程里设置为准双向口模式：
void GPIO_config(void)

```
{
    P3_MODE_IO_PU(GPIO_Pin_All);      //P3.0~P3.7 设置为准双向口
    P3_PULL_UP_ENABLE(GPIO_Pin_All); //P3 口内部上拉电阻使能
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);   //P4.6,P4.7 设置为准双向口
}
```

定时器参数设置内容如下:

```
void Exti_config(void)
{
    EXTI_InitTypeDef   Exti_InitStructure;                  //结构定义

    //中断模式,    EXT_MODE_RiseFall,EXT_MODE_Fall
    Exti_InitStructure.EXTI_Mode        = EXT_MODE_Fall;
    Ext_Inilize(EXT_INT0,&Exti_InitStructure);              //初始化
    NVIC_INT0_Init(ENABLE,Priority_0);          //中断使能, 优先级

    //中断模式,    EXT_MODE_RiseFall,EXT_MODE_Fall
    Exti_InitStructure.EXTI_Mode        = EXT_MODE_Fall;
    Ext_Inilize(EXT_INT1,&Exti_InitStructure);              //初始化
    NVIC_INT1_Init(ENABLE,Priority_0);          //中断使能, 优先级

    NVIC_INT2_Init(ENABLE,NULL);          //中断使能, ENABLE/DISABLE; 无优先级
    NVIC_INT3_Init(ENABLE,NULL);          //中断使能, ENABLE/DISABLE; 无优先级
    NVIC_INT4_Init(ENABLE,NULL);          //中断使能, ENABLE/DISABLE; 无优先级
}
```

通过 Ext_Inilize 函数设置 INT0, INT1 的中断模式, 可选择边沿触发, 或者下降沿触发。由于 INT2、INT3、INT4 只有下降沿模式, 所以不需要进行模式设置。

通过 NVIC_INT0_Init ~ NVIC_INT4_Init 函数设置对应的外部中断使能以及中断优先级。

每项设置可以选择的参数值都可以在第四章对应小节里面找到。

```
void UART_config(void)
{
    COMx_InitDefine        COMx_InitStructure;              //结构定义
    //模式,    UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode          = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;       //波特率,   110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;         //接收允许, ENABLE 或 DISABLE
    //初始化串口  UART1,UART2,UART3,UART4
    UART_Configuration(UART2, &COMx_InitStructure);
    NVIC_UART2_Init(ENABLE,Priority_1);       //中断使能, 优先级
    //串口通道选择,  UART2_SW_P10_P11,UART2_SW_P46_P47
    UART2_SW(UART2_SW_P46_P47);
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）：

```
WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
EAXSFR();   //扩展 SFR(XFR)访问使能
CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等：

```
GPIO_config();
UART_config();
Exti_config();
EA = 1;     //主中断使能，程序中用到任意中断，都要先使能主中断
```

通过串口打印一串数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此判断程序运行情况。

```
PrintString2("STC32G EXINT Wakeup Test Programme!\r\n");//UART 发送一个字符串
```

主循环内容：

由于 INT2、INT3、INT4 只能通过下降沿模式触发中断，所以主循环起始位置先等待外部中断脚位电平为高电平，重复执行两次简单防抖：

```
while(!INT0);  //等待外中断为高电平
while(!INT1);  //等待外中断为高电平
while(!INT2);  //等待外中断为高电平
while(!INT3);  //等待外中断为高电平
while(!INT4);  //等待外中断为高电平
delay_ms(10);//delay 10ms
```

初始化中断源变量：

```
WakeUpSource = 0;
```

该变量在“STC32G_Exti_Isr.c”文件，不同的外部中断函数里面设置成不同的值：

```
void INT0_ISR_Handler (void) interrupt INT0_VECTOR      //进中断时已经清除标志
{
    // TODO: 在此处添加用户代码
    WakeUpSource = 1;
}
void INT1_ISR_Handler (void) interrupt INT1_VECTOR      //进中断时已经清除标志
{
    // TODO: 在此处添加用户代码
    WakeUpSource = 2;
}
void INT2_ISR_Handler (void) interrupt INT2_VECTOR      //进中断时已经清除标志
{
    // TODO: 在此处添加用户代码
    WakeUpSource = 3;
}
void INT3_ISR_Handler (void) interrupt INT3_VECTOR      //进中断时已经清除标志
```

```
{
    // TODO: 在此处添加用户代码
    WakeUpSource = 4;
}
void INT4_ISR_Handler (void) interrupt INT4_VECTOR          //进中断时已经清除标志
{
    // TODO: 在此处添加用户代码
    WakeUpSource = 5;
}
```

通过串口打印一串信息提示用户 MCU 开始进入休眠状态，然后设置 PD 为 1 进入 Power Down 模式，此模式下 CPU 以及全部外设停止工作，振荡器停振。后面连续几个 nop 指令用于 MCU 唤醒后等待主时钟起振并稳定，不能删除!!!

```
PrintString2("MCU 进入休眠状态！\r\n");
PD = 1;       //Sleep
_nop_();
_nop_();
_nop_();
_nop_();
_nop_();
_nop_();
_nop_();
```

接下来判断唤醒源变量，通过串口打印唤醒源：

```
if(WakeUpSource == 1)    PrintString2("外中断 INT0 唤醒   ");
if(WakeUpSource == 2)    PrintString2("外中断 INT1 唤醒   ");
if(WakeUpSource == 3)    PrintString2("外中断 INT2 唤醒   ");
if(WakeUpSource == 4)    PrintString2("外中断 INT3 唤醒   ");
if(WakeUpSource == 5)    PrintString2("外中断 INT4 唤醒   ");
```

最后这段代码是通过一个计数器累计唤醒次数，并通过串口打印出来：

```
WakeUpCnt++;
TX2_write2buff((u8)(WakeUpCnt/100+'0'));
TX2_write2buff((u8)(WakeUpCnt%100/10+'0'));
TX2_write2buff((u8)(WakeUpCnt%10+'0'));
PrintString2("次唤醒\r\n");
```

5.4 看门狗复位测试程序

5.4.1 项目文件

打开例程“04-看门狗复位测试程序”，其中包含编译文件存放目录“list”文件夹，其它文件介绍如下：

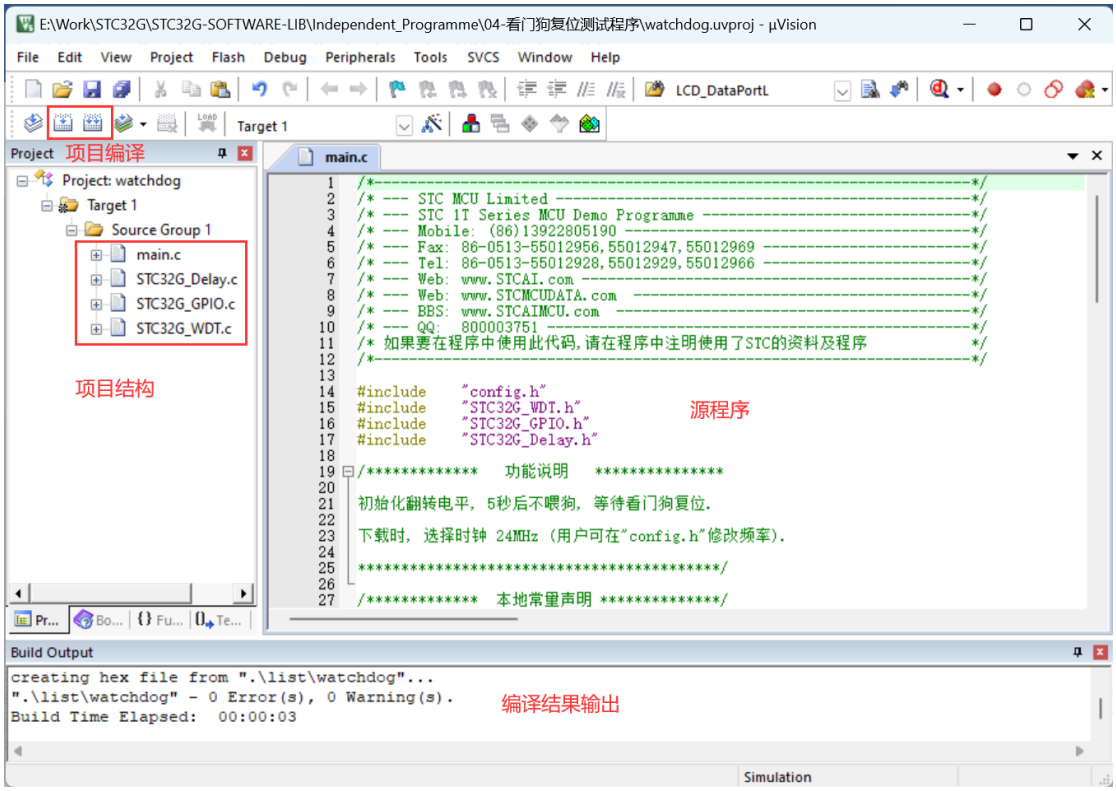
文件	描述
Config.h	用户配置文件，主要是主时钟定义
watchdog (.uvopt .uvproj)	项目文件
Main.c	主函数文件

STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_WDT (.h .c)	嵌套向量中断控制器初始化相关函数库
Type_def.h	数据类型定义文件

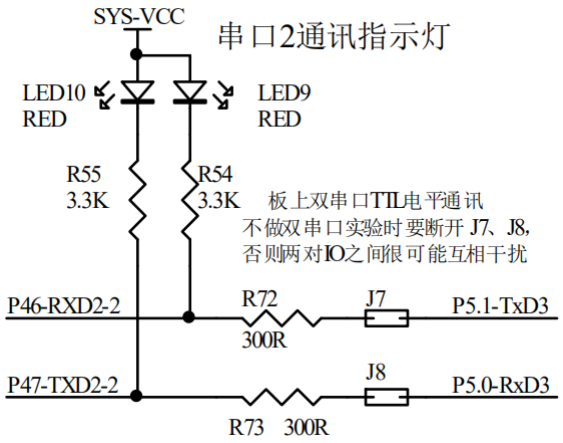
5.4.2 程序框架

双击“watchdog.uvproj”使用 Keil 编译器打开项目。

在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在"main.c"文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程演示看门狗复位功能，下载时，选择时钟 24MHz (用户可在"config.h"修改频率)。



该例程是在 STC32G 实验箱上进行验证，通过实验箱上 P4.7 脚连接的 LED10 指示灯来判断 MCU 是否产生复位。在使用前需要对这个脚位进行模式配置，在例程里设置为准双向口模式：

```
void GPIO_config(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;          //结构定义
    //指定要初始化的 IO, GPIO_Pin_0 ~ GPIO_Pin_7, 或操作
    GPIO_InitStructure.Pin  = GPIO_Pin_7;          //设置 Px.7
    //指定 IO 的输入或输出方式,GPIO_PullUp,GPIO_HighZ,GPIO_OUT_OD,GPIO_OUT_PP
    GPIO_InitStructure.Mode = GPIO_PullUp;         //准双向模式
    GPIO_Inilize(GPIO_P4,&GPIO_InitStructure);     //初始化 P4 口模式
}
```

对看门狗进行设置:

```
void WDT_config(void)
{
    WDT_InitTypeDef  WDT_InitStructure;                 //结构定义

    WDT_InitStructure.WDT_Enable      = ENABLE;     //看门狗使能 ENABLE 或 DISABLE
    //IDLE 模式是否停止计数 WDT_IDLE_STOP,WDT_IDLE_RUN
    WDT_InitStructure.WDT_IDLE_Mode   = WDT_IDLE_STOP;
    WDT_InitStructure.WDT_PS          = WDT_SCALE_16;// 看门狗定时器时钟分频系数
    WDT_Inilize(&WDT_InitStructure);                    //看门狗初始化
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;  //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();      //扩展 SFR(XFR)访问使能
    CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

```
    GPIO_config();
```

让 P4.7 连接的 LED10 闪两下，这样就可以通过 LED10 的状态来判断 MCU 是否产生复位。

```
    P47 = 0;       //LED10 亮
    delay_ms(200);
    P47 = 1;       //LED10 灭
    delay_ms(200);
    P47 = 0;       //LED10 亮
    delay_ms(200);
    P47 = 1;       //LED10 灭
    delay_ms(200);
```

设置并使能看门狗:

```
    WDT_config();

    //设置看门狗复位需要检测 P3.2 的状态，否则看门狗复位后进入 USB 下载模式
    RSTFLAG |= 0x04;
```

主循环里使用延时函数，每次循环历时 1ms，通过 ms_cnt 变量进行计数，累计 1000 次为 1 秒钟，将 second 变量加 1，5 秒之内主循环每次循环都会执行喂狗操作，5 秒之后就不再喂狗，等待看门狗溢出复位：

```
while(1)
{
    delay_ms(1);          //延时 1ms
    if(second <= 5)       //5 秒后不喂狗，将复位
        WDT_Clear();      // 喂狗

    if(++ms_cnt >= 1000)  //计数 1000 次为 1 秒钟
    {
        ms_cnt = 0;       //复位毫秒计数器，重新开始计数
        second++;         //秒变量加 1
    }
}
```

5.5 串口中断模式与电脑收发

5.5.1 项目文件

打开串中断模式与电脑收发测试例程，其中包含编译文件存放目录“list”文件夹，其它文件介绍如下：

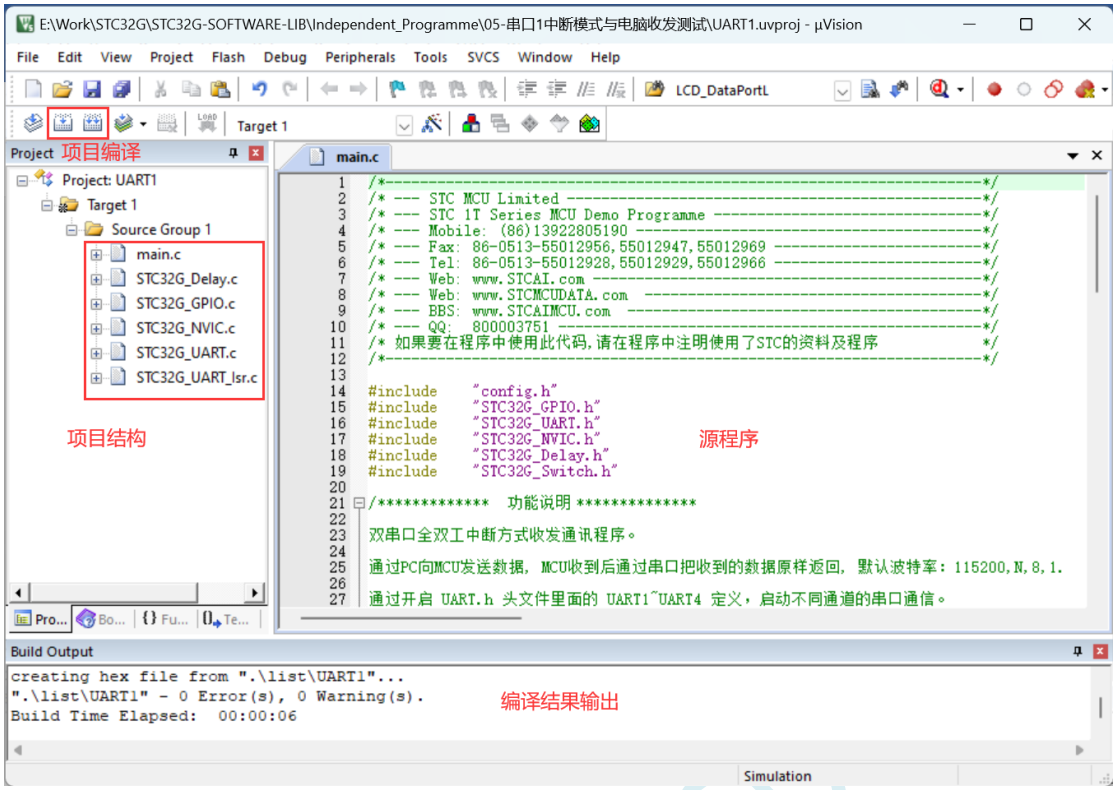
文件	描述
Config.h	用户配置文件，主要是主时钟定义
uartn(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.5.2 程序框架

四个串口使用方法一样，只需修改相应配置就行，这里以串口 1 例子作为介绍。

双击“uartn.uvproj”使用 Keil 编译器打开项目。

在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在"main.c"文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程是在 STC32G 实验箱上进行验证，通过实验箱上的 J17 调试接口可进行串口 1 与电脑通信测试（需断开 USB 接线）。P3.0,P3.1 引脚默认就是准双向模式，安全起见在初始化时最好对需要使用的 IO 口都进行模式配置，避免更换脚位时忘记设置导致功能不正常。在例程里设置为准双向口模式：

```
void GPIO_config(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;        //结构定义
    //指定要初始化的 IO, GPIO_Pin_0 ~ GPIO_Pin_7, 或操作
    GPIO_InitStructure.Pin  = GPIO_Pin_0 | GPIO_Pin_1;      //设置 Px.0, Px.1
    //指定 IO 的输入或输出方式,GPIO_PullUp,GPIO_HighZ,GPIO_OUT_OD,GPIO_OUT_PP
    GPIO_InitStructure.Mode = GPIO_PullUp;      //准双向模式
    GPIO_Inilize(GPIO_P3,&GPIO_InitStructure);  //初始化 P3 口模式
}
```

对串口进行设置：

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;     //结构定义

    //模式, UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode        = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer1, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer1;
    COMx_InitStructure.UART_BaudRate    = 115200ul;     //波特率
    COMx_InitStructure.UART_RxEnable    = ENABLE;       //接收允许
    COMx_InitStructure.BaudRateDouble = DISABLE;        //波特率加倍
    //初始化串口 1 UART1,UART2,UART3,UART4
```

```
    UART_Configuration(UART1, &COMx_InitStructure);
    NVIC_UART1_Init(ENABLE,Priority_1);       //中断使能，优先级

    //UART1_SW_P30_P31,UART1_SW_P36_P37,UART1_SW_P16_P17,UART1_SW_P43_P44
    UART1_SW(UART1_SW_P30_P31);     //通道切换，选择 P3.0, P3.1 引脚通信
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）：

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();       //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等：

```
    GPIO_config();
    UART_config();
    EA = 1;
```

通过串口 1 打印一串数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此判断程序运行情况。

```
    PrintString1("STC32G UART1 Test Programme!\r\n");    //UART 发送一个字符串
```

主循环里使用延时函数，每次循环历时 1ms，判断 RX_TimeOut 超时计数器是否被设置非 0 值，如果被设置的话说明串口中断已经收到数据，每次循环超时计数器减 1，减到 0 说明一串数据已经接收完毕，就可以根据需要进行处理。例程是将接收到的数据原样通过串口发送出来，这样就可以使用串口助手与 MCU 进行通信测试，串口助手发送一串数据给 MCU，如果能够收到 MCU 发送一模一样的数据回来，说明 MCU 的收发功能正常：

```
    while(1)
    {
        delay_ms(1);        //延时 1ms
        if(COM1.RX_TimeOut > 0)       //超时计数
        {
            if(--COM1.RX_TimeOut == 0)
            {
                if(COM1.RX_Cnt > 0)
                {
                    //收到的数据原样返回
                    for(i=0; i<COM1.RX_Cnt; i++)    TX1_write2buff(RX1_Buffer[i]);
                }
                COM1.RX_Cnt = 0;
            }
        }
    }
```

串口 1 中断里本例程用到的代码如下：

```
void UART1_ISR_Handler (void) interrupt UART1_VECTOR
```

```
{
    if(RI)      //判断串口 1 是否接收到一个字节数据
    {
        RI = 0;   //清除标志位
        //判断接收数据长度是否超过缓冲区上限，是的话循环覆盖
        if(COM1.RX_Cnt >= COM_RX1_Lenth)     COM1.RX_Cnt = 0;
        RX1_Buffer[COM1.RX_Cnt++] = SBUF;     //将接收的数据存入缓冲区
        COM1.RX_TimeOut = TimeOutSet1;        //设置超时时间
    }

    if(TI)      //判断串口 1 是否发送完成一个字节数据
    {
        TI = 0;   //清除标志位
        COM1.B_TX_busy = 0;       //清除发送繁忙标志
    }
}
```

例程中 TimeOutSet1 定义为 5，接收一个数据后设置超时时间为 5，然后主循环里开始每毫秒减 1，每次接收都会重置 RX_TimeOut，如果超过 5 毫秒没有收到下一个数据，就说明一串数据已经接收完毕。

实现 printf 打印到串口

printf 函数在 stdio.h 头文件中定义，我们先要添加头文件到程序里。为了实现 printf 重定位到串口，即把数据送到串口，我们需要重写 putchar 函数。

```
void TX1_write2buff(u8 dat)      //串口 1 发送函数
{
    SBUF = dat;                  //写入发送数据
    COM1.B_TX_busy = 1;          //设置发送忙标志
    while(COM1.B_TX_busy);       //等待发送完成
}
char putchar(char c)             //重写 putchar 函数，定向到串口
{
    TX1_write2buff(c);           //使用串口 1 发送数据
    return c;
}
```

如果要用其它串口实现 printf 打印，只要将 putchar 函数里的发送函数改成其它串口发送函数就行。例程可以通过修改“STC32G_UART.h”头文件里的定义来切换 putchar 函数对应的串口。

```
//使用哪些串口就开对应的定义，不用的串口可屏蔽掉定义，节省资源
#define  UART1   1
#define  UART2   2
#define  UART3   3
#define  UART4   4
//设置串口收发数据缓存空间，可选 edata 或者 xdata
#define  UART_BUF_type    edata    //这里选择 edata
//选择 printf 函数所使用的串口，参数 UART1~UART4
```

```
#define  PRINTF_SELECT  UART2   //这里选择串口 2 打印 printf 内容
```

5.6 ADC 转换

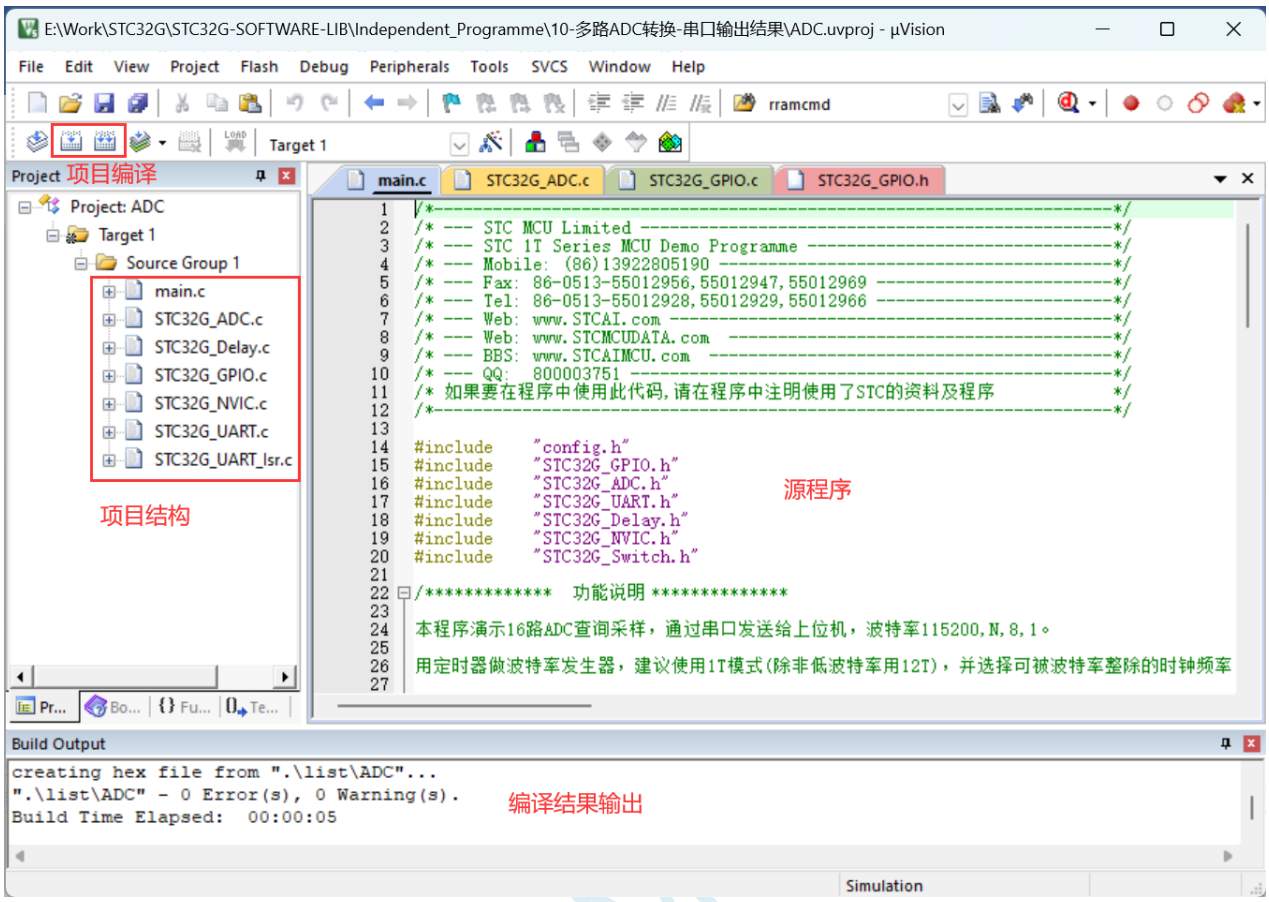
5.6.1 项目文件

打开多路 ADC 转换-串口输出结果例程，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
ADC(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_ADC (.h .c)	ADC 模块初始化及应用相关函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.6.2 程序框架

双击“ADC.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程是在 STC32G 实验箱上进行验证，通过实验箱上的 J17 调试接口可进行串口 1 与电脑通信测试（需断开 USB 接线）。P3.0,P3.1 引脚默认就是准双向模式，安全起见在初始化时最好对需要使用的 IO 口都进行模式配置，避免更换脚位时忘记设置导致功能不正常。在例程里设置为准双向口模式：

```
void GPIO_config(void)
{
    //P0.0~P0.6 设置为高阻输入
    P0_MODE_IN_HIZ(GPIO_Pin_LOW | GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6);
    P1_MODE_IN_HIZ(GPIO_Pin_All);     //P1.0~P1.7 设置为高阻输入
    P3_MODE_IO_PU(GPIO_Pin_0 | GPIO_Pin_1);   //P3.0,P3.1 设置为准双向口
}
```

对 ADC 进行设置：

```
void ADC_config(void)
{
    ADC_InitTypeDef     ADC_InitStructure;     //结构定义
    //ADC 模拟信号采样时间控制, 0~31（注意： SMPDUTY 一定不能设置小于 10）
    ADC_InitStructure.ADC_SMPduty   = 31;
    ADC_InitStructure.ADC_CsSetup   = 0;       //ADC 通道选择时间控制 0(默认),1
    ADC_InitStructure.ADC_CsHold    = 1;       //ADC 通道选择保持时间控制 0,1(默认),2,3
    //设置 ADC 工作时钟频率 ADC_SPEED_2X1T~ADC_SPEED_2X16T
    ADC_InitStructure.ADC_Speed     = ADC_SPEED_2X16T;
    //ADC 结果调整, ADC_LEFT_JUSTIFIED,ADC_RIGHT_JUSTIFIED
    ADC_InitStructure.ADC_AdjResult = ADC_RIGHT_JUSTIFIED;
```

```
    ADC_Inilize(&ADC_InitStructure);      //初始化
    ADC_PowerControl(ENABLE);             //ADC 电源开关, ENABLE 或 DISABLE
    NVIC_ADC_Init(DISABLE,Priority_0);    //关闭中断使能, 优先级
}
```

对串口进行设置:

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;     //结构定义

    //模式, UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode      = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer1, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2)
    COMx_InitStructure.UART_BRT_Use   = BRT_Timer1;
    COMx_InitStructure.UART_BaudRate  = 115200ul;     //波特率
    COMx_InitStructure.UART_RxEnable  = ENABLE;       //接收允许
    COMx_InitStructure.BaudRateDouble = DISABLE;      //波特率加倍
    //初始化串口 1 UART1,UART2,UART3,UART4
    UART_Configuration(UART1, &COMx_InitStructure);
    NVIC_UART1_Init(ENABLE,Priority_1);     //中断使能, 优先级

    //UART1_SW_P30_P31,UART1_SW_P36_P37,UART1_SW_P16_P17,UART1_SW_P43_P44
    UART1_SW(UART1_SW_P30_P31);   //通道切换, 选择 P3.0, P3.1 引脚通信
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();   //扩展 SFR(XFR)访问使能
    CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

```
    GPIO_config();
    UART_config();
    ADC_config();
    EA = 1;
```

通过串口 1 打印一串数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此判断程序运行情况。

```
    PrintString1("STC32G AD to UART Test Programme!\r\n");   //UART 发送一个字符串
```

主循环里使用 for 循环轮询采集 16 个通道的 ADC 值，通过串口将采集结果打印出来。使用延时函数，每次延时 200ms 进行采集打印，方便通过串口助手观察采集结果。

```
    while(1)
    {
        for(i=0; i<16; i++)       //轮询采集 16 个通道 ADC
```

```
    {
        delay_ms(200);      //延时 200ms，降低打印速度，方便观察

        //Get_ADCResult(i);           //参数 0~15,查询方式做一次 ADC，丢弃一次
        j = Get_ADCResult(i);   //参数 0~15,查询方式做一次 ADC，返回值就是结果， 4096 为错误
        printf("AD%02d=%04d ",i,j);  //打印采集结果
        if((i & 7) == 7)       printf("\r\n");       //每打印 8 个通道后换行
    }
    printf("\r\n");       //打印 16 个通道后再次换行
}
```

通过串口助手查看结果如下：

ADC 采样需要注意的地方：

1. ADC_Vref 脚电压是 ADC 采集的基准电压，要求采集结果精确的话，ADC_Vref 脚电压一定要稳定。ADC 采样范围在：0~ADC_Vref 脚电压之间，ADC_Vref 脚电压的范围在：2.4V~VCC 电压之间。
2. ADC 模块电源打开后，需要等待 1ms，等待 MCU 内部 ADC 电源稳定后再开始采集。
3. 适当加长对外部信号的采样时间，也就是加长对 ADC 内部采样保持电容的充电或放电时间，才能保持内部和外部电势相等。

5.7 EEPROM 读写

5.7.1 项目文件

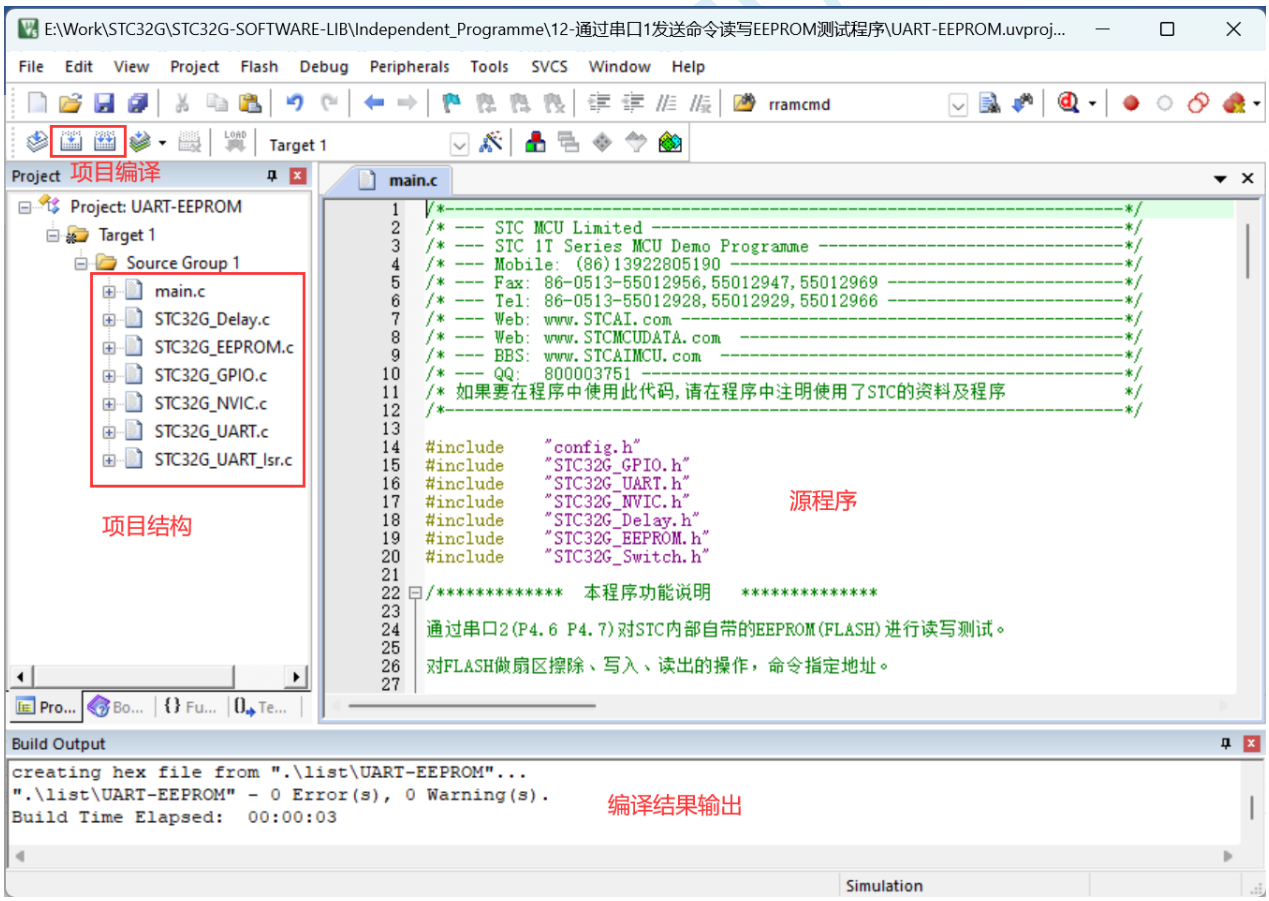
打开多路 ADC 转换-串口输出结果例程，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述

Config.h	用户配置文件，主要是主时钟定义
UART-EEPROM(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_EEPROM (.h .c)	内部 EEPROM(Flash)模块应用相关函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.7.2 程序框架

双击“UART-EEPROM.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程是在 STC32G 实验箱上进行验证，通过实验箱上的 J2 串口 DB9 接口可通过电脑发指令给串口 2（P4.6,P4.7）进行 EEPROM 的读写测试。初始化时对所有用到的 IO 口进行模式配置。在例程里设置为准双向口模式：

void GPIO_config(void)

```
{
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);          //P4.6,P4.7 设置为准双向口
}
```

对串口进行设置:

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;          //结构定义

    //模式, UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode        = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer1, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;      //波特率
    COMx_InitStructure.UART_RxEnable    = ENABLE;        //接收允许
    //初始化串口 UART1,UART2,UART3,UART4
    UART_Configuration(UART2, &COMx_InitStructure);
    NVIC_UART2_Init(ENABLE,Priority_1);      //中断使能, 优先级

    //UART2_SW_P10_P11,UART2_SW_P46_P47
    UART2_SW(UART2_SW_P46_P47);     //通道切换, 选择 P4.6, P4.7 引脚通信
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();     //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

```
    GPIO_config();
    UART_config();
    EA = 1;
```

MCU 上电后通过串口 2 打印一串提示信息，通过这些信息了解例程读取、写入、擦除 EEPROM 所使用的命令格式。注意使用包含 0xfd 编码的汉字时，需要在后面添加“\xfd”转义字符:

```
    PrintString2("STC32 系列单片机 EEPROM 测试程序, 串口命令设置如下:\r\n");
    delay_ms(5);      //等待串口数据发送完成
    PrintString2("E 0x000040                --> 对 0x000040 地址扇区进行擦除\xfd.\r\n");
    delay_ms(5);      //等待串口数据发送完成
    PrintString2("W 0x000040 1234567890   --> 对 0x000040 地址写入字符 1234567890.\r\n");
    delay_ms(5);      //等待串口数据发送完成
    PrintString2("R 0x000040 10             --> 对 0x000040 地址读出 10 个字节内容.\r\n");
    delay_ms(5);      //等待串口数据发送完成
```

由于串口采用队列方式传输数据，如果一次性写入队列的数据量超过缓冲区大小的话，会导致数据溢出而产生覆盖。这里采用发送一段数据，延时一段时间再发送的方式。可以通过“STC32_UART.h”文件修改

设置串口发送数据缓冲区大小来提升一次性可以发送的数据量。或者通过修改“STC32_UART.h”文件串口发送模式 UART_QUEUE_MODE 定义，使用阻塞方式进行串口发送。

主循环里每 1 毫秒检测一次串口 2 的超时计数器，如果计数器非 0 的话每次循环减 1（每毫秒减 1），串口中断里每次收到数据计数器都将重新设置为 5，所以在主循环里如果计数器减到 0 的话，说明已经连续 5 毫秒没有收到串口数据了，表明一串数据已经接收完毕，接下来就可以对接收的数据进行解析，判断内容是否为有效指令，如果是有效指令，则执行相应操作。

```
while(1)
{
    delay_ms(1);        //每 1 毫秒执行一次主循环，也可以使用定时器计时
    if(COM2.RX_TimeOut > 0)        //判断超时计数器
    {
        if(--COM2.RX_TimeOut == 0)
        {
            //收到一串数据，判断数据是否有效指令
        }
}
```

收到一串数据后，通过以下流程判断数据是否为有效指令。

1. 设置一个状态标志，用于判断指令是否有效：

```
status = 0xff;  //状态给一个非 0 值
```

2. 判断数据长度，格式是否符合指令条件。有效指令 10 个字节以上，并且第二字节为空格：

```
if((COM2.RX_Cnt >= 10) && (RX2_Buffer[1] == ' ')) //最短命令为 10 个字节
{
```

3. 将前 10 个字节内容全部转成大小，方便后续判断：

```
    for(i=0; i<10; i++)
    {
        //小写转大写
        if((RX2_Buffer[i] >= 'a') && (RX2_Buffer[i] <= 'z'))      RX2_Buffer[i] = RX2_Buffer[i] - 'a' + 'A';
    }
```

4. 提取数据里代表地址的内容，并判断是否有效：

```
    addr = GetAddress();
    if(addr < 0x00ffffff)      //限制地址范围
    {
```

5. 判断指令类型是否为“E”擦除指令：

```
        if(RX2_Buffer[0] == 'E')      //判断指令类型
        {
```

6. 如果是擦除指令的话执行擦除扇区操作，擦除扇区的地址由第 4 步提取：

```
            EEPROM_SectorErase(addr);                  //擦除扇区
            PrintString2("已擦除\xfd 扇区内容!\r\n");
            status = 0; //命令正确
        }
```

7. 判断指令类型是否为“W”写入指令：

```
        else if((RX2_Buffer[0] == 'W') && (RX2_Buffer[10] == ' '))      //判断指令类型
        {
```

8. 获取写入内容的长度，并判断是否超过预设长度上限：

```
j = COM2.RX_Cnt - 11;
if(j > Max_Length)  j = Max_Length; //越界检测
```

9. 执行写入操作，因为有独立的擦除指令，这里就不绑定擦除操作。注意在同样地址如果之前写入过数据，重新写入之前需要执行擦除操作：

```
//EEPROM_SectorErase(addr);                 //擦除扇区
EEPROM_write_n(addr,&RX2_Buffer[11],j);         //写 N 个字节
PrintString2("已写入");
if(j >= 100)      {TX2_write2buff((u8)(j/100+'0'));     j = j % 100;}
if(j >= 10)       {TX2_write2buff((u8)(j/10+'0'));      j = j % 10;}
TX2_write2buff((u8)(j%10+'0'));
PrintString2("字节！\r\n");
status = 0; //命令正确
}
```

10. 判断指令类型是否为“R”读取指令：

```
else if((RX2_Buffer[0] == 'R') && (RX2_Buffer[10] == ' '))     //读取 N 字节 EEPROM 数据
{
```

11. 获取读取内容的长度，并判断是否超过预设长度上限：

```
j = GetDataLength();
if(j > Max_Length)  j = Max_Length; //越界检测
if(j > 0)
{
```

12. 执行读取操作，并打印读取信息：

```
PrintString2("读出");
TX2_write2buff((u8)(j/10+'0'));
TX2_write2buff((u8)(j%10+'0'));
PrintString2("个字节内容如下：\r\n");
EEPROM_read_n(addr,tmp,j);
for(i=0; i<j; i++)  TX2_write2buff(tmp[i]);
TX2_write2buff(0x0d);
TX2_write2buff(0x0a);
status = 0; //命令正确
}
}
}
}
```

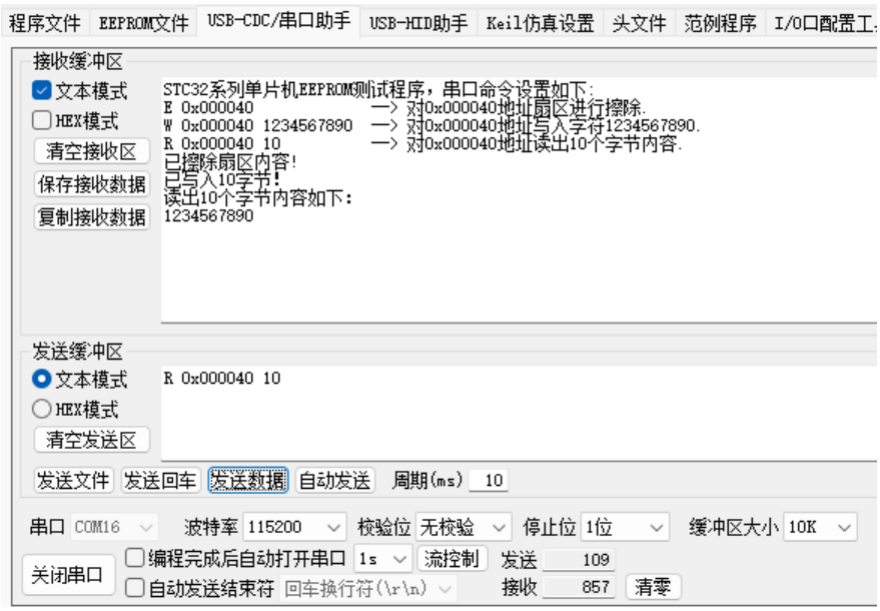
13. 判断状态字节，如果非 0 的话表示命令错误，打印错误信息：

```
if(status != 0) PrintString2("命令错误！\r\n");
```

14. 清除接收字节计数器，下次接收内容从缓冲区的起始位置开始存放：

```
COM2.RX_Cnt = 0;
```

通过串口助手查看结果如下：



5.8 比较器使用

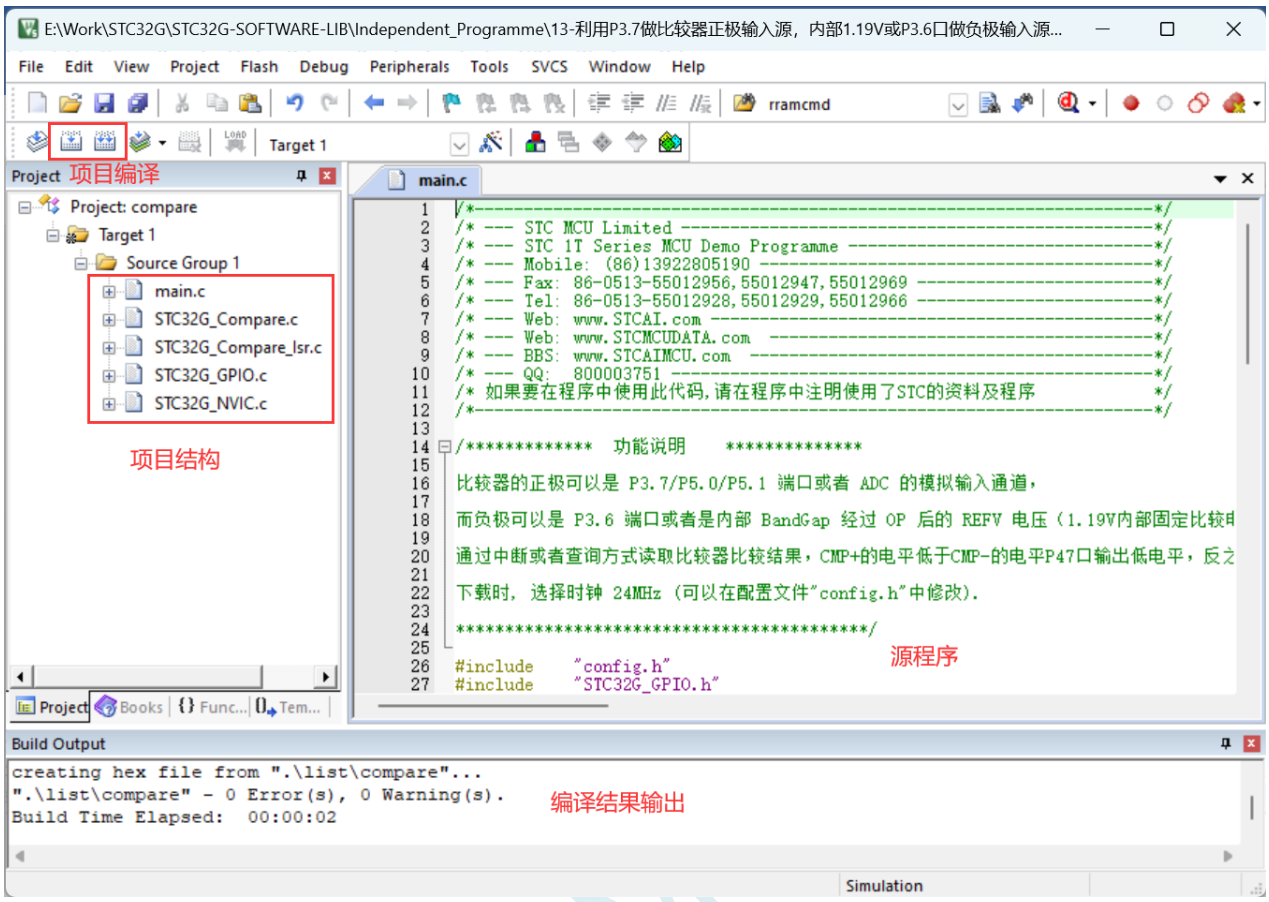
5.8.1 项目文件

打开利用 P3.7 做比较器正极输入源，内部 1.19V 或 P3.6 口做负极输入源例程，其中包含编译文件存放“list”文件夹，其它文件介绍如下:

文件	描述
Config.h	用户配置文件，主要是主时钟定义
compare (.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_Compare (.h .c)	比较器初始化及应用相关函数库
Type_def.h	数据类型定义文件

5.8.2 程序框架

双击“compare.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程是在 STC32G 实验箱上进行验证，比较器的正极可以是 P3.7/P5.0/P5.1 端口或者 ADC 的模拟输入通道，而负极可以是 P3.6 端口或者是内部 BandGap 经过 OP 后的 REFV 电压（1.19V 内部固定比较电压）。通过中断或者查询方式读取比较器比较结果，CMP+的电平低于 CMP-的电平 P47 口输出低电平，反之输出高电平。初始化时对所有用到的 IO 口进行模式配置：

```
void GPIO_config(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;          //结构定义
    //指定要初始化的 IO, GPIO_Pin_0 ~ GPIO_Pin_7, 或操作
    GPIO_InitStructure.Pin  = GPIO_Pin_6 | GPIO_Pin_7;
    //指定 IO 的输入或输出方式,GPIO_PullUp,GPIO_HighZ,GPIO_OUT_OD,GPIO_OUT_PP
    GPIO_InitStructure.Mode = GPIO_HighZ;
    GPIO_Inilize(GPIO_P3,&GPIO_InitStructure);    //初始化 P3.6, P3.7 为高阻输入模式

    //指定要初始化的 IO, GPIO_Pin_0 ~ GPIO_Pin_7, 或操作
    GPIO_InitStructure.Pin  = GPIO_Pin_7;
    //指定 IO 的输入或输出方式,GPIO_PullUp,GPIO_HighZ,GPIO_OUT_OD,GPIO_OUT_PP
    GPIO_InitStructure.Mode = GPIO_PullUp;
    GPIO_Inilize(GPIO_P4,&GPIO_InitStructure);    //初始化 P4.7 为准双向模式
}
```

对比较器进行设置：

```
void CMP_config(void)
{
    CMP_InitDefine CMP_InitStructure;          //结构定义
```

```
    CMP_InitStructure.CMP_EN = ENABLE;          //允许比较器  ENABLE,DISABLE
    //比较器输入正极选择, CMP_P_P37/CMP_P_P50/CMP_P_P51, CMP_P_ADC
    CMP_InitStructure.CMP_P_Select      = CMP_P_P37;
    //比较器输入负极选择,
    //CMP_N_GAP: 选择内部 BandGap 经过 OP 后的电压做负输入,
    //CMP_N_P36: 选择 P3.6 做负输入.
    CMP_InitStructure.CMP_N_Select      = CMP_N_GAP;
    CMP_InitStructure.CMP_InvCMPO       = DISABLE;  //比较器输出取反,  ENABLE,DISABLE
    CMP_InitStructure.CMP_100nsFilter   = ENABLE;   //内部 0.1us 滤波,  ENABLE,DISABLE
    CMP_InitStructure.CMP_Outpt_En      = ENABLE;   //允许比较结果输出,ENABLE,DISABLE
    CMP_InitStructure.CMP_OutDelayDuty = 16;        //比较结果变化延时周期数, 0~63
    CMP_Inilize(&CMP_InitStructure);                //初始化比较器
    NVIC_CMP_Init(RISING_EDGE|FALLING_EDGE,Priority_0);//中断使能,优先级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）：

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();     //扩展 SFR(XFR)访问使能
    CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等：

```
    GPIO_config();
    CMP_config();
    EA = 1;
```

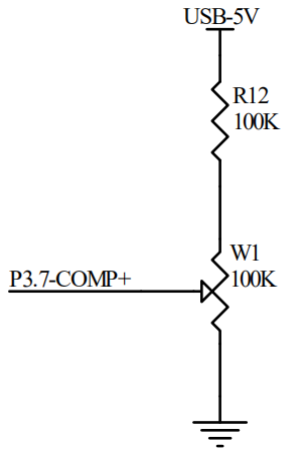
此例程主要验证定时器中断功能，主循环为空。

```
    while (1);
```

在比较器中断里将比较结果输出到 P4.7 口上，实验箱上通过调整 W1 可调电阻的阻值就能通过 P4.7 口连接的 LED10 观察到比较器输出信号产生的翻转：

```
void CMP_ISR_Handler (void) interrupt CMP_VECTOR
{
    CMPIF = 0;              //清除中断标志

    // TODO: 在此处添加用户代码
    P47 = CMPRES;   //中断方式读取比较器比较结果
}
```

W1 可调电阻连接比较器正极 P3.7 口上：



5.9 PWM 输出

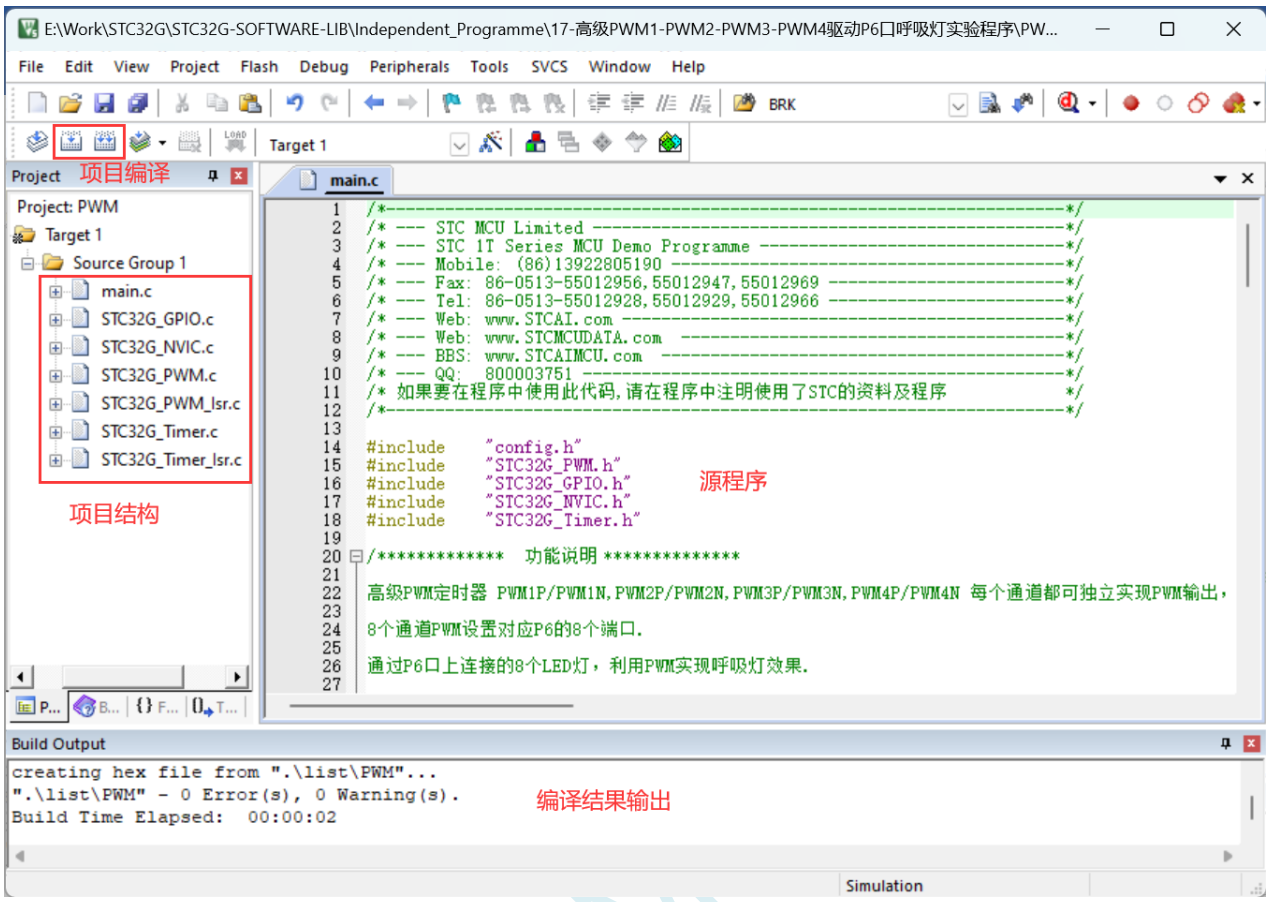
5.9.1 项目文件

打开高级 PWM1-PWM2-PWM3-PWM4 驱动 P6 口呼吸灯实验程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下:

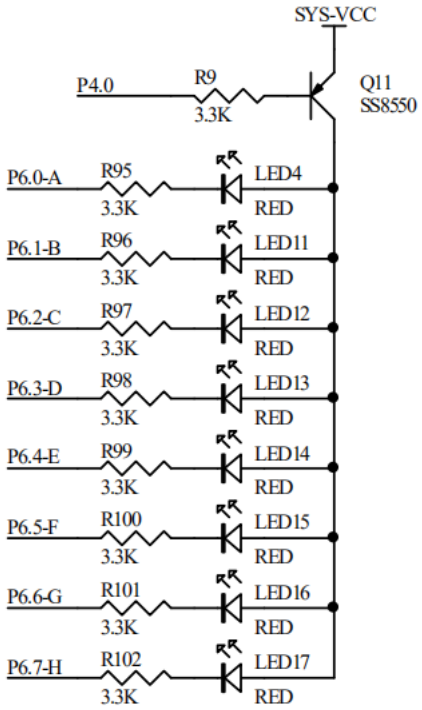
文件	描述
Config.h	用户配置文件，主要是主时钟定义
PWM(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_PWM(.h .c)	PWM 模块初始化相关函数库
STC32G_PWM_Isr.c	PWM 模块中断函数库
STC32G_Timer (.h .c)	定时器模块初始化及应用相关函数库
STC32G_Timer_Isr.c	定时器模块中断函数库
Type_def.h	数据类型定义文件

5.9.2 程序框架

双击“PWM.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在"main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程通过实验箱 P6 口上连接的 8 个 LED 灯，利用 PWM 实现呼吸灯效果。



8 个独立 LED 指示灯实验

初始化时对所有用到的 IO 口进行模式配置:

void GPIO_config(void)

```
{
    P4_MODE_IO_PU(GPIO_Pin_0);      //P4.0 设置为准双向口
//  P6_MODE_IO_PU(GPIO_Pin_All);    //P6 设置为准双向口（启动 PWM 功能后输出脚自动设置为推挽输出模式，所以可以不用设置）
}
```

对定时器进行设置：

```
void UART_config(void)
{
    TIM_InitTypeDef     TIM_InitStructure;      //结构定义
    TIM_InitStructure.TIM_Mode      = TIM_16BitAutoReload;  //16 位自动重载模式
    TIM_InitStructure.TIM_ClkSource = TIM_CLOCK_1T;     //指定 1T 时钟源
    TIM_InitStructure.TIM_ClkOut    = DISABLE;          //不输出高速脉冲
    TIM_InitStructure.TIM_Value     = (u16)(65536UL - (MAIN_Fosc / 1000UL)); //中断频率 1000 次/秒
    TIM_InitStructure.TIM_Run       = ENABLE;           //初始化后启动定时器
    Timer_Inilize(Timer0,&TIM_InitStructure);           //初始化 Timer0
    NVIC_Timer0_Init(ENABLE,Priority_0);    // Timer0 中断使能，优先级 0 级
}
```

对 PWM 进行初始化设置：

```
void PWM_config(void)
{
    PWMx_InitDefine     PWMx_InitStructure;

    PWMA_Duty.PWM1_Duty = 128;      //占空比参数初始化
    PWMA_Duty.PWM2_Duty = 256;
    PWMA_Duty.PWM3_Duty = 512;
    PWMA_Duty.PWM4_Duty = 1024;

    PWMx_InitStructure.PWM_Mode     =   CCMRn_PWM_MODE1; //设置 PWM1 模式
    PWMx_InitStructure.PWM_Duty     = PWMA_Duty.PWM1_Duty;  //PWM 占空比时间, 0~Period
    PWMx_InitStructure.PWM_EnoSelect    = ENO1P | ENO1N;  //输出通道选择
    PWM_Configuration(PWM1, &PWMx_InitStructure);           //初始化 PWM1

    PWMx_InitStructure.PWM_Mode     =   CCMRn_PWM_MODE1;  //设置 PWM1 模式
    PWMx_InitStructure.PWM_Duty     = PWMA_Duty.PWM2_Duty;   //PWM 占空比时间, 0~Period
    PWMx_InitStructure.PWM_EnoSelect    = ENO2P | ENO2N;  //输出通道选择
    PWM_Configuration(PWM2, &PWMx_InitStructure);           //初始化 PWM2

    PWMx_InitStructure.PWM_Mode     =   CCMRn_PWM_MODE1;  //设置 PWM1 模式
    PWMx_InitStructure.PWM_Duty     = PWMA_Duty.PWM3_Duty;   //PWM 占空比时间, 0~Period
    PWMx_InitStructure.PWM_EnoSelect    = ENO3P | ENO3N;  //输出通道选择
    PWM_Configuration(PWM3, &PWMx_InitStructure);           //初始化 PWM3

    PWMx_InitStructure.PWM_Mode     =   CCMRn_PWM_MODE1;  //设置 PWM1 模式
    PWMx_InitStructure.PWM_Duty     = PWMA_Duty.PWM4_Duty;   //PWM 占空比时间, 0~Period
```

```
    PWMx_InitStructure.PWM_EnoSelect    = ENO4P | ENO4N;  //输出通道选择
    PWM_Configuration(PWM4, &PWMx_InitStructure);         //初始化 PWM4

    PWMx_InitStructure.PWM_Period    = 2047;              //周期时间,   0~65535
    PWMx_InitStructure.PWM_DeadTime = 0;                  //死区发生器设置, 0~255
    PWMx_InitStructure.PWM_MainOutEnable= ENABLE;         //主输出使能, ENABLE,DISABLE
    PWMx_InitStructure.PWM_CEN_Enable    = ENABLE;        //使能计数器, ENABLE,DISABLE
    PWM_Configuration(PWMA, &PWMx_InitStructure);         //初始化 PWMA 通用寄存器

    PWM1_USE_P60P61();      //PWM1 选择通道 P60, P61
    PWM2_USE_P62P63();      //PWM2 选择通道 P62, P63
    PWM3_USE_P64P65();      //PWM3 选择通道 P64, P65
    PWM4_USE_P66P67();      //PWM4 选择通道 P66, P67

    NVIC_PWM_Init(PWMA,DISABLE,Priority_0);    // PWM 中断关闭, 优先级 0 级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）：

```
    WTST = 0;  //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();     //扩展 SFR(XFR)访问使能
    CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等：

```
    GPIO_config();
    Timer_config();
    PWM_config();
    EA = 1;
    P40 = 0;      //打开 LED 电源
```

主循环里判断定时器 1ms 标志位，每 1 毫秒改变一次 PWM 通道占空比值，增加到上限后开始减小，减小到下限后开始增加。从而达到呼吸灯效果。

```
    while (1)
    {
        if(B_1ms)     //判断定时器 1ms 中断产生标志
        {
            B_1ms = 0;   //1ms 时间到，清标志位

            if(!PWM1_Flag)    //判断 PWM1 通道占空比是增加还是减小状态
            {
                PWMA_Duty.PWM1_Duty++;    //PWM1 占空比加 1
                //如果 PWM1 占空比达到增加上限，则改变为减小状态
                if(PWMA_Duty.PWM1_Duty >= 2047) PWM1_Flag = 1;
            }
            else
```

```
{
    PWMA_Duty.PWM1_Duty--;      //PWM1 占空比减 1
    //如果 PWM1 占空比达到减小下限，则改变为增加状态
    if(PWMA_Duty.PWM1_Duty <= 0) PWM1_Flag = 0;
}

if(!PWM2_Flag)     //判断 PWM2 通道占空比是增加还是减小状态
{
    PWMA_Duty.PWM2_Duty++;     //PWM2 占空比加 1
    //如果 PWM2 占空比达到增加上限，则改变为减小状态
    if(PWMA_Duty.PWM2_Duty >= 2047) PWM2_Flag = 1;
}
else
{
    PWMA_Duty.PWM2_Duty--;      //PWM2 占空比减 1
    //如果 PWM2 占空比达到减小下限，则改变为增加状态
    if(PWMA_Duty.PWM2_Duty <= 0) PWM2_Flag = 0;
}

if(!PWM3_Flag)     //判断 PWM3 通道占空比是增加还是减小状态
{
    PWMA_Duty.PWM3_Duty++;     //PWM3 占空比加 1
    //如果 PWM3 占空比达到增加上限，则改变为减小状态
    if(PWMA_Duty.PWM3_Duty >= 2047) PWM3_Flag = 1;
}
else
{
    PWMA_Duty.PWM3_Duty--;      //PWM3 占空比减 1
    //如果 PWM3 占空比达到减小下限，则改变为增加状态
    if(PWMA_Duty.PWM3_Duty <= 0) PWM3_Flag = 0;
}

if(!PWM4_Flag)     //判断 PWM4 通道占空比是增加还是减小状态
{
    PWMA_Duty.PWM4_Duty++;     //PWM4 占空比加 1
    //如果 PWM4 占空比达到增加上限，则改变为减小状态
    if(PWMA_Duty.PWM4_Duty >= 2047) PWM4_Flag = 1;
}
else
{
    PWMA_Duty.PWM4_Duty--;      //PWM4 占空比减 1
    //如果 PWM4 占空比达到减小下限，则改变为增加状态
    if(PWMA_Duty.PWM4_Duty <= 0) PWM4_Flag = 0;
}
```

```
            UpdatePwm(PWMA, &PWMA_Duty);        //更新 PWM 占空比
        }
    }
```

5.10 高速 PWM 输出

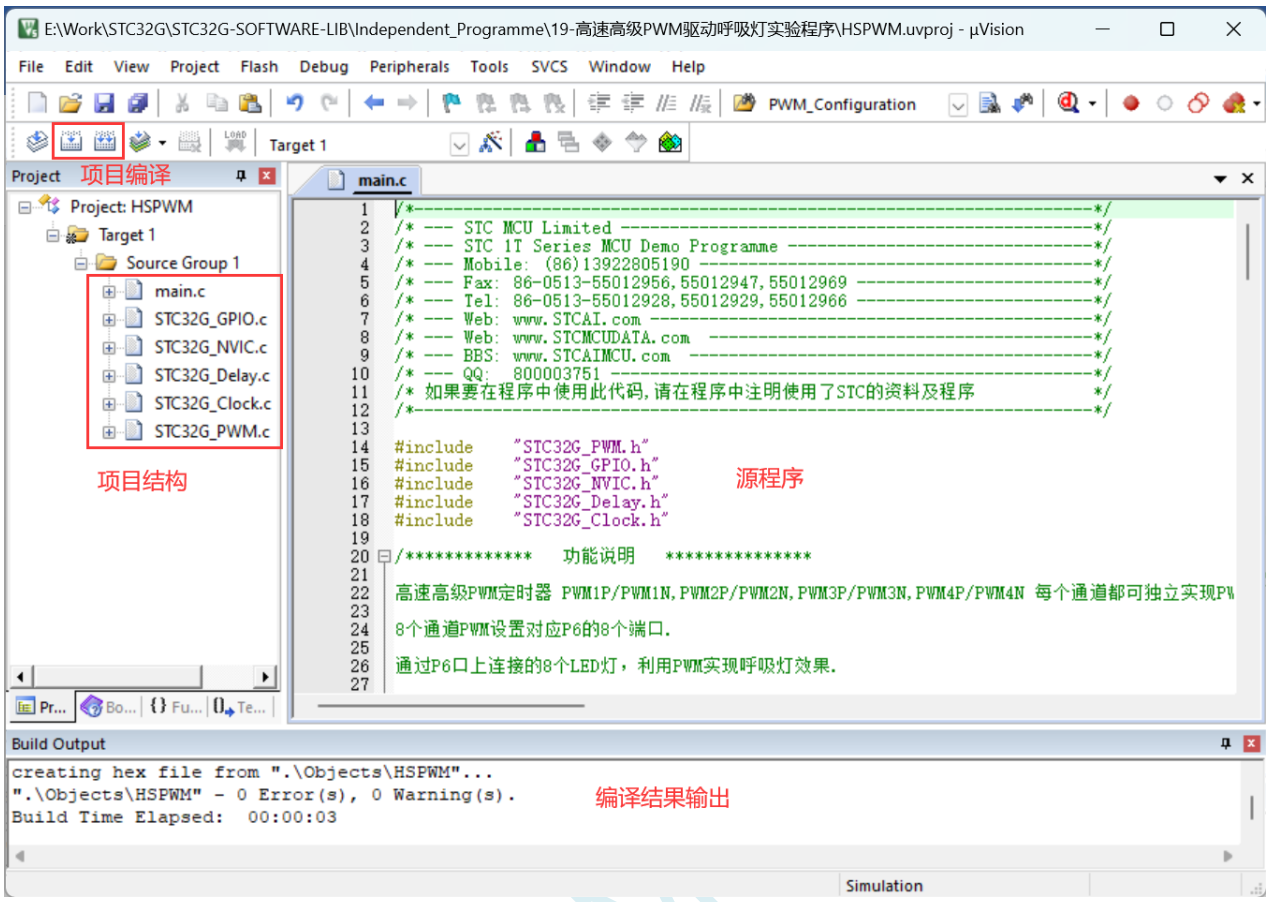
5.10.1 项目文件

打开高速高级 PWM 驱动呼吸灯实验程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

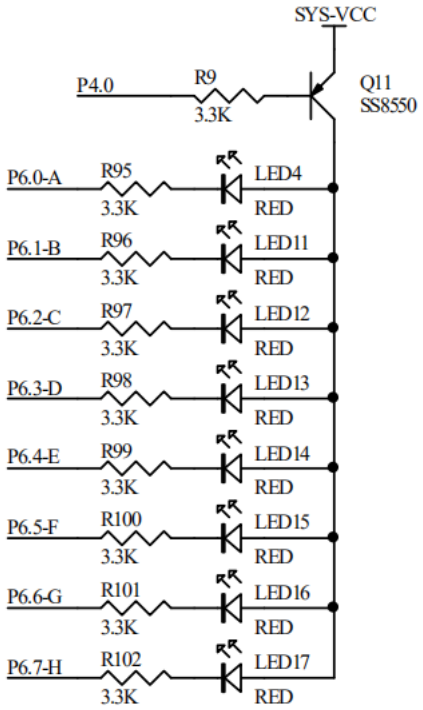
文件	描述
Config.h	用户配置文件，主要是主时钟定义
HSPWM(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_Clock (.h .c)	系统时钟初始化相关函数库
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_PWM(.h .c)	PWM 模块初始化相关函数库
STC32G_PWM_Isr.c	PWM 模块中断函数库
Type_def.h	数据类型定义文件

5.10.2 程序框架

双击“HSPWM.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程通过实验箱 P6 口上连接的 8 个 LED 灯，利用 PWM 实现呼吸灯效果。



8 个独立 LED 指示灯实验

初始化时对所有用到的 IO 口进行模式配置:

void GPIO_config(void)

```
{
    PWM1_USE_P60P61();      //PWM1 选择通道 P60, P61
    PWM2_USE_P62P63();      //PWM2 选择通道 P62, P63
    PWM3_USE_P64P65();      //PWM3 选择通道 P64, P65
    PWM4_USE_P66P67();      //PWM4 选择通道 P66, P67

    PWM5_USE_P74();         //PWM5 选择通道 P74
    PWM6_USE_P75();         //PWM6 选择通道 P75
    PWM7_USE_P76();         //PWM7 选择通道 P76
    PWM8_USE_P77();         //PWM8 选择通道 P77

    P4_MODE_IO_PU(GPIO_Pin_0);      //P4.0 设置为准双向口
    P40 = 0;        //打开实验箱 LED 电源
}
```

对高速 PWM 进行初始化设置：

```
void HSPWM_config(void)
{
    HSPWMx_InitDefine       PWMx_InitStructure;

    PWMx_InitStructure.PWM_EnoSelect=
ENO1P|ENO1N|ENO2P|ENO2N|ENO3P|ENO3N|ENO4P|ENO4N;  //输出通道选择
    PWMx_InitStructure.PWM_Period    = 2047;        //周期时间,    0~65535
    PWMx_InitStructure.PWM_DeadTime = 0;            //死区发生器设置, 0~255
    PWMx_InitStructure.PWM_MainOutEnable= ENABLE;   //主输出使能, ENABLE,DISABLE
    PWMx_InitStructure.PWM_CEN_Enable    = ENABLE;  //使能计数器, ENABLE,DISABLE
    HSPWM_Configuration(PWMA, &PWMx_InitStructure, &PWMA_Duty);//初始化 PWMA 通用寄存器
    PWMx_InitStructure.PWM_EnoSelect= ENO5P|ENO6P|ENO7P|ENO8P; //输出通道选择
    HSPWM_Configuration(PWMB, &PWMx_InitStructure, &PWMB_Duty); //初始化 PWMB 通用寄存器

    HSPllClkConfig(MCLKSEL_HIRC,PLL_96M,0);     //系统时钟选择,PLL 时钟选择,时钟分频系数
    NVIC_PWM_Init(PWMA,DISABLE,Priority_0);
    NVIC_PWM_Init(PWMB,DISABLE,Priority_0);

    NVIC_PWM_Init(PWMA,DISABLE,Priority_0);    // PWM 中断关闭, 优先级 0 级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）：

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();       //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值等：

```
    GPIO_config();
```

```
    HSPWM_config();
    EA = 1;

    PWMA_Duty.PWM1_Duty = 128;      //PWMA 占空比参数初始化
    PWMA_Duty.PWM2_Duty = 256;
    PWMA_Duty.PWM3_Duty = 512;
    PWMA_Duty.PWM4_Duty = 1024;

    PWMB_Duty.PWM5_Duty = 128;      //PWMB 占空比参数初始化
    PWMB_Duty.PWM6_Duty = 256;
    PWMB_Duty.PWM7_Duty = 512;
    PWMB_Duty.PWM8_Duty = 1024;
```

主循环里每 1 毫秒改变一次 PWM 通道占空比值，增加到上限后开始减小，减小到下限后开始增加。从而达到呼吸灯效果。

```
    while (1)
    {
        delay_ms(1);        //主循环延时 1ms 执行 1 次

        if(!PWM1_Flag)    //判断 PWM1 通道占空比是增加还是减小状态
        {
            PWMA_Duty.PWM1_Duty++;    //PWM1 占空比加 1
            //如果 PWM1 占空比达到增加上限，则改变为减小状态
            if(PWMA_Duty.PWM1_Duty >= 2047) PWM1_Flag = 1;
        }
        else
        {
            PWMA_Duty.PWM1_Duty--;    //PWM1 占空比减 1
            //如果 PWM1 占空比达到减小下限，则改变为增加状态
            if(PWMA_Duty.PWM1_Duty <= 0) PWM1_Flag = 0;
        }
        if(!PWM2_Flag)    //判断 PWM2 通道占空比是增加还是减小状态
        {
            PWMA_Duty.PWM2_Duty++;    //PWM2 占空比加 1
            //如果 PWM2 占空比达到增加上限，则改变为减小状态
            if(PWMA_Duty.PWM2_Duty >= 2047) PWM2_Flag = 1;
        }
        else
        {
            PWMA_Duty.PWM2_Duty--;    //PWM2 占空比减 1
            //如果 PWM2 占空比达到减小下限，则改变为增加状态
            if(PWMA_Duty.PWM2_Duty <= 0) PWM2_Flag = 0;
        }
        if(!PWM3_Flag)    //判断 PWM3 通道占空比是增加还是减小状态
        {
```

```
        PWMA_Duty.PWM3_Duty++;      //PWM3 占空比加 1
        //如果 PWM3 占空比达到增加上限，则改变为减小状态
        if(PWMA_Duty.PWM3_Duty >= 2047) PWM3_Flag = 1;
    }
    else
    {
        PWMA_Duty.PWM3_Duty--;      //PWM3 占空比减 1
        //如果 PWM3 占空比达到减小下限，则改变为增加状态
        if(PWMA_Duty.PWM3_Duty <= 0) PWM3_Flag = 0;
    }
    if(!PWM4_Flag)     //判断 PWM4 通道占空比是增加还是减小状态
    {
        PWMA_Duty.PWM4_Duty++;      //PWM4 占空比加 1
        //如果 PWM4 占空比达到增加上限，则改变为减小状态
        if(PWMA_Duty.PWM4_Duty >= 2047) PWM4_Flag = 1;
    }
    else
    {
        PWMA_Duty.PWM4_Duty--;      //PWM4 占空比减 1
        //如果 PWM4 占空比达到减小下限，则改变为增加状态
        if(PWMA_Duty.PWM4_Duty <= 0) PWM4_Flag = 0;
    }

    if(!PWM5_Flag)     //判断 PWM5 通道占空比是增加还是减小状态
    {
        PWMB_Duty.PWM5_Duty++;      //PWM5 占空比加 1
        //如果 PWM5 占空比达到增加上限，则改变为减小状态
        if(PWMB_Duty.PWM5_Duty >= 2047) PWM5_Flag = 1;
    }
    else
    {
        PWMB_Duty.PWM5_Duty--;      //PWM5 占空比减 1
        //如果 PWM5 占空比达到减小下限，则改变为增加状态
        if(PWMB_Duty.PWM5_Duty <= 0) PWM5_Flag = 0;
    }

    if(!PWM6_Flag)     //判断 PWM6 通道占空比是增加还是减小状态
    {
        PWMB_Duty.PWM6_Duty++;      //PWM6 占空比加 1
        //如果 PWM6 占空比达到增加上限，则改变为减小状态
        if(PWMB_Duty.PWM6_Duty >= 2047) PWM6_Flag = 1;
    }
    else
    {
        PWMB_Duty.PWM6_Duty--;      //PWM6 占空比减 1
```

```
            //如果 PWM6 占空比达到减小下限，则改变为增加状态
            if(PWMB_Duty.PWM6_Duty <= 0) PWM6_Flag = 0;
        }

        if(!PWM7_Flag)    //判断 PWM7 通道占空比是增加还是减小状态
        {
            PWMB_Duty.PWM7_Duty++;    //PWM7 占空比加 1
            //如果 PWM7 占空比达到增加上限，则改变为减小状态
            if(PWMB_Duty.PWM7_Duty >= 2047) PWM7_Flag = 1;
        }
        else
        {
            PWMB_Duty.PWM7_Duty--;    //PWM7 占空比减 1
            //如果 PWM7 占空比达到减小下限，则改变为增加状态
            if(PWMB_Duty.PWM7_Duty <= 0) PWM7_Flag = 0;
        }

        if(!PWM8_Flag)    //判断 PWM8 通道占空比是增加还是减小状态
        {
            PWMB_Duty.PWM8_Duty++;    //PWM8 占空比加 1
            //如果 PWM8 占空比达到增加上限，则改变为减小状态
            if(PWMB_Duty.PWM8_Duty >= 2047) PWM8_Flag = 1;
        }
        else
        {
            PWMB_Duty.PWM8_Duty--;    //PWM8 占空比减 1
            //如果 PWM8 占空比达到减小下限，则改变为增加状态
            if(PWMB_Duty.PWM8_Duty <= 0) PWM8_Flag = 0;
        }

        UpdateHSPwm(PWMA, &PWMA_Duty);        //更新 PWMA 占空比
        UpdateHSPwm(PWMB, &PWMB_Duty);        //更新 PWMB 占空比
    }
```

5.11 SPI 主从收发

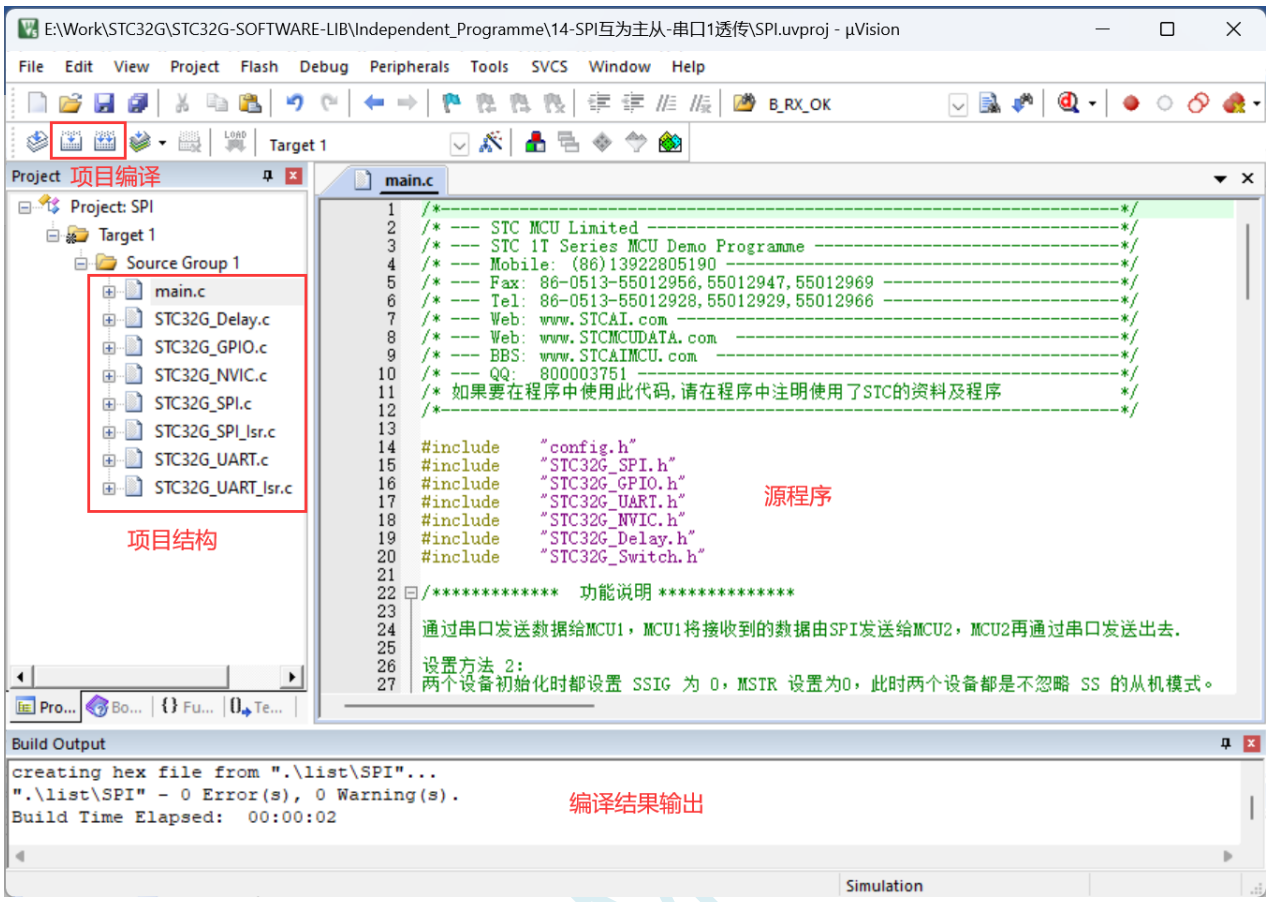
5.11.1 项目文件

打开 SPI 互为主从-串口 1 透传例程，其中包含编译文件存放“list”文件夹，其它文件介绍如下:

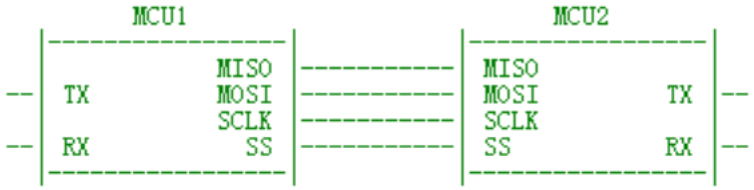
文件	描述
Config.h	用户配置文件，主要是主时钟定义
SPI(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_SPI(.h .c)	SPI 模块初始化相关函数库
STC32G_SPI_Isr.c	SPI 模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.11.2 程序框架

双击“SPI.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程通过串口发送数据给 MCU1，MCU1 将接收到的数据由 SPI 发送给 MCU2，MCU2 再通过串口发送出去。



初始化时对所有用到的 IO 口进行模式配置：

```
void GPIO_config(void)
{
    P2_MODE_IO_PU(GPIO_Pin_All);     //P2 设置为准双向口
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);     //P4.6,P4.7 设置为准双向口
}
```

对串口进行设置：

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;     //结构定义
    //模式,   UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode      = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
//  COMx_InitStructure.UART_BRT_Use   = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate  = 115200ul; //波特率,     110 ~ 115200
```

```
    COMx_InitStructure.UART_RxEnable    = ENABLE;   //接收允许,    ENABLE 或 DISABLE
    //初始化串口 2 UART1,UART2,UART3,UART4
    UART_Configuration(UART2, &COMx_InitStructure);
    NVIC_UART2_Init(ENABLE,Priority_1);       //中断使能,优先级

    UART2_SW(UART2_SW_P46_P47);           //通道选择，UART2_SW_P10_P11,UART2_SW_P46_P47
}
```

对 SPI 接口进行设置：

```
void SPI_config(void)
{
    SPI_InitTypeDef           SPI_InitStructure;
    SPI_InitStructure.SPI_Enable      = ENABLE;     //SPI 启动      ENABLE, DISABLE
    SPI_InitStructure.SPI_SSIG        = DISABLE;    //片选位        ENABLE, DISABLE
    SPI_InitStructure.SPI_FirstBit    = SPI_MSB;    //移位方向      SPI_MSB, SPI_LSB
    //主从选择  SPI_Mode_Master, SPI_Mode_Slave
    SPI_InitStructure.SPI_Mode        = SPI_Mode_Slave;
    //时钟相位     SPI_CPOL_High,     SPI_CPOL_Low
    SPI_InitStructure.SPI_CPOL        = SPI_CPOL_Low;
    //数据边沿     SPI_CPHA_1Edge,    SPI_CPHA_2Edge
    SPI_InitStructure.SPI_CPHA        = SPI_CPHA_2Edge;
    //SPI 速度      SPI_Speed_4, SPI_Speed_8, SPI_Speed_16, SPI_Speed_2
    SPI_InitStructure.SPI_Speed       = SPI_Speed_4;
    SPI_Init(&SPI_InitStructure);
    NVIC_SPI_Init(ENABLE,Priority_3);     //中断使能,优先级
    //SPI_P54_P13_P14_P15,SPI_P22_P23_P24_P25,SPI_P54_P40_P41_P43,SPI_P35_P34_P33_P32
    SPI_SW(SPI_P22_P23_P24_P25);  //通道选择
    SPI_SS_2 = 1;
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）：

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();       //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等：

```
    GPIO_config();
    UART_config();
    SPI_config();
    EA = 1;
```

通过串口打印一串数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此判断程序运行情况。

```
    printf("STC32G UART2 与 SPI 透传程序\r\n");  //UART 发送一个字符串
```

主循环里接收到 UART 数据后通过 SPI 接口发送出去，接收到 SPI 数据后通过 UART 接口发送出去。

```
    while (1)
    {
```

1. 延时 1ms，这样主循环的内容 1ms 执行一次，方便计时：

```
        delay_ms(1);
```

2. 判断串口 2 是否有收到数据，如果收到一串数据就设置串口接收标志（UartReceived）：

```
        if(COM2.RX_TimeOut > 0)
        {
            if(--COM2.RX_TimeOut == 0)
            {
                if(COM2.RX_Cnt > 0)
                {
                    UartReceived = 1;       //设置串口接收标志
                }
            }
        }
```

3. 判断是否有收到串口数据，以及 SPI 是否空闲（SS 脚位是否拉高）。是的话通过 SPI 接收发送数据：

```
        if((UartReceived) && (SPI_SS_2))
        {
            SPI_SS_2 = 0;       //拉低从机 SS 管脚
            SPI_SetMode(SPI_Mode_Master);     //SPI 设置主机模式，开始发送数据
            for(i=0;i<COM2.RX_Cnt;i++)
            {
                SPI_WriteByte(RX2_Buffer[i]);     //SPI 接口输出串口接收数据
            }
            SPI_SS_2 = 1;      //拉高从机的 SS 管脚
            SPI_SetMode(SPI_Mode_Slave);       //SPI 设置从机模式，进入接收状态
            COM2.RX_Cnt = 0;
            UartReceived = 0;
        }
```

4. 判断 SPI 接口是否收到数据，是的话通过串口发送出来：

```
        if(SPI_RxTimerOut > 0) //串口中断收到数据就将 SPI 超时计数器置位
        {
            if(--SPI_RxTimerOut == 0)  //SPI 超时计数器每 1ms 减 1，并判断是否为 0
            {
                if(SPI_RxCnt > 0)   //SPI 接收超时后判断 SPI 接收数据长度是否非零
                {
                    for(i=0; i<SPI_RxCnt; i++)    TX2_write2buff(SPI_RxBuffer[i]);  //通过串口输出数据
                }
                SPI_RxCnt = 0;      //清除 SPI 已接收数据长度
            }
        }
    }
```

5.12 高速 SPI 访问 Flash 芯片

STC32G 系列单片机为 SPI 接口提供了高速模式 (HSSPI)。高速 SPI 以普通 SPI 为基础，增加了高速模式。当系统运行在较低工作频率时，高速 SPI 可工作在高达 144MHz 的频率下，从而可达到降低内核功耗，提升外设性能的目的。

5.12.1 项目文件

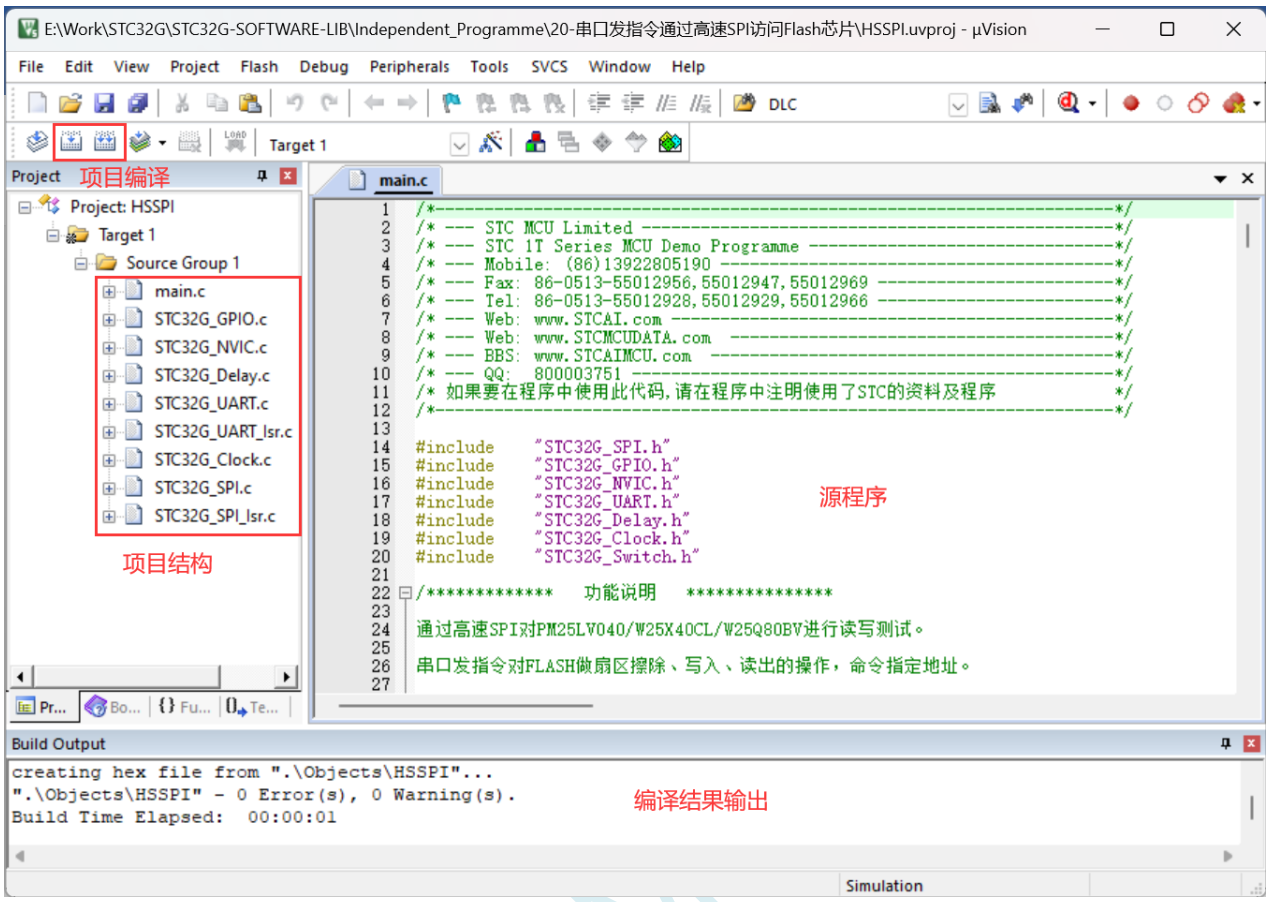
打开串口发指令通过高速 SPI 访问 Flash 芯片例程，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
HSSPI(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_Clock (.h .c)	系统时钟初始化相关函数库
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_SPI(.h .c)	SPI 模块初始化相关函数库
STC32G_SPI_Isr.c	SPI 模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.12.2 程序框架

双击“HSSPI.uvproj”使用 Keil 编译器打开项目。

在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程通过串口发指令给 MCU，对 FLASH 做扇区擦除、写入、读出的操作。
串口命令设置: (字母不区分大小写)
E 0x001234 --> 扇区擦除，指定十六进制地址.
W 0x001234 1234567890 --> 写入操作，指定十六进制地址，后面为写入内容.
R 0x001234 10 --> 读出操作，指定十六进制地址，后面为读出字节.
C --> 如果检测不到 PM25LV040/W25X40CL/W25Q80BV，发送 C 强制允许操作.

初始化时对所有用到的 IO 口进行模式配置:

```
void GPIO_config(void)
{
    P2_MODE_IO_PU(GPIO_Pin_All);        //P2 设置为准双向口
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);        //P4.6,P4.7 设置为准双向口
    //电平转换速度快（提高 IO 口翻转速度）
    P2_SPEED_HIGH(GPIO_Pin_2 | GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5);

    //SPI_P54_P13_P14_P15,SPI_P22_P23_P24_P25,SPI_P54_P40_P41_P43,SPI_P35_P34_P33_P32
    SPI_SW(SPI_P22_P23_P24_P25);
    UART2_SW(UART2_SW_P46_P47);     //UART2_SW_P10_P11,UART2_SW_P46_P47

    SPI_SCK = 0;      //设置默认 SPI CLK 脚电平为低
    SPI_SI = 1;       //设置默认 SPI MOSI 脚电平为高
    SPI_CE = 1;       //设置默认 SPI SS 脚电平为高
```

```
}
```

对 SPI 接口进行设置:

```
void SPI_config(void)
{
    SPI_InitTypeDef         SPI_InitStructure;
    SPI_InitStructure.SPI_Enable    = ENABLE;    //SPI 启动    ENABLE, DISABLE
    SPI_InitStructure.SPI_SSIG      = ENABLE;    //片选位      ENABLE, DISABLE
    SPI_InitStructure.SPI_FirstBit  = SPI_MSB;   //移位方向    SPI_MSB, SPI_LSB
    //主从选择  SPI_Mode_Master, SPI_Mode_Slave
    SPI_InitStructure.SPI_Mode      = SPI_Mode_Slave;
    //时钟相位    SPI_CPOL_High,    SPI_CPOL_Low
    SPI_InitStructure.SPI_CPOL      = SPI_CPOL_High;
    //数据边沿    SPI_CPHA_1Edge,   SPI_CPHA_2Edge
    SPI_InitStructure.SPI_CPHA      = SPI_CPHA_2Edge;
    //SPI 速度    SPI_Speed_4, SPI_Speed_8, SPI_Speed_16, SPI_Speed_2
    SPI_InitStructure.SPI_Speed     = SPI_Speed_4;
    SPI_Init(&SPI_InitStructure);
    NVIC_SPI_Init(DISABLE,Priority_3);    //中断使能,优先级
    SPI_ClearFlag();    //清除  SPIF 和 WCOL  标志

    HSPllClkConfig(MCLKSEL_HIRC,PLL_96M,4);    //系统时钟选择,PLL 时钟选择,时钟分频系数
    HSSPI_Enable();    //使能 SPI 高速模式
}
```

对串口进行设置:

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;     //结构定义
    //模式,    UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode      = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
//  COMx_InitStructure.UART_BRT_Use   = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate  = 115200ul; //波特率,      110 ~ 115200
    COMx_InitStructure.UART_RxEnable  = ENABLE;   //接收允许,    ENABLE 或 DISABLE
    //初始化串口 2 UART1,UART2,UART3,UART4
    UART_Configuration(UART2, &COMx_InitStructure);
    NVIC_UART2_Init(ENABLE,Priority_1);     //中断使能,优先级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();     //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

```
GPIO_config();
UART_config();
SPI_config();
EA = 1;
```

通过串口打印命令提示数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此了解如何发送指令对 SPI Flash 进行擦除、写入、读取等操作。

```
PrintString2("命令设置:\r\n");
PrintString2("E 0x001234                --> 扇区擦除\xfd  十六进制地址\r\n");
PrintString2("W 0x001234 1234567890 --> 写入操作  十六进制地址  写入内容\r\n");
PrintString2("R 0x001234 10             --> 读出操作  十六进制地址  读出字节\r\n");
PrintString2("C                         --> 如果检测不到 PM25LV040/W25X40CL/W25Q80BV，发送 C 强制允许操作.\r\n\r\n");
```

读取 Flash 芯片 ID 号，判断 Flash 芯片型号，并将读取结果通过串口打印出来。

```
FlashCheckID();
FlashCheckID();
if(!B_FlashOK)  PrintString2("未检测到 PM25LV040/W25X40CL/W25Q80BV!\r\n");
else
{
        if(B_FlashOK == 1)
        {
                PrintString2("检测到 PM25LV040!\r\n");
        }
        else if(B_FlashOK == 2)
        {
                PrintString2("检测到 W25X40CL!\r\n");
        }
        else if(B_FlashOK == 3)
        {
                PrintString2("检测到 W25Q80BV!\r\n");
        }
        PrintString2("制造商 ID1 = 0x");
        TX2_write2buff(Hex2Ascii(FLASH_ID1 >> 4));
        TX2_write2buff(Hex2Ascii(FLASH_ID1));
        PrintString2("\r\n          ID2 = 0x");
        TX2_write2buff(Hex2Ascii(FLASH_ID2 >> 4));
        TX2_write2buff(Hex2Ascii(FLASH_ID2));
        PrintString2("\r\n     设备 ID = 0x");
        TX2_write2buff(Hex2Ascii(FLASH_ID >> 4));
        TX2_write2buff(Hex2Ascii(FLASH_ID));
        PrintString2("\r\n");
}
```

主循环里通过软件超时机制，判断接收到一串 UART 数据后调用命令解析函数，判断命令类型，执行相应的操作。

```
    while (1)
    {
        delay_ms(1);
        if(COM2.RX_TimeOut > 0)
        {
            if(--COM2.RX_TimeOut == 0)   //超时,则串口接收结束
            {
                if(COM2.RX_Cnt > 0)
                {
                    RX2_Check();      //串口数据处理
                }
                COM2.RX_Cnt = 0;
            }
        }
    }
```

串口接收数据解析函数分析：

```
void RX2_Check(void)
{
    u8   i,j;
    u8   tmp[EE_BUF_LENGTH];
```

1. 如果串口只接收到 1 个字节数据，并且数据为“C”的话，则允许强制操作 Flash

```
    if((COM2.RX_Cnt == 1) && (RX2_Buffer[0] == 'C'))      //发送 C 强制允许操作
    {
        B_FlashOK = 1;
        PrintString2("强制允许操作 FLASH!\r\n");
    }
```

2. 判断是否允许操作 Flash (读取 Flash ID 有效，或者收到强制操作指令“C”)

```
    if(!B_FlashOK)
    {
        PrintString2("PM25LV040/W25X40CL/W25Q80BV 不存在，不能操作 FLASH!\r\n");
        return;
    }
```

3. 判断接收数据是否符合指令格式

```
    F0 = 0;
    if((COM2.RX_Cnt >= 10) && (RX2_Buffer[1] == ' '))    //最短命令为 10 个字节
    {
//              printf("收到内容如下:   ");
//              for(i=0; i<COM2.RX_Cnt; i++)      printf("%c", RX2_Buffer[i]);      //把收到的数据原样返回，用于测试
//              printf("\r\n");
```

4. 转换指令数据为大写字符，方便后续判断

```
    for(i=0; i<10; i++)
    {
        //小写转大写
        if((RX2_Buffer[i] >= 'a') && (RX2_Buffer[i] <= 'z'))      RX2_Buffer[i] = RX2_Buffer[i] - 'a' + 'A';
    }
```

5. 读取地址位数据，判断地址是否有效

```
    Flash_addr = GetAddress();
    if(Flash_addr < 0x80000000)
    {
```

6. 判断是否为擦除指令，是的话执行擦除操作

```
        if(RX2_Buffer[0] == 'E')      //擦除
        {
            FlashSectorErase(Flash_addr);
            PrintString2("已擦除\xfd 一个扇区内容!\r\n");
            F0 = 1;
        }
```

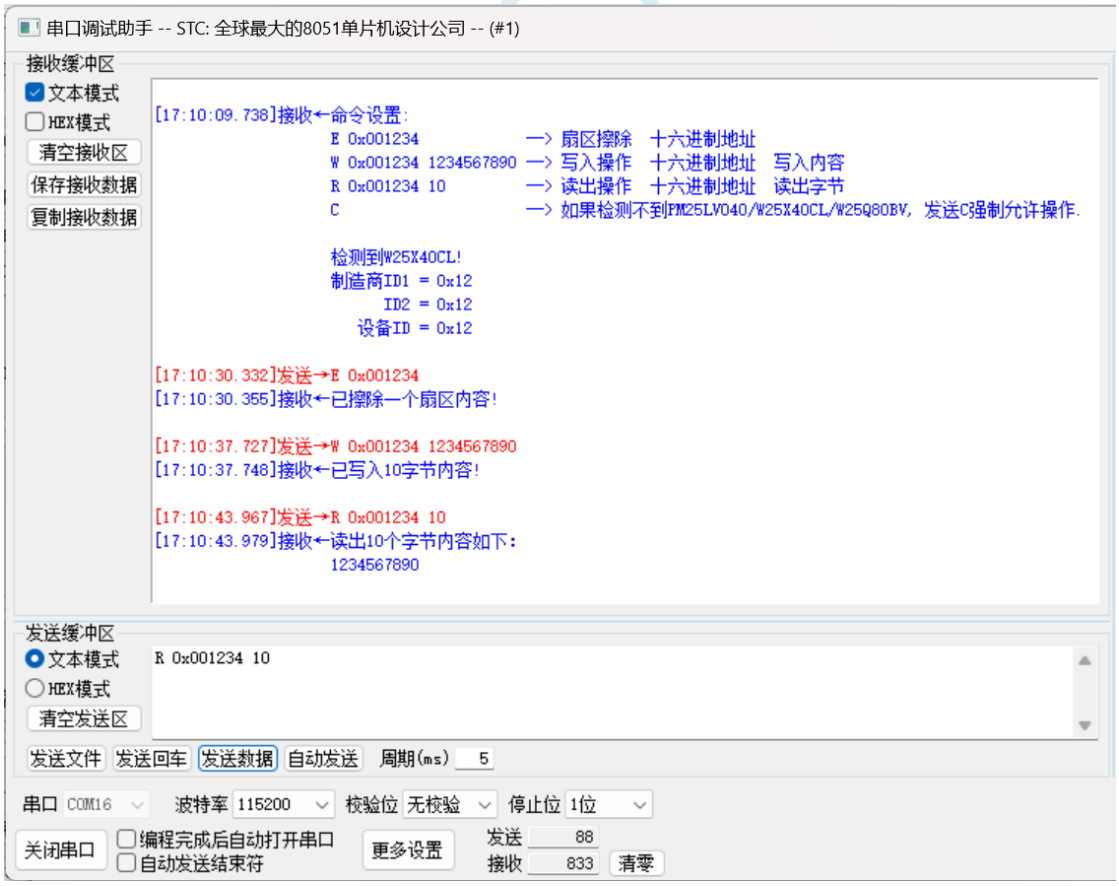
7. 判断是否为写入指令，是写入指令则先判断写入地址空间是否为空，非空的话串口打印提示信息，写入地址为空的话写入数据，然后读出来进行对比，如果与要写入的数据内容一致的话说明写入成功，串口打印已写入 N 字节内容，不一致的话串口打印写入错误

```
        else if((RX2_Buffer[0] == 'W') && (COM2.RX_Cnt >= 12) && (RX2_Buffer[10] == ' '))
        {
            j = COM2.RX_Cnt - 11;
            for(i=0; i<j; i++)    tmp[i] = 0xff;          //检测要写入的空间是否为空
            i = SPI_Read_Compare(Flash_addr,tmp,j);
            if(i > 0)
            {
                PrintString2("要写入的地址为非空,不能写入,请先擦除\xfd!\r\n");
            }
            else
            {
                SPI_Write_Nbytes(Flash_addr,&RX2_Buffer[11],j);        //写 N 个字节
                i = SPI_Read_Compare(Flash_addr,&RX2_Buffer[11],j); //比较写入的数据
                if(i == 0)
                {
                    PrintString2("已写入");
                    if(j >= 100)      {TX2_write2buff((u8)(j/100+'0'));     j = j % 100;}
                    if(j >= 10)       {TX2_write2buff((u8)(j/10+'0'));      j = j % 10;}
                    TX2_write2buff((u8)(j%10+'0'));
                    PrintString2("字节内容!\r\n");
                }
                else            PrintString2("写入错误!\r\n");
            }
            F0 = 1;
        }
```

8. 判断是否为读取指令，是的话读取对应地址内容，并通过串口打印出来

```
            else if((RX2_Buffer[0] == 'R') && (COM2.RX_Cnt >= 12) && (RX2_Buffer[10] == ' '))
            {
                j = GetDataLength();
                if((j > 0) && (j < EE_BUF_LENGTH))
                {
                    SPI_Read_Nbytes(Flash_addr,tmp,j);
                    PrintString2("读出");
                    if(j>=100)    TX2_write2buff((u8)(j/100+'0'));
                    TX2_write2buff((u8)(j%100/10+'0'));
                    TX2_write2buff((u8)(j%10+'0'));
                    PrintString2("个字节内容如下: \r\n");
                    for(i=0; i<j; i++)    TX2_write2buff(tmp[i]);
                    TX2_write2buff(0x0d);
                    TX2_write2buff(0x0a);
                    F0 = 1;
                }
            }
        }
    }
    if(!F0) PrintString2("命令错误!\r\n");
}
```

通过串口助手发送指令测试结果:



5.13 I2C 主从收发

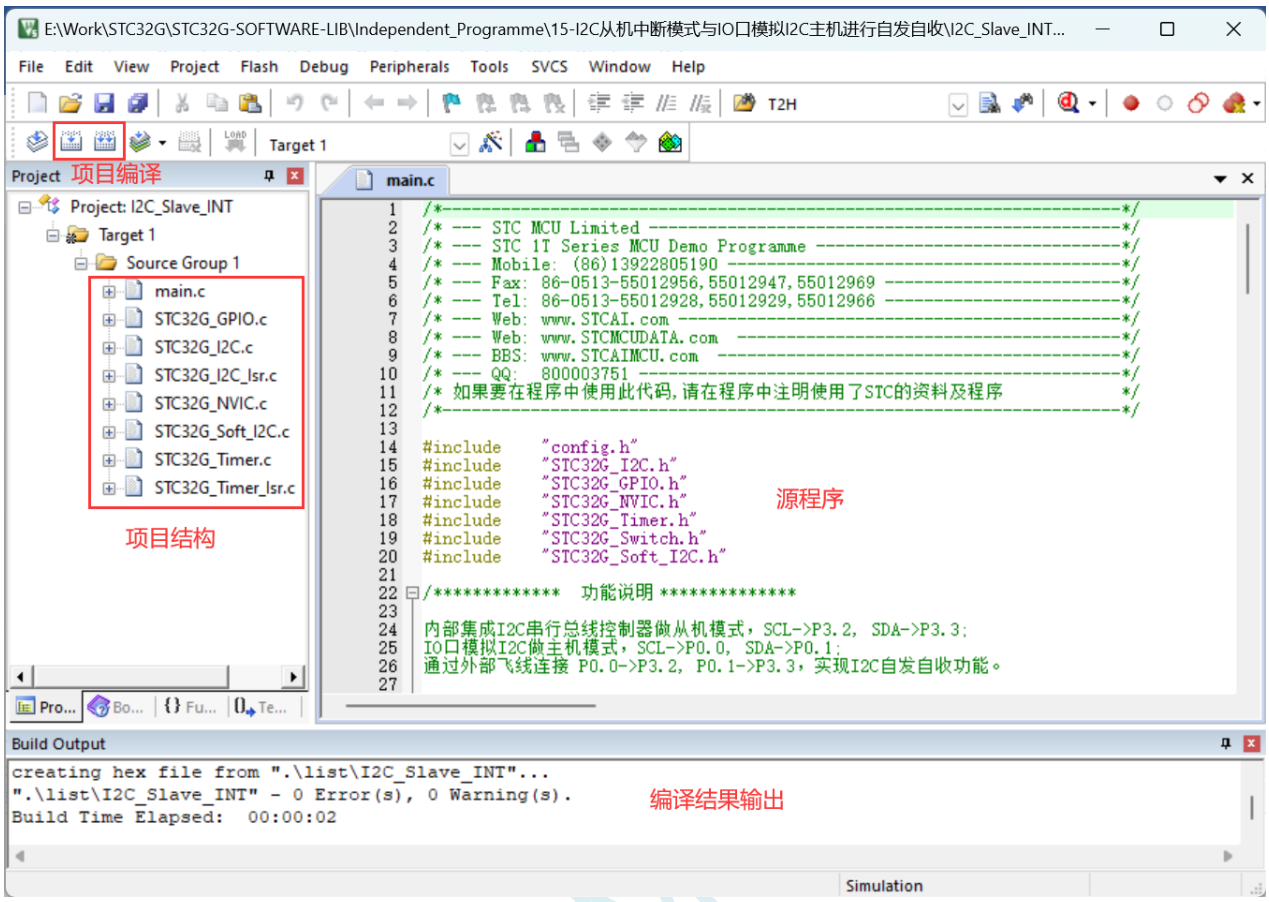
5.13.1 项目文件

打开 I2C 从机中断模式与 IO 口模拟 I2C 主机进行自发自收例程，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
I2C_Slave_INT(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_I2C (.h .c)	I2C 模块初始化相关函数库
STC32G_I2C_Isr.c	I2C 模块中断函数库
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_Soft_I2C (.h .c)	软件模拟 I2C 初始化及应用相关函数库
STC32G_Timer (.h .c)	定时器模块初始化及应用相关函数库
STC32G_Timer_Isr.c	定时器模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.13.2 程序框架

双击“I2C_Slave_INT.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程使用内部集成 I2C 串行总线控制器做从机模式，SCL->P3.2, SDA->P3.3;
IO 口模拟 I2C 做主机模式，SCL->P0.0, SDA->P0.1;
通过飞线连接 P0.0->P3.2, P0.1->P3.3，实现 I2C 自发自收功能。

在“STC32G_Soft_I2C.c”文件里定义从机的读写地址，以及通讯脚位：

```
#define SLAW      0x5A
#define SLAR      0x5B

sbit      SDA = P0^1; //定义 SDA
sbit      SCL = P0^0; //定义 SCL
```

初始化时对所有用到的 IO 口进行模式配置：

```
void GPIO_config(void)
{
    P0_MODE_IO_PU(GPIO_Pin_0 | GPIO_Pin_1);      //P0.0,P0.1 设置为准双向口
    P3_MODE_IO_PU(GPIO_Pin_2 | GPIO_Pin_3);      //P3.2,P3.3 设置为准双向口
    P6_MODE_IO_PU(GPIO_Pin_All);      //P6 设置为准双向口
    P7_MODE_IO_PU(GPIO_Pin_All);      //P7 设置为准双向口
}
```

对定时器进行设置：

```
void Timer_config (void)
{
```

```
    TIM_InitTypeDef         TIM_InitStructure;              //结构定义
//指定工作模式, TIM_16BitAutoReload,TIM_16Bit,TIM_8BitAutoReload,TIM_16BitAutoReloadNoMask
    TIM_InitStructure.TIM_Mode          = TIM_16BitAutoReload;
//指定时钟源,     TIM_CLOCK_1T,TIM_CLOCK_12T,TIM_CLOCK_Ext
    TIM_InitStructure.TIM_ClkSource = TIM_CLOCK_1T;
    TIM_InitStructure.TIM_ClkOut        = DISABLE;          //是否输出高速脉冲, ENABLE 或 DISABLE
    TIM_InitStructure.TIM_Value         = (u16)(65536UL - (MAIN_Fosc / 1000UL));//中断频率, 1000 次/秒
    TIM_InitStructure.TIM_Run           = ENABLE;           //是否初始化后启动定时器, ENABLE 或 DISABLE
    Timer_Inilize(Timer0,&TIM_InitStructure);  //初始化 Timer0
    NVIC_Timer0_Init(ENABLE,Priority_0);        //中断使能,优先级
}
```

对 I2C 接口进行设置:

```
void I2C_config(void)
{
    I2C_InitTypeDef    I2C_InitStructure;
    //主从选择     I2C_Mode_Master, I2C_Mode_Slave
    I2C_InitStructure.I2C_Mode          = I2C_Mode_Slave;
    I2C_InitStructure.I2C_Enable        = ENABLE;           //I2C 功能使能,     ENABLE, DISABLE
    I2C_InitStructure.I2C_SL_MA         = ENABLE;           //使能从机地址比较功能,     ENABLE, DISABLE
    I2C_InitStructure.I2C_SL_ADR        = 0x2d;             //从机设备地址,   0~127   (0x2d<<1 = 0x5a)
    I2C_Init(&I2C_InitStructure);
    //主从模式, I2C_Mode_Master, I2C_Mode_Slave;
    //中断使能, I2C_ESTAI/I2C_ERXI/I2C_ETXI/I2C_ESTOI/DISABLE;
    //优先级(低到高) Priority_0,Priority_1,Priority_2,Priority_3
    NVIC_I2C_Init(I2C_Mode_Slave,I2C_ESTAI|I2C_ERXI|I2C_ETXI|I2C_ESTOI,Priority_0);

    I2C_SW(I2C_P33_P32);                //I2C_P14_P15,I2C_P24_P25,I2C_P76_P77,I2C_P33_P32
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();       //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

```
    GPIO_config();
    Timer_config();
    I2C_config();
    EA = 1;
    display_index = 0;
    DisplayFlag = 0;
    for(i=0; i<8; i++)    LED8[i] = 0x10; //上电消隐
```

主循环里 IO 口模拟 I2C 做主机，每秒钟发送一次计数数据，并在左边 4 个数码管上显示发送内容；硬件 I2C 模块做从机，接收到主机发送的数据后在右边 4 个数码管显示。

```
    while (1)
    {
1.  判断从机硬件 I2C 模块是否有接收到数据，有的话在右边 4 个数码管显示
        if(DisplayFlag)
        {
            DisplayFlag = 0;
            LED8[4] = I2C_Buffer[0];     //将接收到的数据存入数码管显示缓冲区
            LED8[5] = I2C_Buffer[1];
            LED8[6] = I2C_Buffer[2];
            LED8[7] = I2C_Buffer[3];
        }
2.  判断定时器 0 定时 1ms 标志位
        if(B_1ms)
        {
            B_1ms = 0;          //清除 1ms 中断标志
            DisplayScan();      //每 1 毫秒刷新一次数码管显示
3.  累计 1 秒钟执行一次计数，在左边 4 个数码管上显示并通过软件模拟 I2C 发送秒计数值
            if(++msecond >= 1000)     //1ms * 1000 = 1 秒
            {
                msecond = 0;            //清 1000ms 计数
                second++;               //秒计数+1
                if(second >= 10000)     second = 0;     //设置秒计数范围为 0~9999
4.  秒计数值的个、十、百、千位分别保存并存入数码管显示缓冲区
                tmp[0] = second / 1000;
                tmp[1] = (second % 1000) / 100;
                tmp[2] = (second % 100) / 10;
                tmp[3] = second % 10;
                LED8[0] = tmp[0];
                LED8[1] = tmp[1];
                LED8[2] = tmp[2];
                LED8[3] = tmp[3];
5.  通过 I2C 接口发送秒计数值
                SI2C_WriteNbyte(SLAW, 0, tmp, 4);         //通过软件模拟 I2C 发送秒计数值
            }
        }
    }
```

5.14 内部 RTC 时钟

5.14.1 项目文件

打开内部 RTC 时钟测试程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

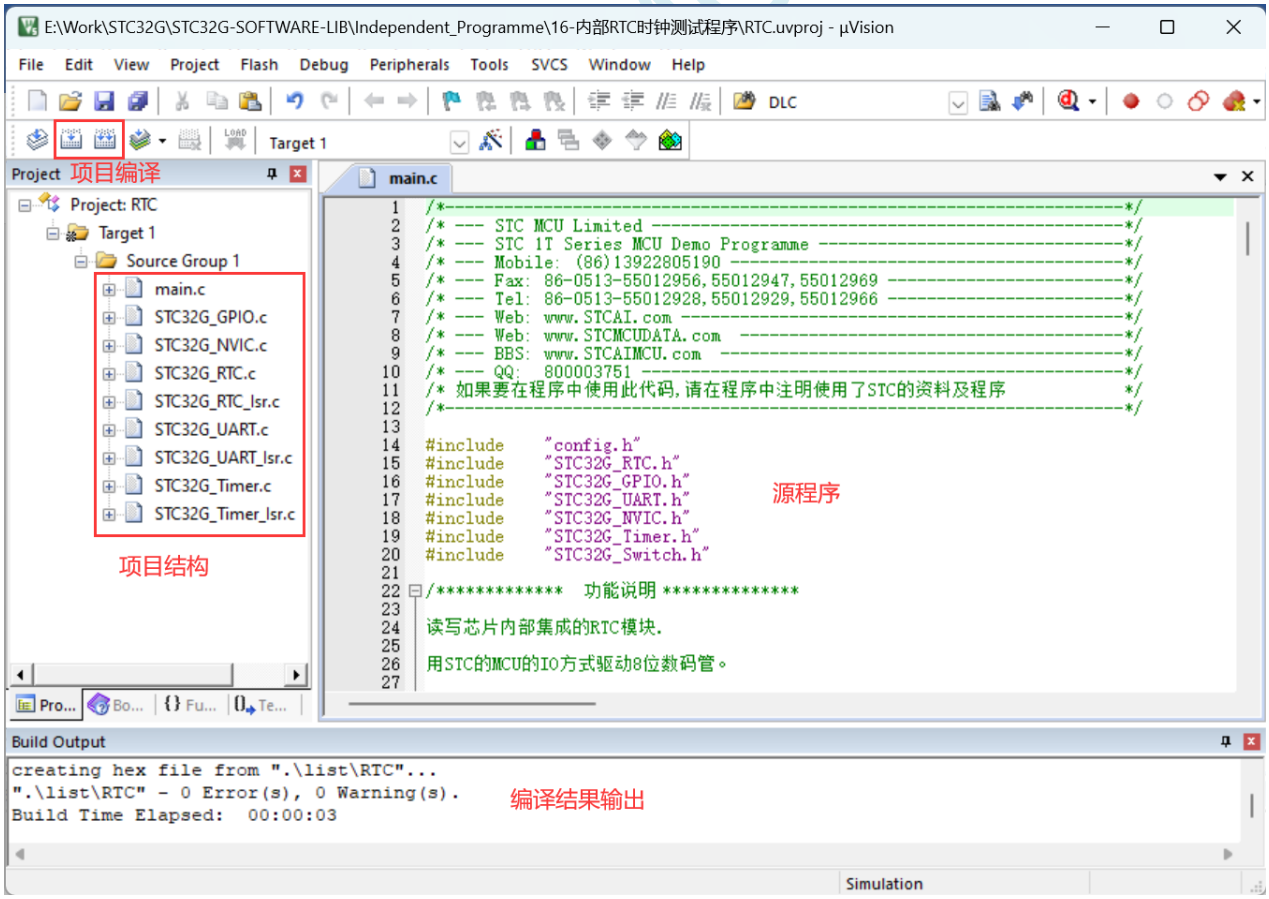
文件	描述

Config.h	用户配置文件，主要是主时钟定义
RTC(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_RTC(.h .c)	RTC 模块初始化相关函数库
STC32G_RTC_Isr.c	RTC 模块中断函数库
STC32G_Timer (.h .c)	定时器模块初始化及应用相关函数库
STC32G_Timer_Isr.c	定时器模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.14.2 程序框架

双击“RTC.uvproj”使用 Keil 编译器打开项目。

在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程使用芯片内部集成 RTC 实时时钟做时钟显示，通过实验箱上的数码管显示当前时间(小时-分钟-秒)，并通过串口 2（P4.6, P4.7）打印时钟内容（年，月，日，时，分，秒）以及闹钟提示（RTC Alarm!）。

利用实验箱上的行列矩阵按键调整时间:
键码 25: 小时+.
键码 26: 小时-.
键码 27: 分钟+.
键码 28: 分钟-.

初始化时对所有用到的 IO 口进行模式配置:

```
void GPIO_config(void)
{
    P0_MODE_IO_PU(GPIO_Pin_All);      //P0 设置为准双向口
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);   //P4.6,P4.7 设置为准双向口
    P6_MODE_IO_PU(GPIO_Pin_All);      //P6 设置为准双向口
    P7_MODE_IO_PU(GPIO_Pin_All);      //P7 设置为准双向口
}
```

对定时器进行设置:

```
void Timer_config (void)
{
    TIM_InitTypeDef     TIM_InitStructure;          //结构定义
//指定工作模式, TIM_16BitAutoReload,TIM_16Bit,TIM_8BitAutoReload,TIM_16BitAutoReloadNoMask
    TIM_InitStructure.TIM_Mode      = TIM_16BitAutoReload;
//指定时钟源,   TIM_CLOCK_1T,TIM_CLOCK_12T,TIM_CLOCK_Ext
    TIM_InitStructure.TIM_ClkSource = TIM_CLOCK_1T;
    TIM_InitStructure.TIM_ClkOut    = DISABLE;      //是否输出高速脉冲, ENABLE 或 DISABLE
    TIM_InitStructure.TIM_Value     = (u16)(65536UL - (MAIN_Fosc / 1000UL));//中断频率, 1000 次/秒
    TIM_InitStructure.TIM_Run       = ENABLE;       //是否初始化后启动定时器, ENABLE 或 DISABLE
    Timer_Inilize(Timer0,&TIM_InitStructure);  //初始化 Timer0
    NVIC_Timer0_Init(ENABLE,Priority_0);       //中断使能,优先级
}
```

对 UART 接口进行设置:

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;             //结构定义
    //模式,   UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode      = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
//  COMx_InitStructure.UART_BRT_Use   = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate  = 115200ul;     //波特率,     110 ~ 115200
    COMx_InitStructure.UART_RxEnable  = ENABLE;       //接收允许,   ENABLE 或 DISABLE
    UART_Configuration(UART2, &COMx_InitStructure);   //初始化串口 2
    NVIC_UART2_Init(ENABLE,Priority_1);       //中断使能,优先级

    UART2_SW(UART2_SW_P46_P47);           //UART2_SW_P10_P11,UART2_SW_P46_P47
}
```

对 RTC 进行设置:

```
void RTC_config(void)
{
    RTC_InitTypeDef         RTC_InitStructure;
    RTC_InitStructure.RTC_Clock    = RTC_X32KCR; //RTC 时钟源选择, RTC_IRC32KCR, RTC_X32KCR
    RTC_InitStructure.RTC_Enable = ENABLE;       //RTC 功能使能,    ENABLE, DISABLE
    RTC_InitStructure.RTC_Year     = 21;         //RTC 年, 00~99, 对应 2000~2099 年
    RTC_InitStructure.RTC_Month    = 12;         //RTC 月, 01~12
    RTC_InitStructure.RTC_Day      = 31;         //RTC 日, 01~31
    RTC_InitStructure.RTC_Hour     = 23;         //RTC 时, 00~23
    RTC_InitStructure.RTC_Min      = 59;         //RTC 分, 00~59
    RTC_InitStructure.RTC_Sec      = 55;         //RTC 秒, 00~59
    RTC_InitStructure.RTC_Ssec     = 00;         //RTC 1/128 秒, 00~127

    RTC_InitStructure.RTC_ALAHour= 00;           //RTC 闹钟时, 00~23
    RTC_InitStructure.RTC_ALAMin = 00;           //RTC 闹钟分, 00~59
    RTC_InitStructure.RTC_ALASec = 00;           //RTC 闹钟秒, 00~59
    RTC_InitStructure.RTC_ALASsec= 00;           //RTC 闹钟 1/128 秒, 00~127
    RTC_Inilize(&RTC_InitStructure);
    //中断使能,
    //RTC_ALARM_INT: 闹钟中断
    //RTC_DAY_INT: 每日中断
    //RTC_HOUR_INT: 每小时中断
    //RTC_MIN_INT: 每分钟中断
    //RTC_SEC_INT: 每秒中断
    //RTC_SEC2_INT: 1/2 秒中断
    //RTC_SEC8_INT: 1/8 秒中断
    //RTC_SEC32_INT: 1/32 秒中断
    //DISABLE: 禁止中断
    //优先级(低到高) Priority_0,Priority_1,Priority_2,Priority_3
    NVIC_RTC_Init(RTC_ALARM_INT|RTC_SEC_INT,Priority_0);
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();       //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

```
    GPIO_config();
    Timer_config();
    UART_config();
    RTC_config();
    EA = 1;
```

```
    display_index = 0;
    for(i=0; i<8; i++)   LED8[i] = 0x10; //上电消隐

    DisplayRTC();       //读取 RTC 时钟值存入数码管显示缓存
    LED8[2] = DIS_;
    LED8[5] = DIS_;

    KeyHoldCnt = 0;  //键按下计时
    KeyCode = 0;     //给用户使用的键码

    IO_KeyState = 0;
    IO_KeyState1 = 0;
    IO_KeyHoldCnt = 0;
    cnt50ms = 0;
```

主循环里每秒钟刷新一次数码管显示内容，并通过串口打印年，月，日，时，分，秒数值。
判断闹钟中断标志，产生闹钟中断的话串口打印“RTC Alarm!”。
每毫秒刷新一次数码管，扫描行列按键判断是否有按键按下，有的话根据键值调整时钟。

```
    while (1)
    {
        if(B_1S)        //判断秒中断标志
        {
            B_1S = 0;
            DisplayRTC();       //读取 RTC 时钟值存入数码管显示缓存
            //通过串口打印年，月，日，时，分，秒数值
            printf("Year=20%d,Month=%d,Day=%d,Hour=%d,Minute=%d,Second=%d\r\n",
            YEAR,MONTH,DAY,HOUR,MIN,SEC);
        }

        if(B_Alarm)         //判断闹钟中断标志
        {
            B_Alarm = 0;
            printf("RTC Alarm!\r\n");     //串口打印提示产生闹钟中断
        }

        if(B_1ms)           //判断 1 毫秒定时器中断标志
        {
            B_1ms = 0;
        #if(SleepModeSet == 1)     //睡眠模式下，MCU 进入低功耗模式
            _nop_();
            _nop_();
            PD = 1;  //STC32G 芯片使用内部 32K 时钟，休眠无法唤醒
            _nop_();
            _nop_();
            _nop_();
```

```
        _nop_();
        _nop_();
        _nop_();
    #else                   //非睡眠模式下通过数码管显示 RTC 时间，并扫描行列矩阵按键
        DisplayScan();

        if(++cnt50ms >= 50)       //50ms 扫描一次行列键盘
        {
            cnt50ms = 0;
            IO_KeyScan();
        }

        if(KeyCode != 0)            //有键按下
        {
            if(KeyCode == 25)     //hour +1
            {
                if(++hour >= 24)       hour = 0;
                WriteRTC();
                DisplayRTC();
            }
            if(KeyCode == 26)     //hour -1
            {
                if(--hour >= 24)       hour = 23;
                WriteRTC();
                DisplayRTC();
            }
            if(KeyCode == 27)     //minute +1
            {
                second = 0;
                if(++minute >= 60)    minute = 0;
                WriteRTC();
                DisplayRTC();
            }
            if(KeyCode == 28)     //minute -1
            {
                second = 0;
                if(--minute >= 60)    minute = 59;
                WriteRTC();
                DisplayRTC();
            }

            KeyCode = 0;
        }
    #endif
    }
```

```
}
```

通过串口助手显示程序运行结果:

5.15 DMA-ADC 数据自动存储

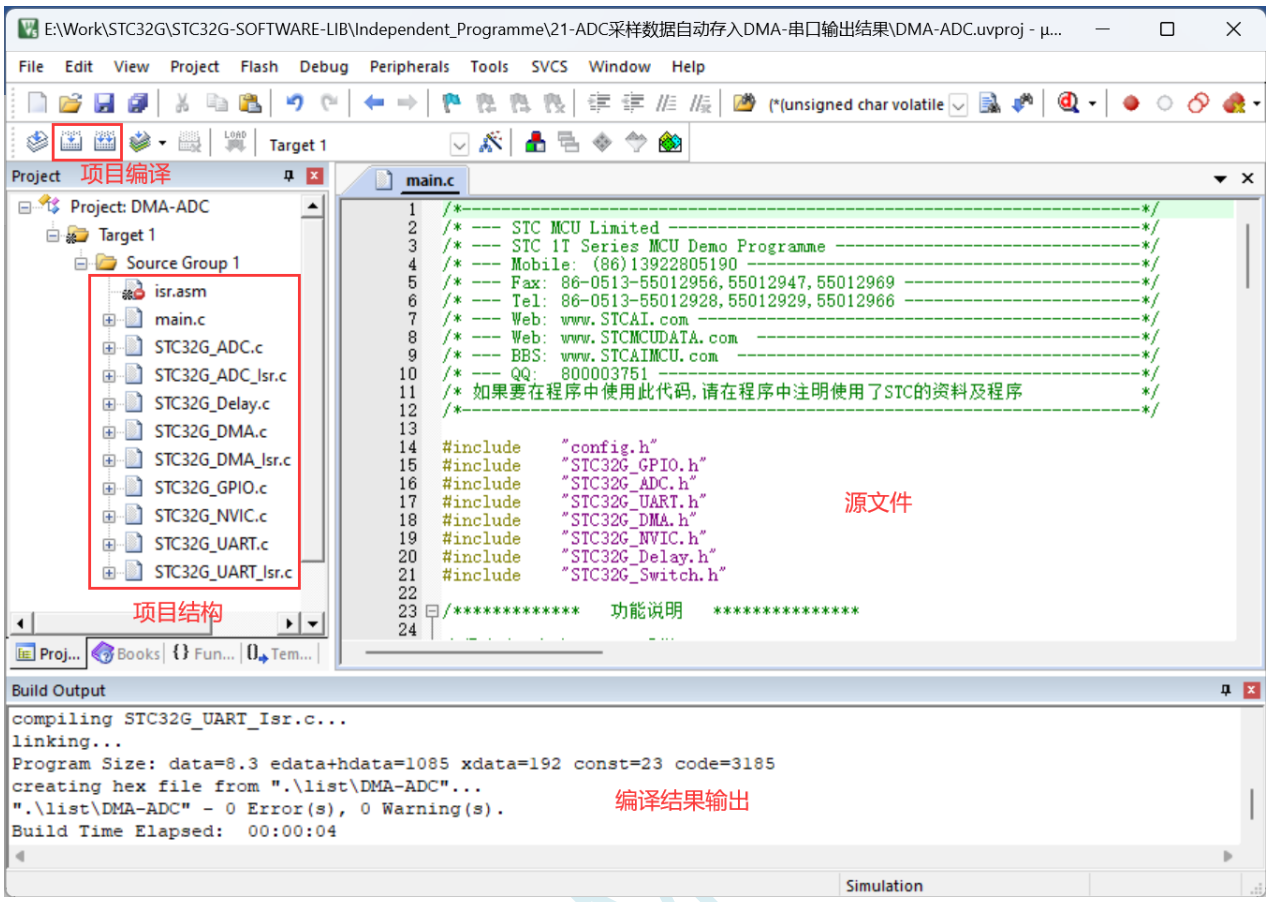
5.15.1 项目文件

打开 ADC 采样数据自动存入 DMA 程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下:

文件	描述
Config.h	用户配置文件，主要是主时钟定义
DMA-ADC(.uvopt .uvproj)	项目文件
Main.c	主函数文件
isr.asm	中断向量映射文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_ADC (.h .c)	ADC 模块初始化及应用相关函数库
STC32G_ADC_Isr.c	ADC 模块中断函数库
STC32G_Delay (.h .c)	标准延时函数
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_DMA(.h .c)	DMA 模块初始化相关函数库
STC32G_DMA_Isr.c	DMA 模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.15.2 程序框架

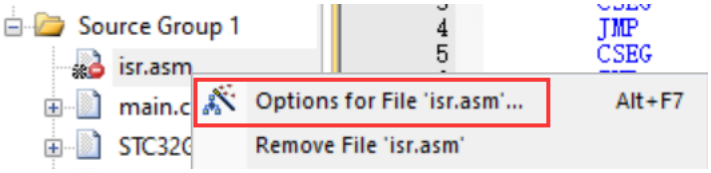
双击“DMA-ADC.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



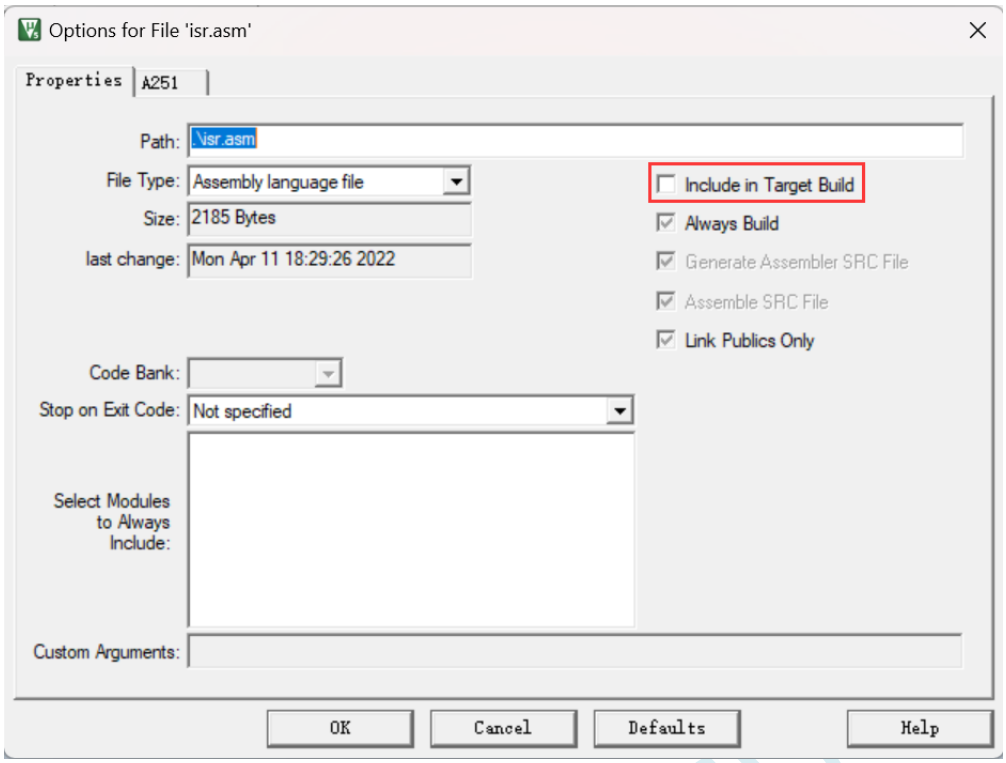
其中的“isr.asm”是中断向量映射文件，由于目前 keil 各版本的 C51 和 C251 编译器均只支持 32 个中断号（0~31），超过 31 的中断向量编译时会报错。暂时只能通过安装中断向量号拓展插件（详情参考说明书 3.3 章节 keil 中断向量号拓展插件使用说明），或者将中断映射到向量号 31 以内没有使用的中断地址。“isr.asm”文件就是将中断向量号超过 31 的部分中断映射到 13 号中断向量地址。

例程使用的是通过安装 keil 中断向量号拓展插件的方法，所以设置“isr.asm”文件不参与编译，如果想用中断映射方式的话通过修改设置让“isr.asm”文件参与编译，并且将中断函数内容添加到 13 号中断函数里即可。

设置“isr.asm”文件参与编译方法，通过右键点击“isr.asm”文件选择“Options for File ‘isr.asm’...”：



在弹出框里勾选“Include in Target Build”：



该例程设置 DMA 自动采集 ADC 数值并存放到指定的 xdata 空间。通过 ADC-DMA 模块可以设置 ADC 轮询采集的通道，以及每个通道采集的次数，并自动求平均值。

程序常量什么位置定义 ADC 转换的通道数与每个通道转换的数据总数：

```
#define ADC_CH      16   /* 1~16, ADC 转换通道数, 需同步修改转换通道 */
#define ADC_DATA    12   /* 6~n, 每个通道 ADC 转换数据总数, 2*转换次数+4, 需同步修改转换次数 */
```

ADC_CH 定义为 16，表示转换 16 个通道（ADC0~ADC15）；
ADC_DATA 值计算公式为：2*转换次数+4。如果每个通道转换 4 次（DMA_Times = ADC_4_Times），那么 ADC_DATA 定义值应设为：2*4+4=12。

ADC 采集数据存储格式如图所示：

ADC 通道	偏移地址	数据
第 1 通道	0	使能的第 1 通道的第 1 次 ADC 转换结果的高字节
	1	使能的第 1 通道的第 1 次 ADC 转换结果的低字节
	2	使能的第 1 通道的第 2 次 ADC 转换结果的高字节
	3	使能的第 1 通道的第 2 次 ADC 转换结果的低字节
	...	...
	2n-2	使能的第 1 通道的第 n 次 ADC 转换结果的高字节
	2n-1	使能的第 1 通道的第 n 次 ADC 转换结果的低字节
	2n	第 1 通道的 ADC 通道号
	2n+1	第 1 通道 n 次 ADC 转换结果取完平均值之后的余数
	2n+2	第 1 通道 n 次 ADC 转换结果平均值的高字节
	2n+3	第 1 通道 n 次 ADC 转换结果平均值的低字节
第 2 通道	(2n+3) + 0	使能的第 2 通道的第 1 次 ADC 转换结果的高字节
	(2n+3) + 1	使能的第 2 通道的第 1 次 ADC 转换结果的低字节
	(2n+3) + 2	使能的第 2 通道的第 2 次 ADC 转换结果的高字节
	(2n+3) + 3	使能的第 2 通道的第 2 次 ADC 转换结果的低字节
	...	...
	(2n+3) + 2n-2	使能的第 2 通道的第 n 次 ADC 转换结果的高字节
	(2n+3) + 2n-1	使能的第 2 通道的第 n 次 ADC 转换结果的低字节
	(2n+3) + 2n	第 2 通道的 ADC 通道号
	(2n+3) + 2n+1	第 2 通道 n 次 ADC 转换结果取完平均值之后的余数
	(2n+3) + 2n+2	第 2 通道 n 次 ADC 转换结果平均值的高字节
	(2n+3) + 2n+3	第 2 通道 n 次 ADC 转换结果平均值的低字节
	...	...

初始化时对所有用到的 IO 口进行模式配置，把要 ADC 转换的引脚设置为高阻输入：

```
void GPIO_config(void)
{
    //P0.0~P0.6 设置为高阻输入
    P0_MODE_IN_HIZ(GPIO_Pin_LOW | GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6);
    P1_MODE_IN_HIZ(GPIO_Pin_All);      //P1.0~P1.7 设置为高阻输入
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);   //P4.6,P4.7 设置为准双向口
}
```

对 UART 接口进行设置：

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;                 //结构定义
    //模式,    UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode        = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
//  COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;     //波特率,     110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;       //接收允许,   ENABLE 或 DISABLE
    UART_Configuration(UART2, &COMx_InitStructure);     //初始化串口 2
```

```
    NVIC_UART2_Init(ENABLE,Priority_1);        //中断使能,优先级

    UART2_SW(UART2_SW_P46_P47);          //UART2_SW_P10_P11,UART2_SW_P46_P47
}
```

对 ADC 进行设置:

```
void ADC_config (void)
{
    ADC_InitTypeDef        ADC_InitStructure;      //结构定义

    //ADC 模拟信号采样时间控制, 0~31（注意: SMPDUTY 一定不能设置小于 10）
    ADC_InitStructure.ADC_SMPduty    = 31;
    ADC_InitStructure.ADC_CsSetup    = 0;          //ADC 通道选择时间控制 0(默认),1
    ADC_InitStructure.ADC_CsHold     = 1;          //ADC 通道选择保持时间控制 0,1(默认),2,3
    ADC_InitStructure.ADC_Speed      = ADC_SPEED_2X16T;    //设置 ADC 工作时钟频率
    ADC_InitStructure.ADC_AdjResult = ADC_RIGHT_JUSTIFIED; //ADC 结果调整, 这里设置右对齐
    ADC_Inilize(&ADC_InitStructure);      //初始化 ADC
    ADC_PowerControl(ENABLE);             //ADC 电源开关, ENABLE 或 DISABLE
    NVIC_ADC_Init(DISABLE,Priority_0);          //中断使能,优先级
}
```

对 DMA 进行设置:

```
void DMA_config(void)
{
    DMA_ADC_InitTypeDef      DMA_ADC_InitStructure;     //结构定义

    DMA_ADC_InitStructure.DMA_Enable = ENABLE;          //DMA 使能   ENABLE, DISABLE
    //ADC 通道使能寄存器, 1:使能, bit15~bit0 对应 ADC15~ADC0
    DMA_ADC_InitStructure.DMA_Channel = 0xffff;
    DMA_ADC_InitStructure.DMA_Buffer = (u16)DmaAdBuffer;   //ADC 转换数据存储地址
    DMA_ADC_InitStructure.DMA_Times = ADC_4_Times;    //每个通道转换次数
    DMA_ADC_Inilize(&DMA_ADC_InitStructure);              //初始化
    NVIC_DMA_ADC_Init(ENABLE,Priority_0,Priority_0);        //中断使能,优先级
    DMA_ADC_TRIG();       //触发启动 ADC 转换
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();      //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、设置变量默认值、IO 口初始电平等:

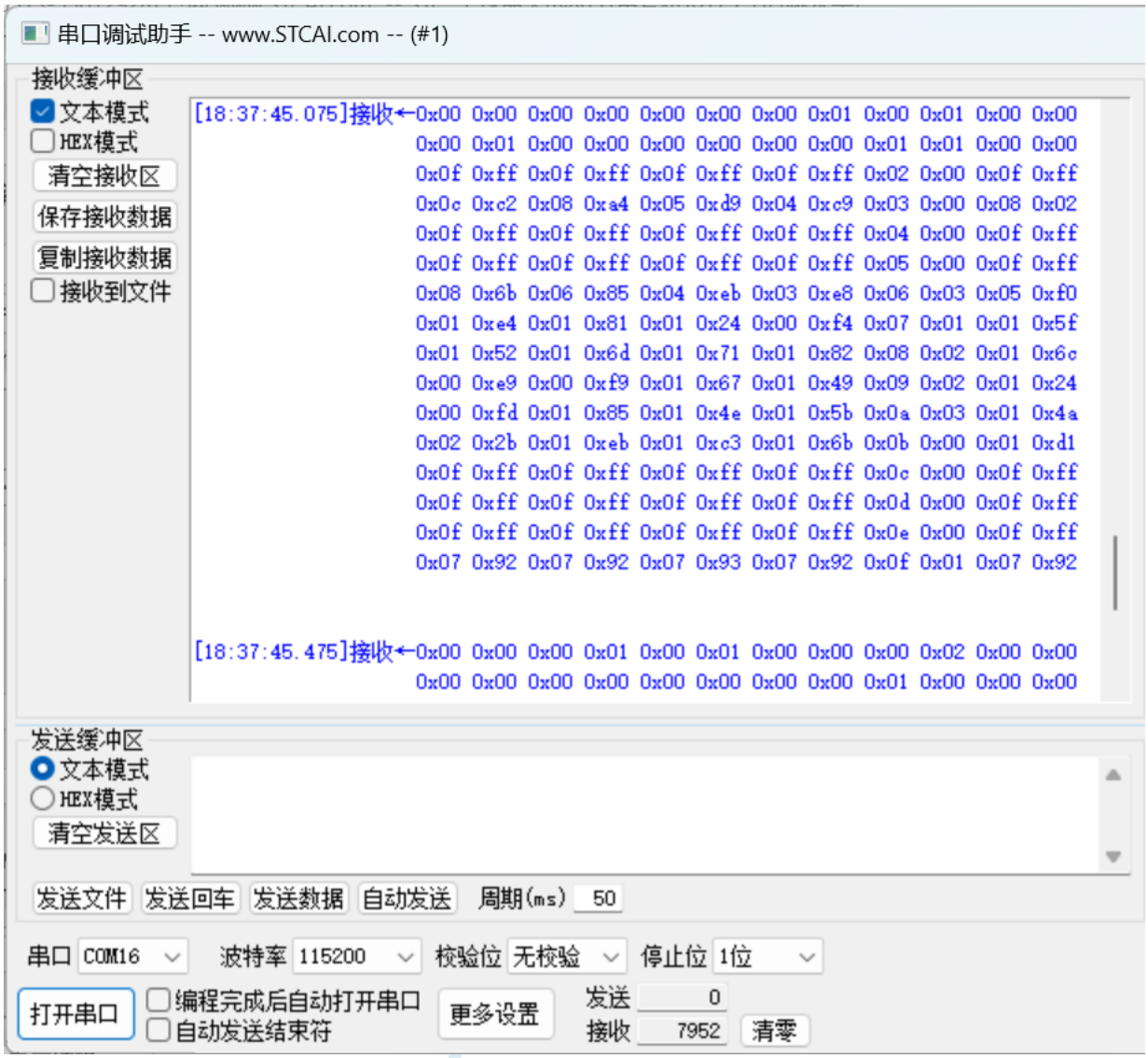
```
    GPIO_config();
    UART_config();
```

```
    ADC_config();
    DMA_config();
    EA = 1;
```

主循环里每 200 毫秒钟读取一次 ADC 采样结果，并通过串口打印出来。

```
    while (1)
    {
        delay_ms(200);      //200ms 输出一次采样结果，方便观察

        if(DmaADCFlag)    //判断 ADC DMA 采样是否完成
        {
            DmaADCFlag = 0;         //清除完成标志
            for(i=0; i<ADC_CH; i++)     //打印每个设置通道的采样结果
            {
                for(n=0; n<ADC_DATA; n++)     //打印当前通道的采样数据
                {
                    //第 1 组数据,...,第 n 组数据,AD 通道,平均余数,平均值
                    printf("0x%02x ",DmaAdBuffer[i][n]);
                }
                printf("\r\n");         //串口输出回车换行符
            }
            printf("\r\n");             //串口输出回车换行符
            DMA_ADC_TRIG();         //重新触发启动下一次转换
        }
    }
```

通过串口助手显示程序运行结果:


串口调试助手 -- www.STCAI.com -- (#1)
接收缓冲区
文本模式
HEX模式
清空接收区
保存接收数据
复制接收数据
接收到文件
[18:37:45.075]接收←0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x01 0x00 0x00
0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x01 0x00 0x00
0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0f 0xff 0x02 0x00 0x0f 0xff
0x0c 0xc2 0x08 0xa4 0x05 0xd9 0x04 0xc9 0x03 0x00 0x08 0x02
0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0f 0xff 0x04 0x00 0x0f 0xff
0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0f 0xff 0x05 0x00 0x0f 0xff
0x08 0x6b 0x06 0x85 0x04 0xeb 0x03 0xe8 0x06 0x03 0x05 0xf0
0x01 0xe4 0x01 0x81 0x01 0x24 0x00 0xf4 0x07 0x01 0x01 0x5f
0x01 0x52 0x01 0x6d 0x01 0x71 0x01 0x82 0x08 0x02 0x01 0x6c
0x00 0xe9 0x00 0xf9 0x01 0x67 0x01 0x49 0x09 0x02 0x01 0x24
0x00 0xfd 0x01 0x85 0x01 0x4e 0x01 0x5b 0x0a 0x03 0x01 0x4a
0x02 0x2b 0x01 0xeb 0x01 0xc3 0x01 0x6b 0x0b 0x00 0x01 0xd1
0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0c 0x00 0x0f 0xff
0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0d 0x00 0x0f 0xff
0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0f 0xff 0x0e 0x00 0x0f 0xff
0x07 0x92 0x07 0x92 0x07 0x93 0x07 0x92 0x0f 0x01 0x07 0x92
[18:37:45.475]接收←0x00 0x00 0x00 0x01 0x00 0x01 0x00 0x00 0x00 0x02 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00
发送缓冲区
文本模式
HEX模式
清空发送区
发送文件
发送回车
发送数据
自动发送
周期(ms) 50
串口 COM16
波特率 115200
校验位 无校验
停止位 1位
打开串口
编程完成后自动打开串口
自动发送结束符
更多设置
发送 0
接收 7952
清零

5.16 DMA-M2M 存储器之间数据交换

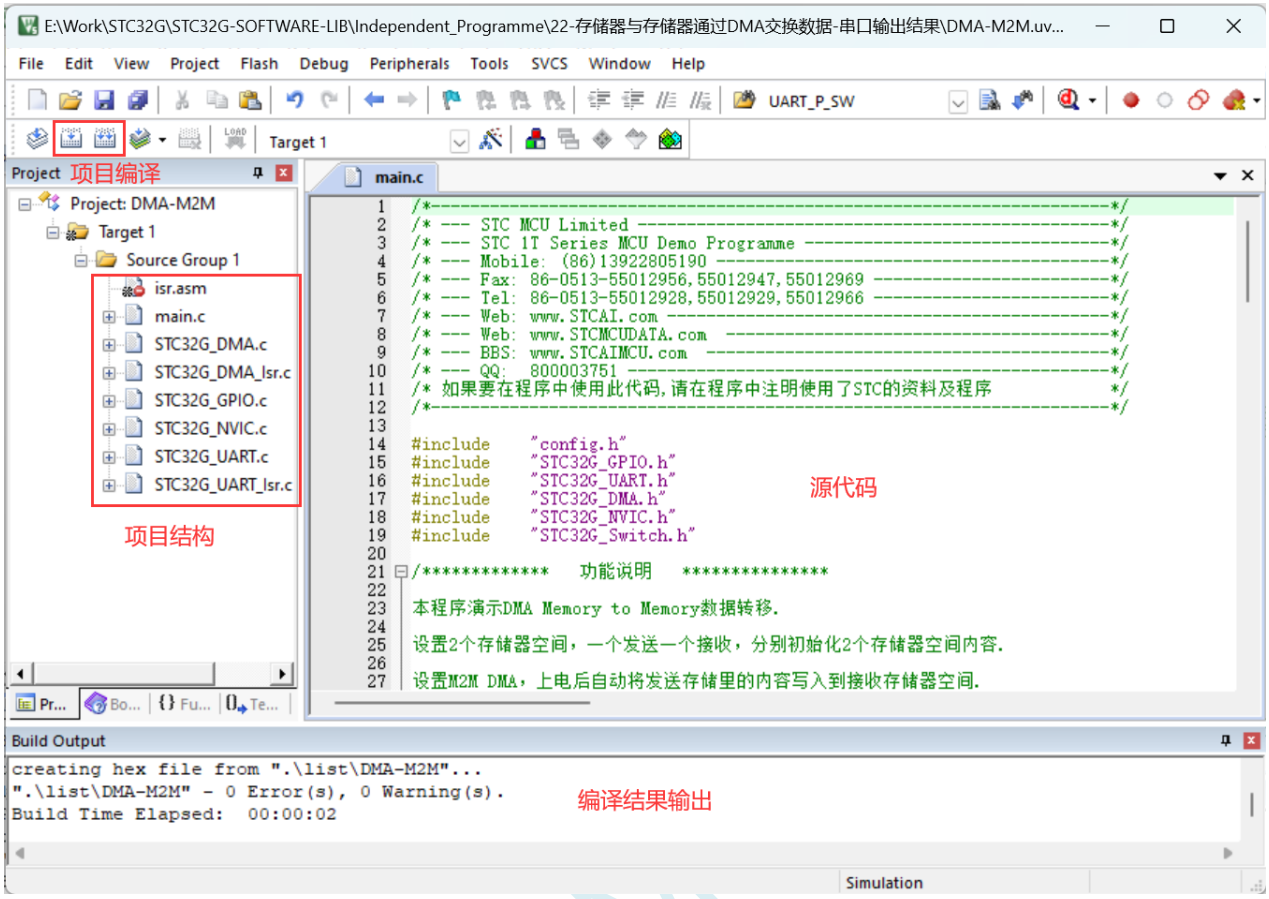
5.16.1 项目文件

打开存储器与存储器通过 DMA 交换数据程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
DMA-M2M(.uvopt .uvproj)	项目文件
Main.c	主函数文件
isr.asm	中断向量映射文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_DMA(.h .c)	DMA 模块初始化相关函数库
STC32G_DMA_Isr.c	DMA 模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.16.2 程序框架

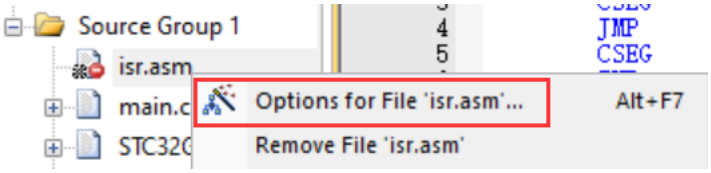
双击“DMA-M2M.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



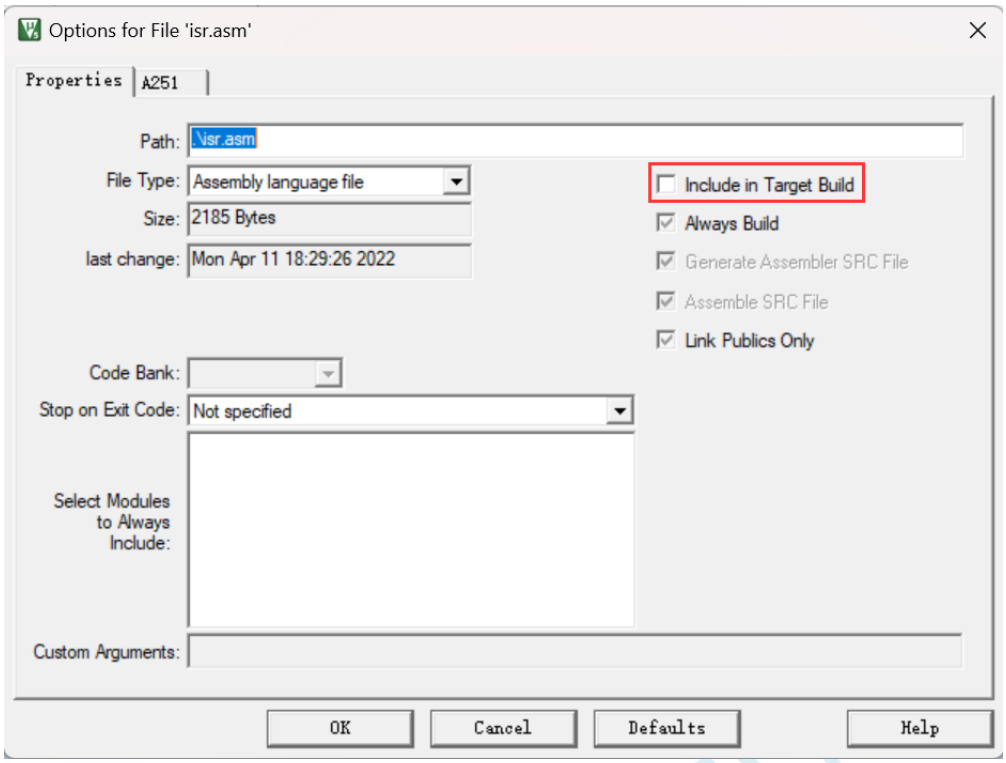
其中的“isr.asm”是中断向量映射文件，由于目前 keil 各版本的 C51 和 C251 编译器均只支持 32 个中断号（0~31），超过 31 的中断向量编译时会报错。暂时只能通过安装中断向量号拓展插件（详情参考说明书 3.3 章节 keil 中断向量号拓展插件使用说明），或者将中断映射到向量号 31 以内没有使用的中断地址。“isr.asm”文件就是将中断向量号超过 31 的部分中断映射到 13 号中断向量地址。

例程使用的是通过安装 keil 中断向量号拓展插件的方法，所以设置“isr.asm”文件不参与编译，如果想用中断映射方式的话通过修改设置让“isr.asm”文件参与编译，并且将中断函数内容添加到 13 号中断函数里即可。

设置“isr.asm”文件参与编译方法，通过右键点击“isr.asm”文件选择“Options for File ‘isr.asm’...”：



在弹出框里勾选“Include in Target Build”：



该例程设置设置 2 个存储器空间，一个发送一个接收，上电后自动将发送存储里的内容写入到接收存储器空间。通过串口 2(P4.6 P4.7)打印接收存储器数据(上电打印一次)。设置不同的读取顺序、写入顺序，接收到不同的数据结果。

初始化时对所有用到的 IO 口进行模式配置：

```
void GPIO_config(void)
{
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);    //P4.6,P4.7 设置为准双向口
}
```

对 UART 接口进行设置：

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;                 //结构定义
    //模式,    UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode        = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
//  COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;         //波特率,       110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;           //接收允许,     ENABLE 或 DISABLE
    UART_Configuration(UART2, &COMx_InitStructure);         //初始化串口 2
    NVIC_UART2_Init(ENABLE,Priority_1);         //中断使能,优先级

    UART2_SW(UART2_SW_P46_P47);             //UART2_SW_P10_P11,UART2_SW_P46_P47
}
```

对 DMA 进行设置，通过设置不同的读取顺序、写入顺序，可接收到不同的数据结果:

```
void DMA_config(void)
{
    DMA_M2M_InitTypeDef         DMA_M2M_InitStructure;      //结构定义

    DMA_M2M_InitStructure.DMA_Enable = ENABLE;          //DMA 使能       ENABLE,DISABLE
    DMA_M2M_InitStructure.DMA_Length = 127;             //DMA 传输总字节数  (0~255) + 1

    DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)DmaTxBuffer;   //发送数据存储地址
    DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)DmaRxBuffer;   //接收数据存储地址
    //数据源地址改变方向   M2M_ADDR_INC,M2M_ADDR_DEC
    DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_INC;
    //数据目标地址改变方向   M2M_ADDR_INC,M2M_ADDR_DEC
    DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_INC;

//  DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)&DmaTxBuffer[255];  //发送数据存储地址
//  DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)DmaRxBuffer;    //接收数据存储地址
    //数据源地址改变方向      M2M_ADDR_INC,M2M_ADDR_DEC
//  DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_DEC;
    //数据目标地址改变方向   M2M_ADDR_INC,M2M_ADDR_DEC
//  DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_INC;

//  DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)DmaTxBuffer;    //发送数据存储地址
//  DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)&DmaRxBuffer[255];  //接收数据存储地址
    //数据源地址改变方向      M2M_ADDR_INC,M2M_ADDR_DEC
//  DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_INC;
    //数据目标地址改变方向   M2M_ADDR_INC,M2M_ADDR_DEC
//  DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_DEC;

//  DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)&DmaTxBuffer[255];  //发送数据存储地址
//  DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)&DmaRxBuffer[255];  //接收数据存储地址
    //数据源地址改变方向      M2M_ADDR_INC,M2M_ADDR_DEC
//  DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_DEC;
    //数据目标地址改变方向   M2M_ADDR_INC,M2M_ADDR_DEC
//  DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_DEC;

    DMA_M2M_Inilize(&DMA_M2M_InitStructure);      //初始化
    NVIC_DMA_M2M_Init(ENABLE,Priority_0,Priority_0);      //中断使能,优先级,总线优先级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();     //扩展 SFR(XFR)访问使能
```

```
CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、IO 口初始电平等:

```
GPIO_config();
UART_config();
DMA_config();
EA = 1;
```

通过串口 1 打印一串数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此判断程序运行情况。

```
printf("STC32G Memory to Memory DMA Test Programme!\r\n");   //UART 发送一个字符串
```

设置变量默认值，然后启动数据交换

```
for(i=0; i<256; i++)
{
    DmaTxBuffer[i] = i;
    DmaRxBuffer[i] = 0;
}
DMA_M2M_TRIG();          //触发启动转换
```

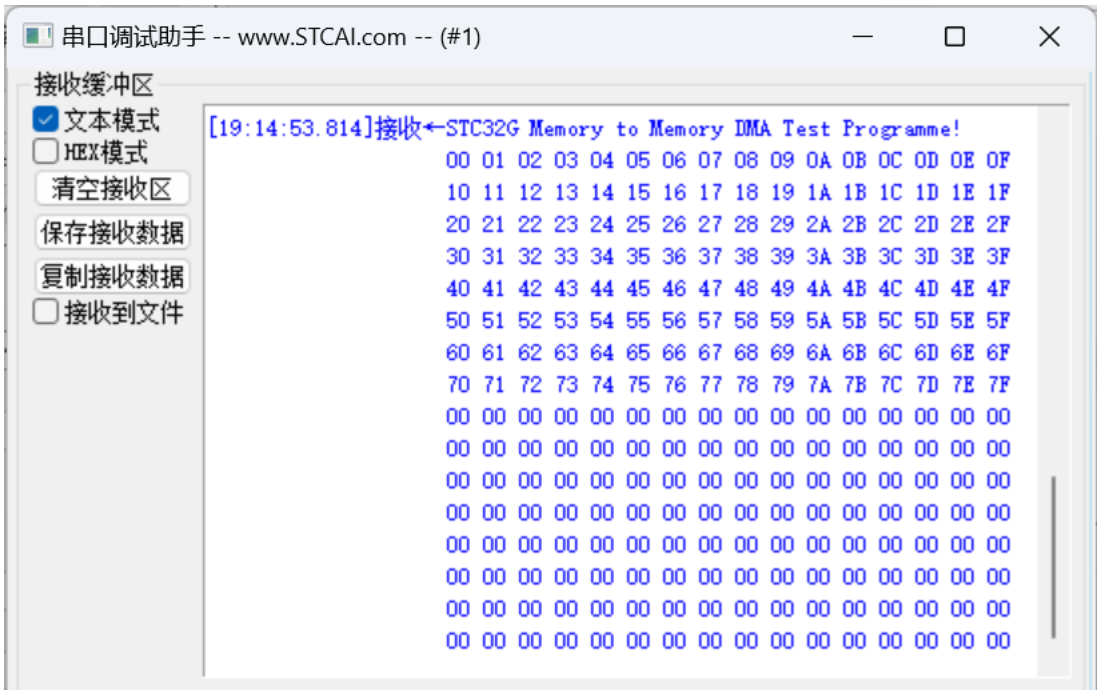
主循环里查询存储器数据交换是否转换完成，转换完成后通过串口打印目标空间的数据内容。

```
while (1)
{
    if(DmaM2MFlag)
    {
        DmaM2MFlag = 0;

        for(i=0; i<256; i++)
        {
            printf("%02X ", DmaRxBuffer[i]);
            if((i & 0x0f) == 0x0f)
                printf("\r\n");
        }
    }
}
```

通过串口助手显示程序运行结果:

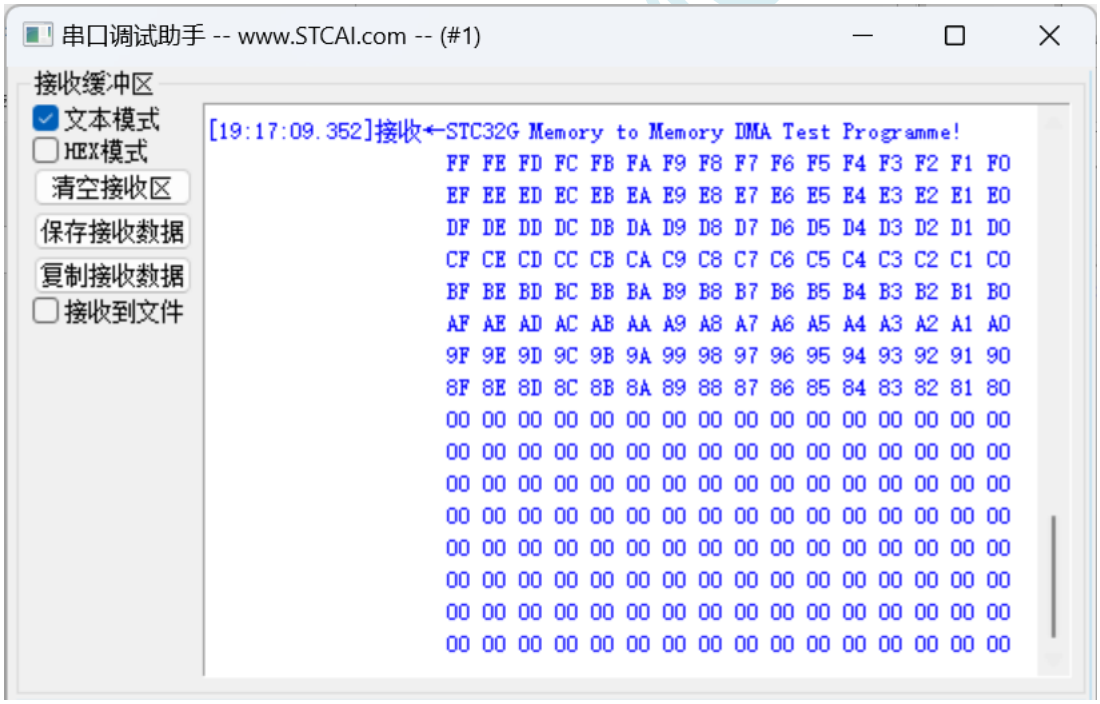
设置 1:

```
DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)DmaTxBuffer; //发送数据存储地址
DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)DmaRxBuffer; //接收数据存储地址
DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_INC;       //数据源地址改变方向      M2M_ADDR_INC,M2M_ADDR_DEC
DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_INC;  //数据目标地址改变方向    M2M_ADDR_INC,M2M_ADDR_DEC
```



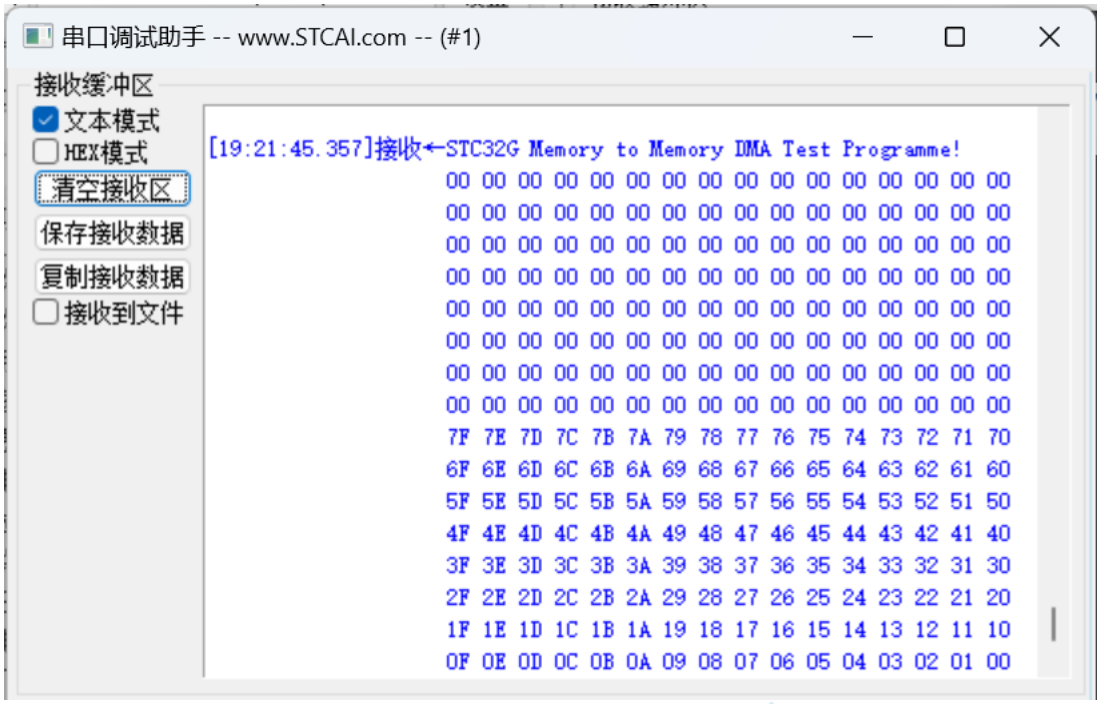
设置 2:

```
DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)&DmaTxBuffer[255];   //发送数据存储地址
DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)DmaRxBuffer; //接收数据存储地址
DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_DEC;       //数据源地址改变方向     M2M_ADDR_INC,M2M_ADDR_DEC
DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_INC;  //数据目标地址改变方向    M2M_ADDR_INC,M2M_ADDR_DEC
```



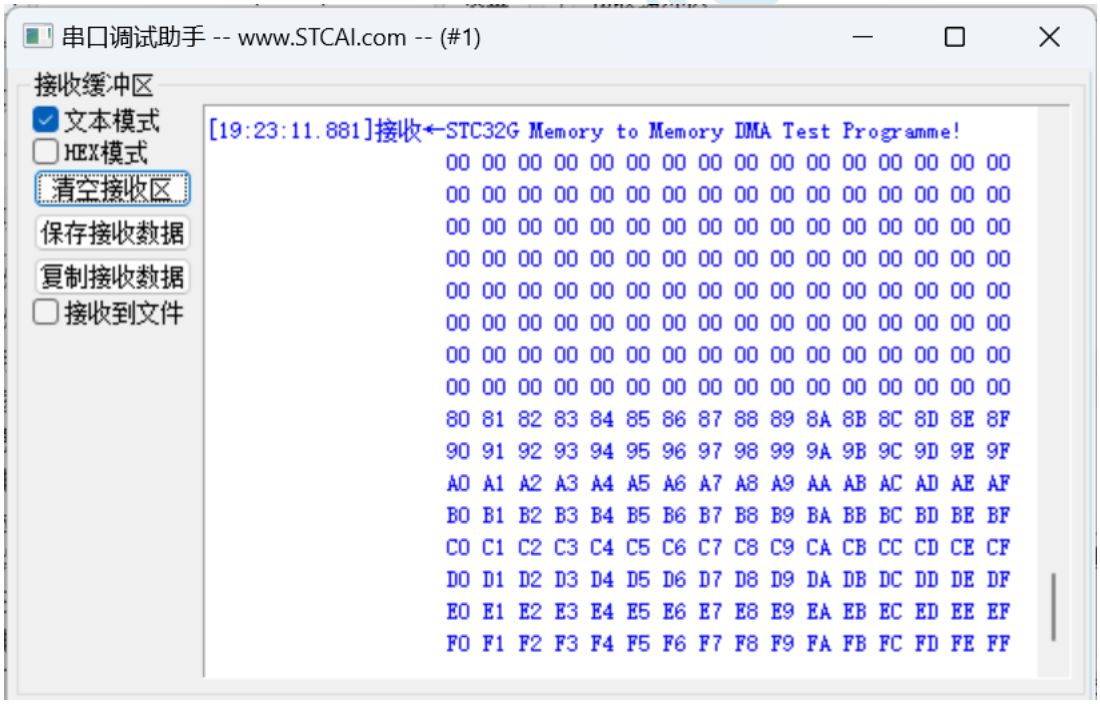
设置 3:

```
DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)DmaTxBuffer; //发送数据存储地址
DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)&DmaRxBuffer[255];   //接收数据存储地址
DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_INC;       //数据源地址改变方向     M2M_ADDR_INC,M2M_ADDR_DEC
DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_DEC;  //数据目标地址改变方向    M2M_ADDR_INC,M2M_ADDR_DEC
```



设置 4:

```
DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)&DmaTxBuffer[255];    //发送数据存储地址
DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)&DmaRxBuffer[255];    //接收数据存储地址
DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_DEC;        //数据源地址改变方向      M2M_ADDR_INC,M2M_ADDR_DEC
DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_DEC;   //数据目标地址改变方向    M2M_ADDR_INC,M2M_ADDR_DEC
```



5.17 DMA-UART 收发数据自动存入存储器

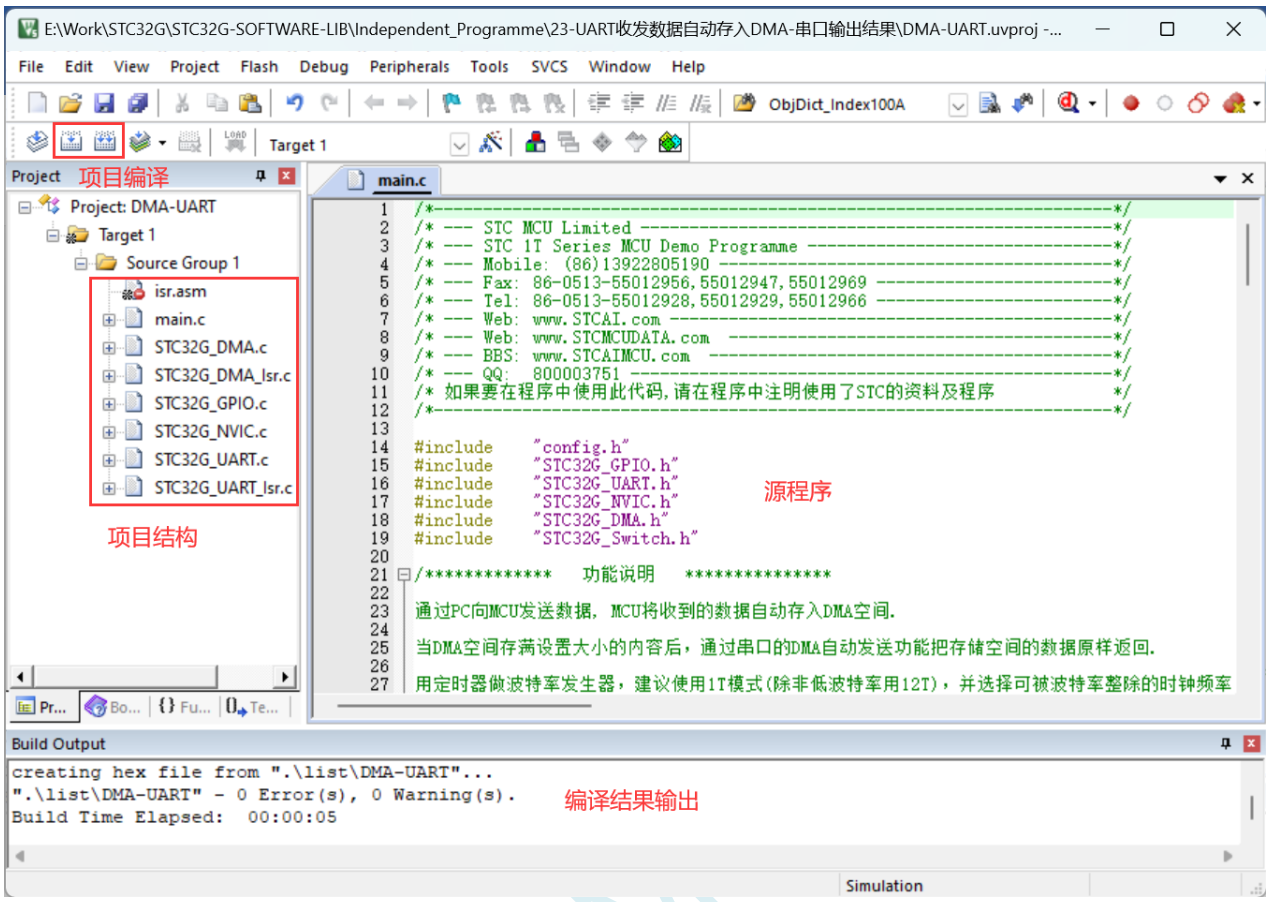
5.17.1 项目文件

打开 UART 收发数据自动存入 DMA 程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下:

文件	描述
Config.h	用户配置文件，主要是主时钟定义
DMA-UART(.uvopt .uvproj)	项目文件
Main.c	主函数文件
isr.asm	中断向量映射文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_DMA(.h .c)	DMA 模块初始化相关函数库
STC32G_DMA_Isr.c	DMA 模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.17.2 程序框架

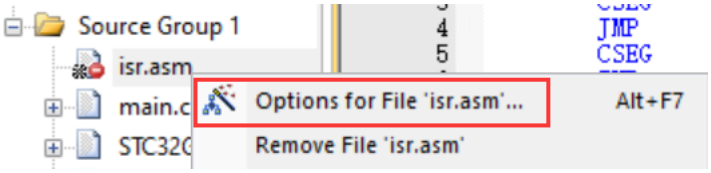
双击“DMA-UART.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



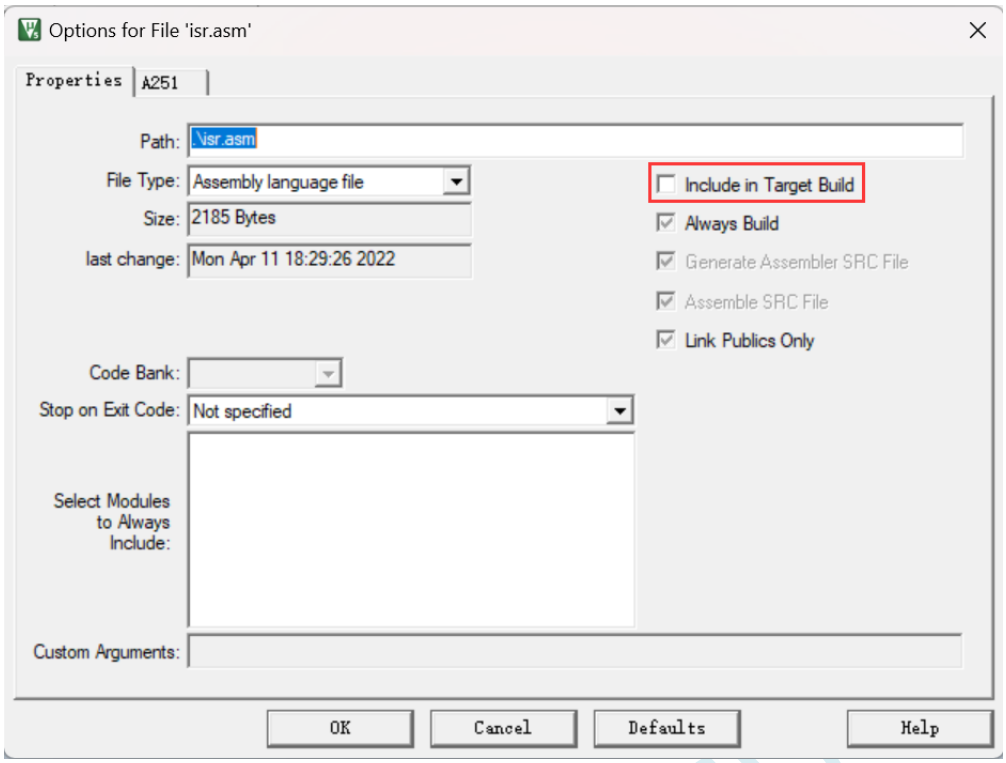
其中的“isr.asm”是中断向量映射文件，由于目前 keil 各版本的 C51 和 C251 编译器均只支持 32 个中断号（0~31），超过 31 的中断向量编译时会报错。暂时只能通过安装中断向量号拓展插件（详情参考说明书 3.3 章节 keil 中断向量号拓展插件使用说明），或者将中断映射到向量号 31 以内没有使用的中断地址。“isr.asm”文件就是将中断向量号超过 31 的部分中断映射到 13 号中断向量地址。

例程使用的是通过安装 keil 中断向量号拓展插件的方法，所以设置“isr.asm”文件不参与编译，如果想用中断映射方式的话通过修改设置让“isr.asm”文件参与编译，并且将中断函数内容添加到 13 号中断函数里即可。

设置“isr.asm”文件参与编译方法，通过右键点击“isr.asm”文件选择“Options for File ‘isr.asm’...”：



在弹出框里勾选“Include in Target Build”：



该例程通过 PC 串口助手向 MCU 发送数据，MCU 将收到的数据自动存入 DMA 空间，当 DMA 空间存满设置大小的内容后，通过串口的 DMA 自动发送功能把存储空间的数据原样返回。

初始化时对所有用到的 IO 口进行模式配置：

```
void GPIO_config(void)
{
//  P3_MODE_IO_PU(GPIO_Pin_0 | GPIO_Pin_1);   //P3.0,P3.1 设置为准双向口 - UART1
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);   //P4.6,P4.7 设置为准双向口 - UART2
//  P0_MODE_IO_PU(GPIO_Pin_0 | GPIO_Pin_1);   //P0.0,P0.1 设置为准双向口 - UART3
//  P0_MODE_IO_PU(GPIO_Pin_2 | GPIO_Pin_3);   //P0.2,P0.3 设置为准双向口 - UART4
}
```

对 UART 接口进行设置：

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;                 //结构定义
    //模式,   UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode        = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;     //波特率,     110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;       //接收允许,   ENABLE 或 DISABLE
//  UART_Configuration(UART1, &COMx_InitStructure);     //初始化串口 1
//  NVIC_UART1_Init(ENABLE,Priority_1);     //串口 1 中断使能,优先级
    UART_Configuration(UART2, &COMx_InitStructure);     //初始化串口 2
```

```
    NVIC_UART2_Init(ENABLE,Priority_1);     //串口 2 中断使能,优先级
//  UART_Configuration(UART3, &COMx_InitStructure);     //初始化串口 3
//  NVIC_UART3_Init(ENABLE,Priority_1);     //串口 3 中断使能,优先级
//  UART_Configuration(UART4, &COMx_InitStructure);     //初始化串口 4
//  NVIC_UART4_Init(ENABLE,Priority_1);     //串口 4 中断使能,优先级

    //UART1_SW_P30_P31,UART1_SW_P36_P37,UART1_SW_P16_P17,UART1_SW_P43_P44
    UART1_SW(UART1_SW_P30_P31);
    UART2_SW(UART2_SW_P46_P47);         //UART2_SW_P10_P11,UART2_SW_P46_P47
    UART3_SW(UART3_SW_P00_P01);         //UART3_SW_P00_P01,UART3_SW_P50_P51
    UART4_SW(UART4_SW_P02_P03);         //UART4_SW_P02_P03,UART4_SW_P52_P53
}
```

对 DMA 进行设置:

```
void DMA_config(void)
{
    DMA_UART_InitTypeDef        DMA_UART_InitStructure;   //结构定义

    DMA_UART_InitStructure.DMA_TX_Length = 255;         //DMA 传输总字节数  (0~65535) + 1
    DMA_UART_InitStructure.DMA_TX_Buffer = (u16)DmaBuffer; //发送数据存储地址
    DMA_UART_InitStructure.DMA_RX_Length = 255;         //DMA 传输总字节数  (0~65535) + 1
    DMA_UART_InitStructure.DMA_RX_Buffer = (u16)DmaBuffer; //接收数据存储地址
    DMA_UART_InitStructure.DMA_TX_Enable = ENABLE;    //DMA TX 使能   ENABLE,DISABLE
    DMA_UART_InitStructure.DMA_RX_Enable = ENABLE;    //DMA RX 使能   ENABLE,DISABLE
//  DMA_UART_Inilize(UART1, &DMA_UART_InitStructure);  //初始化串口 1
    DMA_UART_Inilize(UART2, &DMA_UART_InitStructure);  //初始化串口 2
//  DMA_UART_Inilize(UART3, &DMA_UART_InitStructure);  //初始化串口 3
//  DMA_UART_Inilize(UART4, &DMA_UART_InitStructure);  //初始化串口 4

//  NVIC_DMA_UART1_Tx_Init(ENABLE,Priority_0,Priority_0);  //串口 1 发送中断使能,优先级,总线优先级
//  NVIC_DMA_UART1_Rx_Init(ENABLE,Priority_0,Priority_0);  //串口 1 接收中断使能,优先级,总线优先级
    NVIC_DMA_UART2_Tx_Init(ENABLE,Priority_0,Priority_0);  //串口 2 发送中断使能,优先级,总线优先级
    NVIC_DMA_UART2_Rx_Init(ENABLE,Priority_0,Priority_0);  //串口 2 接收中断使能,优先级,总线优先级
//  NVIC_DMA_UART3_Tx_Init(ENABLE,Priority_0,Priority_0);  //串口 3 发送中断使能,优先级,总线优先级
//  NVIC_DMA_UART3_Rx_Init(ENABLE,Priority_0,Priority_0);  //串口 3 接收中断使能,优先级,总线优先级
//  NVIC_DMA_UART4_Tx_Init(ENABLE,Priority_0,Priority_0);  //串口 4 发送中断使能,优先级,总线优先级
//  NVIC_DMA_UART4_Rx_Init(ENABLE,Priority_0,Priority_0);  //串口 4 接收中断使能,优先级,总线优先级

//  DMA_UR1R_CLRFIFO();     //清空串口 1 接收 DMA FIFO
    DMA_UR2R_CLRFIFO();     //清空串口 2 接收 DMA FIFO
//  DMA_UR3R_CLRFIFO();     //清空串口 3 接收 DMA FIFO
//  DMA_UR4R_CLRFIFO();     //清空串口 4 接收 DMA FIFO
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启

后可以不用再关闭):

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();     //扩展 SFR(XFR)访问使能
    CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、IO 口初始电平等:

```
    GPIO_config();
    UART_config();
    DMA_config();
    EA = 1;
```

通过串口 1 打印一串数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此判断程序运行情况。

```
printf("STC32G UART DMA Test Programme!\r\n");  //UART 发送一个字符串
```

设置变量默认值，然后启动数据交换

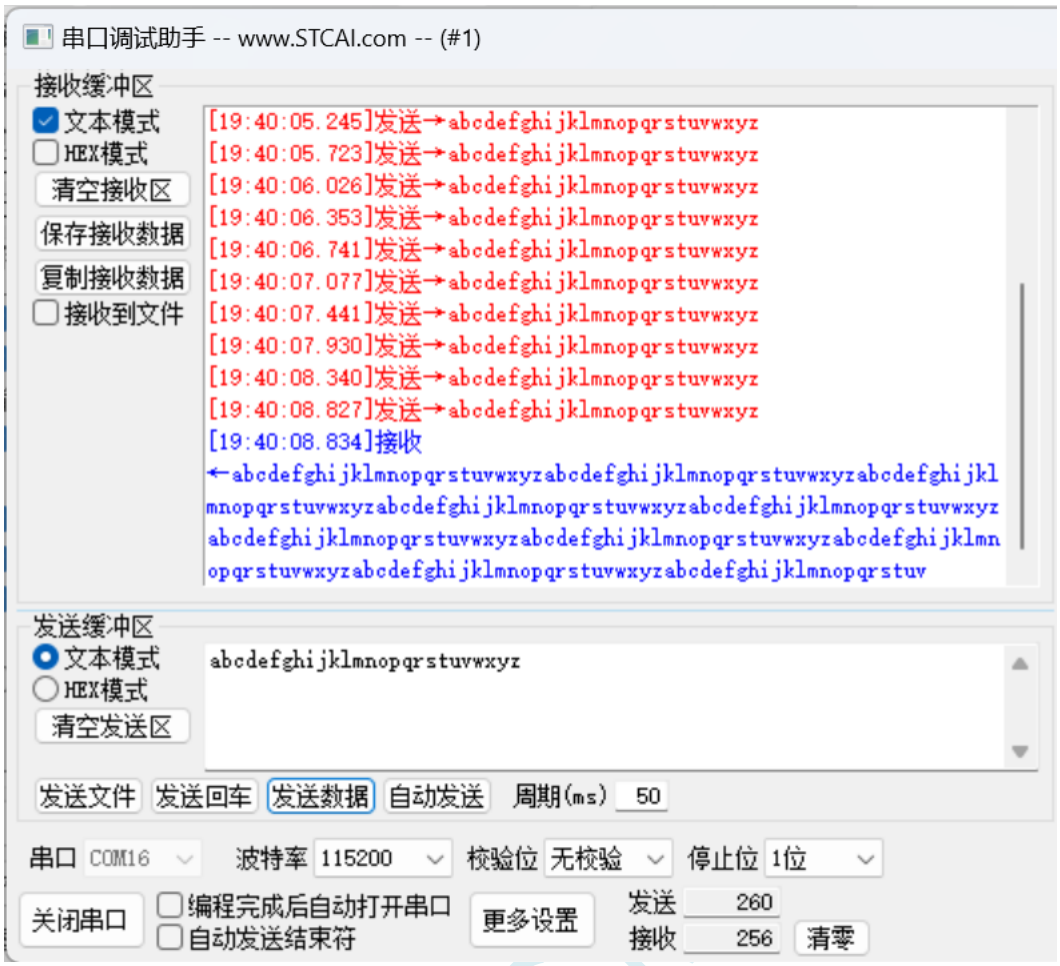
```
    DmaTx1Flag = 0;
    DmaRx1Flag = 0;
    DmaTx2Flag = 0;
    DmaRx2Flag = 0;
    DmaTx3Flag = 0;
    DmaRx3Flag = 0;
    DmaTx4Flag = 0;
    DmaRx4Flag = 0;
    for(i=0; i<256; i++)
    {
        DmaBuffer[i] = i;
    }
//  DMA_UR1T_TRIG();    //触发 UART1 发送功能
//  DMA_UR1R_TRIG();    //触发 UART1 接收功能
    DMA_UR2T_TRIG();    //触发 UART2 发送功能
    DMA_UR2R_TRIG();    //触发 UART2 接收功能
//  DMA_UR3T_TRIG();    //触发 UART3 发送功能
//  DMA_UR3R_TRIG();    //触发 UART3 接收功能
//  DMA_UR4T_TRIG();    //触发 UART4 发送功能
//  DMA_UR4R_TRIG();    //触发 UART4 接收功能
```

主循环里查询串口 DMA 收发是否完成，完成后启动 DMA 发送功能，将接收到的数据发送出来，并触发启动新一轮的接收。

```
    while (1)
    {
        if((DmaTx1Flag) && (DmaRx1Flag))   //通过标志位判断串口 DMA 发送/接收是否完成
        {
            DmaTx1Flag = 0;
            DmaRx1Flag = 0;
```

```
            DMA_UR1T_TRIG();    //重新触发 UART1 发送功能
            DMA_UR1R_TRIG();    //重新触发 UART1 接收功能
        }

        if((DmaTx2Flag) && (DmaRx2Flag))   //通过标志位判断串口 DMA 发送/接收是否完成
        {
            DmaTx2Flag = 0;
            DmaRx2Flag = 0;
            DMA_UR2T_TRIG();    //重新触发 UART2 发送功能
            DMA_UR2R_TRIG();    //重新触发 UART2 接收功能
        }

        if((DmaTx3Flag) && (DmaRx3Flag))   //通过标志位判断串口 DMA 发送/接收是否完成
        {
            DmaTx3Flag = 0;
            DmaRx3Flag = 0;
            DMA_UR3T_TRIG();    //重新触发 UART3 发送功能
            DMA_UR3R_TRIG();    //重新触发 UART3 接收功能
        }

        if((DmaTx4Flag) && (DmaRx4Flag))   //通过标志位判断串口 DMA 发送/接收是否完成
        {
            DmaTx4Flag = 0;
            DmaRx4Flag = 0;
            DMA_UR4T_TRIG();    //重新触发 UART4 发送功能
            DMA_UR4R_TRIG();    //重新触发 UART4 接收功能
        }
    }
```

通过串口助手显示程序运行结果:


串口调试助手 -- www.STCAI.com -- (#1)
接收缓冲区
文本模式
HEX模式
清空接收区
保存接收数据
复制接收数据
接收到文件
[19:40:05.245]发送→abcdefghijklmnopqrstuvwxyz
[19:40:05.723]发送→abcdefghijklmnopqrstuvwxyz
[19:40:06.026]发送→abcdefghijklmnopqrstuvwxyz
[19:40:06.353]发送→abcdefghijklmnopqrstuvwxyz
[19:40:06.741]发送→abcdefghijklmnopqrstuvwxyz
[19:40:07.077]发送→abcdefghijklmnopqrstuvwxyz
[19:40:07.441]发送→abcdefghijklmnopqrstuvwxyz
[19:40:07.930]发送→abcdefghijklmnopqrstuvwxyz
[19:40:08.340]发送→abcdefghijklmnopqrstuvwxyz
[19:40:08.827]发送→abcdefghijklmnopqrstuvwxyz
[19:40:08.834]接收
←abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijkl
mnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmn
opqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuv
发送缓冲区
文本模式
HEX模式
清空发送区
abcdefghijklmnopqrstuvwxyz
发送文件
发送回车
发送数据
自动发送
周期(ms)
50
串口
COM16
波特率
115200
校验位
无校验
停止位
1位
关闭串口
编程完成后自动打开串口
自动发送结束符
更多设置
发送
260
接收
256
清零

5.18 DMA-UART-M2M-SPI 综合演示

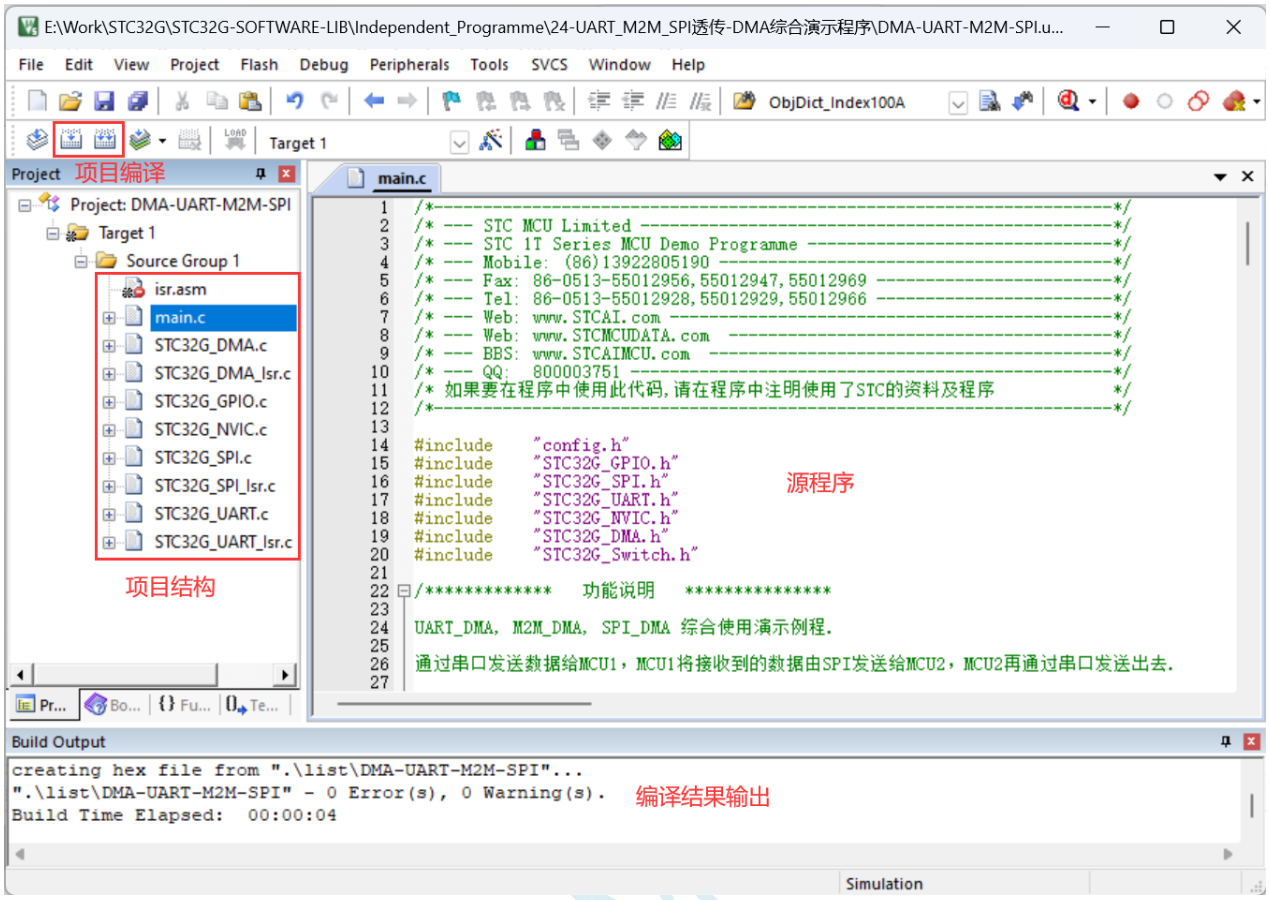
5.18.1 项目文件

打开 UART_M2M_SPI 透传-DMA 综合演示程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
DMA-UART-M2M-SPI(.uvopt .uvproj)	项目文件
Main.c	主函数文件
isr.asm	中断向量映射文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_DMA(.h .c)	DMA 模块初始化相关函数库
STC32G_DMA_Isr.c	DMA 模块中断函数库
STC32G_SPI(.h .c)	SPI 模块初始化相关函数库
STC32G_SPI_Isr.c	SPI 模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.18.2 程序框架

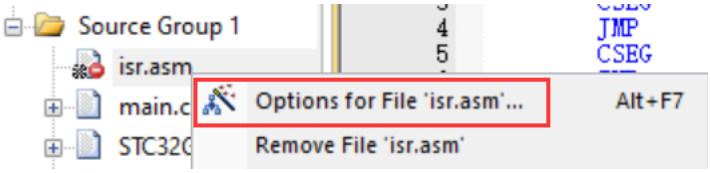
双击“DMA-UART-M2M-SPI.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



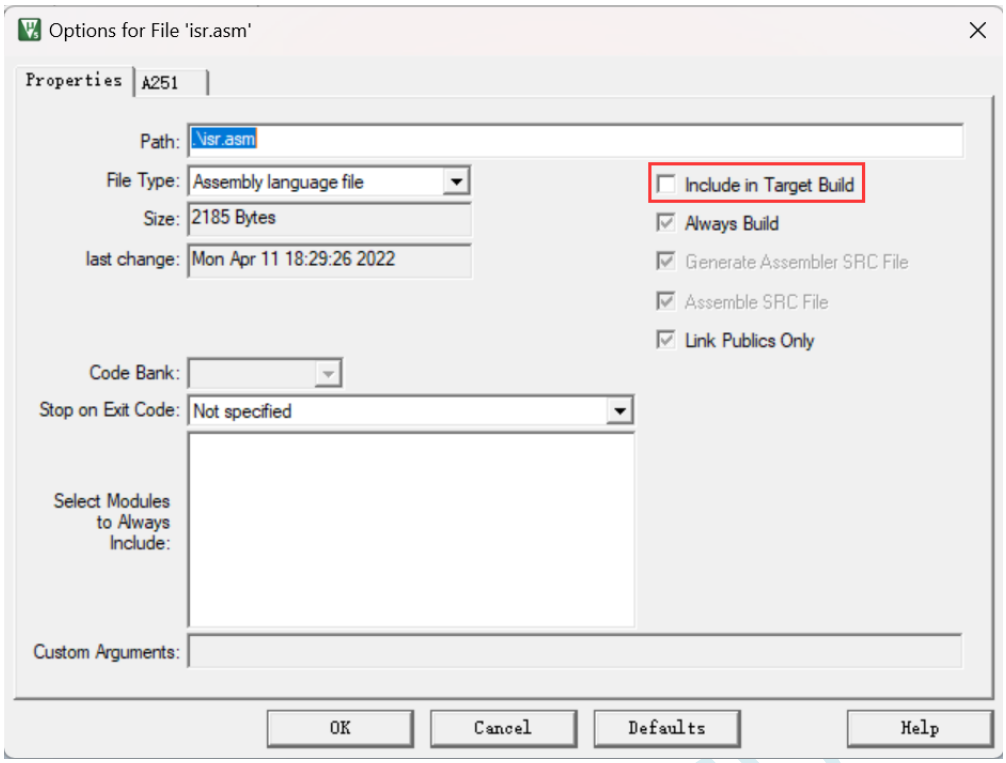
其中的“isr.asm”是中断向量映射文件，由于目前 keil 各版本的 C51 和 C251 编译器均只支持 32 个中断号（0~31），超过 31 的中断向量编译时会报错。暂时只能通过安装中断向量号拓展插件（详情参考说明书 3.3 章节 keil 中断向量号拓展插件使用说明），或者将中断映射到向量号 31 以内没有使用的中断地址。“isr.asm”文件就是将中断向量号超过 31 的部分中断映射到 13 号中断向量地址。

例程使用的是通过安装 keil 中断向量号拓展插件的方法，所以设置“isr.asm”文件不参与编译，如果想用中断映射方式的话通过修改设置让“isr.asm”文件参与编译，并且将中断函数内容添加到 13 号中断函数里即可。

设置“isr.asm”文件参与编译方法，通过右键点击“isr.asm”文件选择“Options for File ‘isr.asm’...”：



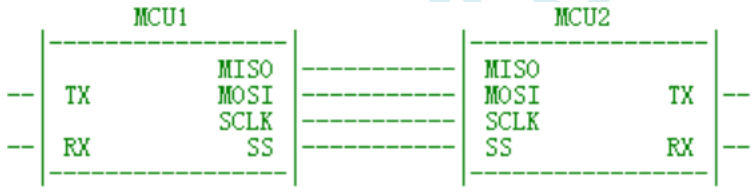
在弹出框里勾选“Include in Target Build”：



该例程通过串口发送数据给 MCU1，MCU1 将接收到的数据由 SPI 发送给 MCU2，MCU2 再通过串口发送出去。通过串口发送数据给 MCU2，MCU2 将接收到的数据由 SPI 发送给 MCU1，MCU1 再通过串口发送出去。

MCU1/MCU2: UART 接收 -> UART Rx DMA -> M2M -> SPI Tx DMA -> SPI 发送

MCU2/MCU1: SPI 接收 -> SPI Rx DMA -> M2M -> UART Tx DMA -> UART 发送



初始化时对所有用到的 IO 口进行模式配置：

```
void GPIO_config(void)
{
    P2_MODE_IO_PU(GPIO_Pin_All);        //P2 设置为准双向口
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);    //P4.6,P4.7 设置为准双向口

    UART2_SW(UART2_SW_P46_P47);     //UART2_SW_P10_P11,UART2_SW_P46_P47
    //SPI_P54_P13_P14_P15,SPI_P22_P23_P24_P25,SPI_P54_P40_P41_P43,SPI_P35_P34_P33_P32
    SPI_SW(SPI_P22_P23_P24_P25);        //SPI 通道选择
    SPI_SS_2 = 1;       //设置 SS 脚默认电平
}
```

对 UART 接口进行设置：

```
void UART_config(void)
```

```
{
    COMx_InitDefine     COMx_InitStructure;             //结构定义
    //模式,   UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode        = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;     //波特率,     110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;       //接收允许,   ENABLE 或 DISABLE
    UART_Configuration(UART2, &COMx_InitStructure);     //初始化串口 2
    NVIC_UART2_Init(ENABLE,Priority_1);     //串口 2 中断使能,优先级
}
```

对 SPI 接口进行设置:

```
void SPI_config(void)
{
    SPI_InitTypeDef     SPI_InitStructure;

    SPI_InitStructure.SPI_Enable    = ENABLE;           //SPI 启动    ENABLE, DISABLE
    SPI_InitStructure.SPI_SSIG      = DISABLE;          //片选位      ENABLE, DISABLE
    SPI_InitStructure.SPI_FirstBit  = SPI_MSB;          //移位方向    SPI_MSB, SPI_LSB
    SPI_InitStructure.SPI_Mode      = SPI_Mode_Slave;   //主从选择    SPI_Mode_Master, SPI_Mode_Slave
    SPI_InitStructure.SPI_CPOL      = SPI_CPOL_Low;     //时钟相位    SPI_CPOL_High,     SPI_CPOL_Low
    SPI_InitStructure.SPI_CPHA      = SPI_CPHA_1Edge;   //数据边沿    SPI_CPHA_1Edge,    SPI_CPHA_2Edge
    //SPI 速度 SPI_Speed_4, SPI_Speed_8, SPI_Speed_16, SPI_Speed_2
    SPI_InitStructure.SPI_Speed     = SPI_Speed_16;
    SPI_Init(&SPI_InitStructure);
    NVIC_SPI_Init(DISABLE,Priority_0);      //中断使能,优先级
}
```

对 DMA 进行设置:

```
void DMA_config(void)
{
    DMA_M2M_InitTypeDef     DMA_M2M_InitStructure;      //结构定义
    DMA_SPI_InitTypeDef     DMA_SPI_InitStructure;      //结构定义
    DMA_UART_InitTypeDef    DMA_UART_InitStructure;     //结构定义

    //-----------------------------------------
    DMA_UART_InitStructure.DMA_TX_Length = BUF_LENGTH;  //DMA 传输总字节数   (0~65535) + 1
    DMA_UART_InitStructure.DMA_TX_Buffer = (u16)UartTxBuffer;   //发送数据存储地址
    DMA_UART_InitStructure.DMA_RX_Length = BUF_LENGTH;  //DMA 传输总字节数   (0~65535) + 1
    DMA_UART_InitStructure.DMA_RX_Buffer = (u16)UartRxBuffer;   //接收数据存储地址
    DMA_UART_InitStructure.DMA_TX_Enable = ENABLE;      //DMA 使能       ENABLE,DISABLE
    DMA_UART_InitStructure.DMA_RX_Enable = ENABLE;      //DMA 使能       ENABLE,DISABLE
    DMA_UART_Inilize(UART2, &DMA_UART_InitStructure);   //初始化
```

```
    NVIC_DMA_UART2_Tx_Init(ENABLE,Priority_0,Priority_0);      //中断使能,优先级,总线优先级
    NVIC_DMA_UART2_Rx_Init(ENABLE,Priority_0,Priority_0);      //中断使能,优先级,总线优先级
    DMA_UR2R_CLRFIFO();       //清空 DMA FIFO
    DMA_UR2R_TRIG();      //触发 UART 接收功能

    //-----------------------------------------
    DMA_M2M_InitStructure.DMA_Enable = ENABLE;              //DMA 使能        ENABLE,DISABLE
    DMA_M2M_InitStructure.DMA_Length = BUF_LENGTH;          //DMA 传输总字节数   (0~65535) + 1
    DMA_M2M_InitStructure.DMA_Tx_Buffer = (u16)UartRxBuffer;     //发送数据存储地址
    DMA_M2M_InitStructure.DMA_Rx_Buffer = (u16)SpiTxBuffer; //接收数据存储地址
    //数据源地址改变方向       M2M_ADDR_INC,M2M_ADDR_DEC
    DMA_M2M_InitStructure.DMA_SRC_Dir = M2M_ADDR_INC;
    //数据目标地址改变方向    M2M_ADDR_INC,M2M_ADDR_DEC
    DMA_M2M_InitStructure.DMA_DEST_Dir = M2M_ADDR_INC;
    DMA_M2M_Inilize(&DMA_M2M_InitStructure);            //初始化
    NVIC_DMA_M2M_Init(ENABLE,Priority_0,Priority_0);       //中断使能,优先级,总线优先级

    //-----------------------------------------
    DMA_SPI_InitStructure.DMA_Enable = DISABLE;             //DMA 使能   ENABLE,DISABLE
    DMA_SPI_InitStructure.DMA_Tx_Enable = ENABLE;           //DMA 发送数据使能
    DMA_SPI_InitStructure.DMA_Rx_Enable = ENABLE;           //DMA 接收数据使能
    DMA_SPI_InitStructure.DMA_Length = BUF_LENGTH;          //DMA 传输总字节数   (0~65535) + 1
    DMA_SPI_InitStructure.DMA_Tx_Buffer = (u16)SpiTxBuffer;     //发送数据存储地址
    DMA_SPI_InitStructure.DMA_Rx_Buffer = (u16)SpiRxBuffer;     //接收数据存储地址
    //自动控制 SS 脚选择   SPI_SS_P12,SPI_SS_P22,SPI_SS_P74,SPI_SS_P35
    DMA_SPI_InitStructure.DMA_SS_Sel = SPI_SS_P22;
    DMA_SPI_InitStructure.DMA_AUTO_SS = DISABLE;            //自动控制 SS 脚使能
    DMA_SPI_Inilize(&DMA_SPI_InitStructure);        //初始化
    //bit7 1:使能  SPI_DMA, bit5 1:开始  SPI_DMA  从机模式, bit0 1:清除  SPI_DMA FIFO
    SET_DMA_SPI_CR(DMA_ENABLE | SPI_TRIG_S | CLR_FIFO);
    NVIC_DMA_SPI_Init(ENABLE,Priority_0,Priority_0);            //中断使能,优先级,总线优先级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）：

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();       //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、IO 口初始电平等：

```
    GPIO_config();
    UART_config();
    SPI_config();
    DMA_config();
    EA = 1;
```

通过串口 1 打印一串数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此判断程序运行情况。

```
printf("STC32G UART-DMA-SPI 互为主从透传程序. \r\n");   //UART 发送一个字符串
```

主循环里查询串口 DMA 接收是否完成，完成后启动 DMA 存储器数据交换功能，将串口接收 DMA 缓冲区的数据转移到 SPI 发送 DMA 缓冲区，然后启动 SPI 的发送 DMA 进行发送。如果 SPI 的 DMA 接收完成的话，将 SPI 接收 DMA 缓冲区内容转到串口发送 DMA 缓冲区，然后启动串口的 DMA 发送功能。

```
while (1)
{
    //UART 接收  -> UART DMA -> SPI DMA -> SPI 发送
```

1. 判断串口的 DMA 接收是否完成，完成的话启动 DMA 存储器数据交换功能，将串口接收 DMA 缓冲区的数据转移到 SPI 发送 DMA 缓冲区。

```
    if(DmaRx2Flag)
    {
        DmaRx2Flag = 0;
        u2sFlag = 1;
        M2M_UART_SPI((u16)UartRxBuffer,(u16)SpiTxBuffer);     //UART Memory -> SPI Memory
    }
```

2. 判断串口接收 DMA 缓冲区的数据转移到 SPI 发送 DMA 缓冲区是否完成，完成后继续开启串口接收 DMA，然后将 SPI 设置为主机模式并触发 SPI 的 DMA 发送功能。

```
    if(SpiSendFlag)
    {
        SpiSendFlag = 0;
        UART_DMA_Rx();          //UART Recive Continue
        SPI_DMA_Master();       //SPI Send Memory
    }
```

3. 判断 SPI 的 DMA 是否发送完成，发送完成后将 SPI 转成从机模式，并触发从机 DMA 接收状态。

```
    if(SpiTxFlag)
    {
        SpiTxFlag = 0;
        SPI_DMA_Slave();        //SPI Slave Mode
    }

    //SPI 接收  -> SPI DMA -> UART DMA -> UART 发送
```

4. 判断 SPI 的 DMA 接收是否完成，完成的话启动 DMA 存储器数据交换功能，将 SPI 接收 DMA 缓冲区的数据转移到串口发送 DMA 缓冲区。

```
    if(SpiRxFlag)
    {
        SpiRxFlag = 0;
        s2uFlag = 1;
        M2M_SPI_UART((u16)SpiRxBuffer, (u16)UartTxBuffer);    //SPI Memory -> UART Memory
    }
```

5. 判断 SPI 接收 DMA 缓冲区的数据转移到串口发送 DMA 缓冲区是否完成，完成后重新触发 SPI 从机模式 DMA 接收状态，并启动串口 DMA 发送功能，将串口发送 DMA 缓冲区内容发送出去。

```
        if(UartSendFlag)
        {
            UartSendFlag = 0;
            SPI_DMA_Slave();        //SPI Slave Mode
            UART_DMA_Tx();          //UART Send Memory
        }
    }
```

5.19 DMA-LCM 液晶屏接口

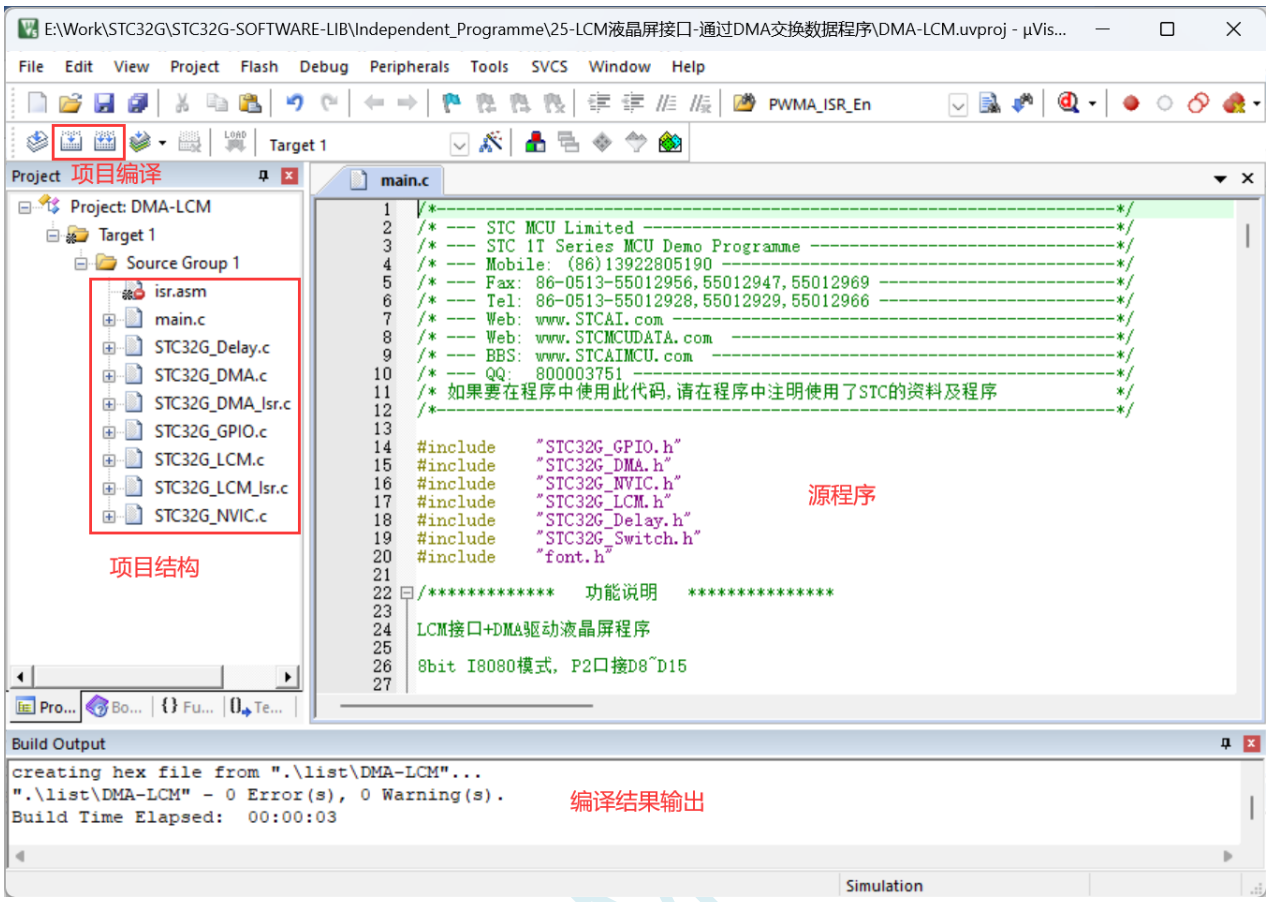
5.19.1 项目文件

打开 LCM 液晶屏接口-通过 DMA 交换数据程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
DMA-LCM(.uvopt .uvproj)	项目文件
Main.c	主函数文件
isr.asm	中断向量映射文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_Delay (.h .c)	标准延时函数
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_DMA(.h .c)	DMA 模块初始化相关函数库
STC32G_DMA_Isr.c	DMA 模块中断函数库
STC32G_LCM(.h .c)	LCM 接口初始化相关函数库
STC32G_LCM_Isr.c	LCM 接口中断函数库
STC32G_UART (.h)	UART 模块初始化及应用相关函数库头文件
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件
font.h	字库文件

5.19.2 程序框架

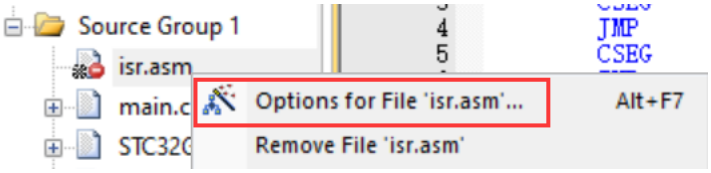
双击“DMA-LCM.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



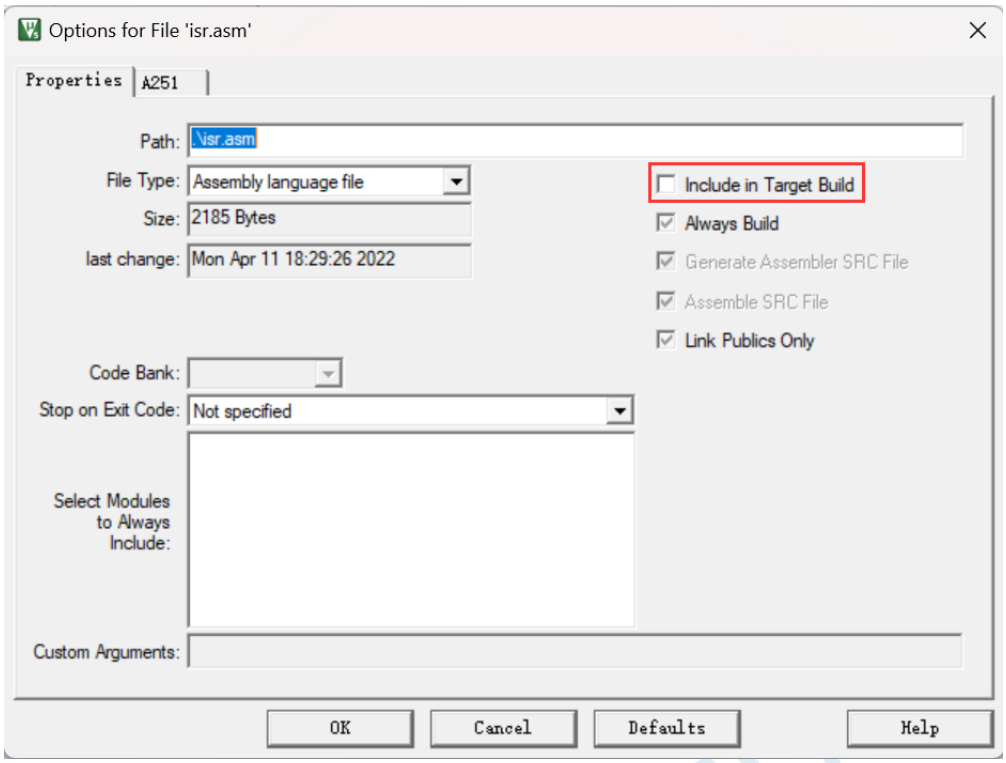
其中的“isr.asm”是中断向量映射文件，由于目前 keil 各版本的 C51 和 C251 编译器均只支持 32 个中断号（0~31），超过 31 的中断向量编译时会报错。暂时只能通过安装中断向量号拓展插件（详情参考说明书 3.3 章节 keil 中断向量号拓展插件使用说明），或者将中断映射到向量号 31 以内没有使用的中断地址。“isr.asm”文件就是将中断向量号超过 31 的部分中断映射到 13 号中断向量地址。

例程使用的是通过安装 keil 中断向量号拓展插件的方法，所以设置“isr.asm”文件不参与编译，如果想用中断映射方式的话通过修改设置让“isr.asm”文件参与编译，并且将中断函数内容添加到 13 号中断函数里即可。

设置“isr.asm”文件参与编译方法，通过右键点击“isr.asm”文件选择“Options for File ‘isr.asm’...”：



在弹出框里勾选“Include in Target Build”：



LCM 硬件接口可用于驱动 I8080 接口和 M6800 接口彩屏，支持 8 位和 16 位数据宽度。该例程通过 LCM 的 DMA 功能对液晶屏进行读写操作。

初始化时对所有用到的 IO 口进行模式配置：

```
void GPIO_config(void)
{
    P3_MODE_OUT_PP(GPIO_Pin_4);     //P3.4 口设置成推挽输出
//P4.2~P4.5 设置成推挽输出
    P4_MODE_OUT_PP(GPIO_Pin_2 | GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5);
//LCM_CTRL_P45_P44_P42,LCM_CTRL_P45_P37_P36,LCM_CTRL_P40_P44_P42,LCM_CTRL_P40_P37_P36
    LCM_CTRL_SW(LCM_CTRL_P45_P44_P42);     //设置控制脚通道
//8 位模式：LCM_D8_NA_P2,LCM_D8_NA_P6
//16 位模式：LCM_D16_P2_P0,LCM_D16_P6_P2,LCM_D16_P2_P7P4,LCM_D16_P6_P7
    LCM_DATA_SW(LCM_D8_NA_P6);          //设置数据脚通道
}
```

对 LCM 接口进行设置：

```
void LCM_config(void)
{
    LCM_InitTypeDef     LCM_InitStructure;     //结构定义

    LCM_InitStructure.LCM_Enable = ENABLE;     //LCM 接口使能   ENABLE,DISABLE
    LCM_InitStructure.LCM_Mode = MODE_I8080;     //LCM 接口模式   MODE_I8080,MODE_M6800
    LCM_InitStructure.LCM_Bit_Wide = BIT_WIDE_8;     //LCM 数据宽度   BIT_WIDE_8,BIT_WIDE_16
    LCM_InitStructure.LCM_Setup_Time = 2;          //LCM 通信数据建立时间   0~7
```

```
    LCM_InitStructure.LCM_Hold_Time = 1;            //LCM 通信数据保持时间  0~3
    LCM_Inilize(&LCM_InitStructure);          //初始化 LCM 接口
    NVIC_LCM_Init(ENABLE,Priority_0);         //中断使能,优先级
}
```

对 DMA 进行设置:

```
void DMA_config(void)
{
    DMA_LCM_InitTypeDef     DMA_LCM_InitStructure;          //结构定义

    DMA_LCM_InitStructure.DMA_Enable = ENABLE;              //DMA 使能  ENABLE,DISABLE
    //DMA 传输总字节数  (0~65535) + 1, 不要超过芯片 xdata 空间上限
    DMA_LCM_InitStructure.DMA_Length = DMA_AMT_LEN;
    DMA_LCM_InitStructure.DMA_Tx_Buffer = (u16)Color;       //发送数据存储地址
    DMA_LCM_InitStructure.DMA_Rx_Buffer = (u16)Buffer;      //接收数据存储地址
    DMA_LCM_Inilize(&DMA_LCM_InitStructure);            //初始化
    NVIC_DMA_LCM_Init(ENABLE,Priority_0,Priority_0);    //中断使能,优先级, 总线优先级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;  //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();     //扩展 SFR(XFR)访问使能
    CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、IO 口初始电平等:

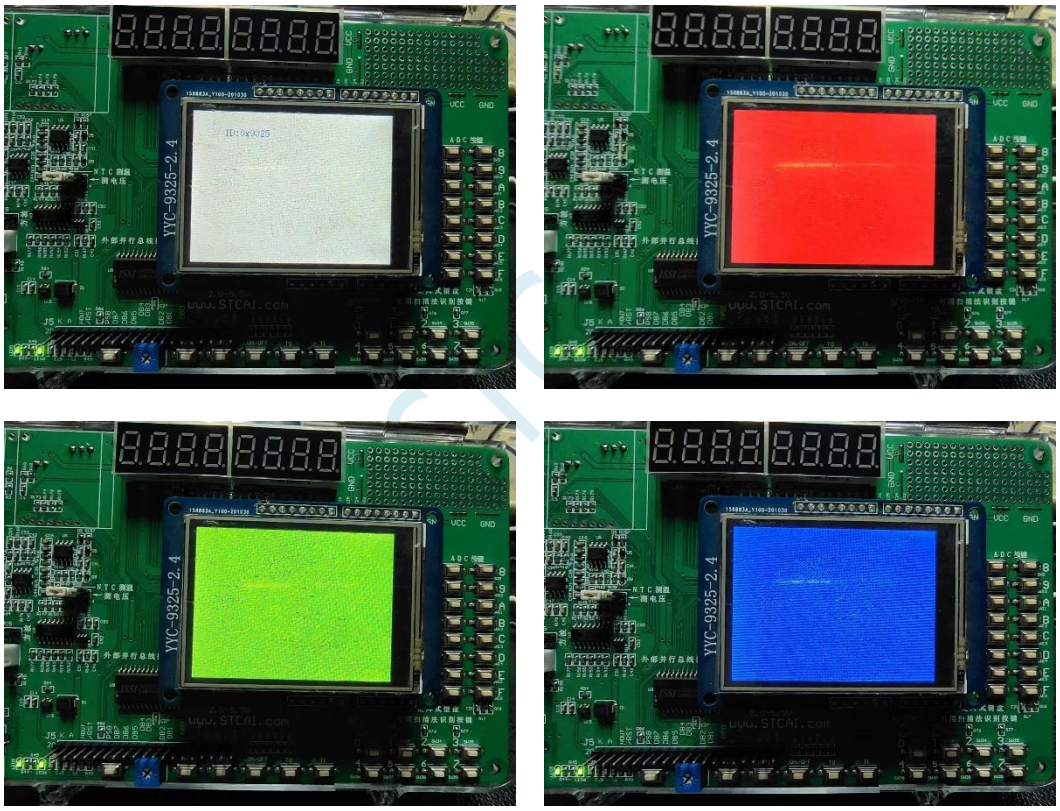
```
    GPIO_config();
    LCM_config();
    DMA_config();
    EA = 1;
    LCD_Init();     //LCM 初始化
```

主循环里读取液晶屏的 ID 号在屏幕上显示，并交替刷新显示不同颜色。

```
    while (1)
    {
        Test_Color();
    }

void Test_Color(void)
{
    u16 lcd_id;
    u8 buf[10] = {0};
    //设置刷新屏幕的 x 轴起始地址, y 轴起始地址, x 轴结束地址, y 轴结束地址, 画笔颜色
    LCD_Fill(0,0,LCD_W,LCD_H,WHITE);   //屏幕填充白色
    while(!LCD_CS);    //等待填充完成
```

```
    SET_LCM_DMA_LEN(0x01);        //设置 DMA 传输数据长度位为 2，n+1
    lcd_id = LCD_Read_ID();       //读取液晶屏 ID 号
    sprintf((char *)buf,"ID:0x%x",lcd_id);   //格式化显示内容
    Show_Str(50,25,BLUE,YELLOW,buf,16,1);  //在屏幕指定位置，指定颜色进行显示
    SET_LCM_DMA_LEN(DMA_AMT_LEN);     //设置 DMA 传输数据长度位为 DMA_AMT_LEN+1

    delay_ms(800);     //延时 800 毫秒，方便观察
    LCD_Fill(0,0,LCD_W,LCD_H,RED);       //填充显示红色画面
    delay_ms(800);     //延时 800 毫秒，方便观察
    LCD_Fill(0,0,LCD_W,LCD_H,GREEN);  //填充显示绿色画面
    delay_ms(800);     //延时 800 毫秒，方便观察
    LCD_Fill(0,0,LCD_W,LCD_H,BLUE);      //填充显示蓝色画面
    delay_ms(800);     //延时 800 毫秒，方便观察
}
```

测试效果如图所示：



5.20 DMA-I2C 接口

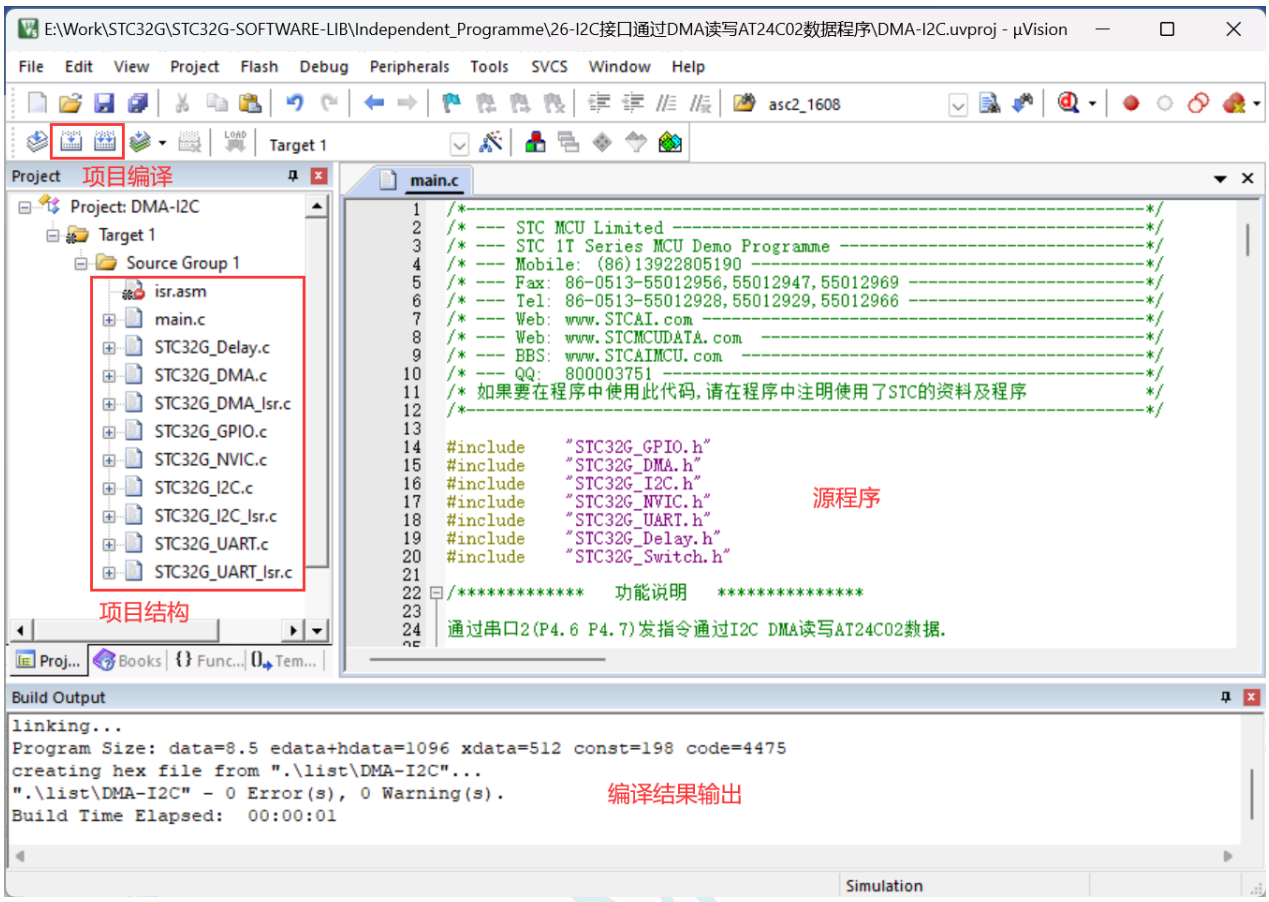
5.20.1 项目文件

打开 I2C 接口通过 DMA 读写 AT24C02 数据程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
DMA-I2C(.uvopt .uvproj)	项目文件
Main.c	主函数文件
isr.asm	中断向量映射文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_Delay (.h .c)	标准延时函数
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_DMA(.h .c)	DMA 模块初始化相关函数库
STC32G_DMA_Isr.c	DMA 模块中断函数库
STC32G_I2C(.h .c)	I2C 接口初始化相关函数库
STC32G_I2C_Isr.c	I2C 接口中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 接口中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.20.2 程序框架

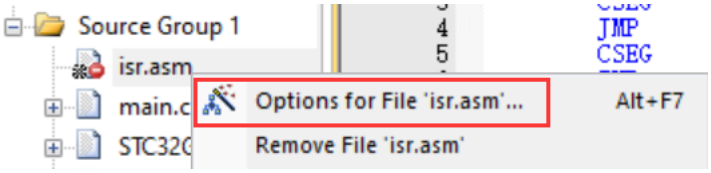
双击“DMA-I2C.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



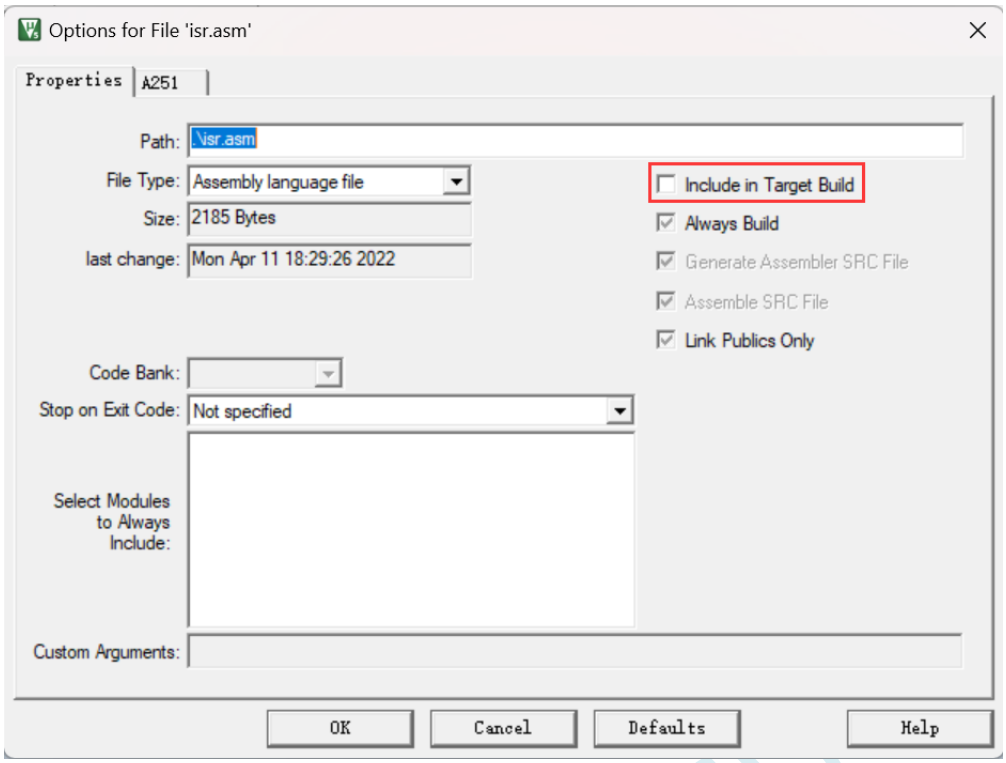
其中的“isr.asm”是中断向量映射文件，由于目前 keil 各版本的 C51 和 C251 编译器均只支持 32 个中断号（0~31），超过 31 的中断向量编译时会报错。暂时只能通过安装中断向量号拓展插件（详情参考说明书 3.3 章节 keil 中断向量号拓展插件使用说明），或者将中断映射到向量号 31 以内没有使用的中断地址。“isr.asm”文件就是将中断向量号超过 31 的部分中断映射到 13 号中断向量地址。

例程使用的是通过安装 keil 中断向量号拓展插件的方法，所以设置“isr.asm”文件不参与编译，如果想用中断映射方式的话通过修改设置让“isr.asm”文件参与编译，并且将中断函数内容添加到 13 号中断函数里即可。

设置“isr.asm”文件参与编译方法，通过右键点击“isr.asm”文件选择“Options for File ‘isr.asm’...”：



在弹出框里勾选“Include in Target Build”：



该例程通过通过串口 2(P4.6 P4.7)发指令通过 I2C DMA 读写 AT24C02 数据。默认波特率: 115200,N,8,1。
串口命令设置: (命令字母不区分大小写)

W 0x12 1234567890 --> 写入操作 十六进制地址 写入内容.
R 0x12 10 --> 读出操作 十六进制地址 读出字节数.

定义 EEPROM 设备读写地址, 以及缓冲区长度:

```
#define SLAW      0xA0
#define SLAR      0xA1
#define EE_BUF_LENGTH          255
```

初始化时对所有用到的 IO 口进行模式配置, 并切换接口通道:

```
void GPIO_config(void)
{
    P2_MODE_IO_PU(GPIO_Pin_4 | GPIO_Pin_5);        //P2.4,P2.5 设置为准双向口
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);        //P4.6,P4.7 设置为准双向口
    I2C_SW(I2C_P24_P25);                  //I2C_P14_P15,I2C_P24_P25,I2C_P76_P77,I2C_P33_P32
    UART2_SW(UART2_SW_P46_P47);           //UART2_SW_P10_P11,UART2_SW_P46_P47
}
```

对 I2C 接口进行设置:

```
void I2C_config(void)
{
    I2C_InitTypeDef          I2C_InitStructure;

    //I2C 主从模式选择   I2C_Mode_Master, I2C_Mode_Slave
```

```
    I2C_InitStructure.I2C_Mode          = I2C_Mode_Master;
    I2C_InitStructure.I2C_Enable        = ENABLE;           //I2C 功能使能,    ENABLE, DISABLE
    I2C_InitStructure.I2C_MS_WDTA       = DISABLE;          //主机使能自动发送,  ENABLE, DISABLE
    I2C_InitStructure.I2C_Speed         = 63;               //总线速度=Fosc/2/(Speed*2+4),    0~63
    I2C_Init(&I2C_InitStructure);
    NVIC_I2C_Init(I2C_Mode_Master,DISABLE,Priority_0);      //主从模式, 中断使能, 优先级
}
```

对 UART 接口进行设置:

```
void UART_config(void)
{
    COMx_InitDefine         COMx_InitStructure;                 //结构定义
    //模式,    UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode          = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;         //波特率,       110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;           //接收允许,    ENABLE 或 DISABLE
    UART_Configuration(UART2, &COMx_InitStructure);         //初始化串口 2
    NVIC_UART2_Init(ENABLE,Priority_1);         //串口 2 中断使能,优先级
}
```

对 DMA 进行设置:

```
void DMA_config(void)
{
    DMA_I2C_InitTypeDef         DMA_I2C_InitStructure;      //结构定义

    DMA_I2C_InitStructure.DMA_TX_Length = EE_BUF_LENGTH; //DMA 传输总字节数   (0~65535) + 1
    DMA_I2C_InitStructure.DMA_TX_Buffer = (u16)I2cTxBuffer;   //发送数据存储地址
    DMA_I2C_InitStructure.DMA_RX_Length = EE_BUF_LENGTH; //DMA 传输总字节数   (0~65535) + 1
    DMA_I2C_InitStructure.DMA_RX_Buffer = (u16)I2cRxBuffer;   //接收数据存储地址
    DMA_I2C_InitStructure.DMA_TX_Enable = ENABLE;       //DMA 使能   ENABLE,DISABLE
    DMA_I2C_InitStructure.DMA_RX_Enable = ENABLE;       //DMA 使能   ENABLE,DISABLE
    DMA_I2C_Inilize(&DMA_I2C_InitStructure);      //初始化

    NVIC_DMA_I2CT_Init(ENABLE,Priority_0,Priority_0);       //中断使能,优先级, 总线优先级
    NVIC_DMA_I2CR_Init(ENABLE,Priority_0,Priority_0);       //中断使能,优先级, 总线优先级
    DMA_I2CR_CLRFIFO();         //清空  DMA FIFO
}
```

主函数起始位置推荐先执行以下几条指令, 提升芯片的性能, 设置扩展寄存器访问使能 (起始位置开启后可以不用再关闭):

```
    WTST = 0;   //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();       //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数、IO 口初始电平等:

```
GPIO_config();
I2C_config();
DMA_config();
UART_config();
EA = 1;
```

通过串口打印命令提示数据出来，使用 USB 转串口工具连接芯片跟电脑，从电脑的串口助手就能收到芯片打印的数据，可以依此了解如何发送指令对 I2C EEPROM 进行写入、读取操作。

```
printf("命令设置:\r\n");
printf("W 0x12 1234567890 --> 写入操作　十六进制地址　写入内容\r\n");
printf("R 0x12 10 --> 读出操作　十六进制地址　读出字节内容\r\n");
```

主循环里通过软件超时机制，判断接收到一串 UART 数据后进行解析，判断命令类型，执行相应的操作。

```
while (1)
{
    delay_ms(1);        //每毫秒执行一次主循环
    if(COM2.RX_TimeOut > 0)        //通过超时计数判断是否接收完成串口数据
    {
        if(--COM2.RX_TimeOut == 0)
        {
//          printf("收到内容如下:  ");
//          //把收到的数据原样返回,用于测试
//          for(i=0; i<COM2.RX_Cnt; i++)    printf("%c", RX2_Buffer[i]);
//          printf("\r\n");
```

1. 接收到一串数据后，通过数据长度、格式判断是否为有效命令

```
status = 0xff;  //状态给一个非 0 值
if((COM2.RX_Cnt >= 8) && (RX2_Buffer[1] == ' ')) //最短命令为 8 个字节
{
```

2. 转换指令数据为大写字符，方便后续判断

```
for(i=0; i<6; i++)
{
    if((RX2_Buffer[i] >= 'a') && (RX2_Buffer[i] <= 'z'))
        RX2_Buffer[i] = RX2_Buffer[i] - 'a' + 'A';  //小写转大写
}
```

3. 读取地址数据，判断地址是否有效（可根据需要设置地址范围，例程不进行限制）

```
addr = I2cGetAddress();
//if(addr <= 255)     //限制地址范围
{
```

4. 判断是否为写入指令，是的话计算需要写入的数据长度，并将数据转到 I2C 的 DMA 缓冲区，然后执行 I2C 写入操作。写入 EEPROM 完成后将对应地址的内容根据长度进行读取，将读写信息通过串口打印出来，依此可以判断操作是否正确。

```
if((RX2_Buffer[0] == 'W')&& (RX2_Buffer[6] == ' '))    //写入 N 个字节
{
```

```
                    j = COM2.RX_Cnt - 7;  //计算命令后面需要接入的数据长度
                    if(j > EE_BUF_LENGTH)   j = EE_BUF_LENGTH; //越界检测

                    for(i=0; i<j; i++)   I2cTxBuffer[i+2] = RX2_Buffer[i+7];
                    WriteNbyte(addr, j);        //写 N 个字节
                    printf("已写入%d 字节内容!\r\n",j);
                    delay_ms(5);

                    ReadNbyte(addr, j);
                    printf("读出%d 个字节内容如下：\r\n",j);
                    for(i=0; i<j; i++)       printf("%c", I2cRxBuffer[i]);
                    printf("\r\n");

                    status = 0; //命令正确
                }
```

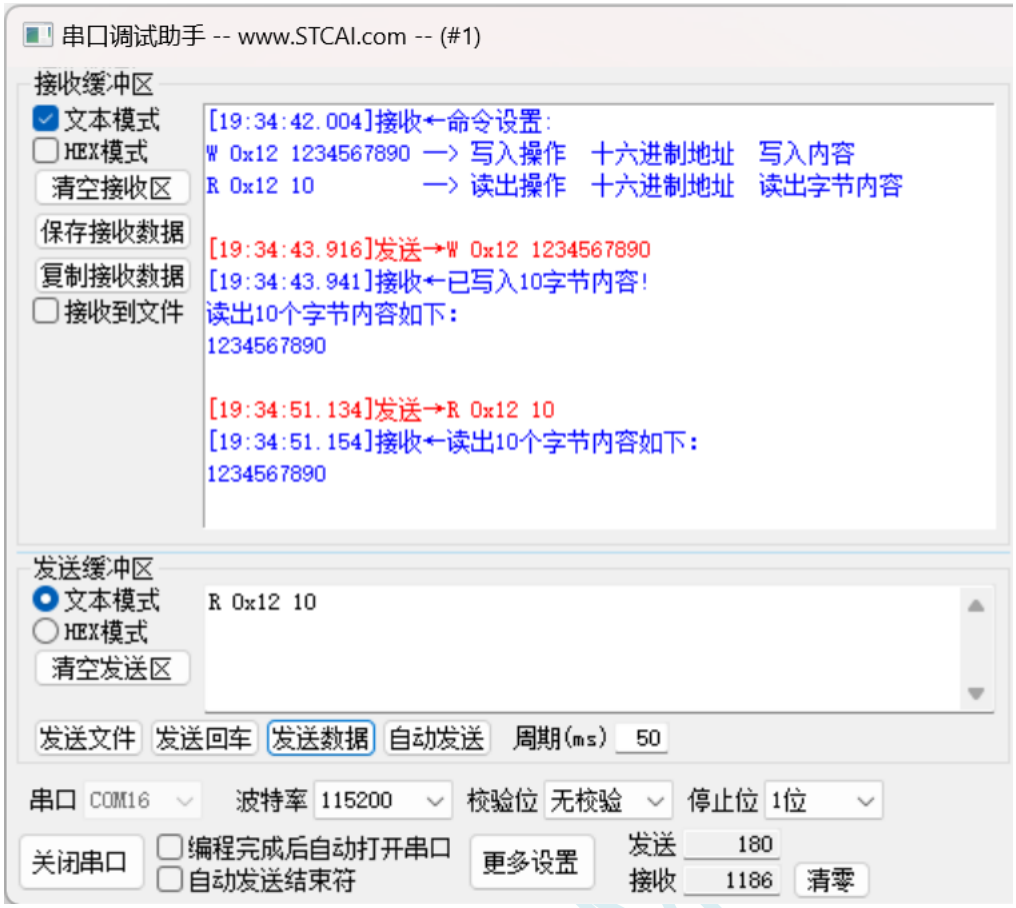
5. 判断是否读取指令，是的话将对应地址的内容根据长度进行读取，将读取数据通过串口打印出来，依此可以判断操作是否正确。

```
                else if((RX2_Buffer[0] == 'R') && (RX2_Buffer[6] == ' '))     //读出 N 个字节
                {
                    j = I2cGetDataLength();      //获取读取数据长度
                    if(j > EE_BUF_LENGTH)   j = EE_BUF_LENGTH; //越界检测
                    if(j > 0)
                    {
                        ReadNbyte(addr, j);
                        printf("读出%d 个字节内容如下：\r\n",j);
                        for(i=0; i<j; i++)       printf("%c", I2cRxBuffer[i]);
                        printf("\r\n");

                        status = 0; //命令正确
                    }
                }
            }
        }
```

6. 判断状态信息，如果不是有效读写命令的话，打印命令错误信息。

```
        if(status != 0) printf("命令错误！\r\n");
        COM2.RX_Cnt = 0;
    }
}
}
```

通过串口助手显示程序测试结果：


串口调试助手 -- www.STCAI.com -- (#1)
接收缓冲区
文本模式
HEX模式
清空接收区
保存接收数据
复制接收数据
接收到文件
[19:34:42.004]接收←命令设置:
W 0x12 1234567890 —> 写入操作 十六进制地址 写入内容
R 0x12 10 —> 读出操作 十六进制地址 读出字节内容
[19:34:43.916]发送→W 0x12 1234567890
[19:34:43.941]接收←已写入10字节内容!
读出10个字节内容如下:
1234567890
[19:34:51.134]发送→R 0x12 10
[19:34:51.154]接收←读出10个字节内容如下:
1234567890
发送缓冲区
文本模式
HEX模式
清空发送区
R 0x12 10
发送文件
发送回车
发送数据
自动发送
周期(ms) 50
串口 COM16
波特率 115200
校验位 无校验
停止位 1位
关闭串口
编程完成后自动打开串口
自动发送结束符
更多设置
发送 180
接收 1186
清零

5.21 CAN 总线接口

5.21.1 项目文件

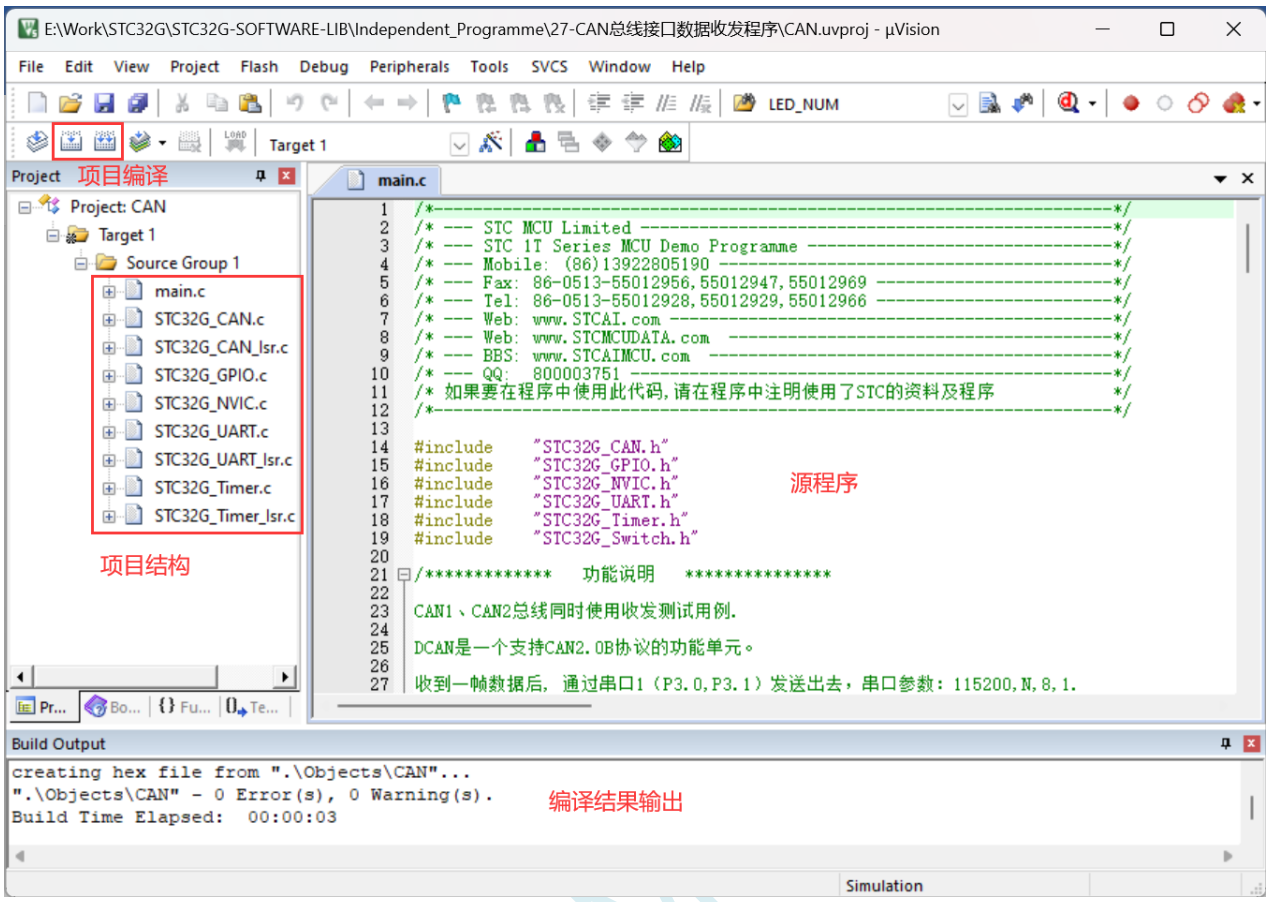
打开 CAN 总线接口数据收发程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
CAN(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_CAN (.h .c)	CAN 模块初始化相关函数库
STC32G_CAN_Isr.c	CAN 接口中断函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_Timer (.h .c)	定时器模块初始化及应用相关函数库
STC32G_Timer_Isr.c	定时器模块中断函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 接口中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.21.2 程序框架

双击“CAN.uvproj”使用 Keil 编译器打开项目。

在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



该例程通过通过串口 1(P3.0 P3.1)打印接收到的 CAN 总线数据。默认波特率: 115200,N,8,1。
需要在"STC32G_UART.h"里设置: #define PRINTF_SELECT UART1 //printf 函数从串口 1 输出
MCU 每秒钟从 CAN1、CAN2 发送一帧固定数据，默认 CAN 总线波特率 500KHz。

CAN 总线波特率=Fclk/((1+(TSG1+1)+(TSG2+1))*(BRP+1)*2)

```
CAN_TSG1    = 2;                //同步采样段 1        0~15
CAN_TSG2    = 1;                //同步采样段 2        1~7 (TSG2 不能设置为 0)
CAN_BRP     = 3;                //波特率分频系数      0~63
```

24000000/((1+3+2)*4*2)=500KHz

初始化时对所有用到的 IO 口进行模式配置，并切换接口通道:

```
void GPIO_config(void)
{
    P3_MODE_IO_PU(GPIO_Pin_0 | GPIO_Pin_1);         //P3.0,P3.1 设置为准双向口
    P5_MODE_IO_PU(GPIO_Pin_LOW);          //P5.0~P5.3 设置为准双向口

    CAN1_SW(CAN1_P50_P51);      //CAN1 切换到 P5.0，P5.1 口
    CAN2_SW(CAN2_P52_P53);      //CAN2 切换到 P5.2，P5.3 口
}
```

对定时器进行初始化设置:

```
void Timer_config(void)
```

```
{
    TIM_InitTypeDef     TIM_InitStructure;      //结构定义
    TIM_InitStructure.TIM_Mode      = TIM_16BitAutoReload;  //16 位自动重载模式
    TIM_InitStructure.TIM_ClkSource = TIM_CLOCK_1T;     //指定 1T 时钟源
    TIM_InitStructure.TIM_ClkOut    = DISABLE;          //不输出高速脉冲
    TIM_InitStructure.TIM_Value     = (u16)(65536UL - (MAIN_Fosc / 1000UL)); // 1 秒钟中断 1000 次
    TIM_InitStructure.TIM_Run       = ENABLE;           //初始化后启动定时器
    Timer_Inilize(Timer0,&TIM_InitStructure);           //初始化 Timer0
    NVIC_Timer0_Init(ENABLE,Priority_0);    // Timer0 中断使能,优先级 0 级
}
```

对 UART 接口进行设置:

```
void UART_config(void)
{
    COMx_InitDefine     COMx_InitStructure;             //结构定义
    //模式,    UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode        = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;     //波特率,       110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;       //接收允许,    ENABLE 或 DISABLE
    UART_Configuration(UART1, &COMx_InitStructure);     //初始化串口 1
    NVIC_UART1_Init(ENABLE,Priority_1);     //串口 1 中断使能,优先级
    UART1_SW(UART1_SW_P30_P31);             //UART1 切换到 P3.0, P3.1
}
```

对 CAN 总线模块进行初始化设置:

```
void CAN_config(void)
{
    CAN_InitTypeDef   CAN_InitStructure;                //结构定义

    CAN_InitStructure.CAN_Enable = ENABLE;          //CAN 功能使能    ENABLE 或 DISABLE
    CAN_InitStructure.CAN_IMR    = CAN_ALLIM;       // CAN 中断寄存器, 打开所有中断
    CAN_InitStructure.CAN_SJW    = 0;               //重新同步跳跃宽度   0~3
    CAN_InitStructure.CAN_SAM    = 0;               //总线电平采样次数   0:采样 1 次; 1:采样 3 次

    //CAN 总线波特率=Fclk/((1+(TSG1+1)+(TSG2+1))*(BRP+1)*2)
    CAN_InitStructure.CAN_TSG1   = 2;               //同步采样段 1        0~15
    CAN_InitStructure.CAN_TSG2   = 1;               //同步采样段 2        1~7 (TSG2  不能设置为 0)
    CAN_InitStructure.CAN_BRP    = 3;               //波特率分频系数      0~63
    //24000000/((1+3+2)*4*2)=500KHz

    CAN_InitStructure.CAN_ListenOnly = DISABLE;     //Listen Only 模式    ENABLE,DISABLE
    //滤波选择   DUAL_FILTER(双滤波),SINGLE_FILTER(单滤波)
    CAN_InitStructure.CAN_Filter = SINGLE_FILTER;
```

```
    CAN_InitStructure.CAN_ACR0      = 0x00;         //总线验收代码寄存器 0~0xFF
    CAN_InitStructure.CAN_ACR1      = 0x00;
    CAN_InitStructure.CAN_ACR2      = 0x00;
    CAN_InitStructure.CAN_ACR3      = 0x00;
    CAN_InitStructure.CAN_AMR0      = 0xff;         //总线验收屏蔽寄存器 0~0xFF
    CAN_InitStructure.CAN_AMR1      = 0xff;
    CAN_InitStructure.CAN_AMR2      = 0xff;
    CAN_InitStructure.CAN_AMR3      = 0xff;
    CAN_Inilize(CAN1,&CAN_InitStructure);           //CAN1 初始化
    CAN_Inilize(CAN2,&CAN_InitStructure);           //CAN2 初始化

    NVIC_CAN_Init(CAN1,ENABLE,Priority_1);      //中断使能, CAN1/CAN2; ENABLE/DISABLE; 优先级
    NVIC_CAN_Init(CAN2,ENABLE,Priority_1);      //中断使能, CAN1/CAN2; ENABLE/DISABLE; 优先级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）：

```
    WTST = 0;   //设置程序指令延时参数，赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();   //扩展 SFR(XFR)访问使能
    CKCON = 0;  //提高访问 XRAM 速度
```

之后调用初始化函数、IO 口初始电平等：

```
    GPIO_config();
    Timer_config();
    UART_config();
    CAN_config();
    EA = 1;
```

设置 CAN 总线的初始化数据。

```
    CAN1_Tx.FF = STANDARD_FRAME;   //标准帧
    CAN1_Tx.RTR = 0;               //0：数据帧，1：远程帧
    CAN1_Tx.DLC = 0x08;            //数据长度
    CAN1_Tx.ID = 0x0567;           //CAN ID
    CAN1_Tx.DataBuffer[0] = 0x11;  //数据内容
    CAN1_Tx.DataBuffer[1] = 0x12;
    CAN1_Tx.DataBuffer[2] = 0x13;
    CAN1_Tx.DataBuffer[3] = 0x14;
    CAN1_Tx.DataBuffer[4] = 0x15;
    CAN1_Tx.DataBuffer[5] = 0x16;
    CAN1_Tx.DataBuffer[6] = 0x17;
    CAN1_Tx.DataBuffer[7] = 0x18;

    CAN2_Tx.FF = EXTENDED_FRAME;   //扩展帧
    CAN2_Tx.RTR = 0;               //0：数据帧，1：远程帧
    CAN2_Tx.DLC = 0x08;            //数据长度
```

```
    CAN2_Tx.ID = 0x03456789;            //CAN ID
    CAN2_Tx.DataBuffer[0] = 0x21;       //数据内容
    CAN2_Tx.DataBuffer[1] = 0x22;
    CAN2_Tx.DataBuffer[2] = 0x23;
    CAN2_Tx.DataBuffer[3] = 0x24;
    CAN2_Tx.DataBuffer[4] = 0x25;
    CAN2_Tx.DataBuffer[5] = 0x26;
    CAN2_Tx.DataBuffer[6] = 0x27;
    CAN2_Tx.DataBuffer[7] = 0x28;
```

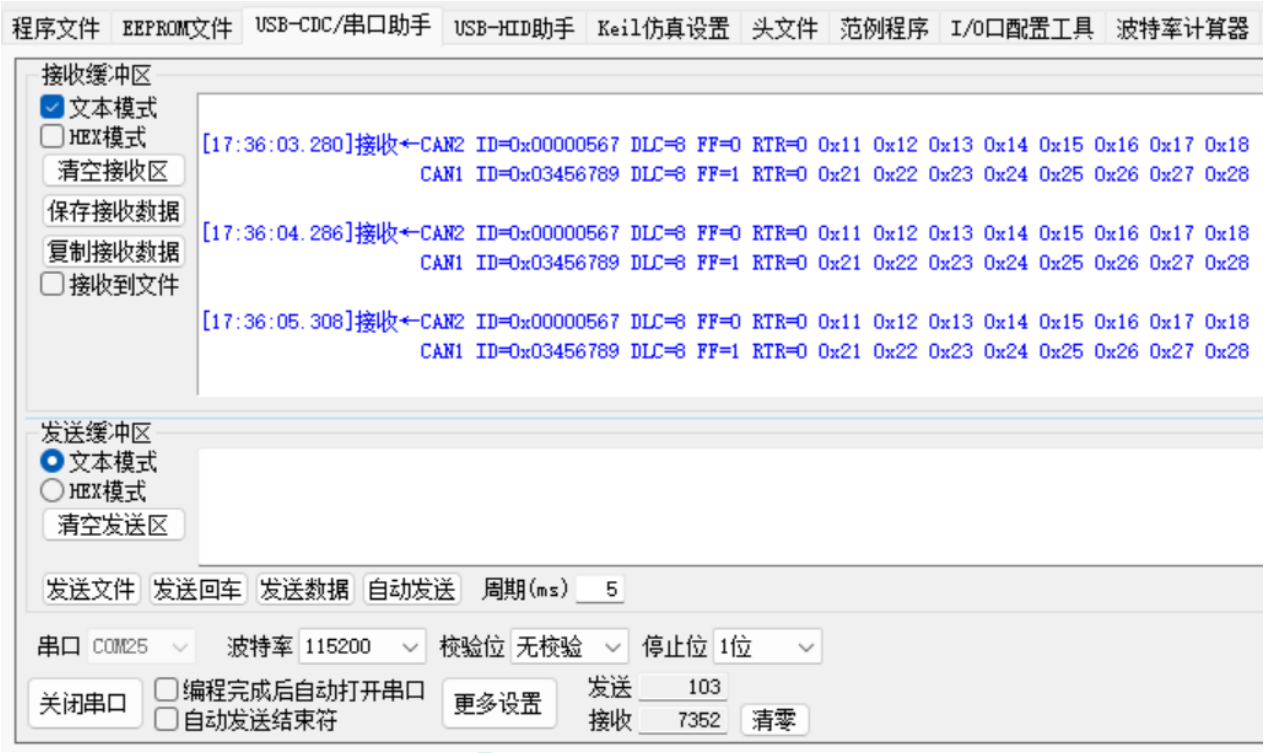
主循环里每 1 秒钟通过 CAN1，CAN2 发送一帧数据到 CAN 总线。判断 CAN1，CAN2 是否有收到 CAN 总线的数据，收到的话通过串口打印接收结果。

```
    while (1)
    {
        if(T0_1ms)    //判断定时器 1ms 中断标志是否置位
        {
            T0_1ms = 0;         //定时 1ms，清除时间标志
            //------------------处理 CAN1 模块-----------------------
            if(++msecond >= 1000)    //1ms *1000=1 秒
            {
                msecond = 0;

                CANSEL = CAN1;        //选择 CAN1 模块
                sr = CanReadReg(SR);  //读取 CAN 状态寄存器

//如果 CAN 控制器收发异常计数超过 255，就会进入 BUS-OFF 状态，此时总线处于忙状态，无法发送也无法接收，需要通过清除 Reset Mode，从 BUS-OFF 状态退出，然后才能正常通信。
                if(sr & 0x01)         //判断是否有 BS:BUS-OFF 状态
                {
                    CANAR = MR;
                    CANDR &= ~0x04;   //清除 Reset Mode，从 BUS-OFF 状态退出
                }
                else
                {
                    CanSendMsg(&CAN1_Tx);    //总线状态正常则发送一帧数据
                }

                //------------------处理 CAN2 模块-----------------------
                CANSEL = CAN2;        //选择 CAN2 模块
                sr = CanReadReg(SR);  //读取 CAN 状态寄存器

                if(sr & 0x01)         //判断是否有 BS:BUS-OFF 状态
                {
                    CANAR = MR;
                    CANDR &= ~0x04;   //清除 Reset Mode，从 BUS-OFF 状态退出
```

```
                }
                else
                {
                    CanSendMsg(&CAN2_Tx);     //总线状态正常则发送一帧数据
                }
            }
        }

        if(B_Can1Read)          //判断 CAN1 是否接收到数据
        {
            B_Can1Read = 0;

            CANSEL = CAN1;          //选择 CAN1 模块
            n = CanReadMsg(CAN1_Rx);       //读取接收内容
            if(n>0)
            {
                for(i=0;i<n;i++)
                {
//                  CanSendMsg(&CAN1_Rx[i]);   //CAN 总线原样返回
                    printf("CAN1 ID=0x%08lX DLC=%d FF=%d RTR=%d
",CAN1_Rx[i].ID,CAN1_Rx[i].DLC,CAN1_Rx[i].FF,CAN1_Rx[i].RTR);   //串口打印帧信息
                    for(j=0;j<CAN1_Rx[i].DLC;j++)
                    {
                        printf("0x%02X ",CAN1_Rx[i].DataBuffer[j]);      //从串口输出收到的数据
                    }
                    printf("\r\n");
                }
            }
        }

        if(B_Can2Read)          //判断 CAN2 是否接收到数据
        {
            B_Can2Read = 0;

            CANSEL = CAN2;          //选择 CAN2 模块
            n = CanReadMsg(CAN2_Rx);       //读取接收内容
            if(n>0)
            {
                for(i=0;i<n;i++)
                {
//                  CanSendMsg(&CAN2_Rx[i]);   //CAN 总线原样返回
                    printf("CAN2 ID=0x%08lX DLC=%d FF=%d RTR=%d
",CAN2_Rx[i].ID,CAN2_Rx[i].DLC,CAN2_Rx[i].FF,CAN2_Rx[i].RTR);   //串口打印帧信息
                    for(j=0;j<CAN2_Rx[i].DLC;j++)
                    {
```

```
                    printf("0x%02X ",CAN2_Rx[i].DataBuffer[j]);      //从串口输出收到的数据
                }
                printf("\r\n");
            }
        }
    }
}
```

通过串口助手显示程序运行结果，CAN2 收到 CAN1 发送的数据，CAN1 收到 CAN2 发送的数据:



5.21.3 CAN 总线帧格式

CAN2.0B 标准帧

CAN 标准帧信息为 11 个字节，包括两部分：信息和数据部分。前 3 个字节为信息部分。

位置	B7	B6	B5	B4	B3	B2	B1	B0
字节 01	FF	RTR	-	-	DLC（数据长度）			
字节 02	CAN ID[10:3]							
字节 03	CAN ID[2:0]			-	-	-	-	-
字节 04	数据 1							
字节 05	数据 2							
字节 06	数据 3							
字节 07	数据 4							
字节 08	数据 5							
字节 09	数据 6							
字节 10	数据 7							
字节 11	数据 8							

字节 1 为帧信息。第 7 位 (FF) 表示帧格式，在标准帧中，FF = 0；第 6 位 (RTR) 表示帧的类型，RTR=0 表示为数据帧，RTR=1 表示为远程帧；DLC 表示在数据帧时实际的数据长度。

字节 2、3 为报文识别码，11 位有效。

字节 4 ~ 11 为数据帧的实际数据，远程帧时无效。

CAN2.0B 扩展帧

CAN 扩展帧信息为 13 个字节，包括两部分，信息和数据部分。前 5 个字节为信息部分。

位置	B7	B6	B5	B4	B3	B2	B1	B0
字节 01	FF	RTR	-	-	DLC（数据长度）			
字节 02	CAN ID[28:21]							
字节 03	CAN ID[20:13]							
字节 04	CAN ID[12:5]							
字节 05	CAN ID[4:0]					-	-	-
字节 06	数据 1							
字节 07	数据 2							
字节 08	数据 3							
字节 09	数据 4							
字节 10	数据 5							
字节 11	数据 6							
字节 12	数据 7							
字节 13	数据 8							

字节 1 为帧信息。第 7 位 (FF) 表示帧格式，在扩展帧中，FF = 1；第 6 位 (RTR) 表示帧的类型，RTR=0 表示为数据帧，RTR=1 表示为远程帧；DLC 表示在数据帧时实际的数据长度。

字节 2 ~ 5 为报文识别码，其高 29 位有效。

字节 6 ~ 13 数据帧的实际数据，远程帧时无效。

5.22 LIN 总线接口

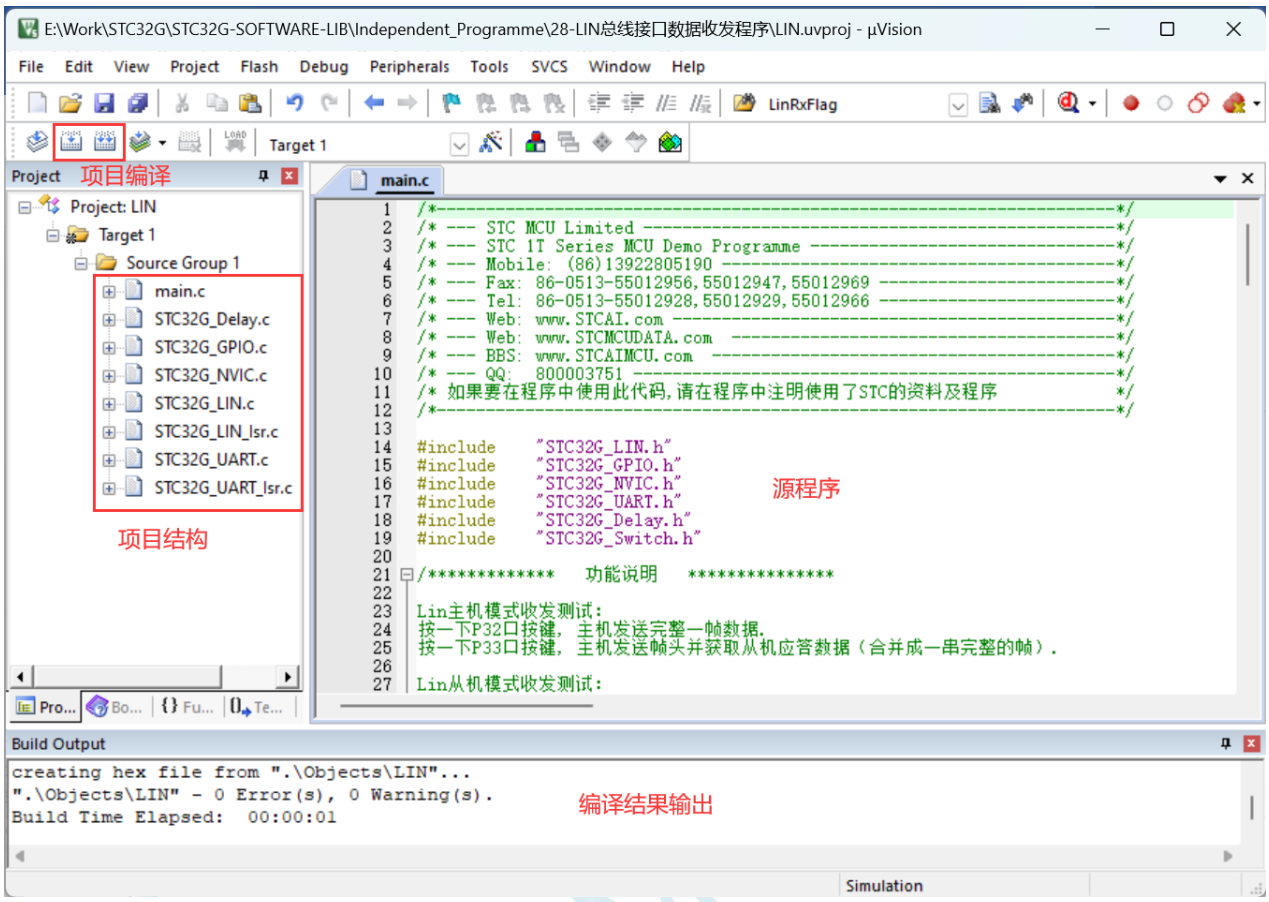
5.22.1 项目文件

打开 LIN 总线接口数据收发程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
LIN(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_Delay (.h .c)	标准延时函数
STC32G_LIN (.h .c)	LIN 模块初始化相关函数库
STC32G_LIN_Isr.c	LIN 接口中断函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 接口中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.22.2 程序框架

双击“LIN.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



Lin 主机模式收发测试:
按一下 P32 口按键，主机发送完整一帧数据.
按一下 P33 口按键，主机发送帧头并获取从机应答数据（合并成一串完整的帧）.

Lin 从机模式收发测试:
收到一个非本机应答的完整帧后通过串口 2 输出，并保存到数据缓存.
收到一个本机应答的帧头后(例如：ID=0x12)，发送缓存数据进行应答.

需要修改头文件 "STC32G_UART.h" 里的定义 "#define PRINTF_SELECT UART1"，通过串口 1 打印信息

默认 LIN 总线传输速率：9600 波特率。
默认串口 2 传输设置： 115200,N,8,1。

根据需要设置 LIN 总线主从模式

```
#define  LIN_MASTER_MODE        1      //0: 从机模式;  1: 主机模式
```

初始化时对所有用到的 IO 口进行模式配置，并切换接口通道:

```
void GPIO_config(void)
{
    P3_MODE_IO_PU(GPIO_Pin_0 | GPIO_Pin_1);        //P3.0,P3.1 设置为准双向口
    P4_MODE_IO_PU(GPIO_Pin_6 | GPIO_Pin_7);        //P4.6,P4.7 设置为准双向口
    P5_MODE_IO_PU(GPIO_Pin_2);        //P5.2 设置为准双向口
```

```
    LIN_SW(LIN_P46_P47);                    //LIN 切换到 P4.6, P4.7 口
    UART1_SW(UART1_SW_P30_P31);             //UART1 切换到 P3.0, P3.1 口
}
```

对 LIN 总线模块进行初始化设置:

```
void LIN_config(void)
{
    LIN_InitTypeDef    LIN_InitStructure;           //结构定义

#if(LIN_MASTER_MODE==0)     //判断主从模式
    LIN_InitStructure.LIN_IE          = LIN_ALLIE;    //LIN 中断使能, 打开所有中断
#else
    LIN_InitStructure.LIN_IE          = DISABLE;      //LIN 中断使能, 关闭所有中断
#endif
    LIN_InitStructure.LIN_Enable      = ENABLE;       //LIN 功能使能     ENABLE,DISABLE
    LIN_InitStructure.LIN_Baudrate = 9600;            //LIN 波特率
    LIN_InitStructure.LIN_HeadDelay = 1;              //帧头延时计数    0~(65535*1000)/MAIN_Fosc
    LIN_InitStructure.LIN_HeadPrescaler = 1;          //帧头延时分频    0~63
    LIN_Inilize(&LIN_InitStructure);                  //LIN 初始化

    NVIC_LIN_Init(ENABLE,Priority_1);           //中断使能, 优先级
}
```

对 UART 接口进行初始化设置:

```
void UART_config(void)
{
    COMx_InitDefine         COMx_InitStructure;                 //结构定义

    //模式,    UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode          = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;       //波特率,       110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;         //接收允许,     ENABLE 或 DISABLE
    UART_Configuration(UART1, &COMx_InitStructure);       //初始化串口 1
    NVIC_UART1_Init(ENABLE,Priority_1);       //串口 1 中断使能,优先级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();       //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数:

```
GPIO_config();
UART_config();
LIN_config();
EA = 1;
```

设置初始化状态和数据:

```
SLP_N = 1;          //收发器进入正常模式
Lin_ID = 0x32;      //LIN 总线 ID
TX_BUF[0] = 0x81;//初始化数据缓冲区
TX_BUF[1] = 0x22;
TX_BUF[2] = 0x33;
TX_BUF[3] = 0x44;
TX_BUF[4] = 0x55;
TX_BUF[5] = 0x66;
TX_BUF[6] = 0x77;
TX_BUF[7] = 0x88;
```

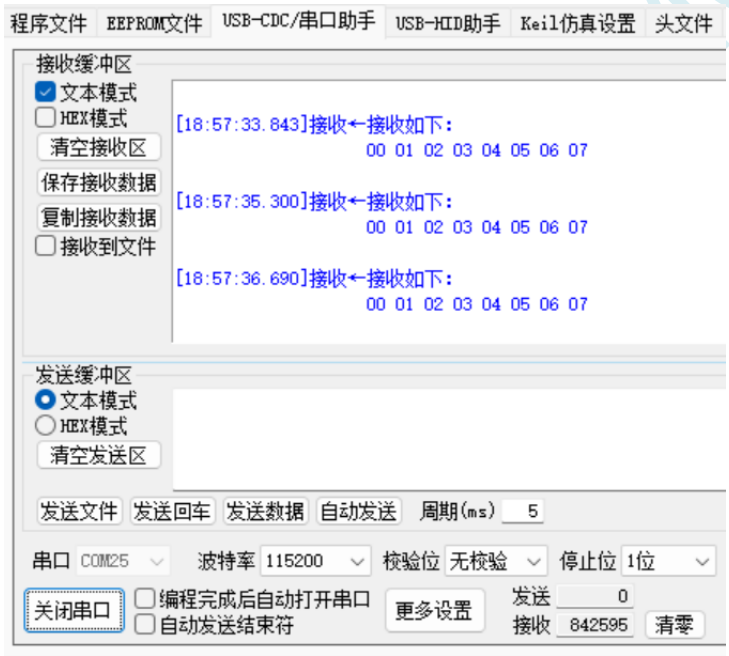
主循环里如果设置主机模式，判断触发 P3.2 口按键，则发送完整一帧数据；判断触发 P3.3 口按键，则发送帧头，然后接收从机回复的数据帧，通过串口打印接收内容。
如果设置从机模式，判断是否收到正确的帧头，如果收到的话判断帧 ID，如果 ID 号是从机响应 ID(0x12)的话，回复帧数据给主机；如果 ID 号不是从机响应 ID(0x12)的话，接收数据帧并通过串口打印接收内容。

```
while (1)
{
#if(LIN_MASTER_MODE==1)     //判断主从模式
    delay_ms(1);      //主循环 1ms 循环一次
    if(!P32)          //判断 P3.2 按键口电平
    {
        if(!Key1_Flag)       //判断按键是否已经触发，避免重复执行按键动作
        {
            Key1_cnt++;
            if(Key1_cnt > 50)        //按键检测 50ms 防抖
            {
                Key1_Flag = 1;
                LinSendFrame(Lin_ID, TX_BUF);   //发送一帧完整数据
            }
        }
    }
    else
    {
        Key1_cnt = 0;
        Key1_Flag = 0;
    }

    if(!P33)          //判断 P3.3 按键口电平
```

```
        {
            if(!Key2_Flag)              //判断按键是否已经触发，避免重复执行按键动作
            {
                Key2_cnt++;
                if(Key2_cnt > 50)       //按键检测 50ms 防抖
                {
                    Key2_Flag = 1;
                    LinSendHeaderRead(0x13,RX_BUF);  //发送帧头，获取数据帧，组成一个完整的帧
                    printf("接收如下：\r\n");
                    for(i=0; i<FRAME_LEN; i++)      printf("%02x ", RX_BUF[i]);    //串口输出接收结果
                    printf("\r\n");
                }
            }
        }
        else
        {
            Key2_cnt = 0;
            Key2_Flag = 0;
        }
#else   //从机模式
        u8 isr;

        if(LinRxFlag == 1)
        {
            LinRxFlag = 0;
            isr = LinReadReg(LID);
            if(isr == 0x12)     //判断是否从机响应 ID
            {
                LinTxResponse(TX_BUF);    //返回响应数据
            }
            else
            {
                LinReadFrame(RX_BUF);                 //接收 Lin 总线数据
                printf("ID=0x%02X, 接收内容：\r\n",isr);
                for(i=0; i<FRAME_LEN; i++)      printf("%02x ", RX_BUF[i]);    //串口输出接收结果
                printf("\r\n");
            }
        }
#endif
    }
```

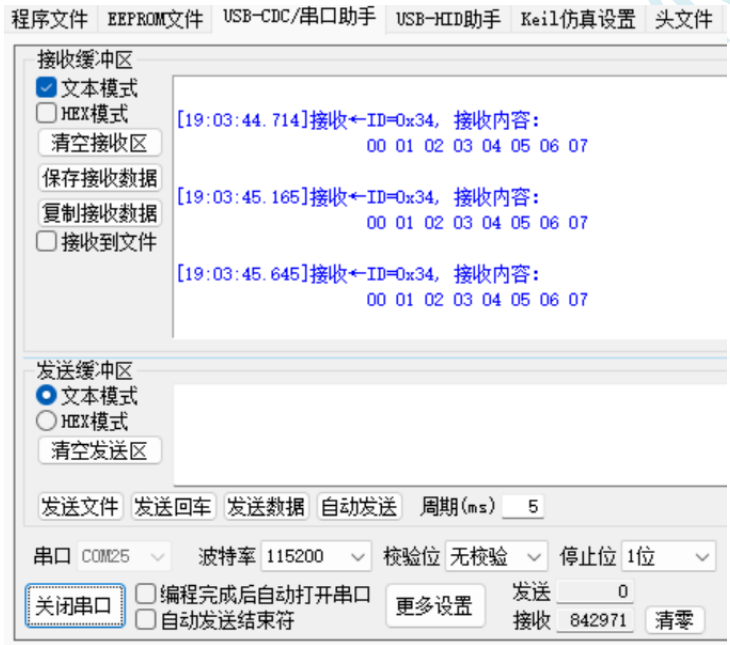
MCU 设置主机模式，通过 LIN 总线测试工具显示程序运行结果：

MCU 设置主机模式，通过串口助手显示从机应答数据：



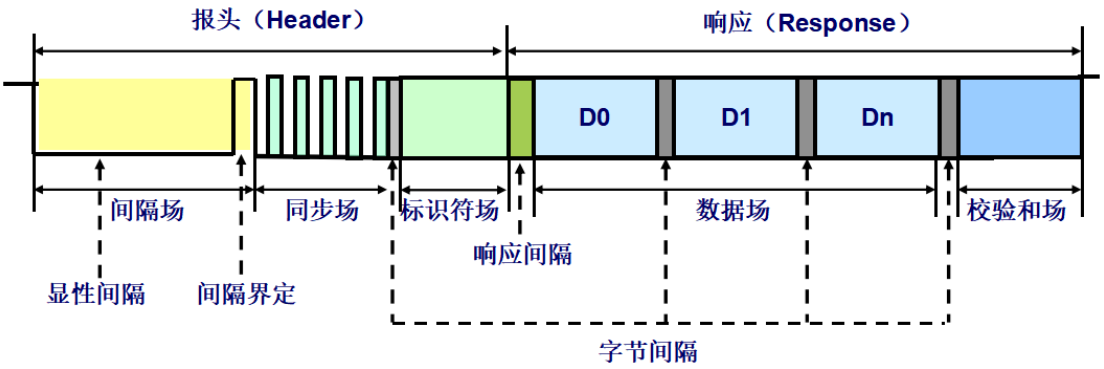
MCU 设置从机模式，LIN 总线测试工具做主机发送完整帧数据:

MCU 设置从机模式，通过串口助手显示接收到的主机发送的数据:



MCU 设置从机模式，LIN 总线测试工具做主机发送帧头，由从机回复应答数据：

5.22.3 LIN 总线时序介绍



1、字节场

1）基于 UART/SCI 的通信格式；

2）发送一个字节需要 10 个位时间（TBIT）

2、间隔场

1）表示一帧报文的起始，由主节点发出；

2）间隔信号至少由 13 个显性位组成；

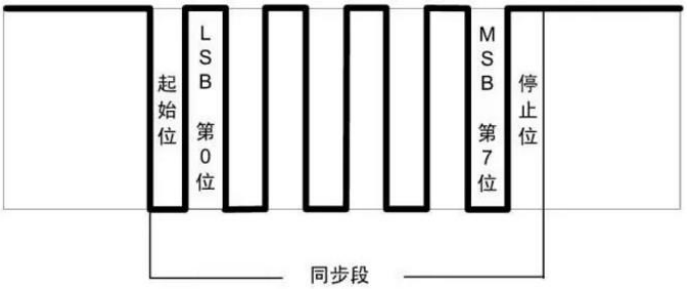
3）间隔界定符至少由 1 个隐形位组成；

4）间隔场是唯一一个不符合字节场格式的场；

5）从节点需要检测到至少连续 11 个显性位才认为是间隔信号；

3、同步场

1) 确保所有从节点使用与节点相同的波特率发送和接收数据;
2) 一个字节，结构固定: 0x55;

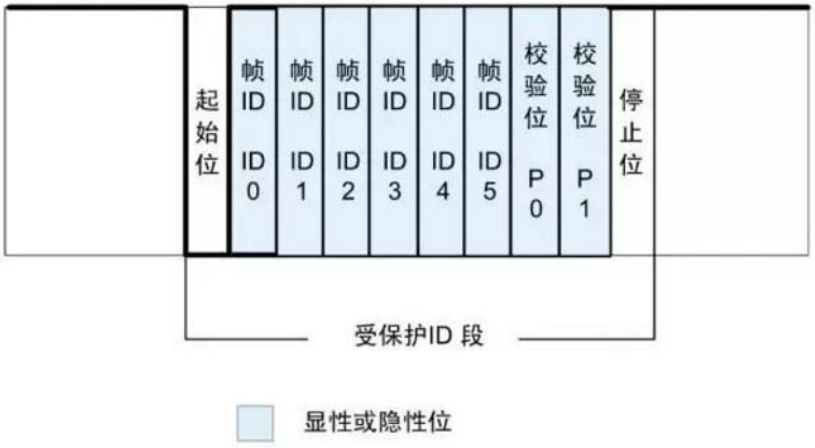


4、标识符场

1) ID 的范围从 0 到 63(0x3f);
2) 奇偶校验符(Parity)P0,P1

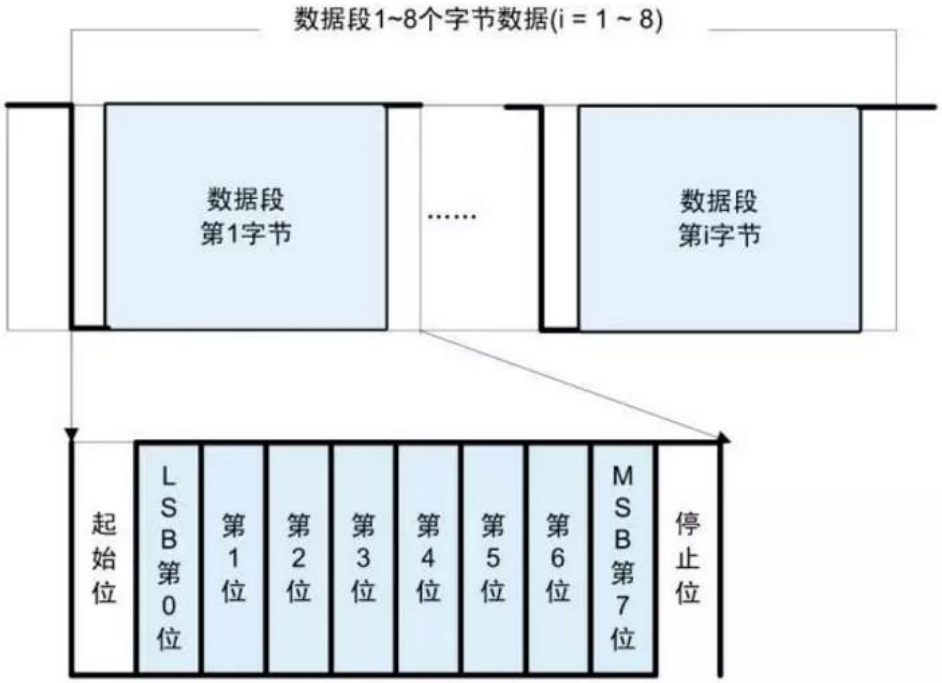
$$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4$$

$$P1 = \neg (ID1 \oplus ID3 \oplus ID4 \oplus ID5)$$



5、数据场

1) 数据场长度 1 到 8 个字节;
2) 低字节先发，低位先发;
3) 如果某信号长度超过 1 个字节采用低位在前的方式发送（小端);



6、校验和场

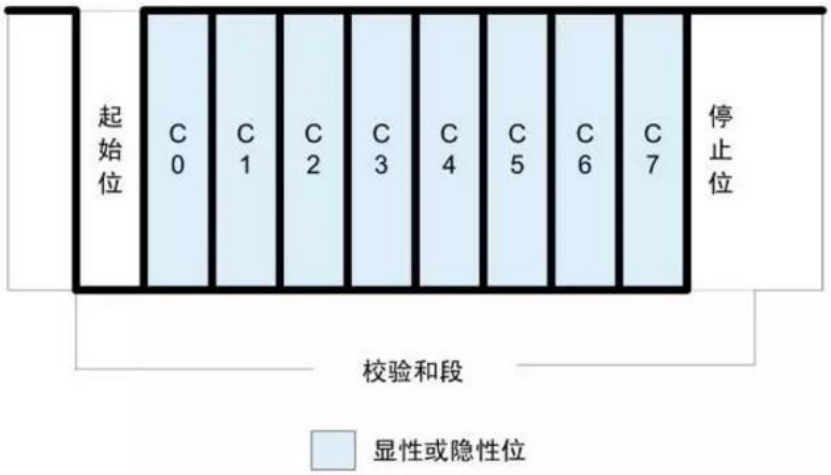
用于校验接收的数据是否正确

1) 经典校验 (Classic Checksum) 仅校验数据场(LIN1.3)

2) 增强校验 (Enhance Checksum) 校验标识符场与数据场内容(LIN2.0、LIN2.1)

标识符为 0x3C 和 0x3D 的帧只能使用经典校验

计算方法: 反转 8 位求和



5.23 USART 接口 LIN 总线

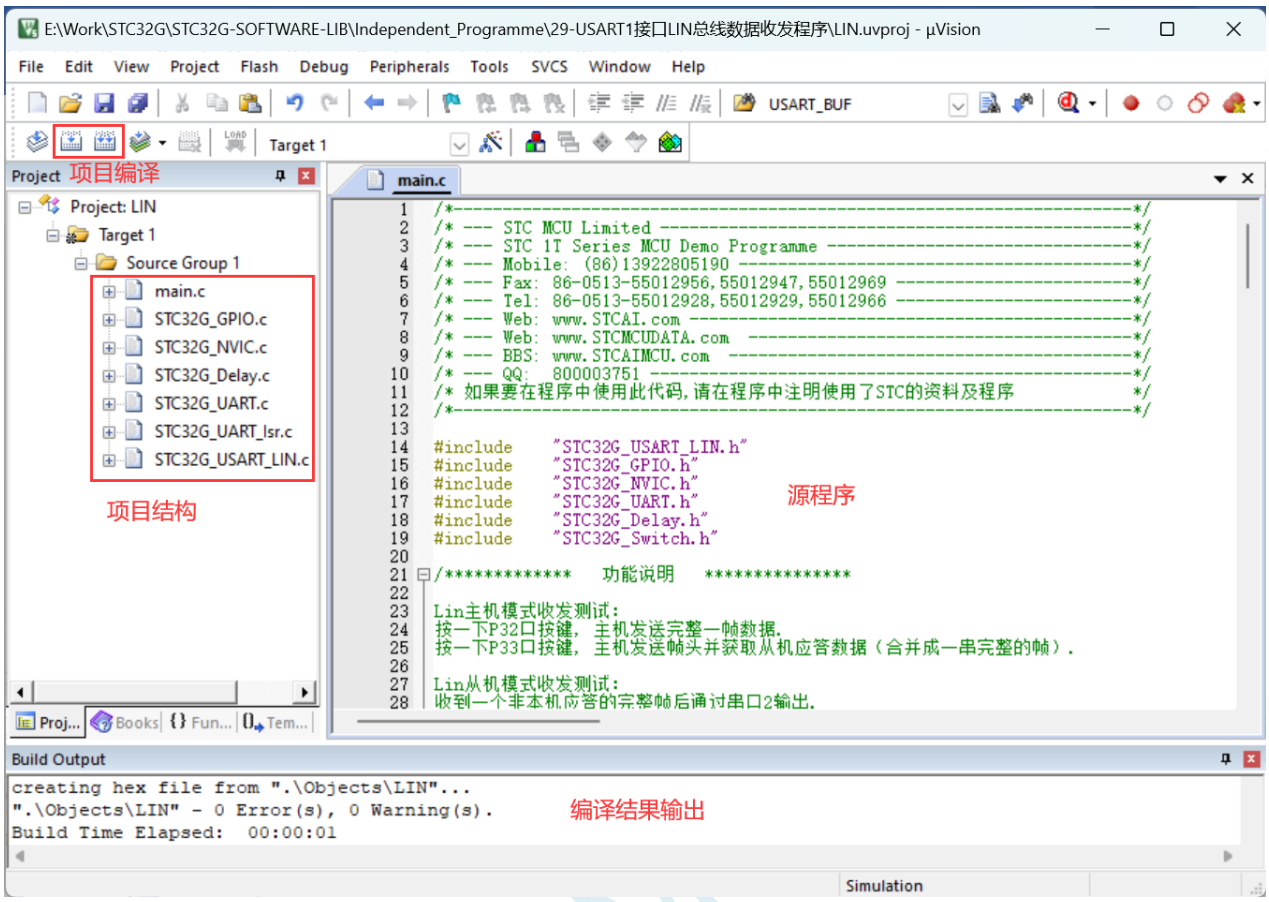
5.23.1 项目文件

打开 USARTn 接口 LIN 总线数据收发程序，其中包含编译文件存放“list”文件夹，其它文件介绍如下：

文件	描述
Config.h	用户配置文件，主要是主时钟定义
LIN(.uvopt .uvproj)	项目文件
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_Delay (.h .c)	标准延时函数
STC32G_USART_LIN (.h .c)	USART 模块 LIN 总线初始化相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_UART_Isr.c	UART 接口中断函数库
STC32G_Switch.h	功能脚切换定义头文件
Type_def.h	数据类型定义文件

5.23.2 程序框架

双击“LIN.uvproj”使用 Keil 编译器打开项目。
在左侧项目窗口里添加项目需要用到的库函数程序文件。程序功能主要在”main.c”文件里实现，双击左侧项目窗口里的“main.c”文件名就可以打开源文件。起始位置包含所需库文件的头文件。



Lin 主机模式收发测试:
按一下 P32 口按键，主机发送完整一帧数据.
按一下 P33 口按键，主机发送帧头并获取从机应答数据（合并成一串完整的帧）.

Lin 从机模式收发测试:
收到一个非本机应答的完整帧后通过串口 2 输出，并保存到数据缓存.
收到一个本机应答的帧头后(例如：ID=0x12)，发送缓存数据进行应答.

需要修改头文件 "STC32G_UART.h" 里的定义 "#define PRINTF_SELECT UART2"，通过串口 2 打印信息

默认 LIN 总线传输速率：9600 波特率。
默认串口 2 传输设置： 115200,N,8,1。

根据需要设置 LIN 总线主从模式

```
#define  LIN_MASTER_MODE        0      //0: 从机模式;  1: 主机模式
```

初始化时对所有用到的 IO 口进行模式配置，并切换接口通道:

```
void GPIO_config(void)
{
    //P4.3,P4.4,P4.6,P4.7 设置为准双向口
    P4_MODE_IO_PU(GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_6 | GPIO_Pin_7);
    P7_MODE_IO_PU(GPIO_Pin_6);              //P7.6 设置为准双向口
```

```
    UART1_SW(UART1_SW_P43_P44);          //USART LIN 切换到 P4.3, P4.4 口
    UART2_SW(UART2_SW_P46_P47);          //UART2 切换到 P4.6, P4.7 口
}
```

对 USART 接口 LIN 总线模块进行初始化设置:

```
void LIN_config(void)
{
    USARTx_LIN_InitDefine LIN_InitStructure;          //结构定义

#if(LIN_MASTER_MODE==1)
    LIN_InitStructure.LIN_Mode = LinMasterMode;       //LIN 总线模式   LinMasterMode,LinSlaveMode
    LIN_InitStructure.LIN_AutoSync = DISABLE;         //自动同步使能   ENABLE,DISABLE
#else
    LIN_InitStructure.LIN_Mode = LinSlaveMode;        //LIN 总线模式   LinMasterMode,LinSlaveMode
    LIN_InitStructure.LIN_AutoSync = ENABLE;          //自动同步使能   ENABLE,DISABLE
#endif
    LIN_InitStructure.LIN_Enable    = ENABLE;         //LIN 功能使能    ENABLE,DISABLE
    LIN_InitStructure.LIN_Baudrate = 9600;            //LIN 波特率
    UASRT_LIN_Configuration(USART1,&LIN_InitStructure);      //LIN 初始化

    NVIC_UART1_Init(ENABLE,Priority_1);      //中断使能, 优先级
}
```

对 UART 接口进行初始化设置:

```
void UART_config(void)
{
    COMx_InitDefine         COMx_InitStructure;                 //结构定义
    //模式,    UART_ShiftRight,UART_8bit_BRTx,UART_9bit,UART_9bit_BRTx
    COMx_InitStructure.UART_Mode          = UART_8bit_BRTx;
    //选择波特率发生器, BRT_Timer2 (注意: 串口 2 固定使用 BRT_Timer2, 所以不用选择)
    COMx_InitStructure.UART_BRT_Use     = BRT_Timer2;
    COMx_InitStructure.UART_BaudRate    = 115200ul;      //波特率,      110 ~ 115200
    COMx_InitStructure.UART_RxEnable    = ENABLE;        //接收允许,    ENABLE 或 DISABLE
    UART_Configuration(UART2, &COMx_InitStructure);      //初始化串口 2
    NVIC_UART2_Init(ENABLE,Priority_1);      //串口 2 中断使能,优先级
}
```

主函数起始位置推荐先执行以下几条指令，提升芯片的性能，设置扩展寄存器访问使能（起始位置开启后可以不用再关闭）:

```
    WTST = 0;   //设置程序指令延时参数, 赋值为 0 可将 CPU 执行指令的速度设置为最快
    EAXSFR();     //扩展 SFR(XFR)访问使能
    CKCON = 0;   //提高访问 XRAM 速度
```

之后调用初始化函数:

```
    GPIO_config();
    UART_config();
    LIN_config();
    EA = 1;
```

设置初始化状态和数据：

```
    SLP_N = 1;          //收发器进入正常模式
    Lin_ID = 0x32;      //LIN 总线 ID
    TX_BUF[0] = 0x81;//初始化数据缓冲区
    TX_BUF[1] = 0x22;
    TX_BUF[2] = 0x33;
    TX_BUF[3] = 0x44;
    TX_BUF[4] = 0x55;
    TX_BUF[5] = 0x66;
    TX_BUF[6] = 0x77;
    TX_BUF[7] = 0x88;
```

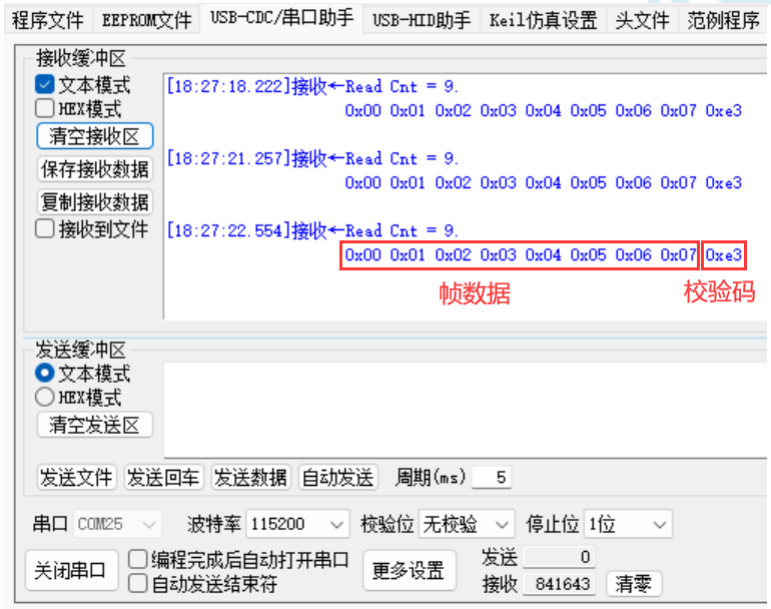
主循环里如果设置主机模式，判断触发 P3.2 口按键，则发送完整一帧数据；判断触发 P3.3 口按键，则发送帧头，然后接收从机回复的数据帧，通过串口打印接收内容。
如果设置从机模式，判断是否收到正确的帧头，如果收到的话判断帧 ID，如果 ID 号是从机响应 ID(0x12)的话，回复帧数据给主机；如果 ID 号不是从机响应 ID(0x12)的话，接收数据帧并通过串口打印接收内容。

```
    while (1)
    {
        delay_ms(1);        //主循环 1ms 循环一次
#if(LIN_MASTER_MODE==1)     //判断主从模式
        if(!P32)            //判断 P3.2 按键口电平
        {
            if(!Key1_Flag)      //判断按键是否已经触发，避免重复执行按键动作
            {
                Key1_cnt++;
                if(Key1_cnt > 50)       //按键检测 50ms 防抖
                {
                    Key1_Flag = 1;
                    UsartLinSendFrame(USART1,ULin_ID, USART_BUF, FRAME_LEN); //发送完整帧数据
                }
            }
        }
        else
        {
            Key1_cnt = 0;
            Key1_Flag = 0;
        }

        if(!P33)            //判断 P3.3 按键口电平
        {
```

```
            if(!Key2_Flag)      //判断按键是否已经触发，避免重复执行按键动作
            {
                Key2_cnt++;
                if(Key2_cnt > 50)       //按键检测 50ms 防抖
                {
                    Key2_Flag = 1;
                    UsartLinSendHeader(USART1,0x13); //发送帧头，获取数据帧，组成一个完整的帧
                }
            }
        }
        else
        {
            Key2_cnt = 0;
            Key2_Flag = 0;
        }
#else   //从机模式
        if((B_ULinRX1_Flag) && (COM1.RX_Cnt >= 2))      //判断是否收到 LIN 总线数据
        {
            B_ULinRX1_Flag = 0;
            //判断帧头以及是否从机响应 ID
            if((RX1_Buffer[0] == 0x55) && ((RX1_Buffer[1] & 0x3f) == 0x12))
            {
                UsartLinSendData(USART1,USART_BUF,FRAME_LEN);           //返回响应数据
                UsartLinSendChecksum(USART1,USART_BUF,FRAME_LEN);   //返回数据校验
            }
        }
#endif

        if(COM1.RX_TimeOut > 0)       //超时计数
        {
            if(--COM1.RX_TimeOut == 0)
            {
                printf("Read Cnt = %d.\r\n",COM1.RX_Cnt);      //打印获取数据长度
                //从串口输出收到的从机数据
                for(i=0; i<COM1.RX_Cnt; i++)       printf("0x%02x ",RX1_Buffer[i]);
                COM1.RX_Cnt   = 0;     //清除字节数
                printf("\r\n");
            }
        }
    }
```

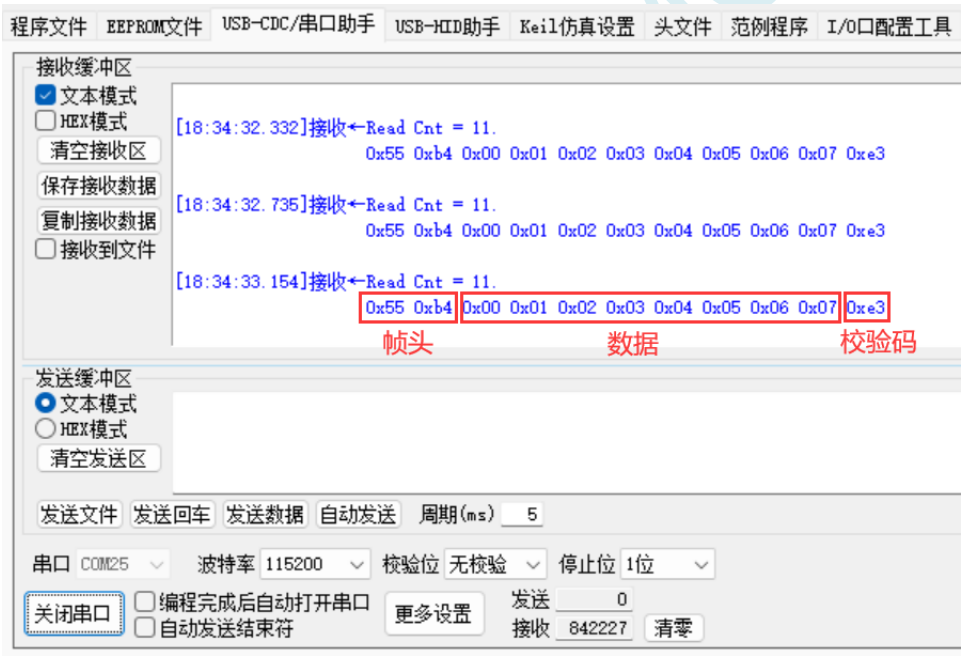
MCU 设置主机模式，通过 LIN 总线测试工具显示程序运行结果：

MCU 设置主机模式，通过串口助手显示从机应答数据：



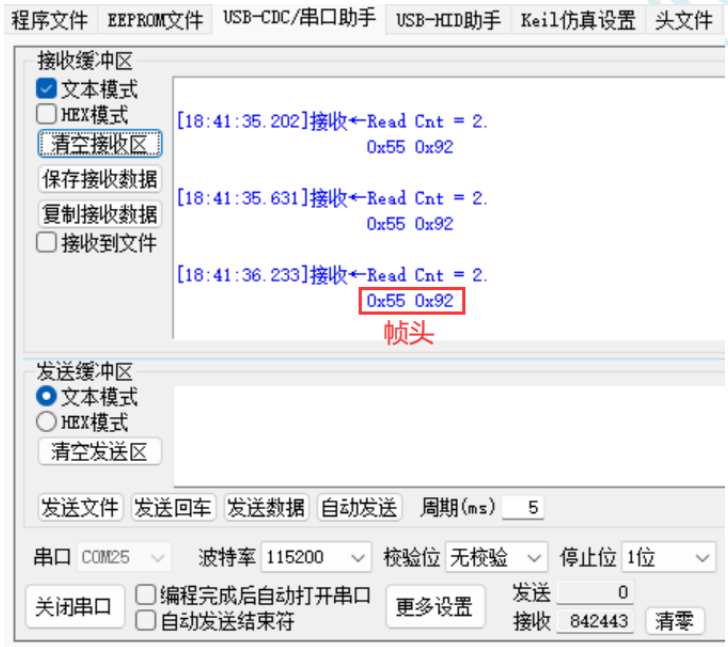
MCU 设置从机模式，LIN 总线测试工具做主机发送完整帧数据:

MCU 设置从机模式，通过串口助手显示接收到的主机发送的数据:



MCU 设置从机模式，LIN 总线测试工具做主机发送帧头，由从机回复应答数据：

MCU 设置从机模式，通过串口助手显示接收到的主机发送的帧头信息：



6 综合例程使用说明

6.1 函数库目录结构

|----App (应用程序目录)
| |----inc (应用程序头文件目录)
| |----src (应用程序源代码目录)
|----Driver (硬件驱动程序目录)
| |----inc (驱动程序头文件目录)
| |----isr (驱动中断程序目录)
| |----src (驱动程序源代码目录)
|----RVMDK (项目及输出文件目录)
| |----list (编译输出文件目录)
|----User (用户程序及配置文件目录)

6.1.1 应用程序模块

文件	描述
APP (.h .c)	应用程序公用变量、函数声明
APP_AD_UART (.h .c)	多路 ADC 查询采样，通过串口发送例程
APP_DMA_AD(.h .c)	STC32G 单片机 ADC DMA 数据批量存储例程
APP_DMA_I2C(.h .c)	STC32G 单片机 I2C 接口+DMA 数据收发例程
APP_DMA_LCM(.h .c)	STC32G 单片机 LCM 接口+DMA 驱动液晶屏例程
APP_DMA_M2M(.h .c)	STC32G 单片机 DMA Memory to Memory 数据转移例程
APP_DMA_SPI_PS(.h .c)	STC32G 单片机 UART_DMA，M2M_DMA，SPI_DMA 综合使用演示例程
APP_DMA_UART(.h .c)	STC32G 单片机串口 DMA 数据批量收发存储例程
APP_INT_UART (.h .c)	INT0~INT4 外部中断将 MCU 从休眠唤醒例程
APP_Lamp (.h .c)	使用 P6 口来演示跑马灯例程
APP_ RTC (.h .c)	STC32G 单片机内置 RTC 模块应用例程
APP_I2C_PS (.h .c)	软件模拟 I2C 与硬件 I2C 自发自收例程
APP_SPI_PS (.h .c)	通过串口发送数据给 MCU1，MCU1 将接收到的数据由 SPI 发送给 MCU2，MCU2 再通过串口发送的透传例程
APP_WDT (.h .c)	看门狗复位使用例程
APP_EEPROM (.h .c)	STC32G 单片机通过串口对内部自带的 EEPROM(FLASH)进行读写例程
APP_PWMA_Output.c	高级 PWMA 组 PWM1、PWM2、PWM3、PWM4 输出呼吸灯效果例程
APP_PWMA_Output.c	高级 PWMB 组 PWM5、PWM6、PWM7、PWM8 输出呼吸灯效果例程

6.1.2 驱动程序模块

文件	描述
STC32G_ADC (.h .c)	ADC 模块初始化及应用相关函数库
STC32G_Compare (.h .c)	比较器模块初始化相关函数库
STC32G_Delay (.h .c)	标准延时函数
STC32G_EEPROM (.h .c)	内部 EEPROM(Flash)模块应用相关函数库
STC32G_Exti (.h .c)	外部中断初始化相关函数库
STC32G_GPIO (.h .c)	IO 口初始化相关函数库
STC32G_I2C (.h .c)	I2C 模块初始化及应用相关函数库
STC32G_NVIC (.h .c)	嵌套向量中断控制器初始化相关函数库
STC32G_Soft_I2C (.h .c)	软件模拟 I2C 初始化及应用相关函数库
STC32G_Soft_UART (.h .c)	软件模拟 UART 初始化及应用相关函数库
STC32G_SPI (.h .c)	SPI 模块初始化及应用相关函数库
STC32G_Timer (.h .c)	定时器模块初始化及应用相关函数库
STC32G_UART (.h .c)	UART 模块初始化及应用相关函数库
STC32G_WDT (.h .c)	看门狗初始化及应用相关函数库
STC32G_PWM (.h .c)	16 位高级 PWM 模块初始化及应用相关函数库
STC32G_DMA (.h .c)	DMA 批量数据传输模块初始化及应用相关函数库
STC32G_CAN (.h .c)	CAN 模块初始化相关函数库
STC32G_LIN (.h .c)	LIN 模块初始化相关函数库
STC32G_USART_LIN (.h .c)	USART 模块 LIN 总线初始化相关函数库
STC32G_Switch.h	功能脚切换定义头文件
STC32G_ADC_Isr.c	ADC 模块中断函数库
STC32G_Compare_Isr.c	比较器模块中断函数库
STC32G_Exti_Isr.c	外部中断模块中断函数库
STC32G_GPIO_Isr.c	IO 口中断函数库
STC32G_I2C_Isr.c	I2C 模块中断函数库
STC32G_SPI_Isr.c	SPI 模块中断函数库
STC32G_Timer_Isr.c	定时器模块中断函数库
STC32G_UART_Isr.c	UART 模块中断函数库
STC32G_PWM_Isr.c	16 位高级 PWM 模块中断函数库
STC32G_DMA_Isr.c	DMA 批量数据传输模块中断函数库
STC32G_CAN_Isr.c	CAN 接口中断函数库
STC32G_LIN_Isr.c	LIN 接口中断函数库

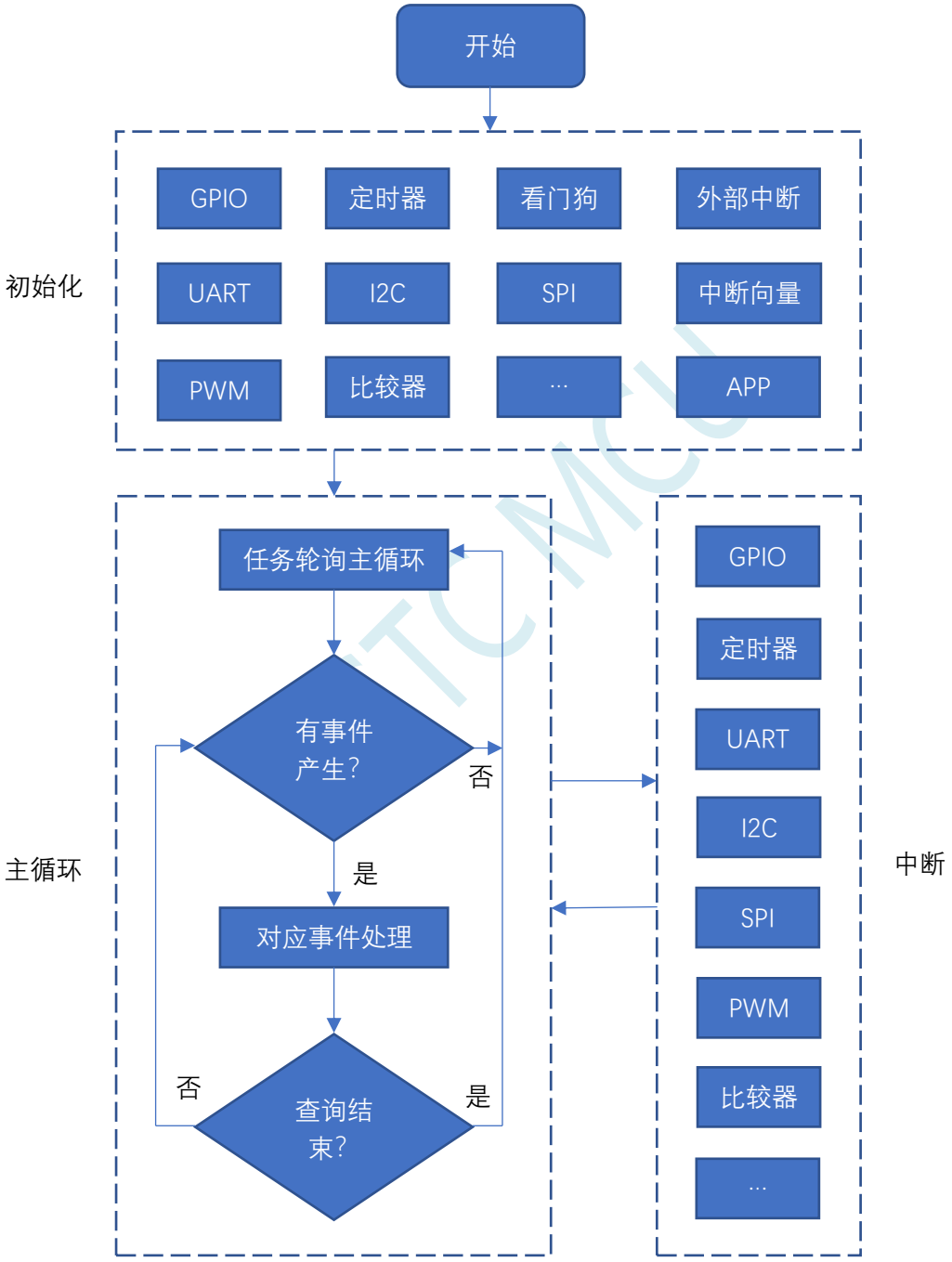
6.1.3 用户程序及配置文件

文件	描述
Config.h	用户配置文件，主要是主时钟定义
Main.c	主函数文件
STC32G.H	STC32G 单片机寄存器定义头文件
System_init (.h .c)	系统初始化配置文件
Task (.h .c)	任务调度配置文件

Type_def.h	数据类型定义文件
isr.asm	中断号大于 31 中断，借用 13 号中断向量转换文件

6.2 系统流程

6.2.1 系统流程图



6.2.2 初始化程序

“system_init.c”文件里存放系统初始化函数，执行全局初始化：

```
//========================================================================
//                                系统初始化
//========================================================================
void  SYS_Init(void)
{
//  GPIO_config();
  Timer_config();
//  ADC_config();
//  UART_config();
//  Exti_config();
//  I2C_config();
//  SPI_config();
//  CMP_config();
  Switch_config();
  EA = 1;

  APP_config();
}
```

6.2.3 任务调度主循环

系统通过定时器 0 设定 1ms 的时间作为各个任务分时调度的基准时钟。
在任务组件数组里面定义每个任务的状态、计数器、周期、执行函数：

```
static TASK_COMPONENTS Task_Comps[]=
{
//状态  计数  周期  函数
  {0, 250, 250, Sample_Lamp},          /* task 1 Period:  250ms */
  {0, 200, 200, Sample_ADtoUART},      /* task 2 Period:  200ms */
//  {0, 20, 20, Sample_INTtoUART},       /* task 3 Period:  20ms */
//  {0, 1, 1, Sample_RTC},               /* task 4 Period:  1ms */
//  {0, 1, 1, Sample_I2C_PS},            /* task 5 Period:  1ms */
//  {0, 1, 1, Sample_SPI_PS},            /* task 6 Period:  1ms */
//  {0, 1, 1, Sample_EEPROM},            /* task 7 Period:  1ms */
//  {0, 100, 100, Sample_WDT},           /* task 8 Period:  100ms */
//  {0, 1, 1, Sample_PWMA_Output},       /* task 9 Period:  1ms */
//  {0, 1, 1, Sample_PWMB_Output},       /* task 10 Period:  1ms */
//  {0, 500, 500, Sample_DMA_AD},        /* task 12 Period:  500ms */
//  {0, 500, 500, Sample_DMA_M2M},       /* task 13 Period:  100ms */
//  {0, 1, 1, Sample_DMA_UART},          /* task 14 Period:  1ms */
//  {0, 1, 1, Sample_DMA_SPI_PS},        /* task 15 Period:  1ms */
//  {0, 1, 1, Sample_DMA_LCM},           /* task 16 Period:  1ms */
//  {0, 1, 1, Sample_DMA_I2C},           /* task 17 Period:  1ms */
  /* Add new task here */
};
```

计数器每毫秒减 1，为 0 时设置状态位，并将周期时间重载到计数器里。任务处理回调函数检查每个任务的状态，如果置位的话则执行对应的函数程序：

```
//========================================================================
// 函数: Task_Pro_Handler_Callback
// 描述: 任务处理回调函数.
// 参数: None.
// 返回: None.
// 版本: V1.0, 2012-10-22
//========================================================================
void Task_Pro_Handler_Callback(void)
{
  u8 i;
  for(i=0; i<Tasks_Max; i++)
  {
    if(Task_Comps[i].Run) /* If task can be run */
    {
      Task_Comps[i].Run = 0;    /* Flag clear 0 */
      Task_Comps[i].TaskHook();  /* Run task */
    }
  }
}
```

执行应用范例程序，在开启任务后需要在“APP.c”文件里启动对应的初始化代码:

```
void APP_config(void)
{
  Lamp_init();
  ADtoUART_init();
//  INTtoUART_init();
//  RTC_init();
//  I2C_PS_init();
//  SPI_PS_init();
//  EEPROM_init();
//  WDT_init();
//  PWMA_Output_init();
//  PWMB_Output_init();
//  DMA_AD_init();
//  DMA_M2M_init();
//  DMA_UART_init();
//  DMA_SPI_PS_init();
//  DMA_LCM_init();
//  DMA_I2C_init();
}
```

用户可根据需要编写各种应用模块，在任务组件数组里面设定时间进行分时调度。

6.3 公共宏定义

“config.h”文件进行系统时钟设置（用户可根据需要自行添加）:

```
//#define MAIN_Fosc       22118400L   //定义主时钟
//#define MAIN_Fosc       12000000L   //定义主时钟
//#define MAIN_Fosc       11059200L   //定义主时钟
//#define MAIN_Fosc        5529600L   //定义主时钟
#define MAIN_Fosc       24000000L   //定义主时钟
```