1　按键中断控制 LED 灯

1.1. 背景介绍

本章我们将简单的介绍一下 AC208 的 SOC 系统设计开发流程，第一个实验主要是对 GPIO 的输出，读取和中断的使用。SOC 系统设计为软硬件协同设计流程，用到的硬件开发环境为智多晶 HQFPGA 编译软件，软件编译环境为 MDK 软件。

1.1.1. GPIO 简介

AHB GPIO 是一个通用的 32 位的 I/O 接口单元，其具有以下几个特点：

1. 可编程的中断生成功能
2. 支持使用地址值进行位屏蔽
3. 支持引脚多路复用的交替功能切换寄存器
4. 通过为控制寄存器提供单独的设置和清除地址来实现线程安全操作
5. 使用双触发器对输入进行采样，避免亚稳态问题

1.1.2. GPIO 寄存器映射

下表 1-1 展示了 GPIO 的内存映射，需要注意的是寄存器地址=基地址+基址偏移量，GPIO 的基地址为 0x4001_0000。

表 1- 1GPIO 寄存器映射表

名字	基址偏移量	类型	位宽	重置值	描述
DATA	0x0000	RW	32	0x----_----	Data value [31:0]: **Read:** Sampled at pin. **Write:** To data output register.Read back value goes through double flip-flop synchronization logic with a delay of two cycles.
DATAOUT	0x0004	RW	32	0x0000_0000	Data output Register value [31:0]: **Read:** Current value of data output register. **Write:** To data output register.
Reserved	0x0008~0x000C	-	-	-	Reserved
OUTENSET	0x0010	RW	32	0x0000_0000	Output enable set [31:0]: **Write:** 1-Set the output enable bit. 0-No effect. **Read back:** 0-Indicates the signal direction as input. 1-Indicates the signal direction as output
OUTENCLR	0x0014	RW	32	0x0000_0000	Output enable set [31:0]:

					Write: 1-Set the output enable bit. 0-No effect **Read back:** 0-Indicates the signal direction as input. 1-Indicates the signal direction as output
ALTFUNCSET	0x0018	RW	32	0xFFFF_FFFF	Alternative function set [31:0]: **Write:** 1-Sets the ALTFUNC bit. 0-No effect **Read back:** 0-For I/O. 1-For an alternate function
ALTFUNCCLR	0x001C	RW	32	0xFFFF_FFFF	Alternative function clear [31:0]: **Write:** 1-Clears the ALTFUNC bit. 0-No effect. **Read back:** 0-For I/O. 1-For an alternate function
INTENSET	0x0020	RW	32	0x0000_0000	Interrupt enable set [31:0]: **Write:** 1-Sets the enable bit. 0-No effect. **Read back:** 0-Interrupt disabled. 1-Interrupt enabled.
INTENCLR	0x0024	RW	32	0x0000_0000	Interrupt enable clear [31:0]: **Write:** 1-Clear the enable bit. 0-No effect. **Read back:** 0-Interrupt disabled. 1-Interrupt enabled
INTTYPESET	0x0028	RW	32	0x0000_0000	Interrupt type clear [31:0]: **Write:** 1-Clears the interrupt type bit. 0-No effect **Read back:** 0-For LOW or HIGH level. 1-For falling edge or rising edge
INTPOLSET	0X0030	RW	32	0x0000_0000	Polarity-level, edge IRQ configuration [31:0]: **Write:** 1-Sets the interrupt polarity bit. 0-No effect. **Read back:** 0-For LOW level or falling edge. 1-For HIGH level or rising edge.
INTPOLCLR	0x0034	RW	32	0x0000_0000	Polarity-level, edge IRQ configuration [31:0]: **Write:**

					1-Clears the interrupt polarity bit. 0-No effect. **Read back:** 0-For LOW level or falling edge. 1-For HIGH level or rising edge.
INTSTATUS, INTCLEAR	0x0038	RW	32	0x0000_0000	Write one to clear interrupt request: Write [31:0] IRQ status clear Register. **Write:** 1--To clear the interrupt request. 0--No effect **Read back:** [31:0] IRQ status Register
MASKBYTE0	0x0400-0x07FC	RW	32	0x----_----	bits[7:0] masked access. Bits[9:2] of the address value are used as enable bit mask for the access: [31:8]: Not used. RAZ/WI. [7:0]: Data for byte0 access, with bits[9:2] of address value used as enable mask for each bit
MASKBYTE1	0x0800-0x0BFC	RW	32	0x----_----	bits[15:8] masked access. Bits[9:2] of the address value are used as enable bit mask for the access: [31:16]: Not used. RAZ/WI. [15:8]: Data for byte1 access, with bits[9:2] of address value used as enable mask for each bit. [7:0]: Not used. RAZ/WI.
MASKBYTE2	0x0C00-0x0FFC	RW	32	0x----_----	bits[23:16] masked access. Bits[9:2] of the address value are used as enable bit mask for the access: [31:24]: Not used. RAZ/WI. [23:16]: Data for byte2 access, with bits[9:2] of address value used as enable mask for each bit. [15:0]: Not used. RAZ/WI.
MASKBYTE3	0x1000-0x13FC	RW	32	0x----_----	bits[31:24] masked access. Bits[9:2] of the address value are used as enable bit mask for the access: [31:24]: Data for byte3 access, with bits[9:2] of address value used as enable mask for each bit. [15:0]: Not used. RAZ/WI

1.1.3. GPIO 复用功能

GPIO 复用功能用于支持两套外围设备，对于每个引脚，必须设置 ALTFUNC 的相关位用于复用功能，对应的就是每个 GPIO 的 ALTFUNCSET 位和 ALTFUNCCLR 位。在 Seal Cortex-M3 块中，GPIO 复用功能使用以下引脚，如下表 1-2 所示，由下表可以看出 GPIO0~20 都是具有复用功能的。

表 1- 2 GPIO 复用功能表

GPIO bit	INPUT ALT Function	OUTPUT ALT Function
0	timer0extin	value 0
1	timer1extin	value 0
2	uart0_rxd	value 0
3	unused	uart0_txd
4	uart1_rxd	value 0
5	unused	uart1_txd
6	adc_etr	value 0
7	unused	spi0_clk_out
8	unused	spi0_sel
9	spi0_data_in[0]	spi0_data_out[0]
10	spi0_data_in[1]	spi0_data_out[1]
11	unused	spi1_clk_out
12	unused	spi1_sel
13	spi1_data_in[0]	spi1_data_out[0]
14	spi1_data_in[1]	spi1_data_out[1]
15	i2c_scl_in	i2c_scl_out
16	i2c_sda_in	i2c_sda_out
17	spi0_data_in[2]	spi0_data_out[2]
18	spi0_data_in[3]	spi0_data_out[3]
19	spi1_data_in[2]	spi1_data_out[2]
20	spi1_data_in[2]	spi1_data_out[3]
21~31	unused	unused

1.1.4. 中断的概念

本次实验使用按键中断的方式控制 LED 灯，中断就是指当出现需要时，CPU 暂时停止当前程序的执行转而去执行和处理新情况的过程。即在程序运行过程中，系统出现了一个必须由 CPU 立即处理的情况，此时，CPU 暂时中止程序的执行转而处理这个新的情况的过程就叫做中断。

当我们开启了中断模式，就是说 CPU 不主动访问这些设备，只管处理自己的任务。如果有设备要与 CPU 联系，或要 CPU 处理一些事情，它会给 CPU 发一个中断请求信号。这时 CPU 就会放下正在进行的工作而去处理这个外设的请求。处理完中断后，CPU 返回去继续执行中断以前的工作。这样做的优点就是可以使 CPU 和外设同时工作，使系统可以及时地响应外部事件，也可允许多个外设同时工作，大大提高了 CPU 的利用率，也提高了数据输入、输出的速度。同时也可以使 CPU 能够及时处理各种软硬件故障。

CPU 响应外部按键的方式有两种，一种是查询方式，一种是中断方式。

查询方式：CPU 不断的查询是否有按键按下，如果有按键按下的话，就执行相应的程序，否则继续查询。中断方式：CPU 处理自己的工作，如果有按键按下，向 CPU 发出中断请求。CPU 停下现在正在处理的工作，转去执行中断程序，执行之后回来继续刚才的工作。

当 CPU 工作于查询方式的时候，要不间断的对外部按键进行查询，其间

CPU 不能进行其他的任何工作，CPU 的大量时间可能会浪费在原地踏步的查询操作上。为进一步提高 CPU 的工作效率，采用中断方式更好。其过程为：CPU 执行主体程序，如果有按键按下，向 CPU 发出中断请求，CPU 就会停下现在正在处理的工作，转去执行中断处理程序，执行完后回来继续刚才的工作；如果没有按键按下，CPU 就做自己的工作，不理睬外部按键。

1.2. 实验介绍

本次实验运用到了 GPIO 的输入、输出、中断的功能。通过按键中断的方式改变 LED 灯的亮灭状态。程序运行后，首先让 LED1 和 LED3 点亮，LED0 和 LED4 熄灭，按下按键 0 改变 LED2 和 LED3 的状态，松开恢复到之前的状态，按下按键 1 改变 LED0 和 LED1 的状态，松开恢复到之前的状态。

1.3. 建立 HQPFGA 工程

在 HQFPGA 编译环境下建立硬件工程，在 RTL 代码中以原语的方式调用 CM3 内核。

1. 打开 HQFPGA 软件，点击右边导航栏中的新建工程一栏，弹出如下图 1.1 所示的界面，依次选择工程存放的位置，这里新建一个“CMS_System”文件夹用来存放 HQ 工程，然后给工程命名，点击下一步。

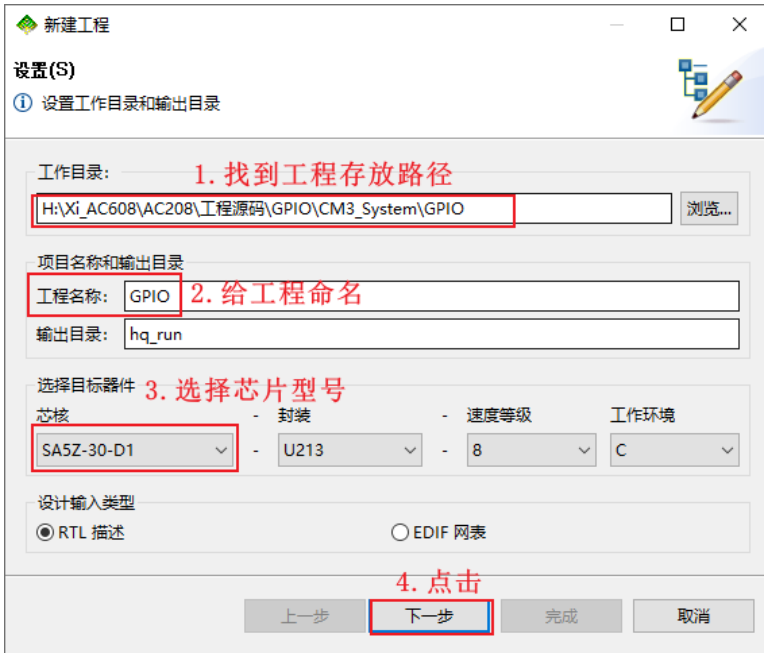

图 1. 1 建立 HQ 工程

弹出添加源文件的界面，如下图 1.2 所示，添加需要的源文件，如果没有，则直接点击完成即可。

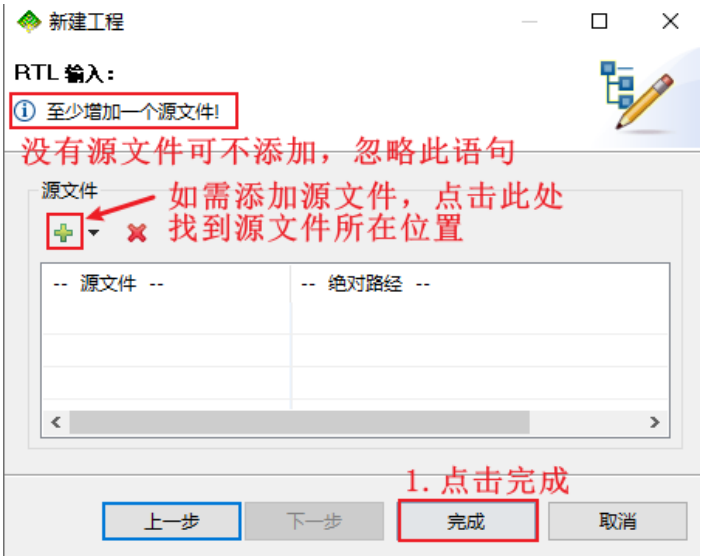

图 1. 2 添加源文件

1.4. 添加顶层文件

1.4.1.1. 新建文件

点击设计管理，进入设计管理器，依次点击【文件】->【新建文件】如下图 1.3 所示。需要注意的是，如果已经打开了其他工具，如调试、下载、物理约束、时序约束等其它窗口，必须先将其关闭，才能打开设计管理。

图 1. 3 建立顶层文件

1.4.1.2. 添加顶层文件代码

本次实验的顶层文件代码如下所示，这里主要调用了“cm3_system”模块，主要是用来初始化 CM3 内核：

```
module LED_KEY(
    input    wire                MAIN_CLK,        //CM3 时钟
    output   wire                CLKOUT,
    output   wire                PCLK,
    output   wire                MTX_CLK,
```

```
    input    wire                    MTXRSTN,        //CM3 复位

    input    wire                    swdclk,         //debug 接口
    inout    wire                    swddio,         //debug 接口
    inout[31:0]                      GPIO            //32 位通用 GPIO
);
cm3_system cm3_system(
    .MAIN_CLK(MAIN_CLK),
    .CLKOUT(CLKOUT),
    .PCLK(PCLK),
    .MTX_CLK(MTX_CLK),
    .MTXRSTN(MTXRSTN),
    .swdclk(swdclk),
    .swddio(swddio),
    .GPIO(GPIO)
);
endmodule
```

下面将简单的介绍一下初始化 CM3 内核代码内容，完整代码如下所示。

```
module cm3_system(
input    wire                    MAIN_CLK,         //CM3 时钟
output   wire                    CLKOUT,
output   wire                    PCLK,
output   wire                    MTX_CLK,
input    wire                    MTXRSTN,        //CM3 复位

input    wire                    swdclk,         //debug 接口
inout    wire                    swddio,         //debug 接口
inout[31:0]                      GPIO            //32 位通用 GPIO
);

wire  [31:0]    GPIOO;
wire  [31:0]    GPIOOEN;
wire            int_jtag_tmso;
wire            int_jtag_tmsoen;

wire  [31:0]    TARGEXP0HADDR;
wire  [2:0]     TARGEXP0HBURST;
wire            TARGEXP0HMASTLOCK;
wire  [3:0]     TARGEXP0HPROT;
wire  [31:0]    TARGEXP0HRDATA;
wire            TARGEXP0HREADYMUX;
wire            TARGEXP0HREADYOUT;
wire            TARGEXP0HRESP;
wire            TARGEXP0HSEL;
wire  [2:0]     TARGEXP0HSIZE;
wire  [1:0]     TARGEXP0HTRANS;
wire  [31:0]    TARGEXP0HWDATA;
```

```
wire            TARGEXP0HWRITE;

wire  [31:0]    TARGEXP1HADDR;
wire  [2:0]     TARGEXP1HBURST;
wire            TARGEXP1HMASTLOCK;
wire  [3:0]     TARGEXP1HPROT;
wire  [31:0]    TARGEXP1HRDATA;
wire            TARGEXP1HREADYMUX;
wire            TARGEXP1HREADYOUT;
wire            TARGEXP1HRESP;
wire            TARGEXP1HSEL;
wire  [2:0]     TARGEXP1HSIZE;
wire  [1:0]     TARGEXP1HTRANS;
wire  [31:0]    TARGEXP1HWDATA;
wire            TARGEXP1HWRITE;
wire            cib_clk;
xsCM3 inst(
    .DFTSE              (1'b0),
    .DFTSI0             (1'b0),
    .DFTSI1             (1'b0),
    .DFTSI2             (1'b0),
    .DFTTM              (1'b0),
    .PLL_OUT            (),
    .PLL_USRCLK         (),
    .CIBCLK             (MAIN_CLK),
    .CLKOUT             (CLKOUT),
    .PCLKEN             (),

    .DBG_SWDI_TMS       (swddio),
    .DBG_SWDO           (int_jtag_tmso),
    .DBG_SWDO_EN        (int_jtag_tmsoen),
    .TDO_ENABLE         (),
    .TDO_TMS            (),
    .PB24_DO_C          (),
    .PB24_TO_C          (),
    .PB24_DO_D          (),
    .PB24_TO_D          (),
    .CS_TDI             (1'b0),             // JTAG TDI
    .CS_TCK             (swdclk),           // SWD Clk / JTAG TCK

    .CPURSTN            (1'b1),             //CPURSTN
    .EXTINT             (16'b0),
    .GPIOI              (GPIO),
    .GPIOO              (GPIOO),
    .GPIOOEN            (GPIOOEN),// DMA request signals
    .DMACBREQ           (4'b0),   // DMA burst transfer request
    .DMACLBREQ          (4'b0), // DMA last burst transfer request
```

```
    .DMACSREQ           (4'b0),   // DMA single transfer request
    .DMACLSREQ          (4'b0), // DMA last single transfer request
     // DMA response signals
    .DMACCLR            (),           // DMA request clear
    .DMACTC             (),           // DMA terminal count

    .INITEXP0HADDR      (32'b0),
    .INITEXP0HBURST     (3'b0),
    .INITEXP0HMASTLOCK  (1'b0),
    .INITEXP0HPROT      (4'b0),
    .INITEXP0HRDATA     (),
    .INITEXP0HREADY     (),
    .INITEXP0HRESP      (),
    .INITEXP0HSEL       (1'b0),
    .INITEXP0HSIZE      (3'b0),
    .INITEXP0HTRANS     (2'b0),
    .INITEXP0HWDATA     (32'b0),
    .INITEXP0HWRITE     (1'b0),
    .INITEXP1HADDR      (32'b0),
    .INITEXP1HBURST     (3'b0),
    .INITEXP1HMASTLOCK  (1'b0),
    .INITEXP1HPROT      (4'b0),
    .INITEXP1HRDATA     (),
    .INITEXP1HREADY     (),
    .INITEXP1HRESP      (),
    .INITEXP1HSEL       (1'b0),
    .INITEXP1HSIZE      (3'b0),
    .INITEXP1HTRANS     (2'b0),
    .INITEXP1HWDATA     (32'b0),
    .INITEXP1HWRITE     (1'b0),
    .MTX_CLK            (MTX_CLK),//AHB CLK
    .MTXRSTN            (MTXRSTN),//AHB rst
    .NSRST              (1'b1), //CM3 内核
    .NTRST              (1'b1), //cm3 内核
    .PCLK               (PCLK),
    .TARGEXP0HADDR      (TARGEXP0HADDR),
    .TARGEXP0HBURST     (TARGEXP0HBURST),
    .TARGEXP0HMASTLOCK  (TARGEXP0HMASTLOCK),
    .TARGEXP0HPROT      (TARGEXP0HPROT),
    .TARGEXP0HRDATA     (TARGEXP0HRDATA),
    .TARGEXP0HREADYMUX  (TARGEXP0HREADYMUX),
    .TARGEXP0HREADYOUT  (TARGEXP0HREADYOUT),
    .TARGEXP0HRESP      (TARGEXP0HRESP),
    .TARGEXP0HSEL       (TARGEXP0HSEL),
    .TARGEXP0HSIZE      (TARGEXP0HSIZE),
    .TARGEXP0HTRANS     (TARGEXP0HTRANS),
    .TARGEXP0HWDATA     (TARGEXP0HWDATA),
```

```
    .TARGEXP0HWRITE     (TARGEXP0HWRITE),
    .TARGEXP1HADDR      (TARGEXP1HADDR),
    .TARGEXP1HBURST     (TARGEXP1HBURST),
    .TARGEXP1HMASTLOCK  (TARGEXP1HMASTLOCK),
    .TARGEXP1HPROT      (TARGEXP1HPROT),
    .TARGEXP1HRDATA     (TARGEXP1HRDATA),
    .TARGEXP1HREADYMUX  (TARGEXP1HREADYMUX),
    .TARGEXP1HREADYOUT  (TARGEXP1HREADYOUT),
    .TARGEXP1HRESP      (TARGEXP1HRESP),
    .TARGEXP1HSEL       (TARGEXP1HSEL),
    .TARGEXP1HSIZE      (TARGEXP1HSIZE),
    .TARGEXP1HTRANS     (TARGEXP1HTRANS),
    .TARGEXP1HWDATA     (TARGEXP1HWDATA),
    .TARGEXP1HWRITE     (TARGEXP1HWRITE),
    .TRACECLK           (),
    .TRACEDATA          (),
    .TREECLK            (1'b0)
);

defparam inst.PCLK_DIV = 0;             // 分频 PCLK_DIV 0-7
defparam inst.CORECLK = "CIB_CLK";      // 时钟来源 "CLK_TREE"
"CLK_PAD" "PLL_OUT" "CIB_CLK" "PLL_USRCLK"
defparam inst.RSTN_ENABLE = "TRUE";
defparam inst.MTXCLK = "CORECLK";       // AHB 总线时钟 "CORECLK"
"SOURCECLK"
defparam inst.CORECLK_EN = "TRUE";
defparam inst.CORE_SET = "TRUE";
//GPIO 的三态处理
genvar          i;
generate
    for(i=0;i<32;i=i+1)begin:setb
        assign GPIO[i] = GPIOOEN[i] ? GPIOO[i] : 1'bz;
    end
endgenerate
//SW 模式的 SWDIO 信号三态处理
assign  swddio = int_jtag_tmsoen ? 1'bz : int_jtag_tmso;
endmodule
```

CM3 端口包括 debug 端口、32 个 GPIO、DMA、AHB_master 以及 AHB_slave 端口。其中 32 个 GPIO 端口可复用为 UART、I2C、SPI 及中断等端口，并且需要连接至工程顶层通过 FPGA 引脚连接至片外。CM3 内核的初始化可通过参数进行设置，主要包括 APB 总线分频系数、时钟来源及复位使能，本次实验设置的具体参数如下所示：

```
defparam inst.PCLK_DIV = 0;             // 分频 PCLK_DIV 0-7
defparam inst.CORECLK = "CIB_CLK";      // 时钟来源 "CLK_TREE"
"CLK_PAD" "PLL_OUT" "CIB_CLK" "PLL_USRCLK"
```

```
defparam inst.RSTN_ENABLE = "TRUE";
defparam inst.MTXCLK = "CORECLK";        // AHB 总线时钟 "CORECLK"
"SOURCECLK"
defparam inst.CORECLK_EN = "TRUE";
defparam inst.CORE_SET = "TRUE";
```

在对 SA5Z 系列 SoC FPGA 中集成的 Cortex-M3 CPU 进行调试和下载，需要引出对应的引脚，并对 swdio 做双向数据传输使能，代码如下所示：

```
.DBG_SWDI_TMS        (swddio),
.DBG_SWDO            (int_jtag_tmso),
.DBG_SWDO_EN         (int_jtag_tmsoen),
.CS_TCK              (swdclk),              // SWD Clk / JTAG TCK
assign  swddio = int_jtag_tmsoen ? 1'bz : int_jtag_tmso;
```

同样地，在 FPGA 工程中，需要将 GPIO 描述为双向端口，代码如下所示：

```
genvar          i;
generate
    for(i=0;i<32;i=i+1)begin:setb
        assign GPIO[i] = GPIOOEN[i] ? GPIOO[i] : 1'bz;
    end
endgenerate
```

1.4.1.3. 保存顶层文件

编写完成之后，点击保存图标（不支持 Ctrl+S）以保存文件，在弹出的界面中，建立一个 rtl 文件用来存放源码，然后给文件命名，其名字与你定义的顶层模块名一致，步骤如下图 1.4 所示。

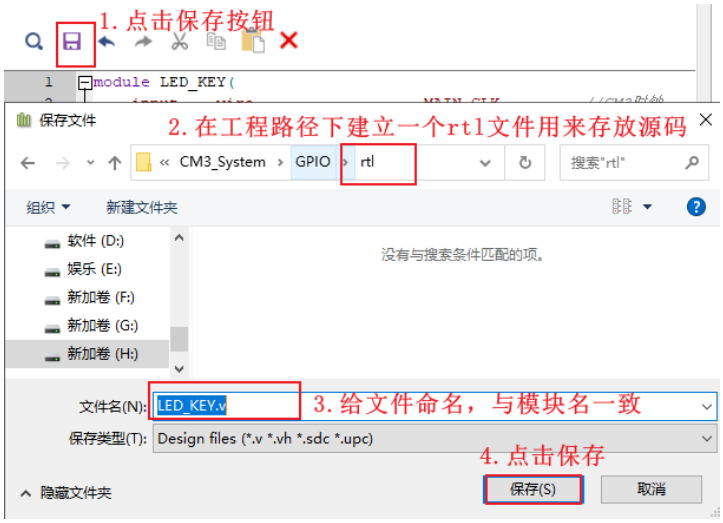

图 1. 4 保存顶层文件代码

1.4.1.4. 将顶层文件添加进工程

由于设计管理器不会自动对创建的文件添加进工程，因此需要我们手动点击添加文件按钮，选择我们刚刚保存的“LED_KEY.v”文件，操作步骤如下图

1.5 所示。

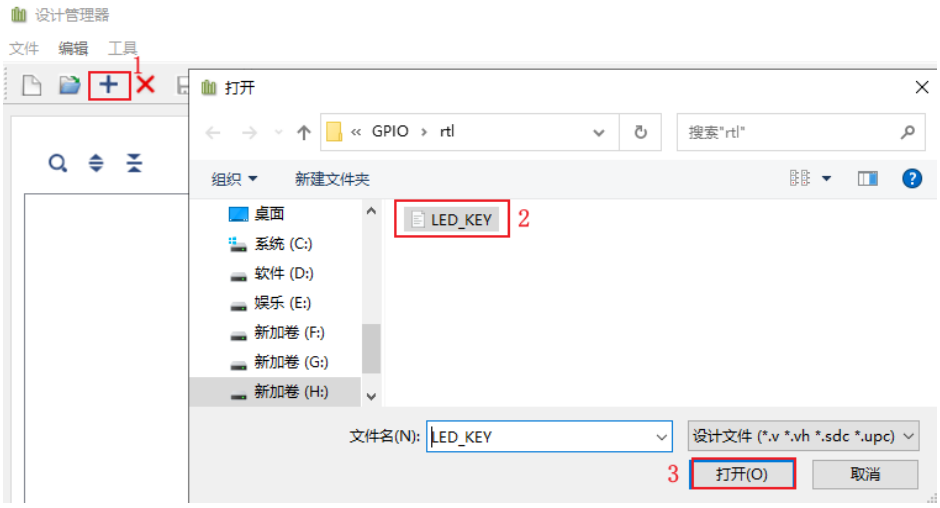

图 1.5 添加顶层文件至工程

将文件添加完成之后，这时我们可以看到，会弹出语法检查失败的提示框，并且提示如下图 1.6 所示的错误

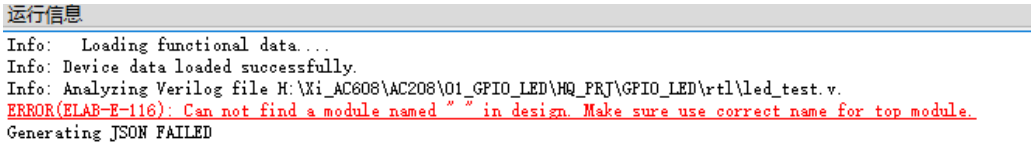

图 1.6 顶层文件提示错误

上图显示错误的原因是工程刚刚创建，还没有指定工程的顶层设计文件，导致设计管理器无法进行语法检测。此时，我们只需要直接关闭设计管理器，弹出如下图 1.7 所示的提示框，直接点击 ok 就可以了。

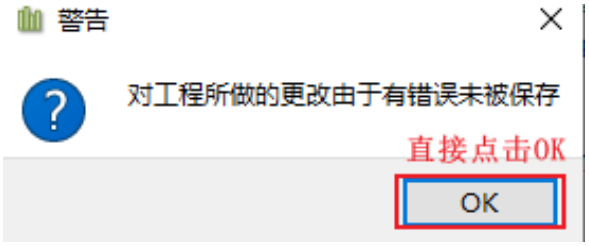

图 1.7 关闭设计管理器错误提示框

上诉步骤操作完成之后，点击工程属性，找到下方的添加源文件的位置，点击加号，找到我们刚刚保存的“LED_KEY.v”文件，操作步骤如下图 1.8 所示。需要注意的是，由于我们调用了“cm3_system”模块，在添加源文件的时候，也需要将该模块进行添加。

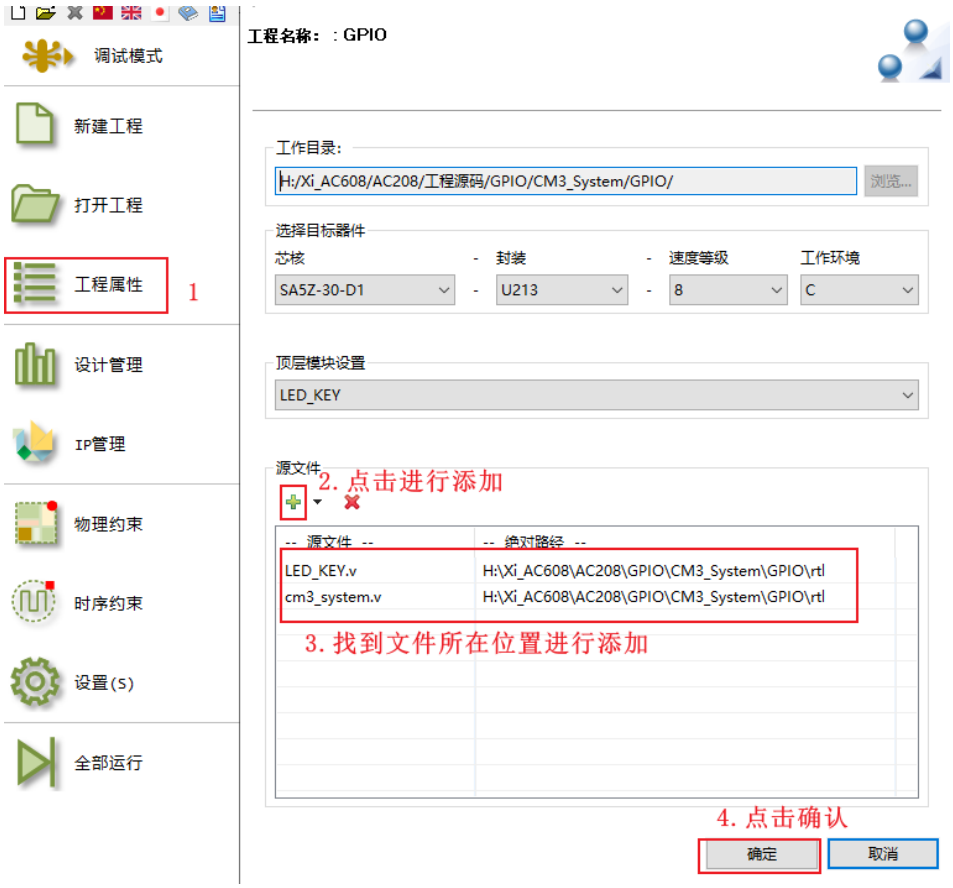

图 1. 8 添加源文件至工程

添加完成之后，再次打开设计管理器，就能够看到设计文件已经添加好了，这时点击语法检查，也能够通过，如果提示语法错误，请根据错误提示的位置进行修改。

1.5. 物理管脚约束

1. 点击物理约束，弹出如下图 1.9 所示的提示框。

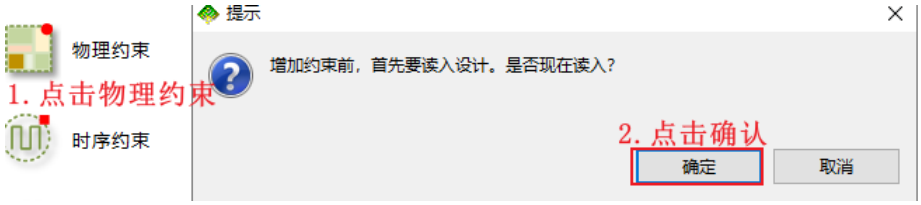

图 1. 9 点击物理约束提示框

2. 读入完成之后，会弹出选择约束方式的对话框，依次点击【约束编辑器】->【确定】如下图 1.10 所示。

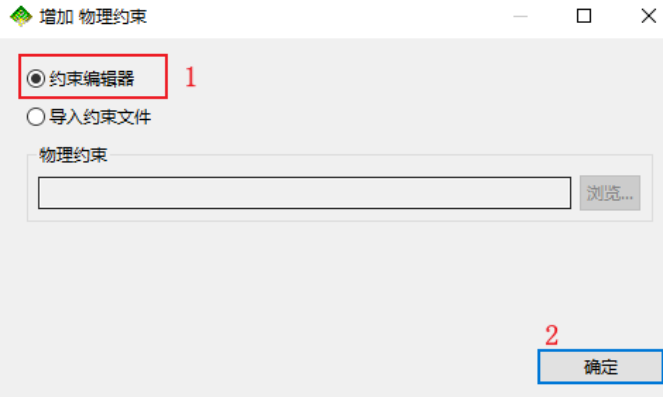

图 1. 10 添加物理约束界面

3. 在弹出的物理约束界面中，根据物理电路板信息，输入对应的管脚名，本次实验所需要使用引脚分配如下表 1-3 所示。

表 1- 3 引脚分配表

引脚名称	AC208-SA5Z 对应的引脚编号	AC208 对应的管脚编号
MAIN_CLK	H13	C12
MTXRSTN	F5(KEY2)	R13(KEY2)
swddio	C1	P6
swdclk	H1	R5
GPIO[21]	D15(LED0)	P8(LED0)
GPIO[22]	G15(LED1)	K10(LED1)
GPIO[23]	F14(LED2)	K9(LED2)
GPIO[24]	E14(LED3)	L6(LED3)
GPIO[26]	E5(KEY1)	L3(KEY1)
GPIO[27]	D5(KEY0)	N2(KEY0)

将上述的引脚编号添加至对应的引脚位置，添加完成之后如下图 1.11 所示。

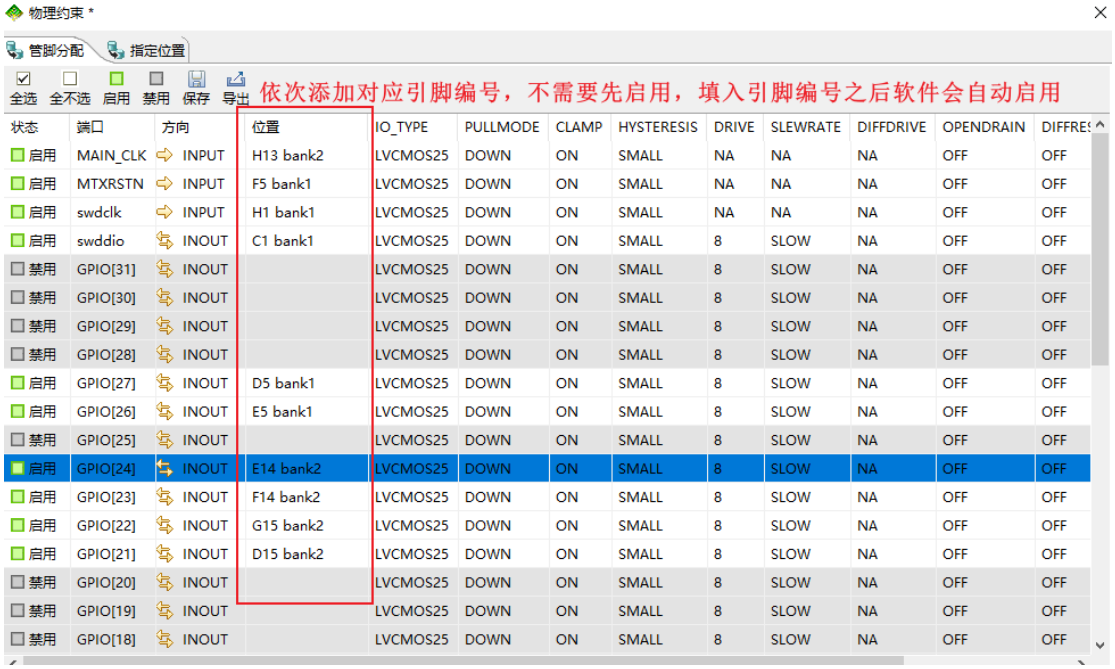

图 1. 11 添加引脚界面示意图

4. 引脚添加完成之后，先别保存，点击上方的导出按钮，导出引脚约束文件，然后选择约束文件存放位置，并给约束文件命名，如下图 1.12 所示。

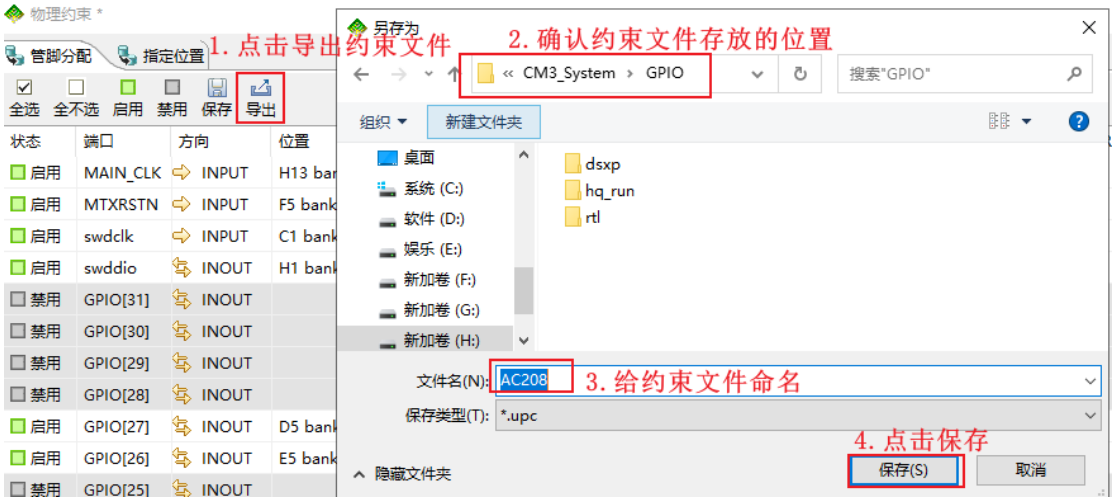

图 1. 12 导出引脚约束文件

5. 关闭物理约束界面，会弹出如下图 1.13 提示框，点击“否”。

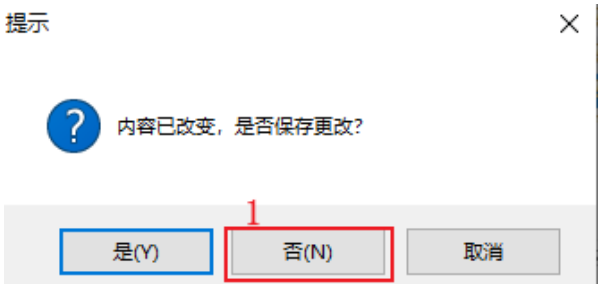

图 1. 13 关闭物理约束提示框

6. 再次点击物理约束，找到我们刚刚导出的物理约束文件，进行添加。添加物理约束文件的操作方式如下图 1.14 所示。对于 AC208 开发板，我们也提供有对应的引脚约束文件，里面对于比较常用的引脚，都已经分配完成，用户在使用的时候可以直接将其复制至工程目录下进行添加。

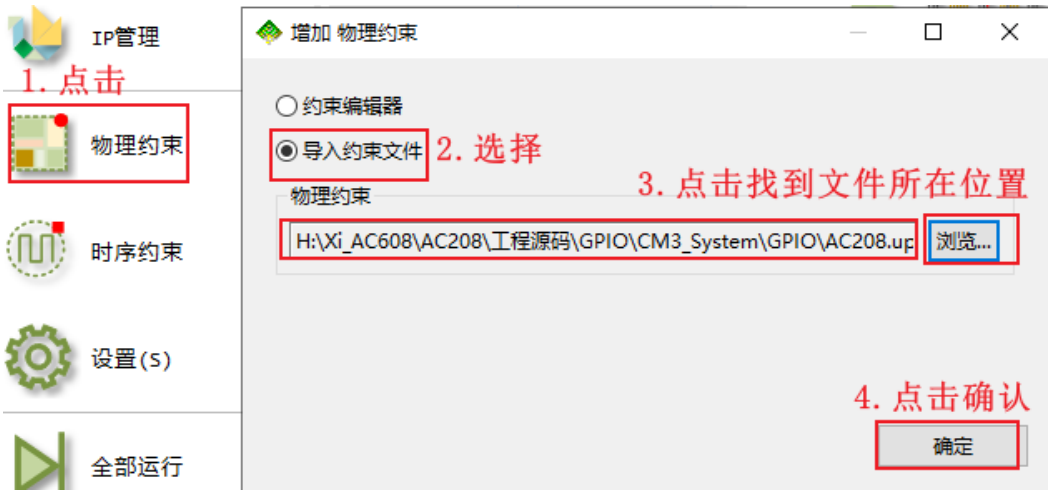

图 1. 14 添加物理约束文件

如需查看物理约束文件，进入设计管理器，点击下方的设计文件，即可查看，具体操作步骤如下图 1.15 所示。

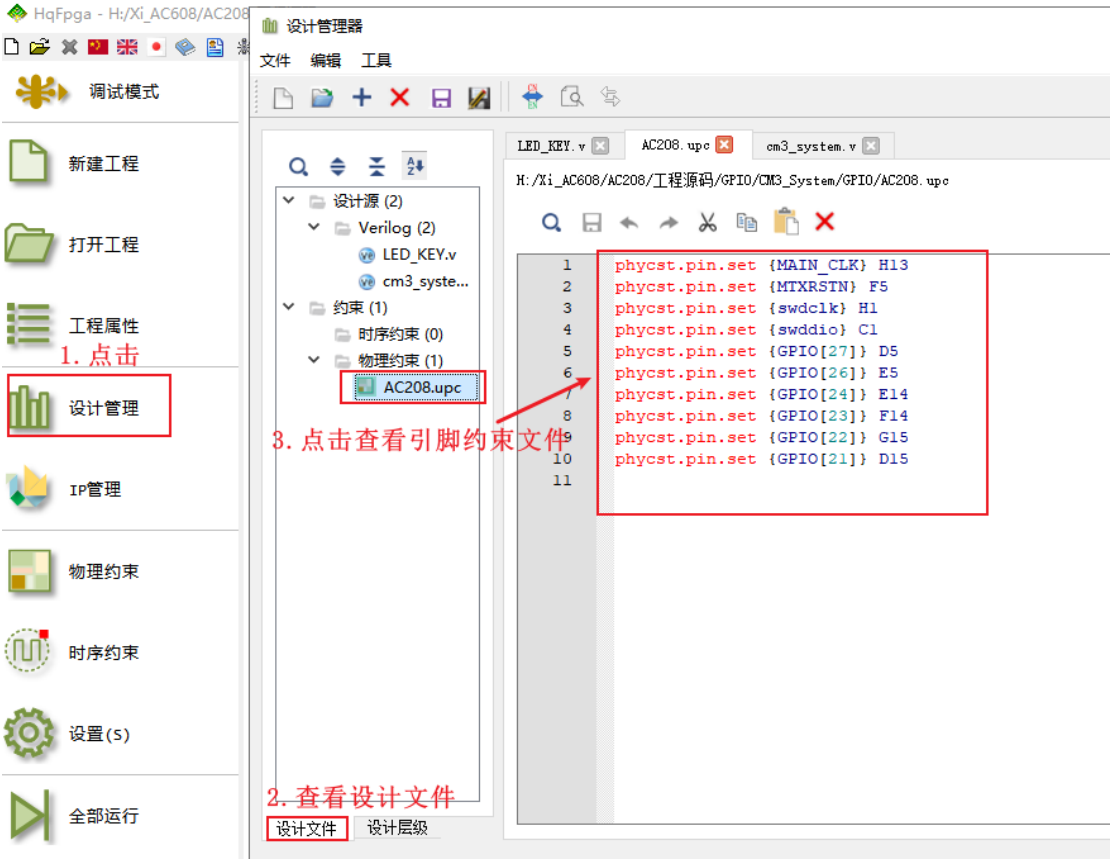

图 1. 15 查看引脚约束文件

1.6. 编译设计

对于 SA5Z 系列 FPGA，HQFPGA 软件默认生成 bit 格式的配置文件，位于工程目录下的 hq_run 文件夹下，该格式文件就像 Quartus 软件的 sof 文件，以及 Vivado 软件的 bit 文件，可以用来在线烧写到 FPGA 的 sram 中运行。但是无法烧录到 FPGA 中运行。

如果需要将 FPGA 程序烧录到 FPGA 的配置 Flash 器件中，实现上电自动配置运行，则需要生成 bin 格式的烧录文件。而且 bin 格式也可以像 bit 一样直接下载进 FPGA 的 SRAM 运行。

生成 bin 格式的文件非常简单，只需要先在 HQFPGA 打开对应工程，然后点击设置按钮，在打开的界面中的“位流生成”中勾选上“生成二进制文件格式”，具体操作步骤如下图 1.16 所示。

图 1. 16 生成 bin 格式的文件操作示意图

点击“全部运行”按钮，对设计进行全编译并生成编程 bin 文件。操作步骤如下图 1.17 所示。

图 1. 17 编译运行整个文件

运行完成之后，我们可以查看对应的资源利用率，查看步骤如下图 1.18 所示。

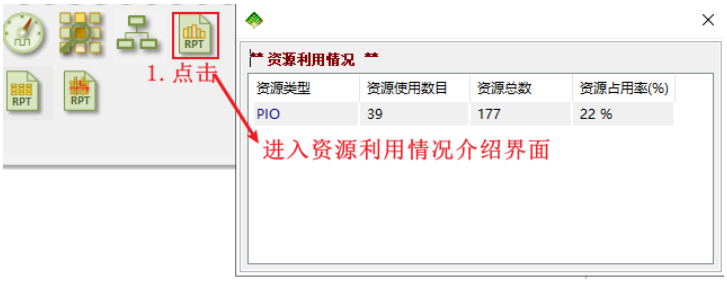

图 1. 18 查看资源利用情况操作示意图

由上图可以看出，此时只有 PIO 的资源占用情况，这是因为这个工程没有包括 FPGA 侧的逻辑，所以看到的资源占用只有 PIO。如果使用到了 FPGA 的逻辑设计，其资源占用情况会不一样，例如在笔者自己设计的一个既包括 CM3 硬核，又包括 FPGA 逻辑设计的工程中，就能看到其他资源（PLL、IOLOGIC、SLICE 等）的占用情况，如下图 1.19 所示。

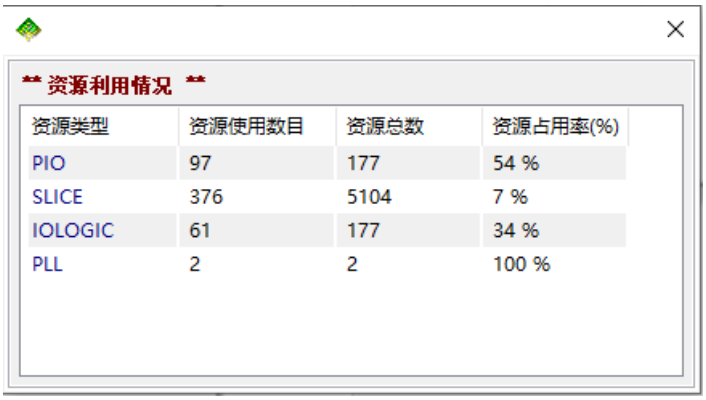

图 1. 19 包含 FPGA 逻辑设计的资源利用情况图

1.7. 建立 MDK 工程

SA5Z-30 FPGA 的 CM3 核是通过 SOC 方式集成在芯片内部，在对其进行软件编程的时候需要采用第三方的编译器软件，在这里我们使用的编译器是 Keil uVision5，工程建立步骤如下所示。

1.7.1. 创建新工程

1. 打开 MDK 软件，点击 Project->New uVision Project，创建一个新工程，如下图 1.20 所示。

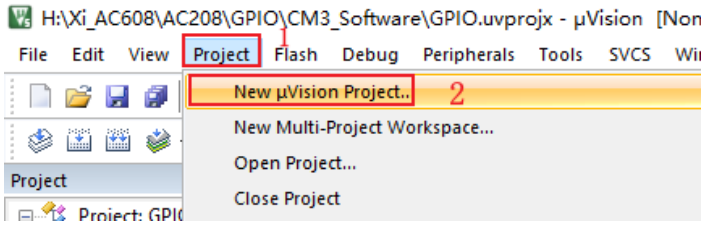

图 1. 20 新建 MDK 工程

2. 在弹出的保存文件的界面中，新建一个 CM3_Softerware 文件夹，用来存放 MDK 工程，并将工程命名为 GPIO，保存工程操作界面如下图 1.21 所示。

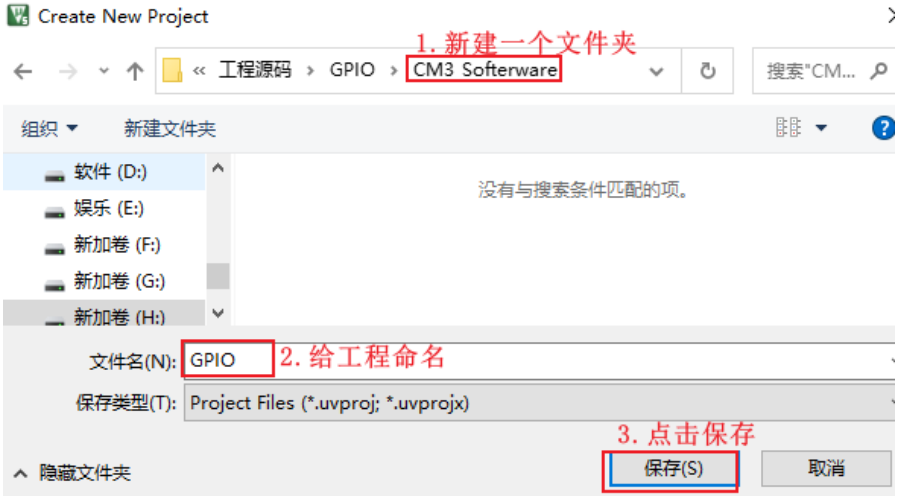

图 1. 21 保存工程界面

3. 在弹出的选择器件类型的界面中，依次选择 ARM-> ARM Cortex M3-> ARM CM3，如下图 1.22 所示。

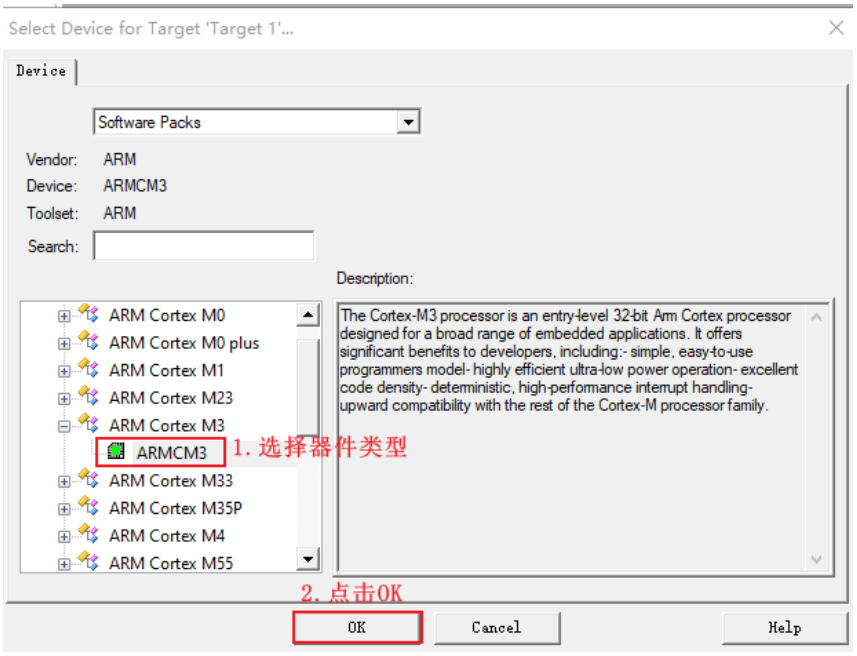

图 1. 22 选择器件类型

4. 弹出 Manage Run-Time Environment 的对话框，如下图 1.23 所示。

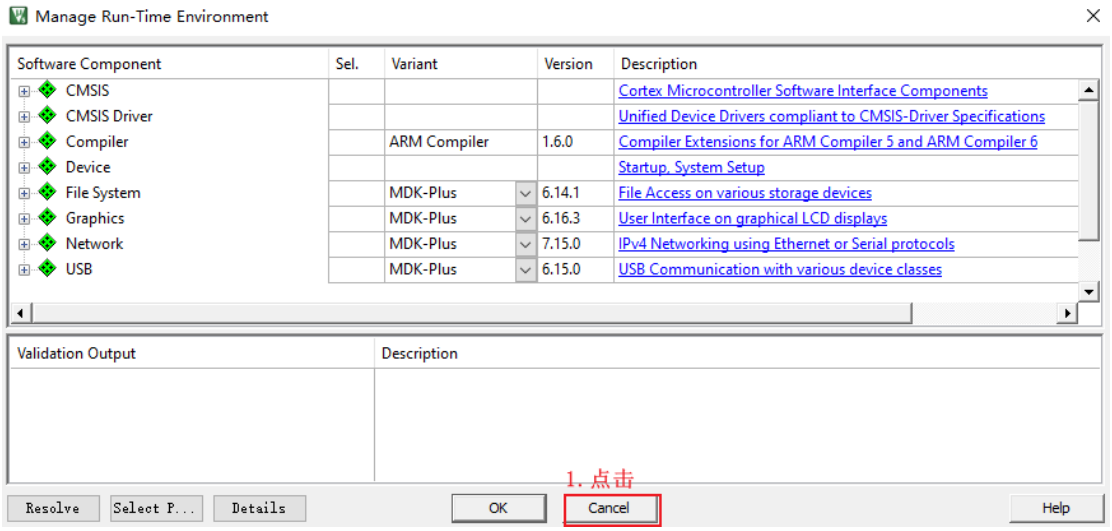

图 1. 23 Manage Run-Time Environment 界面

上图显示的界面中，我们可以添加自己需要的组件，从而方便构建开发环境，这里我们不需要改变，直接点击 Cancel 即可，得到如下图 1.24 所示的界面。这里，我们只是建立一个基本的框架，还需要根据自己的需求添加一个启动文件。

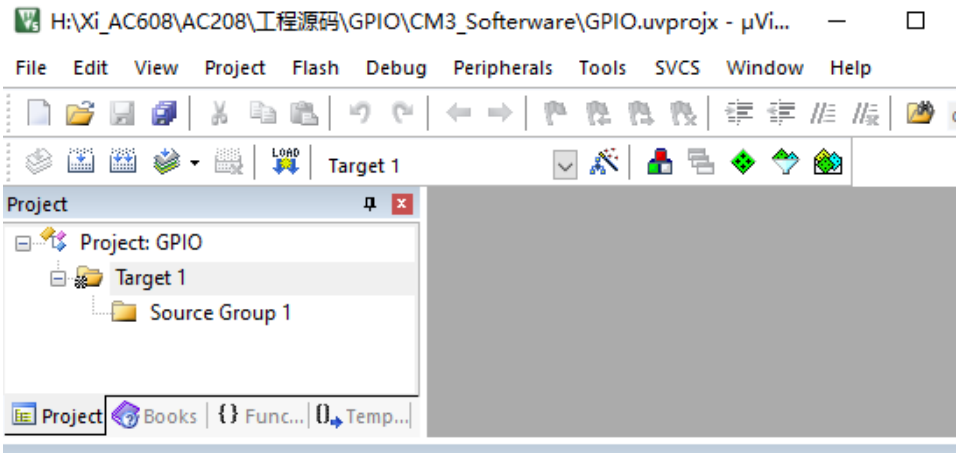

图 1. 24 工程初步建立

1.7.2. 添加所需文件

1. 首先需要将所需的文件复制至我们的工程目录下，可以找到我们提供的例程中其中名为“CMSIS”、“generic”的文件夹，这两个文件夹存放有我们所需的库文件和启动文件，然后建立一个 HARDWARE 文件，用来存放我们编写的库文件，如下图 1.25 所示。

图 1. 25 复制工程文件至新工程目录下

2. 在工程目录下新建 4 个分组（SRC、Startup、HARDWARE、CMSIS），分别用来存放启动文件、库函数文件和自己编写的文件，点击 ，在 Manage Project Items 中点击建立新的分组并命名，操作步骤如下图 1.26 所示。

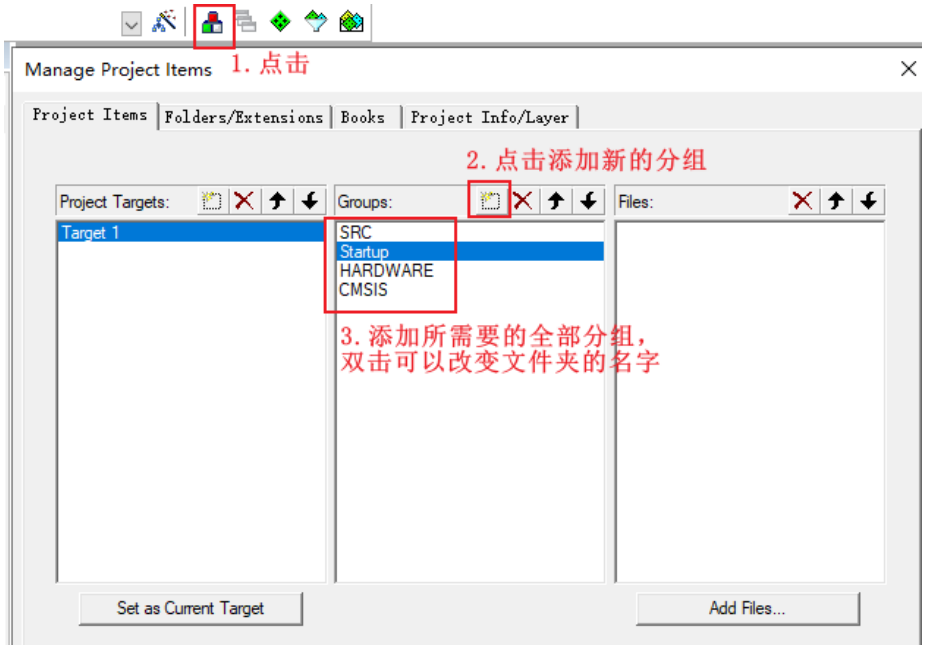

图 1. 26 添加新分组并命名

3. 在上述界面中点击 Add Files 选择需要添加所需的文件。添加启动文件的方式如下图 1.27 所示。

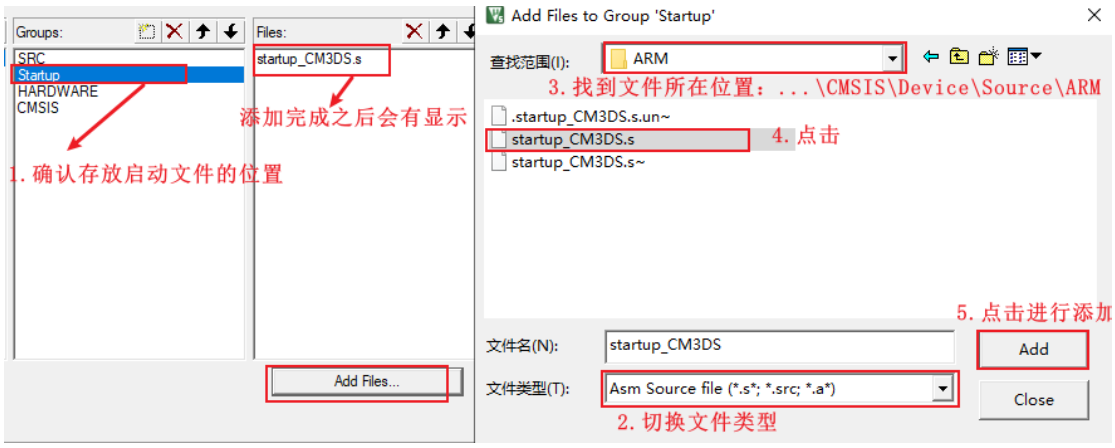

图 1. 27 添加启动文件

4. 添加实验需要的库函数文件，这里添加的都是.c 的文件，所以不需要切换文件类型，操作方式如下图 1.28 所示。

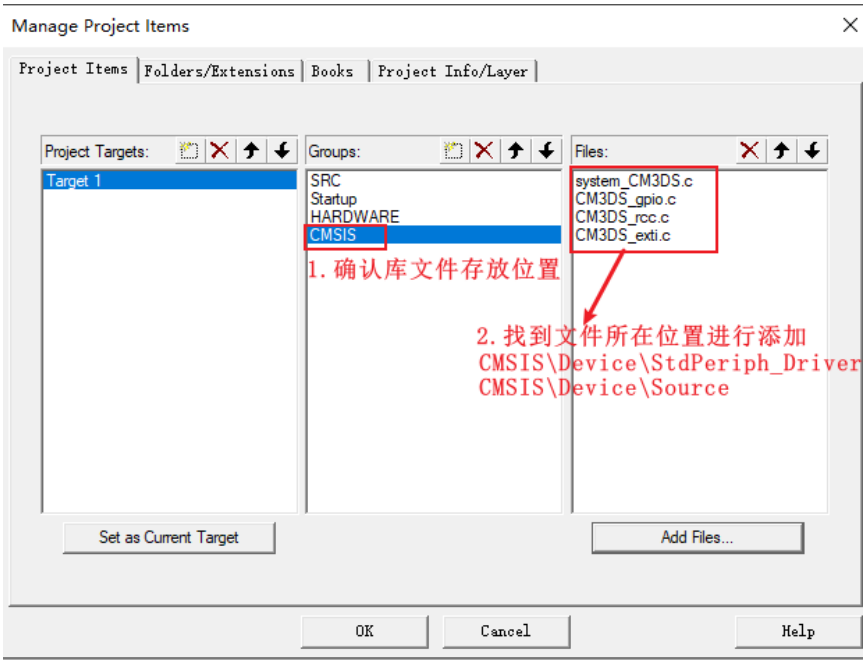

图 1. 28 添加库函数文件

5. 添加 main.c 文件至 SRC 文件夹当中，操作方式如下图 1.29 所示。

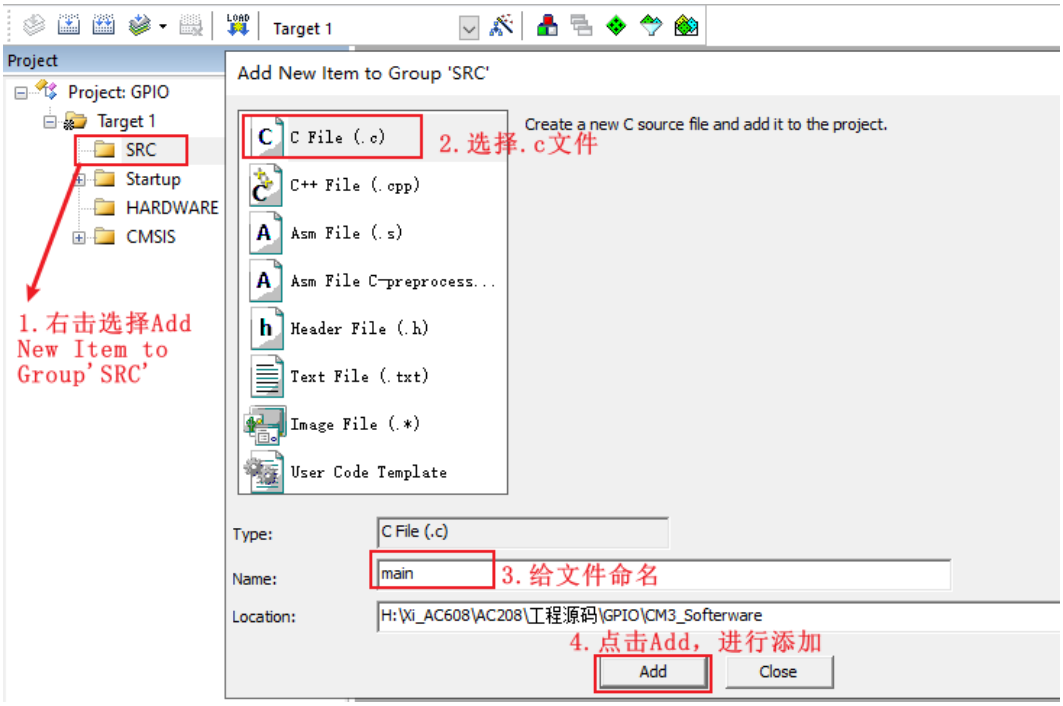

图 1. 29 添加 main 函数文件

6. 添加文件路径，点击，选择 C/C++，然后点击 Include Paths 后面的，添加对应的文件路径，操作步骤如下图 1.30 所示。

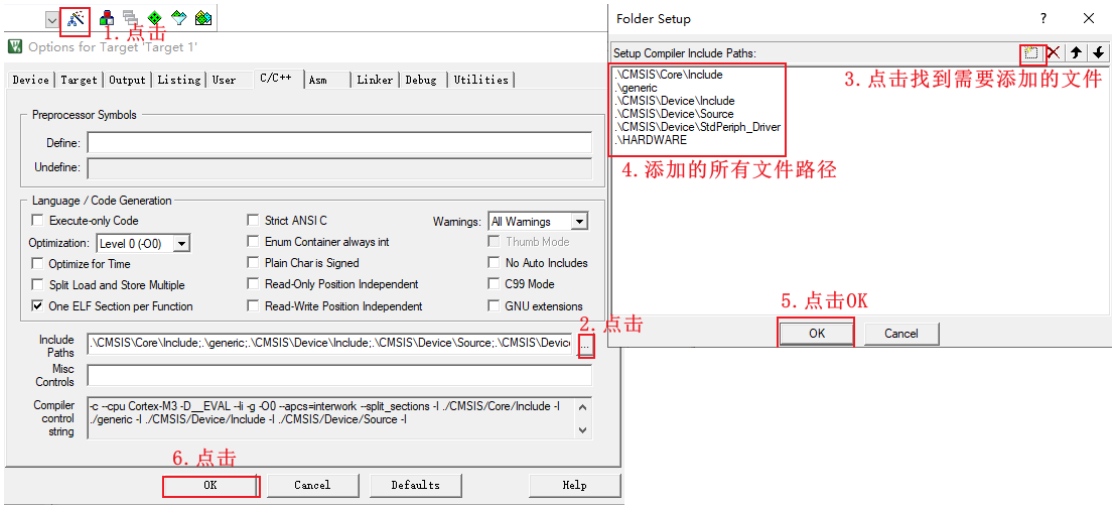

图 1. 30 添加文件路径

1.7.3. 调试软件设置

在需要使用到 Debug 环境进行调试的时候，首先需要在 MDK 软件中进行设置，这里使用到的调试工具是 DAP Link，具体的配置步骤如下所示。

1. 添加 Flash 文件，找到调试文件将其复制到 Keil 安装目录的 Flash 文件夹下，操作步骤如下图 1.31 所示。调试文件“SA30K_EFlash.FLM”在我们提供

的第一个实验“GPIO”例程中的 CM3_System 文件夹之下。

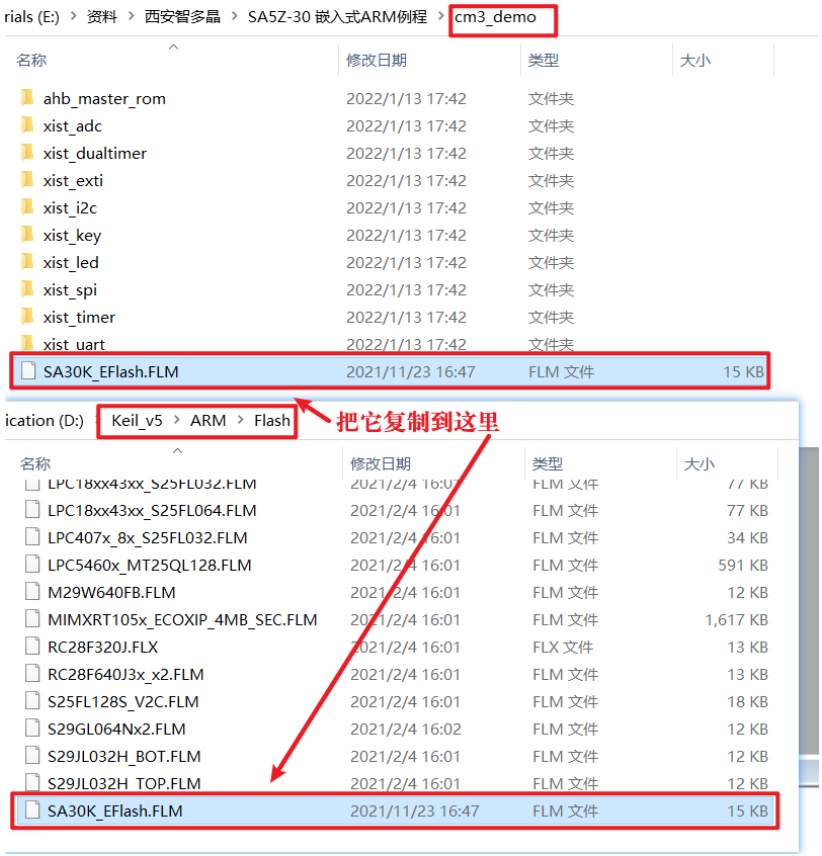

图 1. 31 添加 Flash 文件

2. 设置 Target 栏，需要对 RAM 的 SIZE 进行修改，将其修改为“0x10000”，并且将 ARM Compiler 选项设置为“Use default compiler version5”，如下图 1.32 所示。

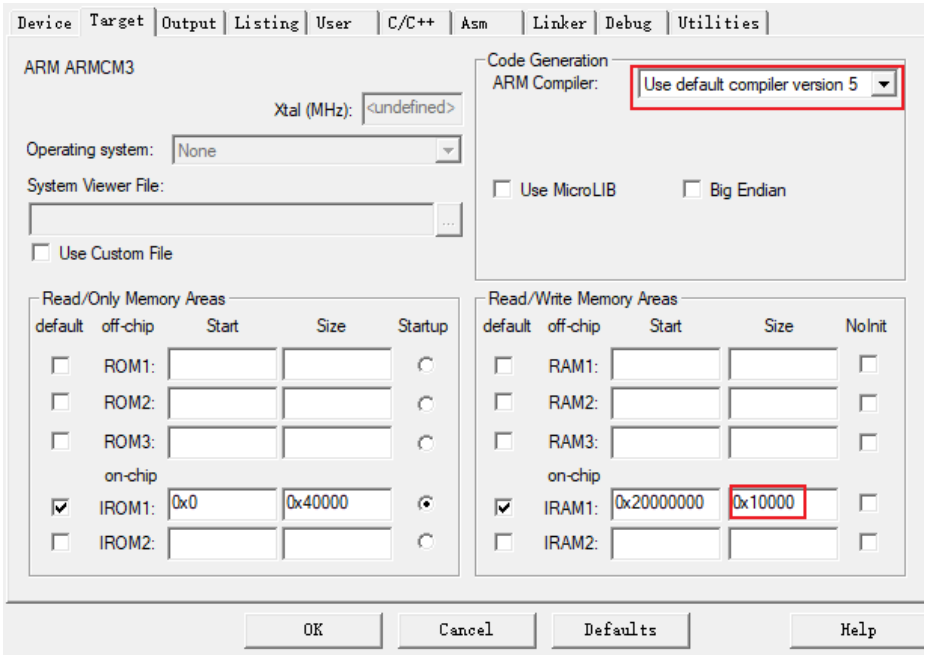

图 1. 32 Target 界面参数设置

3. 设置 Debug 栏，选择 CMSIS-DAP Debugger，如下图 1.33 所示。

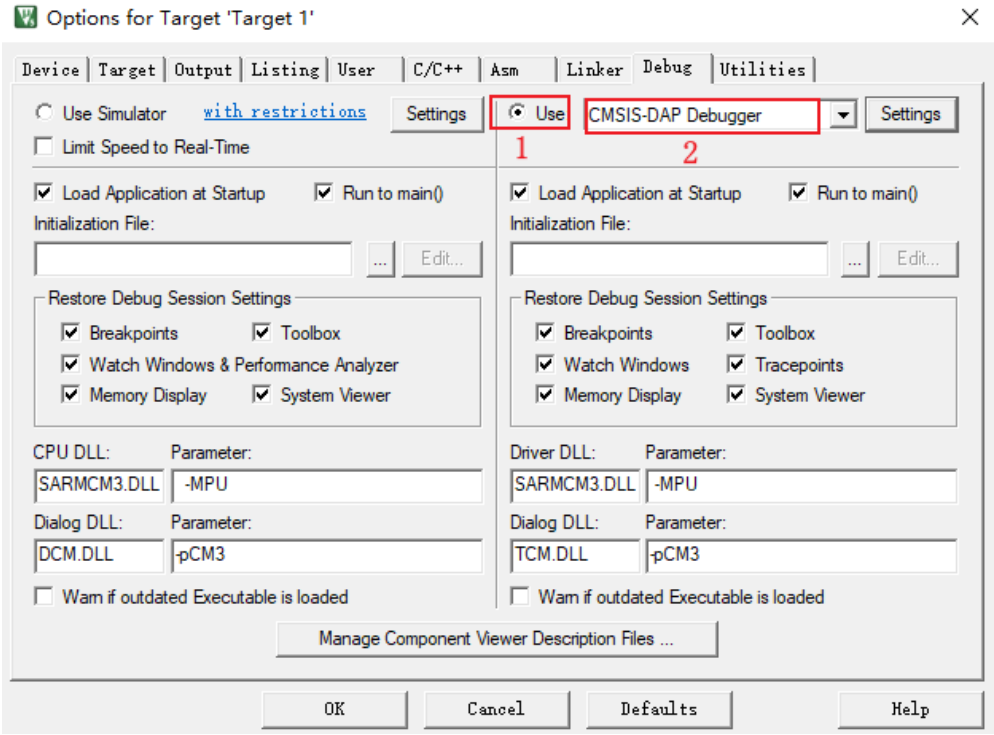

图 1. 33 Debug 界面的设置

然后点击上图中的 “settings”按钮对话框，弹出如下图 1.34 所示的对话框进行设置。需要注意的是下图 IDCORE 中显示有检测到，是因为我们将 DAP Link 已经连接到了开发板上，如果用户不知道怎么连接，参考 1.9.2 节硬件连接中的内容。

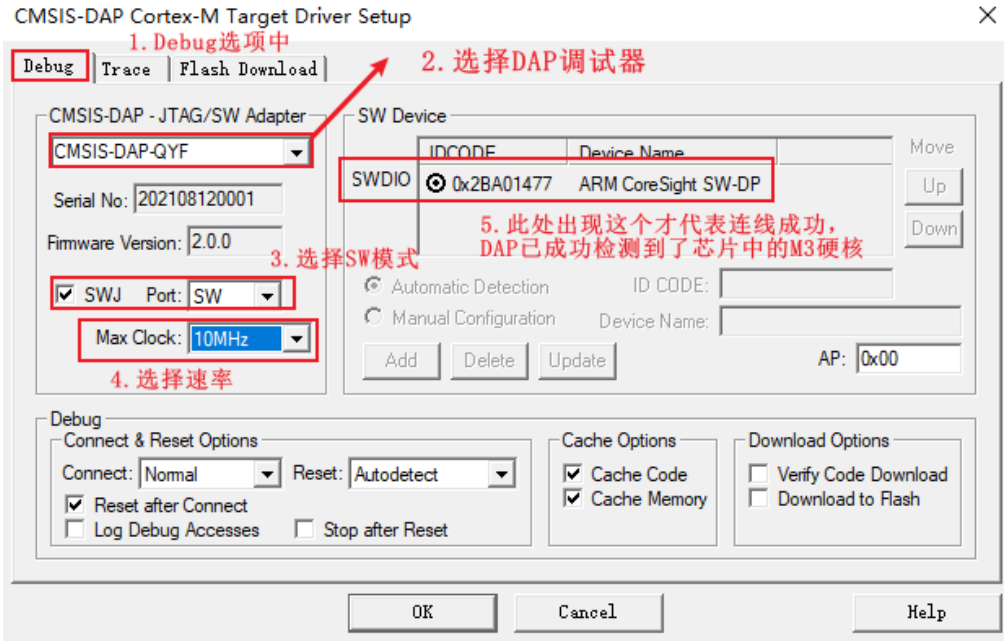

图 1. 34 设置 Debug 选项

在上述的界面中选择 Flash Download，勾选上“Reset and Run”，勾选上之后即可实现下载程序之后自动复位并直接运行，不用手动复位。然后点击“Add”，进入“Add Flash Programming Algorithm”界面选择“Xsit Micro 128k Flash V1.0”选项进行添加。设置方式如下图 1.35 所示。

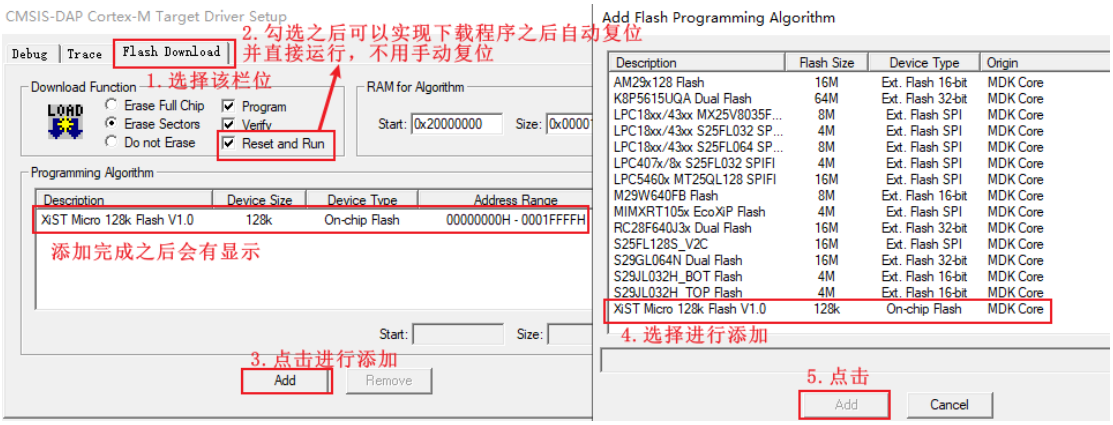

图 1. 35 进入 Flash Download 界面示意图

4. 设置 Utilities 栏，首先取消勾选“Use Debug Driver”，然后选择器件，具体操作步骤如下图 1.36 所示。

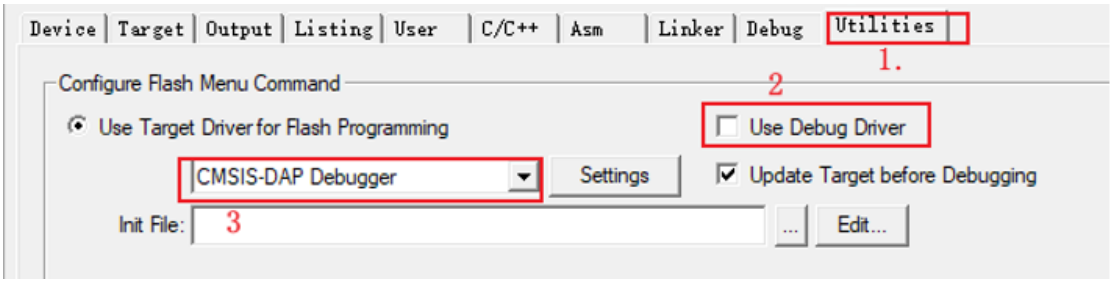

图 1. 36 设置 Utilities 栏

1.7.4. 输出文件的设置

1. 设置 User 一栏，使其输出 bin 文件，设置步骤如下图 1.37 所示。所要添加的路径如下。添加的时候需要注意的是 output 后面的文件名需要与我们的工程名保持一致。

```
$K\ARM\ARMCC\BIN\fromelf.exe #L -c -d -e -s --output GPIO.lst
$K\ARM\ARMCC\BIN\fromelf.exe #L --bin -o GPIO.bin
```

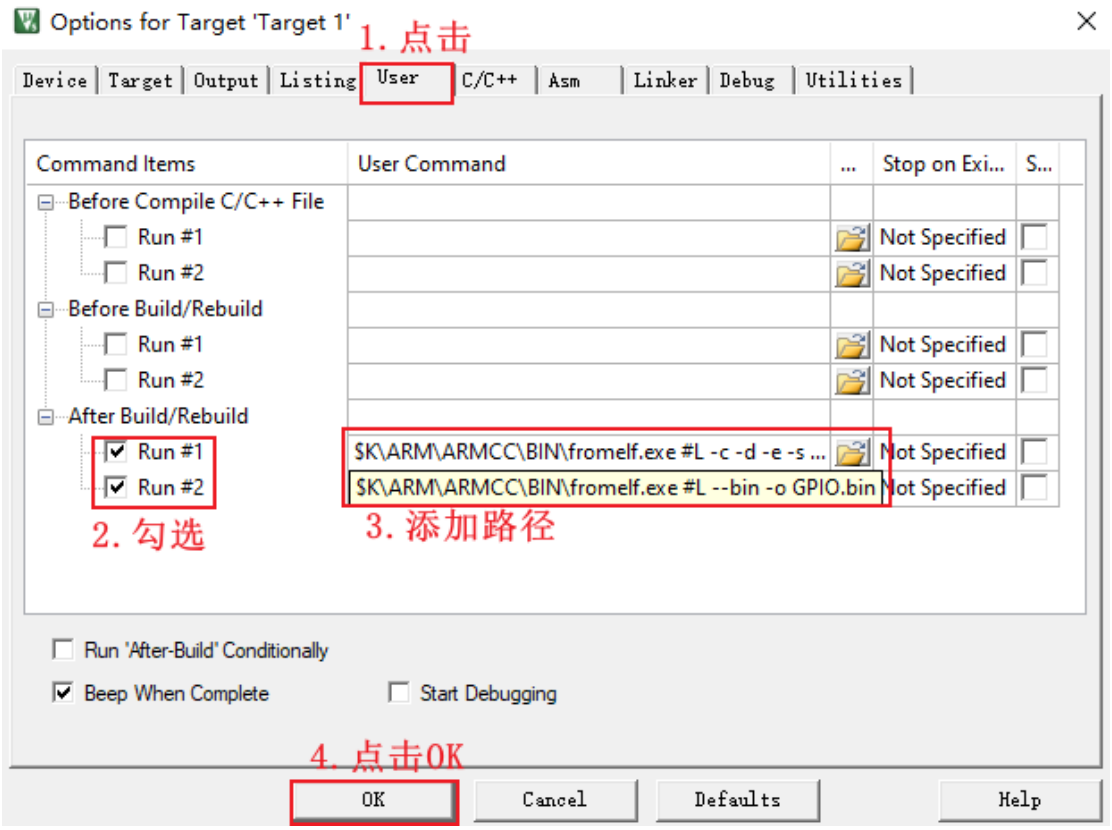

图 1. 37 设置 User

2. 配置 Output 一栏，配置界面如下图 1.38 所示。

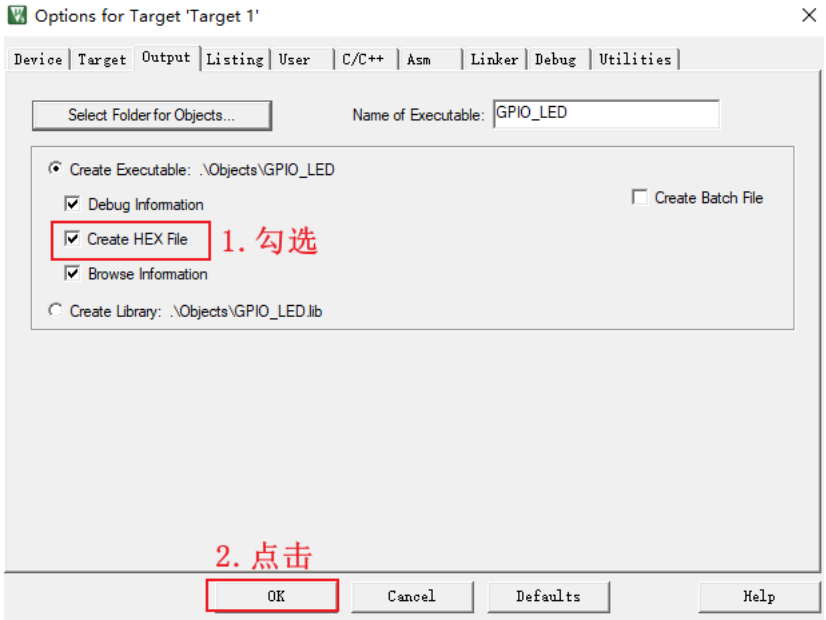

图 1. 38 配置 Output

1.8. 软件设计

本次实验所需要运用到的功能就是对 GPIO 口输出高低电平以及中断的控

制，下面我们将来介绍一下运用到官方库。

1.8.1. GPIO 库

对于 GPIO 口的控制，所需要的函数都在“CM3DS_gpio.c”这个文件之下，下面对于该库中的函数做一下简单的介绍。

1.8.1.1. GPIO_DeInit

该函数的主要功能就是将 GPIO 的外围寄存器初始化为默认重置值，函数变量描述如下表 1-4 所示。

表 1- 4 GPIO_DeInit 函数变量说明表

变量名称	变量意义
CM3DS_MPS2_GPIOx	确定使用哪一组 GPIO，AHB GPIO 只提供了一组 IO 接口，因此第一个参数只能设置为 CM3DS_MPS2_GPIO0

函数内容：首先是检测参数是否符合所定义的，然后就是对 GPIO 口的 ALTFUNCCLR 进行设置，代码如下所示：

```
if (CM3DS_MPS2_GPIOx == CM3DS_MPS2_GPIO0)
{
    CM3DS_MPS2_GPIOx->ALTFUNCCLR = 0xffffffff;
}
```

上述代码所实现的功能就是向 ALTFUNCCLR 写入 0xffffffff。ALTFUNCCLR 代表的是其复用清除功能寄存器，向该寄存器中写入 1，关闭其复用功能。如需查看其在代码中的定义，可以将鼠标放在 CM3DS_MPS2_GPIO_TypeDef 处，然后按下键盘上的 F12 就可以看到其定义的 GPIO 寄存器的结构体，如下图 1.3 9 所示。

```
/*-------------------- General Purpose Input Output (GPIO) --------------------*/
typedef struct
{
  __IO   uint32_t  DATA;                    /* Offset: 0x000 (R/W) DATA Register */
  __IO   uint32_t  DATAOUT;                 /* Offset: 0x004 (R/W) Data Output Latch Register */
         uint32_t  RESERVED0[2];
  __IO   uint32_t  OUTENABLESET;            /* Offset: 0x010 (R/W) Output Enable Set Register */
  __IO   uint32_t  OUTENABLECLR;            /* Offset: 0x014 (R/W) Output Enable Clear Register */
  __IO   uint32_t  ALTFUNCSET;              /* Offset: 0x018 (R/W) Alternate Function Set Register */
  __IO   uint32_t  ALTFUNCCLR;              /* Offset: 0x01C (R/W) Alternate Function Clear Register */
  __IO   uint32_t  INTENSET;                /* Offset: 0x020 (R/W) Interrupt Enable Set Register */
  __IO   uint32_t  INTENCLR;                /* Offset: 0x024 (R/W) Interrupt Enable Clear Register */
  __IO   uint32_t  INTTYPESET;              /* Offset: 0x028 (R/W) Interrupt Type Set Register */
  __IO   uint32_t  INTTYPECLR;              /* Offset: 0x02C (R/W) Interrupt Type Clear Register */
  __IO   uint32_t  INTPOLSET;               /* Offset: 0x030 (R/W) Interrupt Polarity Set Register */
  __IO   uint32_t  INTPOLCLR;               /* Offset: 0x034 (R/W) Interrupt Polarity Clear Register */
  union {
    __I    uint32_t  INTSTATUS;             /* Offset: 0x038 (R/ ) Interrupt Status Register */
    __O    uint32_t  INTCLEAR;              /* Offset: 0x038 ( /W) Interrupt Clear Register */
    };
         uint32_t RESERVED1[241];
  __IO   uint32_t BYTE0_MASKED[256];          /* Offset: 0x400 - 0x7FC Lower byte Masked Access Register (R/W) */
  __IO   uint32_t BYTE1_MASKED[256];          /* Offset: 0x800 - 0xBFC Upper byte Masked Access Register (R/W) */
  __IO   uint32_t BYTE2_MASKED[256];          /* Offset: 0xC00 - 0xFFC Lower byte Masked Access Register (R/W) */
  __IO   uint32_t BYTE3_MASKED[256];          /* Offset: 0x1000 - 0x13FC Upper byte Masked Access Register (R/W) *
} CM3DS_MPS2_GPIO_TypeDef;
```

图 1. 39 GPIO 寄存器结构体定义

函数的使用参考方式如下所示：

```
GPIO_DeInit(CM3DS_MPS2_GPIO0);
```

1.8.1.2. GPIO_ReadInputData

该函数的功能就是读取某一个 GPIO 口的值，函数的变量说明如下表 1-5 所示。

表 1- 5 GPIO_ReadInputData 函数变量说明表

变量名称	变量意义
CM3DS_MPS2_GPIOx	确定使用哪一组 GPIO，AHB GPIO 只提供了一组 IO 接口，因此第一个参数只能设置为 CM3DS_MPS2_GPIO0
GPIO_Pin_x	对应使用哪个 GPIO 口，一共只有 32 个 GPIO 口供我们使用，所以其取值范围为：GPIO_Pin_0~ GPIO_Pin_31。

函数主要内容：清除输出使能位（OUTENABLECLR），也就是将 GPIO 口设置为输入，然后读取其数据寄存器中的值，读到的是 0，则返回 0，是 1 则返回 1。函数的部分代码如下所示：

```
CM3DS_MPS2_GPIOx->OUTENABLECLR = (1 << GPIO_Pin_x);
readValue = CM3DS_MPS2_GPIOx->DATA;
if (( readValue & ( Bit_SET << GPIO_Pin_x)) == 0){
    bitStatus = (uint8_t) Bit_RESET;
}
else{
    bitStatus = (uint8_t) Bit_SET;
}
return bitStatus;
```

函数的使用方式如下所示：

```
GPIO_ReadInputData(CM3DS_MPS2_GPIO0,GPIO_Pin_2);//读取 GPIO2 的值
```

1.8.1.3. GPIO_SetBit

函数的功能就是设置某一个GPIO口输出为高电平，函数使用的变量说明如下表 1-6 所示。

表 1- 6 GPIO_SetBit 函数变量说明表

变量名称	变量意义
CM3DS_MPS2_GPIOx	确定使用哪一组 GPIO，AHB GPIO 只提供了一组 IO 接口，因此第一个参数只能设置为 CM3DS_MPS2_GPIO0
GPIO_Pin_x	对应使用哪个 GPIO 口，一共只有 32 个 GPIO 口供我们使用，所以其取值范围为：GPIO_Pin_0~ GPIO_Pin_31。

函数的主要内容：将需要使用的某一个 GPIO 口的输出使能寄存器（OUTENABLESET）置 1，将其设置为输出，然后将复用清除寄存器（ALTFUNCCLR）置 1，关闭其复用功能，将中断清除使能寄存器（INTENCLR）置 1，关闭其中断功能，然后根据不同的 GPIO 口，向不同的字节屏蔽访问寄存器写入 1，对于字节屏蔽访问寄存器的划分如下表 1-7 所示。

表 1- 7 字节屏蔽访问寄存器划分表

GPIO 口	字节屏蔽访问寄存器
GPIO0~GPIO7	BYTE0_MASKED

GPIO8~GPIO15	BYTE1_MASKED
GPIO16~GPIO23	BYTE2_MASKED
GPIO24~GPIO32	BYTE3_MASKED

函数的部分代码如下所示：

```
CM3DS_MPS2_GPIOx->OUTENABLESET |= ( 1 << GPIO_Pin_x );
CM3DS_MPS2_GPIOx->ALTFUNCCLR  = ( 1 << GPIO_Pin_x );
CM3DS_MPS2_GPIOx->INTENCLR = ( 1 << GPIO_Pin_x );
if (CM3DS_MPS2_GPIOx  == CM3DS_MPS2_GPIO0)
{
    gpioDatamask = ( 1 << GPIO_Pin_x % 8);
    if (GPIO_Pin_x<= 7) {
        CM3DS_MPS2_GPIOx->BYTE0_MASKED[gpioDatamask] |= ( 1 << GPIO_Pin_x );
    } else if ( (8 <= GPIO_Pin_x) &&  (GPIO_Pin_x <= 15) ) {
        CM3DS_MPS2_GPIOx->BYTE1_MASKED[gpioDatamask] |= ( 1 << GPIO_Pin_x );
    } else if ( (16 <= GPIO_Pin_x) &&  (GPIO_Pin_x <= 23) ) {
            CM3DS_MPS2_GPIOx->BYTE2_MASKED[gpioDatamask] |= ( 1 << GPIO_Pin_x );
    } else if ( (24 <= GPIO_Pin_x) &&  (GPIO_Pin_x <= 31) ) {
        CM3DS_MPS2_GPIOx->BYTE3_MASKED[gpioDatamask] |= ( 1 << GPIO_Pin_x );
     }
}
```

函数的使用方式如下所示：

```
GPIO_SetBit(CM3DS_MPS2_GPIO0,GPIO_Pin_2);//设置 GPIO2 为高电平
```

1.8.1.4. GPIO_ResetBit

设置某一个 GPIO 口为低电平，函数主要变量说明如下表 1-8 所示。

表 1- 8 GPIO_ResetBit 函数变量说明表

变量名称	变量意义
CM3DS_MPS2_GPIOx	确定使用哪一组 GPIO，AHB GPIO 只提供了一组 IO 接口，因此第一个参数只能设置为 CM3DS_MPS2_GPIO0
GPIO_Pin_x	对应使用哪个 GPIO 口，一共只有 32 个 GPIO 口供我们使用，所以其取值范围为：GPIO_Pin_0~ GPIO_Pin_31。

其函数的内容和 GPIO_SetBit 函数唯一的不同就是向不同的字节屏蔽寄存器中写入 0，使其输出为低电平，函数的部分代码如下所示：

```
CM3DS_MPS2_GPIOx->OUTENABLESET |= ( 1 << GPIO_Pin_x );
CM3DS_MPS2_GPIOx->ALTFUNCCLR  = ( 1 << GPIO_Pin_x );
CM3DS_MPS2_GPIOx->INTENCLR = ( 1 << GPIO_Pin_x );
if (CM3DS_MPS2_GPIOx  == CM3DS_MPS2_GPIO0)
{
    gpioDatamask = ( 1 << GPIO_Pin_x % 8);
    if (GPIO_Pin_x <= 7) {
        CM3DS_MPS2_GPIOx->BYTE0_MASKED[gpioDatamask] &= (~( 1 << GPIO_Pin_x ));
    } else if ( (8 <= GPIO_Pin_x) &&  (GPIO_Pin_x <= 15) ) {
        CM3DS_MPS2_GPIOx->BYTE1_MASKED[gpioDatamask] &= (~( 1 << GPIO_Pin_x ));
    } else if ( (16 <= GPIO_Pin_x) &&  (GPIO_Pin_x <= 23) ) {
        CM3DS_MPS2_GPIOx->BYTE2_MASKED[gpioDatamask] &= (~( 1 << GPIO_Pin_x ));
```

```
    } else if ( (24 <= GPIO_Pin_x) &&  (GPIO_Pin_x <= 31) ) {
        CM3DS_MPS2_GPIOx->BYTE3_MASKED[gpioDatamask] &= (~( 1 << GPIO_Pin_x ));
    }
}
```

函数的使用方式如下所示：

```
GPIO_ResetBit(CM3DS_MPS2_GPIO0,GPIO_Pin_2);//设置 GPIO2 为低电平
```

1.8.1.5. GPIO_WriteBit

向某一个 GPIO 写入 0（输出低电平），写入 1（输出高电平），函数变量说明如下表 1-9 所示。

表 1- 9 GPIO_WriteBit 函数变量说明表

变量名称	变量意义
CM3DS_MPS2_GPIOx	确定使用哪一组 GPIO，AHB GPIO 只提供了一组 IO 接口，因此第一个参数只能设置为 CM3DS_MPS2_GPIO0
GPIO_Pin_x	对应使用哪个 GPIO 口，一共只有 32 个 GPIO 口供我们使用，所以其取值范围为：GPIO_Pin_0~ GPIO_Pin_31。
BitAction BitVal	该参数定义的一个结构体，其中就只有两个值：Bit_RESET 和 Bit_SET。

函数内容：将需要使用的 GPIO 口的输出使能寄存器（OUTENABLESET）置 1，将其设置为输出，然后将复用清除寄存器（ALTFUNCCLR）置 1，关闭其复用功能，将中断清除使能寄存器（INTENCLR）置 1，关闭其中断功能，然后判断我们需要写入的值是 0 还是 1，如果是 0 的话，向 DATA 寄存器中写入 0，是 1 的话，向 DATA 寄存器中写入 1。函数的部分代码如下所示：

```
CM3DS_MPS2_GPIOx->OUTENABLESET =  ( 1 << GPIO_Pin_x );
CM3DS_MPS2_GPIOx->ALTFUNCCLR  = ( 1 << GPIO_Pin_x );
CM3DS_MPS2_GPIOx->INTENCLR = ( 1 << GPIO_Pin_x );
if (BitVal != Bit_RESET){
    CM3DS_MPS2_GPIOx->DATA |= ( Bit_SET << GPIO_Pin_x );
}
else{
    CM3DS_MPS2_GPIOx->DATA &= (~( Bit_SET << GPIO_Pin_x ));
}
```

函数的使用方式如下所示：

```
GPIO_WriteBit(CM3DS_MPS2_GPIO0,GPIO_Pin_2,1);//向 GPIO2 写 1，输出高电平
```

1.8.1.6. GPIO_PinRemapConfig

用以设置 GPIO 的复用功能，可设置 UART、ADC、Timer、I2C、SPI，该函数变量说明如下表 1-10 所示。

表 1- 10 GPIO_PinRemapConfig 函数变量说明表

变量名称	变量意义
CM3DS_MPS2_GPIOx	确定使用哪一组 GPIO，AHB GPIO 只提供了一组 IO 接口，因此第一个参数只能设置为 CM3DS_MPS2_GPIO0

GPIO_Remap	如需查看可以在“CM3DS_gpio.h”文件中查看，如下图 1.40 所示
NewState	是否使能复用，该结构体定义了两个值：DISABLE：不使能；ENABLE：使能

```
/**
  * @defgroup GPIO_Remap_define
  * @{
  */
#define GPIO_Remap_TIM0EXTIN        ((uint32_t)0x00000001)    /*! < TIM0EXTIN  Alternate Function mapping */
#define GPIO_Remap_TIM1EXTIN        ((uint32_t)0x00000002)    /*! < TIM1EXTIN  Alternate Function mapping*/
#define GPIO_Remap_USART0_RXD       ((uint32_t)0x00000004)    /*! < USART0RXD  Alternate Function mapping*/
#define GPIO_Remap_USART0_TXD       ((uint32_t)0x00000008)    /*! < USART0TXD  Alternate Function mapping*/
#define GPIO_Remap_USART1_RXD       ((uint32_t)0x00000010)    /*! < USART1RXD  Alternate Function mapping*/
#define GPIO_Remap_USART1_TXD       ((uint32_t)0x00000020)    /*! < USART1TXD  Alternate Function mapping*/
#define GPIO_Remap_ADCETR           ((uint32_t)0x00000040)    /*! < ADCETR Alternate Function mapping*/
#define GPIO_Remap_SPI0_CLK_OUT     ((uint32_t)0x00000080)    /*! < SPI0_CLK_OUT  Alternate Function mapping*/
#define GPIO_Remap_SPI0_SEL         ((uint32_t)0x00000100)    /*! < SPI0_SEL  Alternate Function mapping*/
#define GPIO_Remap_SPI0_DATA0       ((uint32_t)0x00000200)    /*! < SPI0_DATA0 Alternate Function mapping*/
#define GPIO_Remap_SPI0_DATA1       ((uint32_t)0x00000400)    /*! < SPI0_DATA1 Alternate Function mapping*/
#define GPIO_Remap_SPI0_DATA2       ((uint32_t)0x00000800)    /*! <SPI0_DATA2  Alternate Function mapping */
#define GPIO_Remap_SPI0_DATA3       ((uint32_t)0x00001000)    /*! <SPI0_DATA3  Alternate Function mapping */

#define GPIO_Remap_SPI1_CLK_OUT     ((uint32_t)0x00002000)
#define GPIO_Remap_SPI1_SEL         ((uint32_t)0x00004000)
#define GPIO_Remap_SPI1_DATA0       ((uint32_t)0x00008000)
#define GPIO_Remap_SPI1_DATA1       ((uint32_t)0x00010000)
#define GPIO_Remap_SPI1_DATA2       ((uint32_t)0x00020000)
#define GPIO_Remap_SPI1_DATA3       ((uint32_t)0x00040000)

#define GPIO_Remap_I2C0SCL          ((uint32_t)0x00080000)
#define GPIO_Remap_I2C0SDA          ((uint32_t)0x00100000)
```

图 1. 40 复用功能引脚定义

函数内容：是否启用复用功能，主要需要操作的寄存器就是：复用使能寄存器（ALTFUNCSET）和复用清除寄存器（ALTFUNCCLR），分别向这两个寄存器中写入 1，代表使能和屏蔽复用功能，函数的部分代码如下所示：

```
assert_param(IS_GPIO_ALL_PERIPH(CM3DS_MPS2_GPIOx));
assert_param(IS_GPIO_REMAP(GPIO_Remap));
assert_param(IS_FUNCTIONAL_STATE(NewState));
if ( NewState != DISABLE)
{
        CM3DS_MPS2_GPIOx->ALTFUNCSET   = GPIO_Remap;
}
else
{
    CM3DS_MPS2_GPIOx->ALTFUNCCLR   = GPIO_Remap;
}
```

1.8.1.7. GPIO_PinInterruptTypeConfig

该函数的功能主要就是设置中断的触发方式，可设置为低电平、高电平、下降沿、上升沿四种。函数变量说明如下表 1-11 所示。

表 1- 11 GPIO_PinRemapConfig 函数变量说明表

变量名称	变量意义
CM3DS_MPS2_GPIOx	确定使用哪一组 GPIO，AHB GPIO 只提供了一组 IO 接口，因此第一个参数只能设置为 CM3DS_MPS2_GPIO0
GPIO_Pin_x	对应使用哪个 GPIO 口，一共只有 32 个 GPIO 口供我们使用，所以其取值范围为：GPIO_Pin_0~ GPIO_Pin_31。
InterruptState NewState	是否启用中断：Interrupt_DISABLE：不启用；Interrupt_ENABLE：启用。
BitInterruptType	中断触发方式：可设置为低电平、高电平、下降沿、上升沿四种，其用结构

INTTYPE	体定义在“CM3DS_gpio.h”文件中，如下所示，需要使用什么方式，选择即可。
	`typedef enum`
	`{`
	`Bit_LOWLEVEL = 0,`
	`Bit_HIGHLEVEL,`
	`Bit_FALLINGEDGE,`
	`Bit_RISINGEDGE`
	`}BitInterruptType;`

函数内容：首先根据我们输入的参数 InterruptState NewState 的值判断是否启用中断，如果需要启用中断的话，需要向中断使能寄存器（INTENSET）中写入 1，来启用中断，然后根据中断的触发方式，依次去设置中断类型清除寄存器（INTTYPECLR）、中断类型启用寄存器（INTTYPESET）、中断极性设置寄存器（INTPOLSET）、中断极性清除寄存器（INTPOLCLR），具体的配置如下表 1-12 所示。

表 1- 12 中断触发方式寄存器配置表

中断触发类型	寄存器配置			
	INTTYPECLR	INTTYPESET	INTPOLCLR	INTPOLSET
Bit_LOWLEVEL	1	0	1	0
Bit_HIGHLEVEL	1	0	0	1
Bit_FALLINGEDGE	0	1	1	0
Bit_RISINGEDGE	0	1	0	1

函数的使用方式如下所示，启用 GPIO27 的中断，中断触发类型为低电平中断：

```
GPIO_PinInterruptTypeConfig(CM3DS_MPS2_GPIO0,GPIO_Pin_27,Bit_LOWLEVEL,Interrupt_ENABLE);
```

1.8.2. 按键中断函数

将我们提供的按键中断库函数“key_exit”添加至 HARDWARE 文件夹下，首先需要将提供的库函数复制至文件夹之下，然后点击 之后进行添加，添加方式如下图 1.41 所示。

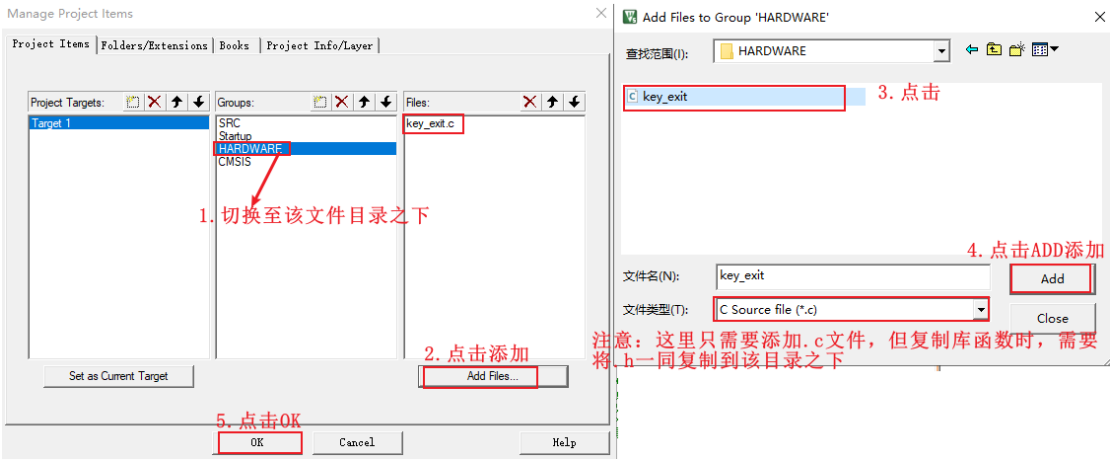

图 1. 41 添加按键中断库函数

1.8.2.1. KEY_INT_Init

按键中断初始化函数，函数的参数一共有两个：KEYx：对应哪个按键；PORT0_x_IRQn：按键对应的特定的中断号，对于中断号的定义都可以在“CM3DS_MPS2.h”文件找到。这两个参数的取值都在“key_exit.h”文件中有定义，定义如下：

```
//KEY3 在这里将其作为复位按键，KEY0 和 KEY1 供我们使用
#define KEY0 GPIO_Pin_27
#define KEY1 GPIO_Pin_26
//KEY0 和 KEY1 对应的中断号
#define PORT0_KEY0_IRQn PORT0_27_IRQn
#define PORT0_KEY1_IRQn PORT0_26_IRQn
```

函数内容：首先需要通过 GPIO_PinInterruptTypeConfig 函数启用对应按键的中断以及中断触发方式，这里将其设置为低电平触发，然后清除挂起的中断，最后启用外部的中断，函数内容如下所示：

```
void KEY_INT_Init(unsigned char KEYx,unsigned short PORT0_x_IRQn)
{
    //这里使能中断，设置低电平触发
    GPIO_PinInterruptTypeConfig(CM3DS_MPS2_GPIO0,KEYx,Bit_LOWLEVEL,In
terrupt_ENABLE);
    //清除挂起的中断
    NVIC_ClearPendingIRQ(PORT0_x_IRQn);
    //启用外部中断
    NVIC_EnableIRQ(PORT0_x_IRQn);
}
```

函数的使用方式如下所示：

```
KEY_INT_Init(KEY0,PORT0_KEY0_IRQn);//启用按键 KEY0 的中断
```

1.8.2.2. PORT0_x_Handler

中断服务函数，其中 x 代表对应的哪个 GPIO 口，比如本次实验使用到的按

键 0 和按键 1，其对应的 GPIO 口为 GPIO_Pin_27 和 GPIO_Pin_26，那么就需要将中断服务函数定义为 PORT0_27_Handler 和 PORT0_26_Handler。

以按键 1 的中断服务函数为例，首先添加按键中断处理事件，然后通过 EXTI_ClearFlag 函数清除中断标志，清除中断标志的方式就是向中断清除寄存器（INTCLEAR）中写入 1，代码内容如下所示：

```
void PORT0_26_Handler()
{
    /*************按键 KEY1 中断处理事件***********/

    /****************************************/
    EXTI_ClearFlag(CM3DS_MPS2_GPIO0,KEY1);//清除中断标志
}
```

1.8.3. 添加用户代码

本次实验需要实现的功能是：程序下载之后，LED1 和 LED3 亮，LED0 和 LED4 灭，按下按键 0 改变 LED2 和 LED3 的状态，松开恢复成之前的状态，按下按键 1 改变 LED0 和 LED1 的状态，松开恢复成之前的状态。

首先查看 LED 灯的电路原理图如下图 1.42 所示。从图中可以看出，如果需要点亮 LED 灯，就需使对应的 GPIO 口输出低电平，可以通过 GPIO_ResetBit 函数或者 GPIO_WriteBit 函数直接写 0 实现，如果需要熄灭 LED 灯的话，就需要使对应的 GPIO 口输出高电平，可以通过 GPIO_SetBit 函数或者 GPIO_WriteBit 函数写入 1 实现。

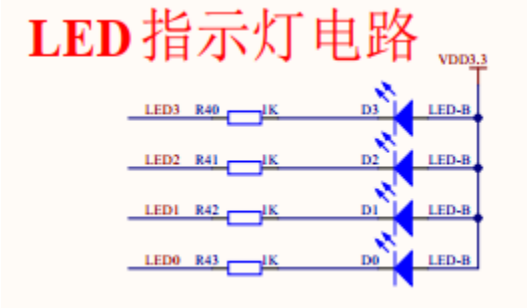

图 1. 42 LED 灯原理图

根据实验所需要实现的功能，编写主函数代码如下所示：

```
#include "CM3DS_gpio.h"
#include "CM3DS_rcc.h"
#include "CM3DS_MPS2.h"
#include "CM3DS_exti.h"
#include "key_exit.h"
int main()
{
    KEY_INT_Init(KEY0,PORT0_KEY0_IRQn);//启用按键 KEY0 的中断
    KEY_INT_Init(KEY1,PORT0_KEY1_IRQn);//启用按键 KEY1 的中断
```

```
    while(1){
        GPIO_SetBit(CM3DS_MPS2_GPIO0,LED0);//熄灭 LED0
        GPIO_ResetBit(CM3DS_MPS2_GPIO0,LED1);//点亮 LED1
        GPIO_SetBit(CM3DS_MPS2_GPIO0,LED2);//熄灭 LED2
        GPIO_ResetBit(CM3DS_MPS2_GPIO0,LED3);//点亮 LED3
    }
}
```

按键 0 的中断服务函数如下所示：

```
void PORT0_27_Handler()
{
    /*************按键 KEY0 中断处理事件***********/

    GPIO_WriteBit(CM3DS_MPS2_GPIO0,LED2,0);//点亮 LED2
    GPIO_WriteBit(CM3DS_MPS2_GPIO0,LED3,1);//熄灭 LED3

    /*****************************************/
    EXTI_ClearFlag(CM3DS_MPS2_GPIO0,KEY0);//清除中断标志
}
```

按键 1 的中断服务函数如下所示：

```
void PORT0_26_Handler()
{
    /*************按键 KEY1 中断处理事件***********/

    GPIO_WriteBit(CM3DS_MPS2_GPIO0,LED0,0);//点亮 LED0
    GPIO_WriteBit(CM3DS_MPS2_GPIO0,LED1,1);//熄灭 LED1

    /*****************************************/
    EXTI_ClearFlag(CM3DS_MPS2_GPIO0,KEY1);//清除中断标志
}
```

1.9. 板级验证

1.9.1. 实验所需硬件

(1) AC208 开发板

(2) FPGA 下载器：XIST USB Cable

(3) CM3 仿真器：DAP Link

(4) 电源线一根

1.9.2. 硬件连接

在 MDK 中，使用 DAPLink 可以实现对 Cortex-M3 CPU 的程序下载和调试（Debug），任何一个符合标准的 DAP Link 调试器都能对 SA5Z 系列 SoC FPGA 中集成的 Cortex-M3 CPU 进行调试和下载，使用 DAPLink 对 SA5Z 系列 SoC FPGA 中集成的 Cortex-M3 CPU 进行调试和下载时，只需要连接 SW 模式下的

SWCLK、SWDIO 两根信号线和 GND 信号线即可。具体的连接方式如下图 1.43 所示。

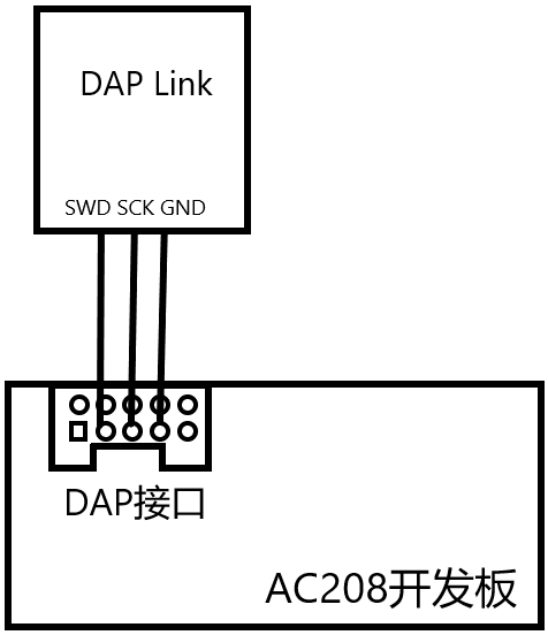

图 1. 43 DAP Link 连接示意图

系统的硬件连接图如下图 1.44 所示。其中黑色的线接到 DAP Link 的 GND，红色的线接到 DAP Link 的 DIO（有的标注为 SWD），褐色的线接到 CLK（有的标注为 SCK）引脚上。

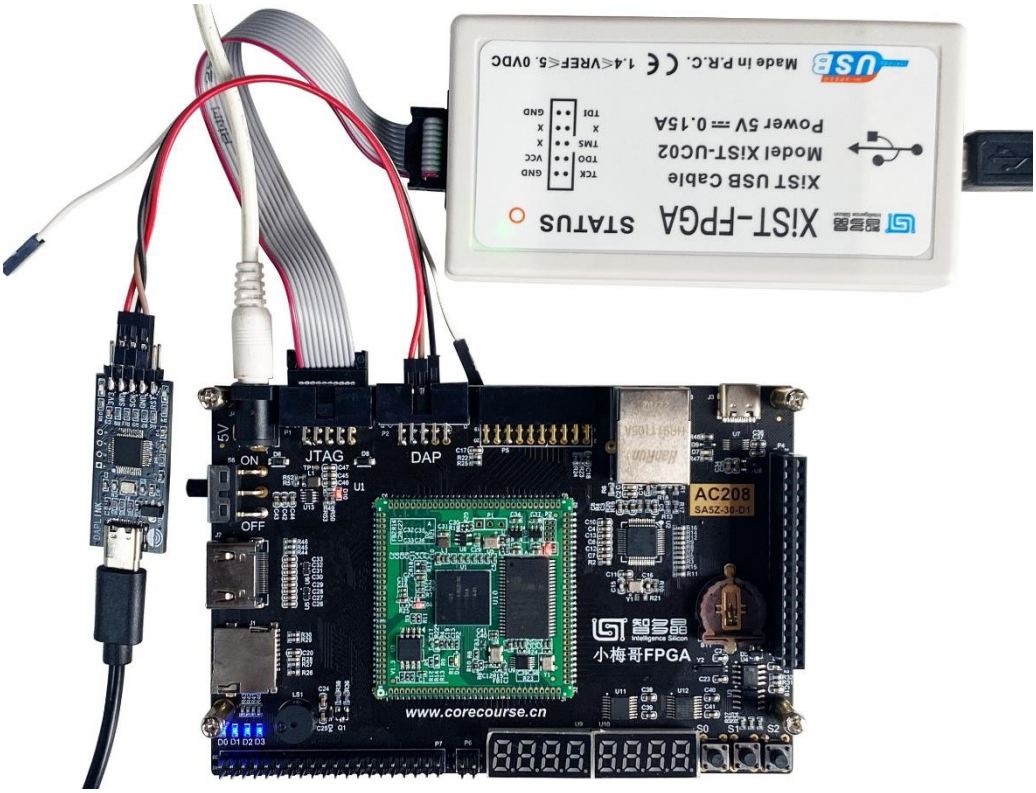

图 1. 44 AC208-SA5Z 硬件连接图

对于 AC208 开发板由于没有 DAP 的专用插口，需要选择两个 IO 口用杜邦线进行 DAPLink 上对应端口的连接。这里选择的是 GPIO0_0 和 GPIO0_1（用户可以参照原理图和引脚表任选），对应在硬件上就是分别将 SWD（红线） 和 SCK（褐线）连接在 P3 的 1 号引脚和 2 号引脚，并将 GND（黑线）连接在 12 号引脚上（也是 GND）。

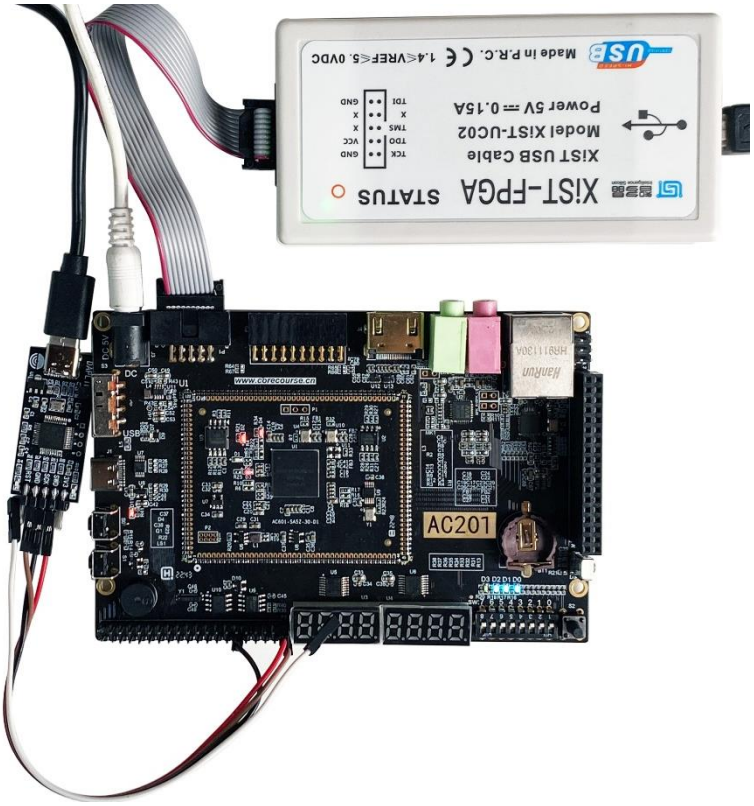

图 1. 45 AC208 硬件连接图

如果使用的是我们的 DAP Link，可以用转接线连接 DAP Link 和开发板，此时硬件连接图如下图 1. 46 所示。

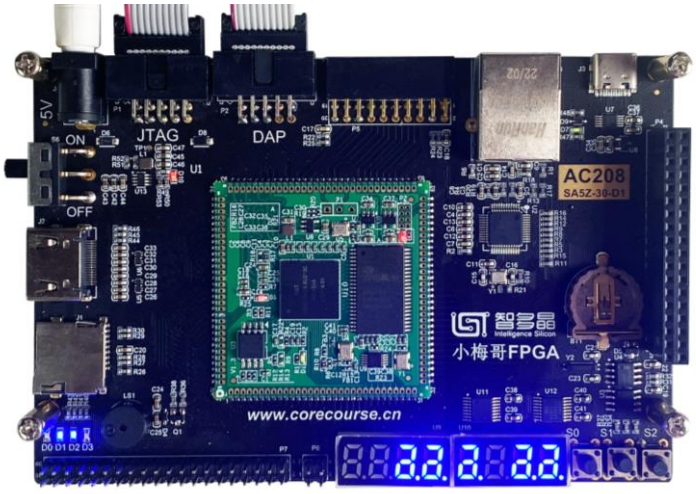

图 1. 46 AC208-SA5Z 硬件连接图 2

1.1.1 下载文件至目标板

下载文件的方式总共分为两种，一种是分别下载硬件设计代码和软件设计代码，当我们需要调试软件程序代码的时候，这种方式比较适合；另外一种是合并成一个 bin 文件进行下载，这种方式适合在代码已经测试没有问题，需要量产的时候使用。

1.9.2.1. 分别下载

1. 下载硬件设计文件，硬件测试文件在 HQ 工程目录下的 hq_run 文件夹下，具体操作方式如下图 1. 47 所示。

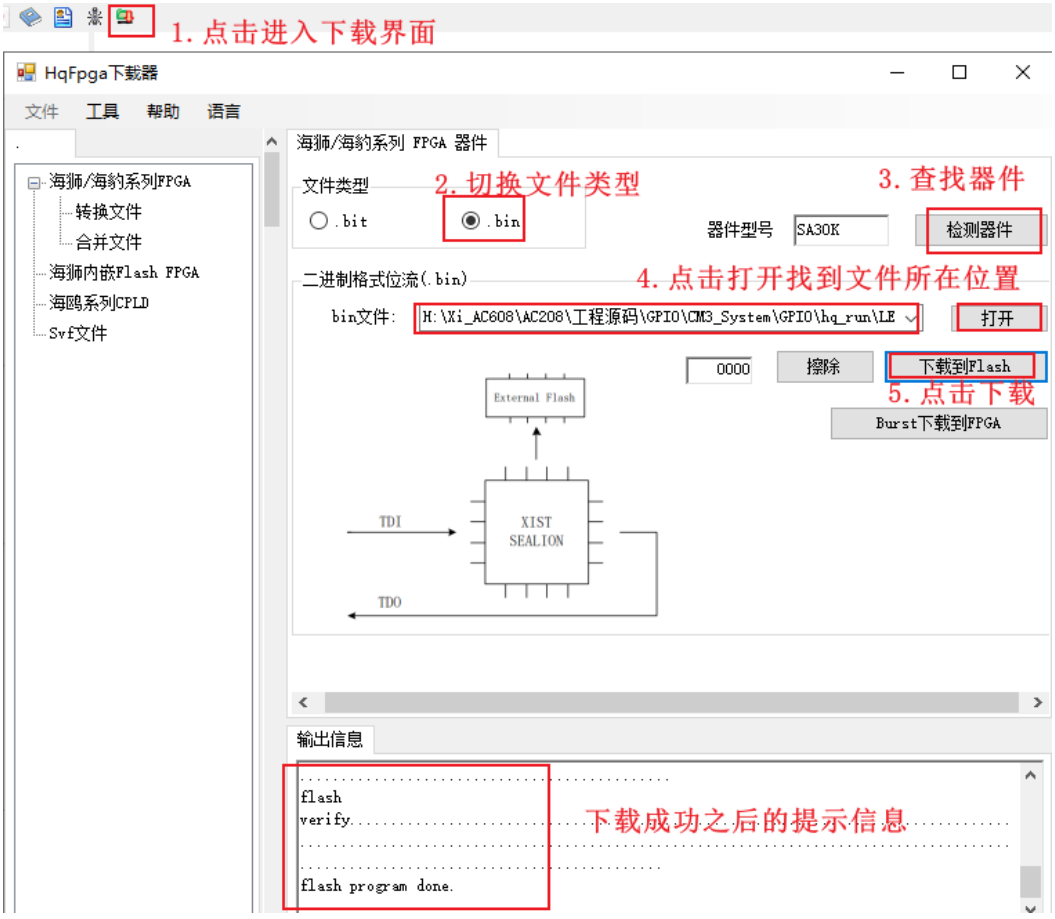

图 1. 47 下载硬件设计文件

2. 下载软件设计文件，操作方式如下图 1. 48 所示。

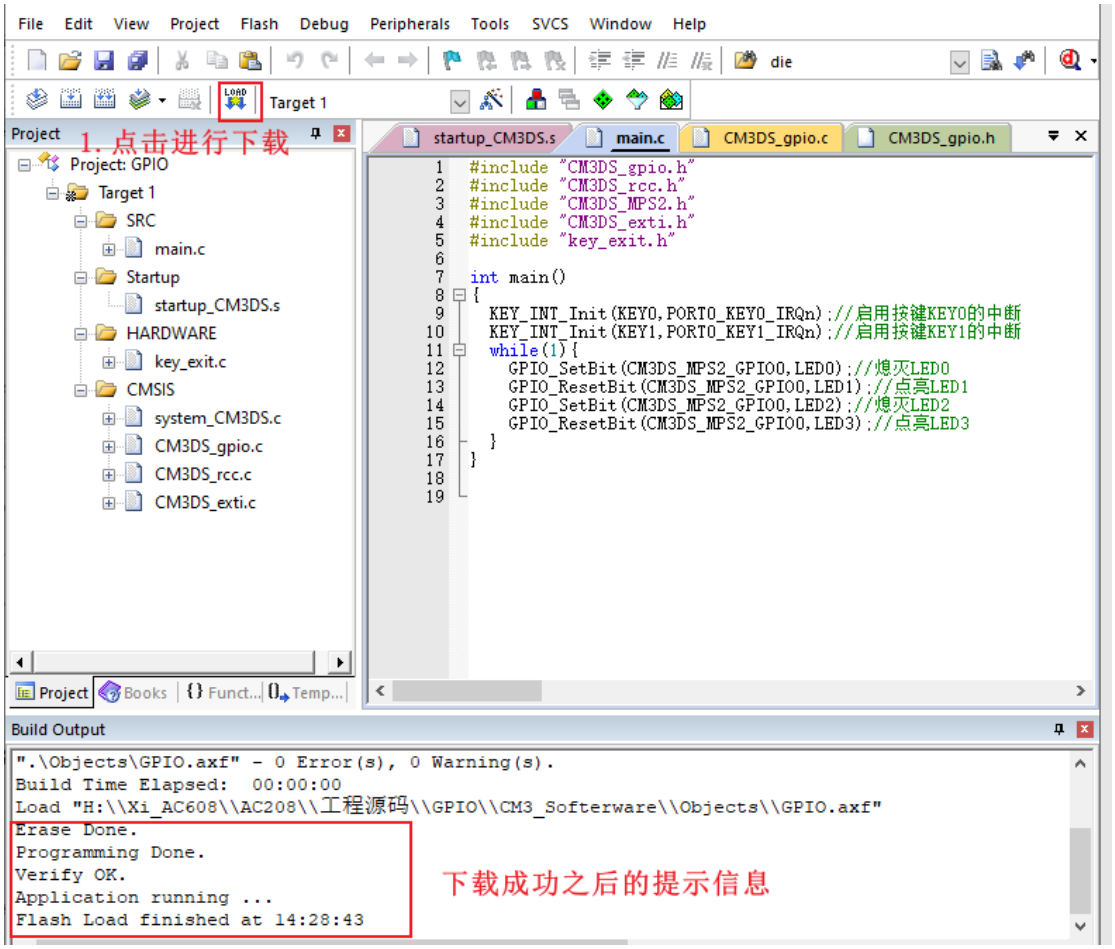

图 1. 48 下载软件设计文件

1.9.2.2. 合并下载

将 FPGA 的 bin 文件和 CM3 的 bin 文件合成一个 bin 文件下载至开发板，合并文件操作步骤如下图 1. 49 所示。

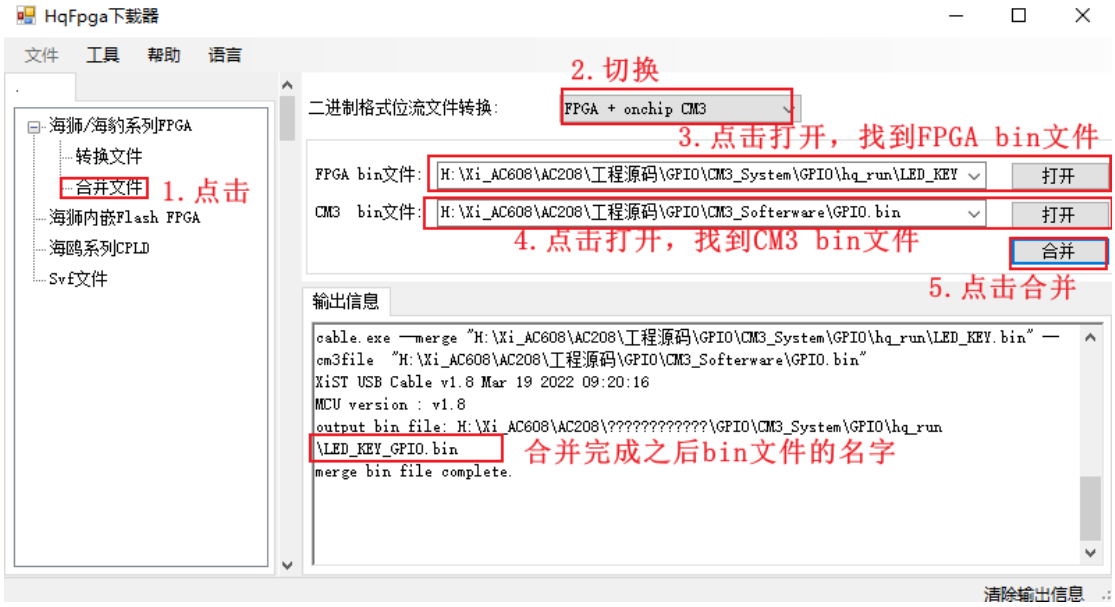

图 1. 49 合并文件操作示意图

合并成功之后进行下载，合并之后的文件同样也存放在 hq_run 文件下，下载步骤如下图 1. 50 所示。

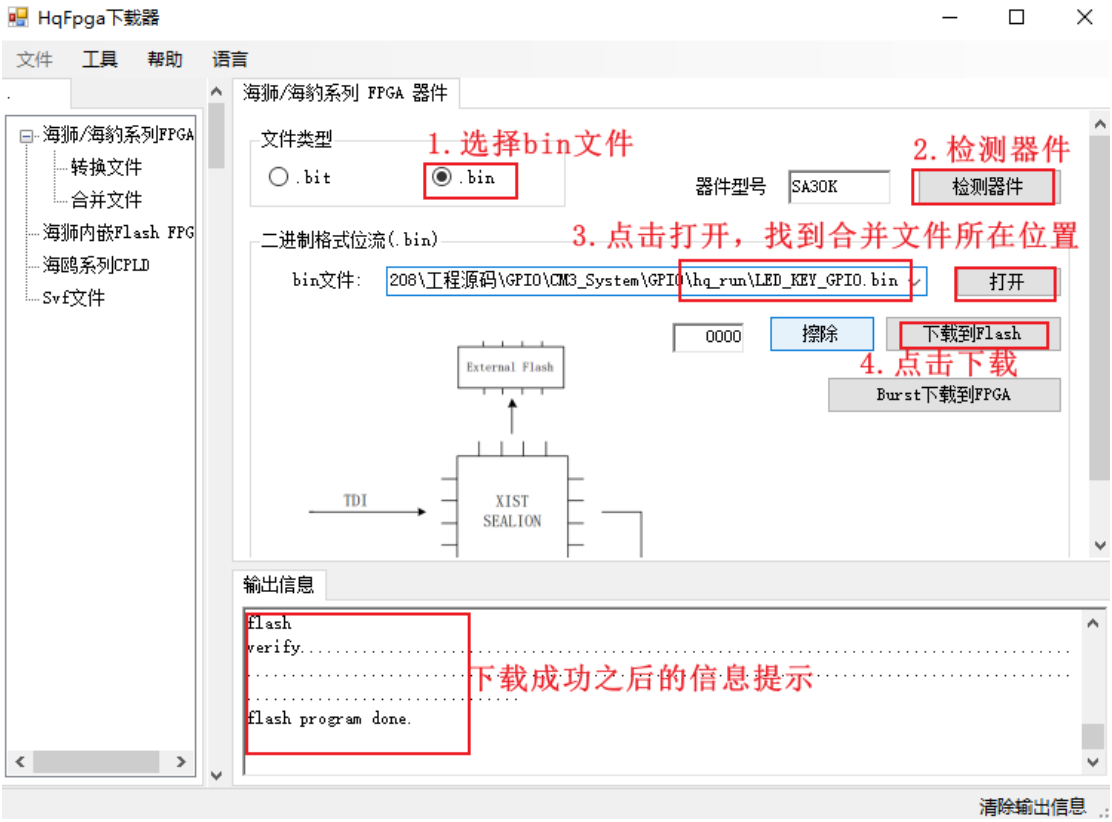

图 1. 50 合并文件下载步骤示意图

1.9.3. AC208-SA5Z 功能演示

程序下载之后，开发板上的 LED0 和 LED2 是熄灭的，LED1 和 LED3 被点亮如下图 1. 51 所示。

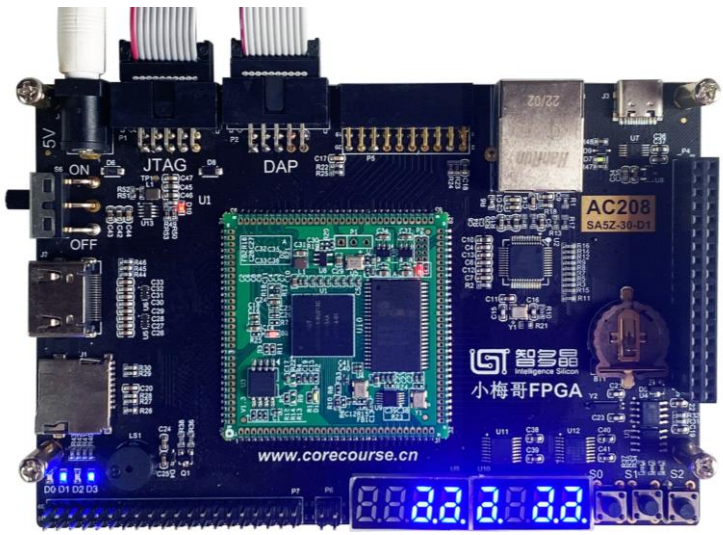

图 1. 51 程序下载之后的结果显示

按下按键 0，LED2 和 LED3 的状态被改变，如下图 1. 52 所示。

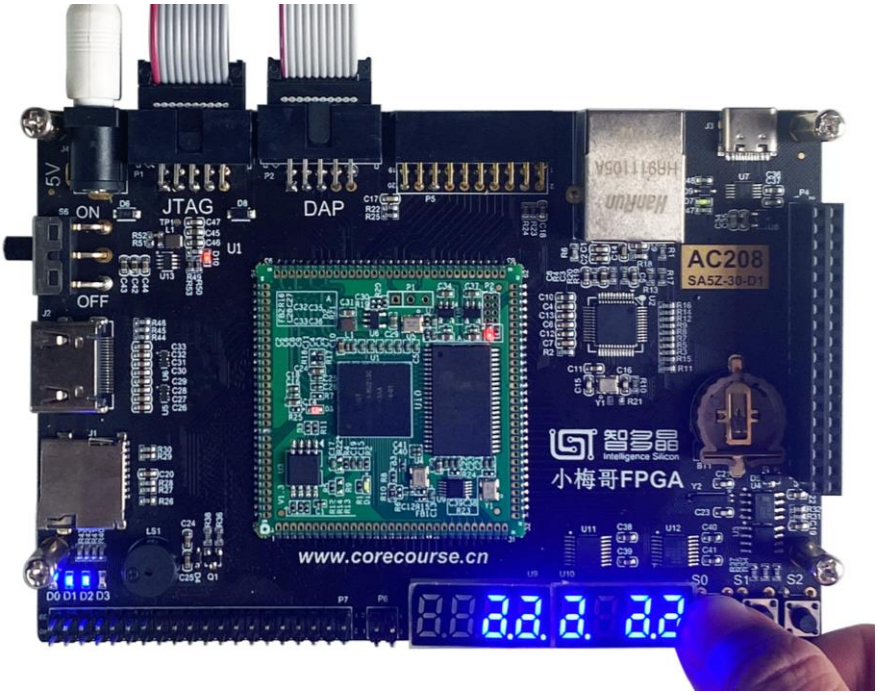

图 1. 52 按下按键 0 之后的结果显示

按下按键 1，LED0 和 LED1 的状态被改变，如下图 1. 53 所示。

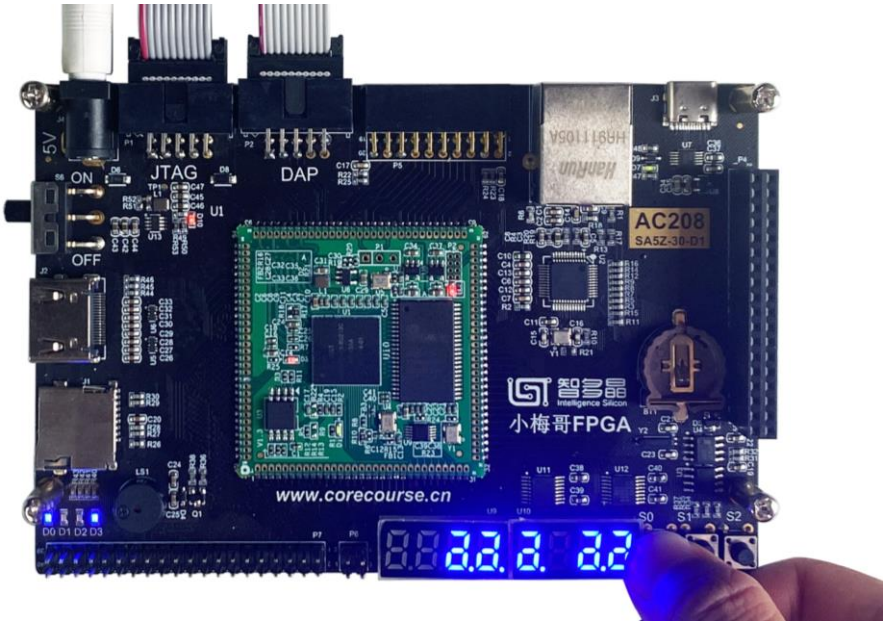

图 1. 53 按下按键 1 之后的结果显示

1.1.2 AC208 功能演示

程序下载之后，开发板上的 LED0 和 LED2 是熄灭的， LED1 和 LED3 被点亮如下图所示。

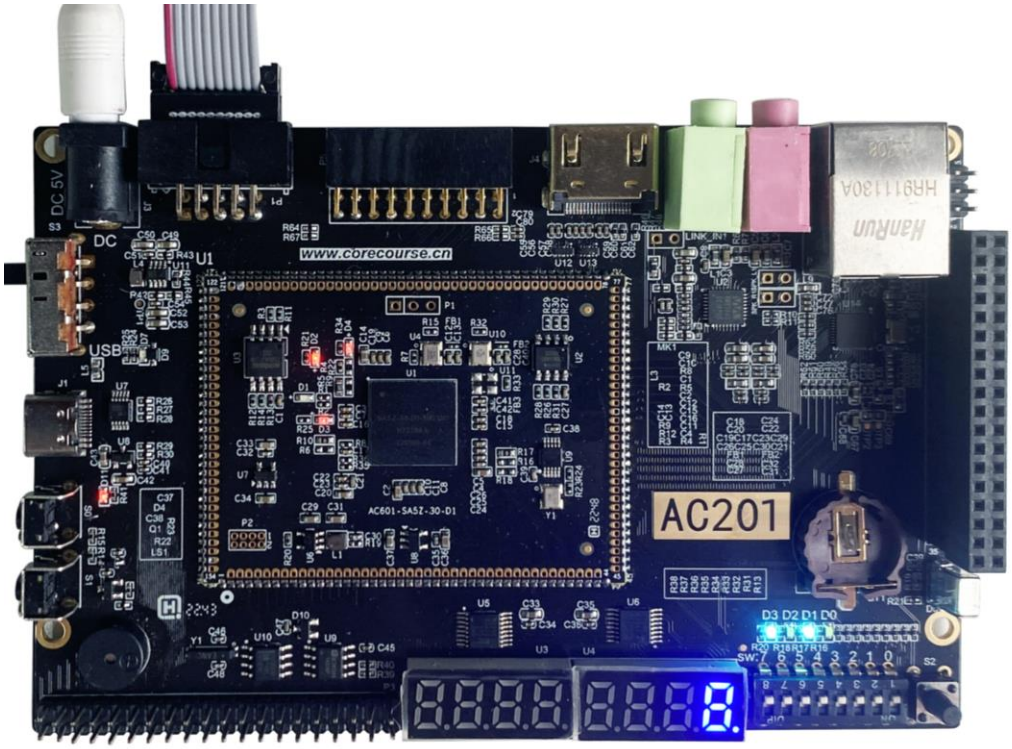

图 1. 54 上电下载 bin 文件后不做任何操作时的结果显示

按下按键 0， LED2 和 LED3 的状态被改变，如下图所示。

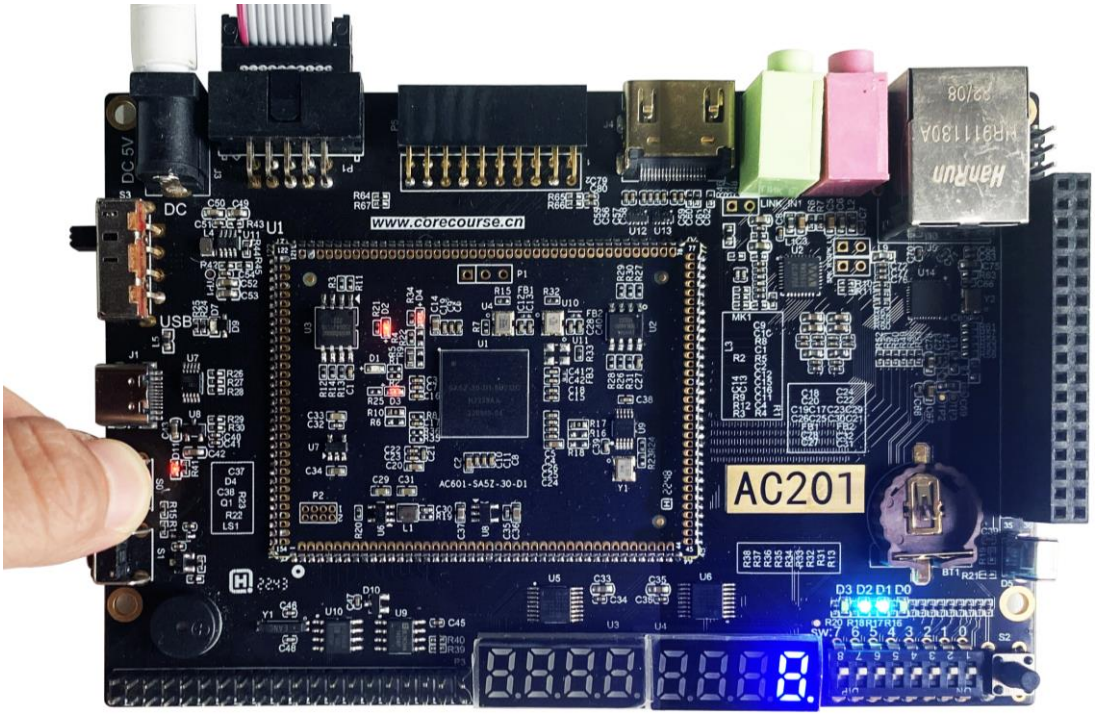

图 1. 55 按下按键 0 之后的结果显示

按下按键 1，LED0 和 LED1 的状态被改变，如下所示。

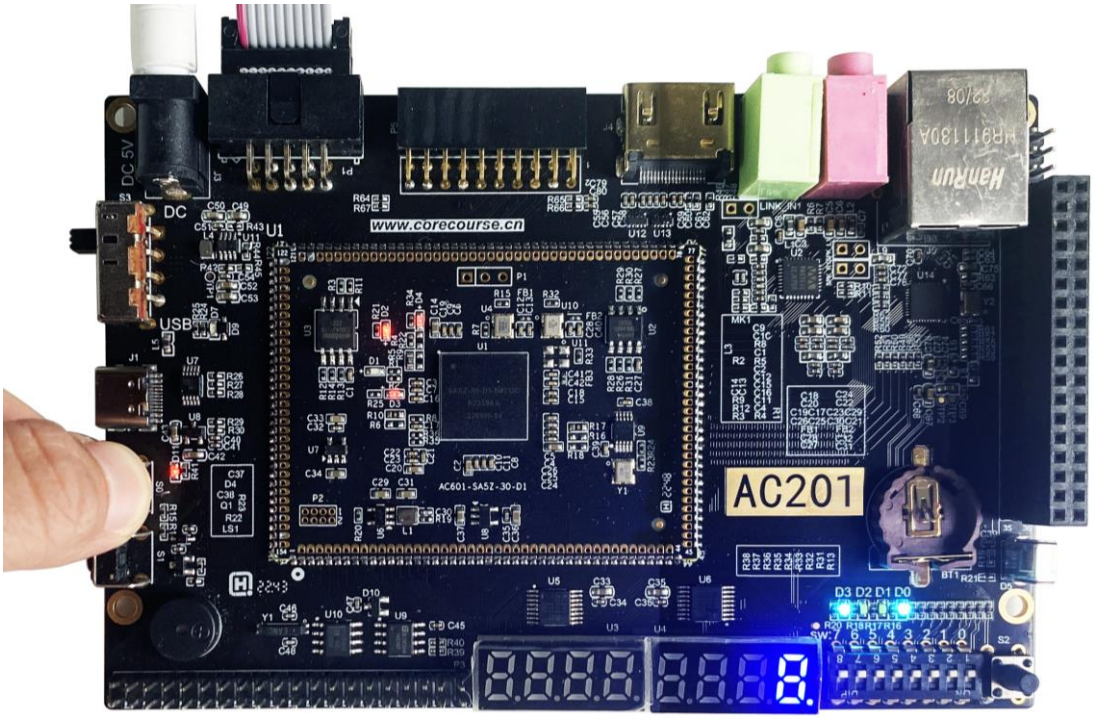

图 1. 56 按下按键 1 之后的结果显示

1.1.3 板级调试

通常我们在编写完成代码之后，下载到我们的开发板上时，有可能达不到我们想要的效果，这个时候就需要进行代码的调试，虽然我们第一个实验并没有用到调试功能，但是如果要实现比较复杂的功能的时候，就需要我们进行调试，接下来简单的描述一下怎么去调试代码。

在 MDK 软件中点击 图标，进入调试界面，如下图所示。如果用户对程序进行了修改，需要退出调试界面，重新编译一下程序，然后再进入调试界面。

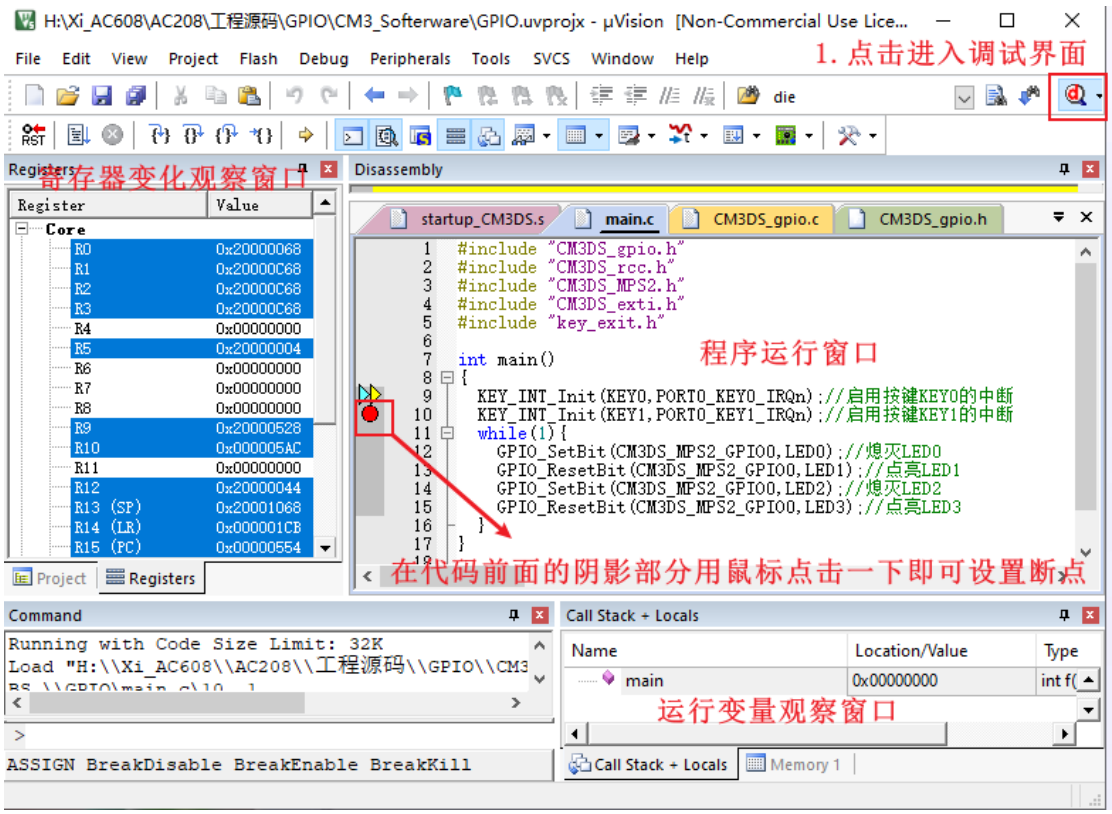

图 1. 57 调试界面显示

用户可以选取下图所示的通用调试控制工具来跟踪程序的运行。在程序运行观察窗口中可以观察程序的运行情况；在运行变量观察窗口中可以观察变量的值；在寄存器变换观察窗口能够查看寄存器、存储器中的值。如果想要退出调试界面，再点击一下  图标就可以了。

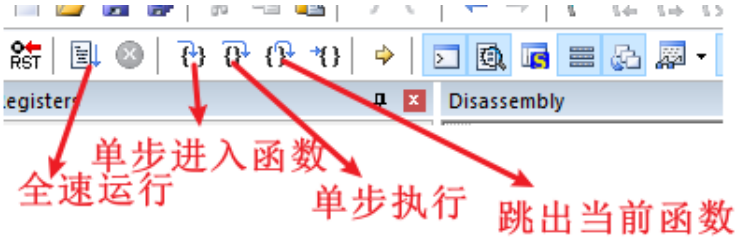

图 1. 58 调试窗口介绍

1.10.思考与总结

本章实验通过对 GPIO 的使用，完成了按键对 LED 灯状态的控制，作为第一个实验，对于整个设计流程作了比较详细的介绍，在实验的过程中，有几点需要大家注意的：

1. 在应用到调试功能的时候，在 cm3_system 文件中一定要添加一行代码，否则程序将无法进行调试，需要添加的代码如下所示。

```
assign  swddio = int_jtag_tmsoen ? 1'bz : int_jtag_tmso;
```

2. 使用 DAPLink 对 SA5Z 系列 SoC FPGA 中集成的 Cortex-M3 CPU 进行调试和下载时，只需要连接 SW 模式下的 SWCLK 和 SWDIO 两根信号线和 GND 信号线即可。

3. 当下载程序的时候出现 DAP Link 连接不上的问题时，如下图所示。

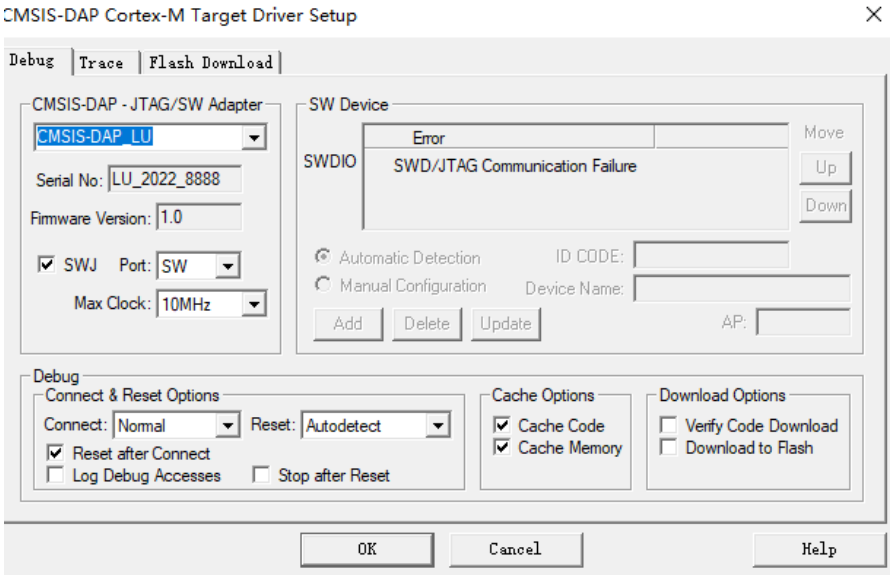

图 1. 59 DAP Link 连接不上

出现上图所示的问题时，从下面几个方面去一一排查问题：

(1) 下载程序的时候，开发板需要上电；

(2) 需要先下载 FPGA 侧的程序，然后再通过 MDK 软件下载自己编写的应用程序；

(3) 检查 DAP Link 的连接方式是否出错，参考 1.9.1 节中的内容；

(4) 检查是否添加了引脚约束文件，引脚是否分配错误。

4. 当编译的时候出现“Overlapping of Algorithms at Address 00000000H”的问题的时候，检查一下自己的 Flash Download 的设置，也就是 1.7.3 节第 3 点设置 Debug 选项的时候，在“Add Flash Programming Algorithm”的时候，是否只留下了“Xist Micro 128k Flash V1.0”,如下图所示。