乐升半导体
LEVETOP

LT32U02

High Performance 32bit Micro Controller

应 用 手 册

1.2

www.levetop.cn

Levetop Semiconductor Co., Ltd.

目　录

1. LT32U02 简介 4

1.1 引脚图 4

1.2 特点 5

1.3 内部方块图 5

2. 程序库说明 6

2.1 有关时钟配置(CLOCK) 6

2.1.1 时钟设置 6

2.1.2 时钟选择控制 6

2.2 中断控制(EIC) 9

2.3 EPT 定时器设置(EPT) 10

2.4 PIT 定时器设置(PITx) 11

2.5 GPIO 口配置和外部中断设置(EPORTx) 12

2.6 串口设置(UART0) 14

2.7 脉冲宽度调制(PWMx) 16

2.8 ADC 设置(ADC) 19

2.9 DMA 设置(LDMA) 21

2.10 COMP 比较器设置(COMPx) 24

2.11 I2C 硬件设置(I2C) 26

2.12 SPI 设置(SPI) 28

2.13 看门狗设置(WDTx) 32

2.14 Flash 操作 34

2.15 其它选项设置(CACHE 、EFM、CCM) 35

2.16 函数库列表 37

3. 开发板介绍 45

3.1 LT32U02 开发板 45

3.2 I2C 45

3.3 SPI 46

3.4 UART 46

3.5 ADC 46

3.6 PWM 46

©Levetop Semiconductor Co., Ltd.

3.7 EPORT 47
3.8 其它 47
3.9 开发板原理图 48

4. 烧录说明 49

4.1 多线烧录 49
4.2 SWD 烧录 52
4.3 脱机烧录 53
4.4 拓展功能 54

5. IDE 工具使用说明 57

5.1 软件安装 57
5.2 Stlink 驱动安装 61
5.3 创建工程及配置 62
5.4 导入 72
5.5 工程配置 74
5.6 调试注意事项 75
5.7 Telnet 工具使用 76
5.8 常见的调试命令和调试菜单 77
5.9 常见问题及原因 80

6. 版本记录 81

7. 版权说明 81

©Levetop Semiconductor Co., Ltd.

1. LT32U02 简介

LT32U02 是乐升半导体新推出的两款高效能、低单价的 32 位 MCU。LT32U02 内部包含了一个高性能的 C0 32 位精简指令（RISC）核心，拥有最高 72MHz 的工作频率，还有高速嵌入式存储内存，包括 64KB 闪存、8KB SRAM、1KB Cache Memory、Full Speed USB 2.0 接口。除此之外 LT32U02 也提供了各式标准的通信接口，包括两个 SPI、一个 I2C、一个 SCI、8 路高分辨率 PWM，还有一组 8 路 12 位的 ADC 与 2 路的模拟比较器，及多组通道的 GPIO 接口。

LT32U02 拥有优异的乘法运算能力，搭配 USB 2.0 传输接口，可适用于各类中高端家用电器、游戏鼠标、智能电子玩具、智能门锁、电子仪器、电子设备、飞行控制器，及需要 USB 传输的各式电子产品上。

1.1 引脚图

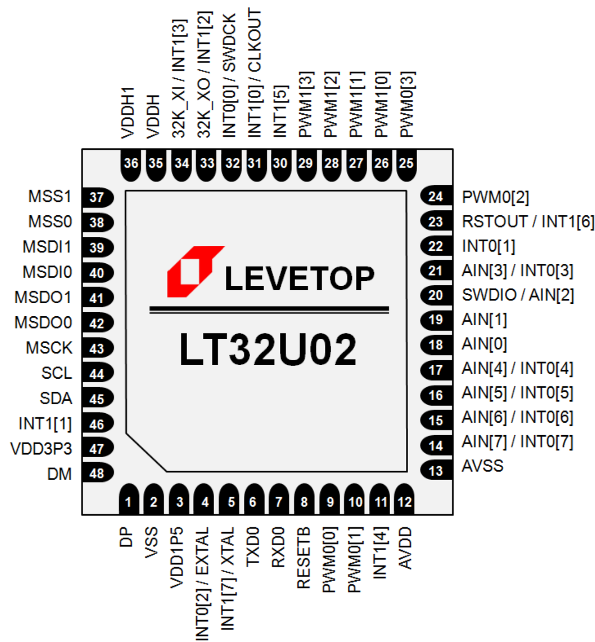


图 1–1: LT32U02 引脚图

表 1-1

型 号	封 装
LT32U02	48Pin QFN (QFNWB6x6-48L)

©Levetop Semiconductor Co., Ltd.

1.2 特点

1) 32 位 C0 精简指令集处理器
2) 支持 32 位×32 位单周期硬件整数乘法器，及 32 位 3~13 周期硬件除法器
3) 内建 64KBytes Flash、8K Bytes SRAM 和 1K Bytes Cache Memory
4) 内建 USB2.0 （Full Speed / 8 EP）
5) 内建 4 个 16 位周期性间隔定时器（PIT）
6) 支持 2 路串行 SPI、1 路 SCI/UART 接口
7) 支持 2 组 8 通道脉宽调制器（PWM）
8) 内建 12 位 ADC 转换器支持 8 个模拟输入
9) 内建 2 路模拟比较器（Comparator）
10) 支持 I2C 主/从 串行通信接口
11) 内建 16 个 GPIO 接口
12) 内建 2 组 Watch-Dog 看门狗定时器
13) 支持 3 通道 DMA 控制器
14) 支持串行线调试（SWD）
15) 内建电源管理单元（PMU）
16) 内建 3.3V LDO 供内、外部使用
17) 内建 72MHz PLL 内部振荡器
18) 可接外部 32,768Hz 晶振
19) 工作电压：2.0/2.8V~5.5V

1.3 内部方块图

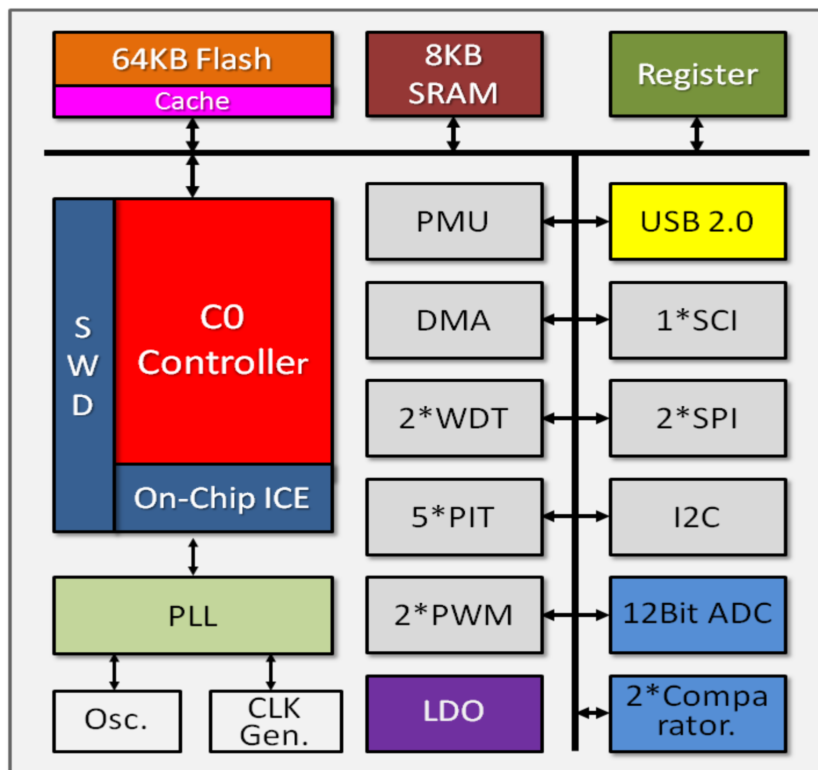


图 1-2：内部方块图

©Levetop Semiconductor Co., Ltd.

2. 程序库说明

2.1 有关时钟配置(CLOCK)

LT32U02 内部最高时钟可达 72MHZ，包含三种时钟 system_clock，IPG_clock，ADC_clock。

- System_clock：系统时钟，可来源于内部提供的经过 2 分频的 144MHZ 或外部提供的高速晶振；
- IPG_clock：设备时钟，通过在 system_clock 的基础上分频后得到的时钟，作为大部分外设的时钟源；
- ADC_clock：ADC 时钟，通过在 system_clock 的基础上分频后得到的时钟，ADC 特有的时钟源。

而作为提供时钟频率的晶振也分为 4 种，包含了 FIRC、SIRC、FXOSC 和 SXOSC。

- FIRC ：内部高速的 144MHZ 晶振
- SIRC ：内部低速的 128KHZ 晶振
- FXOSC：外部高速晶振，范围为 0~20MHZ
- SXOSC：外部低速晶振 32.768KHZ

功能封装函数档案名：clk_lib.c
功能封装函数声明档案名：clk_lib.h

2.1.1 时钟设置

```
void CLOCK_SystemCLkDivider(UINT8 div);      //设置系统时钟分频值，频率最高可达 72MHZ

void CLOCK_IPGCLkDivider(UINT8 div);         //设置外设时钟分频值，频率最高可达到
                                             // system_clock 的速度
```

ADC_clock 时钟配置在后面的 ADC 模块会提到。

举例：如果要将 system_clock 设置为 36MHZ；将 IPG _clock 设置为 18MHZ

```
CLOCK_SystemCLkDivider(2);
CLOCK_IPGCLkDivider(2);
```

2.1.2 时钟选择控制

```
void CLOCK_CLKOutSelect(CLOCK_CLKOutMode mode);     //输出时钟的选择

void CLOCK_SIRCCmd(FunctionalState state);          //是否使能 SIRC 时钟

void CLOCK_FXOSCCmd(FunctionalState state);         //是否使能打开 FXOSC 时钟
```


©Levetop Semiconductor Co., Ltd.

```
void CLOCK_EnterLowPowerModeCmd(FunctionalState state);//是否使能允许低功耗模式

void CLOCK_SIRCLowPowerCmd(FunctionalState state);        //是否使能 SIRC 低功耗模式

void CLOCK_FXOSCLowPowerCCmd(FunctionalState state);     //是否使能 FXOSC 低功耗模式

void CLOCK_EFlashLowPowerCCmd(FunctionalState state);     //是否使能 FLASH 低功耗模式

void CLOCK_SleepModeCmd(FunctionalState state);              //是否使能进入休眠模式

void CLOCK_ADCCLKCmd(FunctionalState state);                 //是否使能 ADC 时钟

void CLOCK_StandbyOperationControl(CLOCK_StandbyMode mode);//选择待机操作的模式

void CLOCK_SIRCStableTime(UINT8 value);                      //设置 SIRC 稳定时间

void CLOCK_SXOSCStableTime(UINT16 value);                    //设置 SXOSC 稳定时间

void CLOCK_SXOSCCrystalPortCmd(FunctionalState state);       //是否使能 SXOSC 晶振管脚

void CLOCK_SXOSCPowerSupplyControl(CLOCK_PowerSupply state);//选择供应 SXOSC 的
                                                             //电源
void CLOCK_WDTCLKSelectionControl(CLOCK_WDTClock state);//选择 WDT 的时钟源

void CLOCK_WDT0CLKCmd(FunctionalState state);                //是否使能 WDT0 时钟

void CLOCK_WDT1CLKCmd(FunctionalState state);                //是否使能 WDT1 时钟

int CLOCK_GetSIRCReadyStatus(void);                          //获取 SIRC 的准备状态

int CLOCK_GetSXOSCReadyStatus(void);                         //获取 SXOSC 的准备状态

int CLOCK_GetWDTCLKSwitchDoneStatus(void);                   //获取 WDT 的转换完成和
                                                             //正常工作状态

void CLOCK_ModuleStopCmd(UINT32 module,FunctionalState state);//是否停止相关的模块

void CLOCK_OpenEPTClockDivider(UINT16 div);                 //打开 EPT 时钟和设置预分频
值
```


©Levetop Semiconductor Co., Ltd.

举例：编写一个唤醒源为 WDT1 的休眠函数

```
void Sleep(void)
{
  CLOCK_SIRCLowPowerCmd(DISABLE);                    //禁止 SIRC 低功耗模式
  CLOCK_EFlashLowPowerCCmd(ENABLE);                  //使能 FLASH 低功耗模式
  CLOCK_SIRCLowPowerCmd(ENABLE);                     //使能 SIRC 低功耗模式
  CLOCK_StandbyOperationControl(Clock_StanbyMode_3);
  //选择待机模式为模式 3：
  //ADC clock disable，system clock disable，clock source disable，LDO standby

  CHIP_VREFShutDownCmd();                            //关闭 VREF
  CHIP_WakeupSourceFilterCmd(ENABLE);                //使能唤醒源滤波器
  CHIP_SelectWKUPSource(WakeupSource_WDT1Interrupt);//选择相关的唤醒源
  CHIP_PinReduceDriveCmd(Pin_All,ENABLE);            //选择所有管脚都降低驱动
  CLOCK_ModuleStopCmd(Clock_ALL,ENABLE);             //使能停止所有的时钟
  CLOCK_SIRCCmd(ENABLE);                             //使能 SIRC
  CLOCK_SleepModeCmd(ENABLE);                        //进入休眠模式

/*休眠被唤醒后会在此处继续执行程序*/

  CLOCK_ModuleStopCmd(Clock_ALL,DISABLE);            //恢复打开所有的时钟
}
/*休眠函数需要调用 misc.c 的部分函数，如上例程所示*/
```


©Levetop Semiconductor Co., Ltd.

2.2 中断控制(EIC)

EIC 中断控制器包含了 32 个中断源。

功能封装函数档案名：misc.c
功能封装函数声明档案名：misc.h

```
void EIC_IRQChannelCmd(EIC_IER channel,FunctionalState state);      //是否使能相关的中断源

void EIC_PriorityLevelSelect(EIC_IER channel,EIC_Priority value);   //相关的中断源优先级设置

void EIC_SetInterruptPending(EIC_IER channel);                      //保留挂起的相关中断源

void EIC_ClearInterruptPending(EIC_IER channel);                    //清除挂起的相关中断源
```

举例：开启 PIT1 和 PIT2 的中断源和设置优先级

```
EIC_IRQChannelCmd(IE_PIT0,ENABLE);
EIC_PriorityLevelSelect(IE_PIT0, Priority_128);
EIC_IRQChannelCmd(IE_PIT1,ENABLE);
EIC_PriorityLevelSelect(IE_PIT1, Priority_128);
```


©Levetop Semiconductor Co., Ltd.

2.3 EPT 定时器设置(EPT)

EPT 的时钟源为 system_clock，EPT 中断优先级为最高。

功能封装函数档案名：ept_lib.c
功能封装函数声明档案名：ept_lib.h

```
void EPT_SetReload(UINT32 RLD);                          //设置 EPT 重载值

void EPT_InterruptRequestCmd(FunctionalState state);     //是否使能 EPT 中断请求

void EPT_CLKSourceSelect(EPT_CLKSource clk);             //选择 EPT 的时钟源

void EPT_CounterCmd(FunctionalState state);              //是否使能 EPT 计数器

int EPT_GetCountDownStatus(void);                        //获取计数为零标志的状态
```

举例：配置一个定时为 100ms 的 EPT（默认 system_clock 为 72MHZ）

```
void EPT_Init(void)
{
EPT_SetReload(7200000);                   //设置 EPT 重载值为 7200000
EPT_InterruptRequestCmd(ENABLE);          //使能 EPT 中断请求
EPT_CLKSourceSelect(EPT_CoreClock);       //选择内部时钟源
EPT_CounterCmd(ENABLE);                   //使能 EPT 计数器
}
```


©Levetop Semiconductor Co., Ltd.

2.4 PIT 定时器设置(PITx)

PIT 定时器的时钟源是 system_clock,分别有 PIT0、PIT1、PIT2 和 PIT3 四个定时器

功能封装函数档案名：pit_lib.c
功能封装函数声明档案名：pit_lib.h

```
void PITCLKConfig(pit *PITx,PIT_PreClock_Div PreClk_DIVx, UINT16 count);//配置时钟分频值和计数
                                                                          //重载值
void PIT_InterruptCmd(pit *PITx,FunctionalState state);                   //是否使能 PIT 中断

void PIT_ReloadCmd(pit *PITx,PIT_ReloadState state);                      //选择重装载的模式

void PIT_Cmd(pit *PITx,FunctionalState state);                            //是否使能 PIT

int PIT_GetInterruptStatus(pit *PITx);                                    //获取 PIT 中断状态

void PIT_ClearInterruptPendingBit(pit *PITx);                             //清除 PIT 中断标志
```

举例：配置一个定时为 1ms 的 PIT0（默认 system_clock 为 72MHZ）

```
Void PIT0_Init(void)
{
PITCLKConfig(PIT0,PreClk_DIV16,4500);     //选择 PIT0，将 system_clock 时钟 16 分频，设置 4500
                                          //为计数装载值
PIT_ReloadCmd(PIT0,Reloaded_From_PMR);//选择从 PMR 中获取的值为计数装载值
PIT_InterruptCmd(PIT0,ENABLE);            //PIT0 中断使能
PIT_Cmd(PIT0,ENABLE);                     //PIT0 使能
EIC_IRQChannelCmd(IE_PIT0,ENABLE);        //开启 PIT0 中断源
```


©Levetop Semiconductor Co., Ltd.

2.5 GPIO 口配置和外部中断设置(EPORTx)

EPORT 包含 EPORT0 和 EPORT1，而每个 EPORT 又包含 8 个外部引脚，每个 pin 同时具有 GPIO 的特性和外部中断的功能。

功能封装函数档案名：eport_lib.c
功能封装函数声明档案名：eport_lib.h

void EFM_ADCChannelShutdown(UINT8 EPORT0_PORTx,FunctionalState state);
//由于 EPORT03~EPORT07 与对应的 ADC[3:7]通道共用,在使用该 IO 口时需先禁止对应的 ADC 通道

void EPORT_PortDirectionControl(eport *EPORTx,UINT8 EPORT_PORTx,EPORT_Direction state);
//配置 I/O 口的方向

void EPORT_OpenDrainOutputCmd(eport *EPORTx,UINT8 EPORT_PORTx,FunctionalState state);
//是否使能 I/O 口为开漏输出

void EPORT_PullupCmd(eport *EPORTx,UINT8 EPORT_PORTx,FunctionalState state);
//是否使能 I/O 口为上拉

void EPORT_SetPortData(eport *EPORTx,UINT8 EPORT_PORTx,EPORT_Level state);
//设置 I/O 口输出的电平

int EPORT_GetPortDataStatus(eport *EPORTx,UINT8 EPORT_PORTx);
//获取 I/O 口的电平状态

void EPORT_PortTriggerConfigure(eport *EPORTx,UINT8 EPORT_PORTx,
EPORT_PinAssignment state);
//选择外部中断边沿触发的方式

void EPORT_PortLevelPolarityControl(eport *EPORTx,UINT8 EPORT_PORTx,EPORT_Level state);
//选择外部中断电平触发的电平高低

void EPORT_InterruptCmd(eport *EPORTx,UINT8 EPORT_PORTx,FunctionalState state);
//是否使能 I/O 口外部中断

int EPORT_GetEdgePortStatus(eport *EPORTx,UINT8 EPORT_PORTx);
//获取指定的外部中断触发状态

void EPORT_ClearEdgePortPendingBit(eport *EPORTx,UINT8 EPORT_PORTx);
//清除 I/O 口中断标志

©Levetop Semiconductor Co., Ltd.

void EPORT_DigitalFilterCmd(eport *EPORTx,FunctionalState state);
//是否使能开启 EPORT 的数字滤波

void EPORT_SetDigitalFilterPulseWidth(eport *EPORTx,UINT8 width);
//设置滤波器脉冲的宽度

举例：配置 GPIO 功能和实现外部中断功能

```
EFM_ADCChannelShutdown(EPORT0_PORT3|EPORT0_PORT4,ENABLE);//使能对 ADC 的 3 通道和
                                                         //4 通道的禁止
/*配置 EPORT03 为普通 IO 口*/
Void EPORT03_Init(void)
{
EPORT_PortDirectionControl(EPORT0,EPORT_PORT3, EPORT_Output); //配置 EPORT03 为输出
EPORT_OpenDrainOutputCmd(EPORT0,EPORT_PORT3, DISABLE);       //禁止 EPORT03 为开漏输
出
EPORT_SetPortData(EPORT0,EPORT_PORT3, EPORT_Highlevel);       //配置 EPORT03 为高电平
int a=EPORT_GetPortDataStatus(EPORT0,EPORT_PORT3);            //获取 EPORT03 的电平状态
}

/*配置 EPORT04 为外部中断口*/
Void EPORT04_Init(void)
{
EPORT_PortTriggerConfigure(EPORT0,EPORT_PORT4,EPORT_Level_Sensitive);//配置 EPORT04 为
                                                                     //电平触发
EPORT_PortLevelPolarityControl(EPORT0, EPORT_PORT4,EPORT_Lowlevel);  //配置 EPORT04 为
                                                                     //低电平触发
EPORT_InterruptCmd(EPORT0, EPORT_PORT4,ENABLE);            //使能 EPORT04 外部中断
EPORT_DigitalFilterCmd(EPORT0, ENABLE);                    //使能数字滤波器开启
EPORT_SetDigitalFilterPulseWidth(EPORT0,127);              //设置数字滤波脉冲宽度为 127
EIC_IRQChannelCmd(IE_EPORT04,ENABLE);                      //开启 EPORT04 中断源
}
```


©Levetop Semiconductor Co., Ltd.

2.6 串口设置(UART0)

UART 的时钟源为 IPG_clock，通过运算转化，实现对波特率的简单配置。

功能封装函数档案名：uart_lib.c
功能封装函数声明档案名：uart_lib.h

```
void UART_BaudRateConfigure(UINT32 BAUDRATE);                //设置需要的波特率

void UART_DataFormatMode(UART_Format mode);                  //选择传输帧的格式

void UART_ParityControl(UART_Parity state);                  //选择奇偶校验的模式

void UART_TransmitterCmd(FunctionalState state);             //是否使能发送器

void UART_ReceiverCmd(FunctionalState state);                //是否使能接收器

void UART_LoopModeSelect(UART_LoopSelect mode);              //选择传输模式是否要循环模式

void UART_ITConfig(UINT8 UART_IT, FunctionalState State);    //是否使能 UART 相应的中断

int UART_GetFlagStatus(UINT16 UART_FLAG);                    //获取 UART 相应标志位状态

void UART_SendData(UINT16 Data);                             //数据发送函数

UINT16 UART_ReceiveData(void);                               //数据接收函数
```

举例：实现 UART 初始化和串口接收数据功能

```
/* UART 初始化函数*/
void Uart_Init(void)
{
 UART_BaudRateConfigure(115200);                             //设置波特率为 115200
 UART_DataFormatMode(UART_1StartBit_8DataBits_1StopBit);     //选择 1 位起始位、8 位数据位和
                                                             //1 位停止位的帧格式
 UART_LoopModeSelect(UART_NormalMode);                       //选择正常传输模式
 UART_ReceiverCmd(ENABLE);                                   //使能接收器的开启
 UART_ITConfig(UART_IE_RIE,ENABLE);                          //使能接收器中断
 UART_ITConfig(UART_IE_ILIE,ENABLE);                         //使能空闲中断
 EIC_IRQChannelCmd(IE_UART,ENABLE);                          //开启 UART 中断源
}
```


©Levetop Semiconductor Co., Ltd.

```
/* UART 中断服务函数*/
void SCI0_Handler(void)
{
 if(UART_GetFlagStatus(UART_FLAG_RDRF))                    //获取接收器满标志的状态
 {
  RxBuf[RxIndex]=UART_ReceiveData();
  UART_SendData(RxBuf[RxIndex]);
  RxIndex++;
  if(RxIndex==256)
  {
   RxIndex=0;
  }
 }

 if(UART_GetFlagStatus(UART_FLAG_IDLE))                    //获取空闲标志的状态
 {
  RxBuf[RxIndex]=UART_ReceiveData();                       //接收数据
  RxBuf[RxIndex]='\0';
  RxIndex=0;
 }
}
```


©Levetop Semiconductor Co., Ltd.

2.7 脉冲宽度调制(PWMx)

PWM 的时钟源是 IPG _clock，包含 PWM0 和 PWM1，而每个 PWM 含有 4 个定时器通道。

功能封装函数档案名：pwm_lib.c
功能封装函数声明档案名：pwm_lib.h

void PWM_CLKSelectPrescale(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,UINT8 div, PWM_CLKDivide clkdiv);
//配置 PWM 相关定时器预分频值和选择分频时钟

void PWM_ModeSelect(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,PWM_Mode state);
//选择 PWM 模式为单次触发还是自动重装

void PWM_InverterTurnOnCmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx, FunctionalState state);
//是否使能 PWM 相关定时器输出反转

void PWM_DeadZoneGeneratorCmd(pwm *PWMx,PWM_DeadZone DeadZonex, FunctionalState state);
//是否使能 PWM 死区发生器

void PWM_SetPeriod(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,UINT16 period);
//设置 PWM 相关定时器周期计数值

void PWM_SetDuty(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,UINT16 duty);
//设置 PWM 相关定时器占空比较值

void PWM_InterruptCmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx, FunctionalState state);
//是否使能 PWM 相关定时器中断

int PWM_GetInterruptStatus(pwm *PWMx,PWM_TimerChannel PWM_TIMERx);
//获取 PWM 相关定时器中断状态

void PWM_OutputCmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,FunctionalState state);
//是否使能 PWM 相关引脚为输出

void PWM_PullupCmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,FunctionalState state);
//是否使能 PWM 相关引脚为上拉

©Levetop Semiconductor Co., Ltd.

void PWM_PortDataConfigure(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,
PWM_PortLevel state);
//设置 PWM 相关引脚的电平

int PWM_GetPortDataStatus(pwm *PWMx,PWM_TimerChannel PWM_TIMERx);
//获取 PWM 相关引脚的电平状态

void PWM_Cmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,FunctionalState state);
//是否使能 PWM 相关定时器

Void PWM_CaptureCmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,
FunctionalState state);
//是否使能 PWM 相关定时器的通道捕获

void PWM_CaptureFallingInterruptCmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,
FunctionalState state);
//是否使能 PWM 相关定时器的下降沿捕获中断

void PWM_CaptureRisingInterruptCmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,
FunctionalState state);
//是否使能 PWM 相关定时器的上升沿捕获中断

void PWM_CaptureInverterCmd(pwm *PWMx,PWM_TimerChannel PWM_TIMERx,
FunctionalState state);
//是否使能 PWM 相关定时器捕获反转

int PWM_GetCaptureFallingStatus(pwm *PWMx,PWM_TimerChannel PWM_TIMERx) ;
//获取 PWM 相关定时器捕获下降沿标志的状态

int PWM_GetCaptureRisingStatus(pwm *PWMx,PWM_TimerChannel PWM_TIMERx) ;
//获取 PWM 相关定时器捕获上升沿标志的状态

int PWM_GetCaptureInterruptStatus(pwm *PWMx,PWM_TimerChannel PWM_TIMERx);
//获取 PWM 相关定时器捕获中断标志的状态

void PWM_ClearCaptureFallingPendingBit(pwm *PWMx,PWM_TimerChannel PWM_TIMERx);
//清除 PWM 相关定时器捕获下降沿标志

void PWM_ClearCaptureRisingPendingBit(pwm *PWMx,PWM_TimerChannel PWM_TIMERx);
//清除 PWM 相关定时器捕获上升沿标志

©Levetop Semiconductor Co., Ltd.

```
void PWM_ClearCaptureInterruptPendingBit(pwm *PWMx,PWM_TimerChannel PWM_TIMERx);
//清除 PWM 相关定时器捕获中断标志
```

举例：设置一个占空比为 50%的 PWM 定时器

```
void PWM0_Init(void)
{
 PWM_CLKSelectPrescale(PWM0,PWM_Timer1,1,PWM_CLK_DividedBy4);//设置 PWM01 预分频值
                                                            //为 1，选择 4 分频时钟
  PWM_ModeSelect(PWM0,PWM_Timer1,PWM_AutoLoadMode);//选择 PWM01 为动装载模式
  PWM_SetPeriod(PWM0,PWM_Timer1,180);               //设置 PWM01 周期计数值为 180
  PWM_SetDuty(PWM0,PWM_Timer1,90);                  //设置 PWM01 占空比较值为 90
  PWM_OutputCmd(PWM0,PWM_Timer1,ENABLE);            //使能 PWM01 为输出
  PWM_Cmd(PWM0,PWM_Timer1,ENABLE);                  //使能 PWM01
}
```


©Levetop Semiconductor Co., Ltd.

2.8 ADC 设置(ADC)

ADC 的时钟源为 ADC_clock。

功能封装函数档案名：adc_lib.c
功能封装函数声明档案名：adc_lib.h

```
void CLOCK_ADCCLKConfig(ADC_Clock clkdiv,ADC_PreClock_Div PreClk_DIVx);//配置 ADC 时钟分
                                                                        //频值和预分频值
void ADC_SelectChannelLength(ADC_Length SEQ_LENx);            //选择转换序列的长度

void ADC_ContinuousConvModeCmd(FunctionalState state);        //是否使能连续转换模式

void ADC_DataLeftAlignmentCmd(FunctionalState state);         //是否使能数据左对齐方式

void ADC_Resolution(ADC_Accuracy accuracy);                   //选择转换的分辨率

void ADC_SelectChannel(ADC_Ccw ADC_CCWx,ADC_CH channel);//选择转换序列的位置和对
                                                        //应的转换通道

void ADC_Samplingtime(UINT16 qclock);                         //设置采样时间

void ADC_DMACmd(FunctionalState state);                       //是否使能 DMA 请求控制

void ADC_EnCmd(FunctionalState state);                        //是否使能 ADC

void ADC_DisCmd(FunctionalState state);                       //是否禁止 ADC

void ADC_StartConvCmd(FunctionalState state);                 //是否使能开始转换 ADC

void ADC_StopConvCmd(FunctionalState state);                  //是否使能停止转换 ADC

void ADC_InterruptConfig(UINT8 ADC_IT,FunctionalState state); //是否使能 ADC 相应的中断

int ADC_GetInterruptStatus(UINT8 ADC_FLAG);                   //获取 ADC 相应中断标志位状态

void ADC_ClearInterruptPendingBit(UINT8 ADC_FLAG);            //清除 ADC 相应的标志位

int ADC_GetCRStatus(UINT8 ADC_CR_FLAG);                       //获取 ADC 相应控制标志位状态

int ADC_GetConversionValue(void);                             //获取 ADC 对应通道的转换值
```


©Levetop Semiconductor Co., Ltd.

举例：ADC 单通道转换

```
/* ADC 初始化*/
void ADC_Init()
{
 CLOCK_ADCCLKConfig (ADCCLK_DIVBY6,PreClk_DIV12); //先将 system_clock6 分频，然后再对其12 分频
                                                 //若系统时钟 72MHZ,ADC 时钟为 1MHZ
 ADC_ContinuousConvModeCmd(DISABLE);             //禁止 ADC 连续转换
 ADC_SelectChannelLength(SEQ_LEN1);              //选择序列为 1 个通道的长度
 ADC_Resolution(Accuracy12);                     //选择 12 位的分辨率
 ADC_ DataLeftAlignmentCmd(DISABLE);             //禁止数据左对齐
 ADC_SelectChannel(ADC_CCW0,ADC_CH7);            //选择转换序列的位置为 CCW0，要转换的
                                                 //通道为 ADC7
 ADC_Samplingtime(4);                            //采样时间为 4 个单位时钟周期
 ADC_EnCmd(ENABLE);                              //使能 ADC
 while(!ADC_GetInterruptStatus(ADC_FLAG_ADRDY)); //等待 ADC 的准备状态
}

/*获取 ADC 对应通道的转换值*/
UINT16 Get_ADC_Val(ADC_CH ch)
{
 UINT16 Val = 0;
 ADC_SelectChannel(ADC_CCW0,ch);                 //选择转换序列的位置为 CCW0，选择要转换
                                                 //的通道：ch
 ADC_StartConvCmd(ENABLE);                       //使能 ADC 开始转换
 while(!ADC_GetInterruptStatus(ADC_FLAG_EOSEQ)); //等待 ADC 序列转换结束状态
 ADC_ClearInterruptPendingBit(ADC_FLAG_EOSEQ);   //清除 ADC 序列转换结束标志
 Val=ADC_GetConversionValue();                   //获取对应 ADC 通道的转换值
 return Val;
}
```


©Levetop Semiconductor Co., Ltd.

2.9 DMA 设置(LDMA)

DMA 模块提供了 5 个传输通道，外设对应选择，支持字节、半字和字的传输。

功能封装函数档案名：dma_lib.c
功能封装函数声明档案名：dma_lib.h

```
void DMA_PeripheralSelect(DMA_Channel DMA_CHx);
//选择 DMA 相关通道对应的外设

void DMA_MemoryBaseAddr(DMA_Channel DMA_CHx,UINT32 MBAddr);
//设置 DMA 相关通道的内存基地址

void DMA_BufferSize_DataSize(DMA_Channel DMA_CHx,LDMA_Data_Width width,
UINT16 byte_Count);
//设置 DMA 相关通道的数据宽度和需传输的总字节数

void DMA_TransferDirection(DMA_Channel DMA_CHx,DMA_TD state);
//选择 DMA 相关通道的传输方向

void DMA_LeftByteTransferFirstCmd(DMA_Channel DMA_CHx,FunctionalState state);
//是否使能 DMA 相关通道的左边字节先传输的功能

void DMA_MemoryChangeMode(DMA_Channel DMA_CHx,DMA_MINC_MDEC mode);
//选择 DMA 相关通道的内存变化的模式

void DMA_CircularModeCmd(DMA_Channel DMA_CHx,FunctionalState state);
//是否使能 DMA 相关通道的循环模式

void DMA_HardwareTriggerModeCmd(DMA_Channel DMA_CHx,FunctionalState state);
//是否使能 DMA 相关通道的硬件触发模式

void DMA_CompleteTransferInterruptCmd(DMA_Channel DMA_CHx,FunctionalState state);
//是否使能 DMA 相关通道的传输完成中断

void DMA_HalfTransferInterruptCmd(DMA_Channel DMA_CHx,FunctionalState state);
//是否使能 DMA 相关通道的半传输完成中断

void DMA_Cmd(DMA_Channel DMA_CHx,FunctionalState state);
//是否使能 DMA 相关通道
```


©Levetop Semiconductor Co., Ltd.

int DMA_GetConfigurationErrorStatus(DMA_Channel DMA_CHx);
//获取 DMA 相关通道的配置错误状态

int DMA_GetTransferBusyStatus(DMA_Channel DMA_CHx);
//获取 DMA 相关通道的传输忙状态

int DMA_GetCompleteTransferStatus(DMA_Channel DMA_CHx);
//获取 DMA 相关通道的传输完成状态

int DMA_GetHalfTransferStatus(DMA_Channel DMA_CHx);
//获取 DMA 相关通道的半传输完成状态

void DMA_ClearCompleteTransferInterruptPendingBit(DMA_Channel DMA_CHx);
//清除 DMA 相关通道的传输完成中断标志位

void DMA_ClearHalfTransferInterruptPendingBit(DMA_Channel DMA_CHx);
//清除 DMA 相关通道的半传输完成中断标志位

举例：利用 DMA 实现 ADC 多通道的转换

```
/* DMA_ CH0 初始化*/
void ADC_LDMA_Inite(UINT32 Rx_Addr)
{
 DMA_PeripheralSelect(DMA_CH0);                      //选择对应外设 ADC read 的
DMA_CH0
 DMA_Cmd(DMA_CH0,DISABLE);                           //禁止 DMA_CH0
 DMA_MemoryBaseAddr(DMA_CH0,Rx_Addr);                //设置对应通道内存基地地址 Rx_Addr
 DMA_BufferSize_DataSize(DMA_CH0, Data_Width16,10);  //选择的数据宽度为 2BYTE，因为要传
                                                     //输 5 个数据，则总共要传输 10 个字节
 DMA_LeftByteTransferFirstCmd(DMA_CH0,DISABLE);      //禁止 DMA_CH0 的左边字节先传输
 DMA_TransferDirection(DMA_CH0,Peripheral_To_Memory);//选择 DMA_CH0 的传输方向为
                                                     //外设到内存
 DMA_MemoryChangeMode(DMA_CH0,Memory_Address_Increment);//选择 DMA_CH0 内存递增
                                                        //模式
 DMA_CircularModeCmd(DMA_CH0,DISABLE)                //使能 DMA_CH0 循环模式
 ADC_DMACmd(ENABLE);                                 //使能 DMA_CH0
 DMA_CompleteTransferInterruptCmd(DMA_CH0,ENABLE);   //使能 DMA_CH0 传输完成的中断
 EIC_IRQChannelCmd(IE_LDMAC,ENABLE);                 //开启 DMA 中断源
 while(DMA_GetConfigurationErrorStatus(DMA_CH0));    //等待 DMA_CH0 配置错误状态
 DMA_Cmd(DMA_CH0,ENABLE);                            //使能 DMA_CH0
}
```


©Levetop Semiconductor Co., Ltd.

```
/*ADC 多通道初始化*/
void ADC_Init1()
{
 ADCCLKConfig(ADCCLK_DIVBY6,PreClk_DIV12);
 ADC_ContinuousConvMode(ENABLE);
 ADC_NbrOfChannel(SEQ_LEN5);                    //设置 ADC 转换序列为 5 个通道的长度
 ADC_Resolution(Accuracy12);
 ADC_DataAlign_Left(DISABLE);

 ADC_SelectChannel(ADC_CCW0,ADC_CH2);
 ADC_SelectChannel(ADC_CCW1,ADC_CH3);
 ADC_SelectChannel(ADC_CCW2,ADC_CH4);
 ADC_SelectChannel(ADC_CCW3,ADC_CH5);
 ADC_SelectChannel(ADC_CCW4,ADC_CH6);       //分别选择对应的序列位置和要转换的 ADC 通道

 ADC_Samplingtime(4);
 ADC_EnCmd(ENABLE);
 while(!ADC_GetInterruptStatus(ADC_FLAG_ADRDY));
 ADC_StartConvCmd(ENABLE);
}
/*经过 DMA 传输，ADC 多通道的转换值分别依次地存放在内存基地址上*/
```


©Levetop Semiconductor Co., Ltd.

2.10 COMP 比较器设置(COMPx)

COMP 包含 COMP0 和 COMP1。

功能封装函数档案名：comp_lib.c
功能封装函数声明档案名：comp _lib.h

void COMP_Cmd(comp *COMPx,FunctionalState state);
//是否使能 COMP

void COMP_HysteresisControl(comp *COMPx,COMP_Hysteresis state);
//选择 COMP 滞后电压大小

void COMP_OutputToPadCmd(comp *COMPx,FunctionalState state);
//是否使能 COMP 输出

void COMP_RisingEdgeInterruptCmd(comp *COMPx,FunctionalState state) ;
//是否使能 COMP 上升沿中断

void COMP_FallingEdgeInterruptCmd(comp *COMPx,FunctionalState state);
//是否使能 COMP 下降沿中断

void COMP_SelectWakeUpMode(comp *COMPx,COMP_WakeupMode state);
//选择 COMP 唤醒的模式

void COMP_HighSpeedModeCmd(comp *COMPx,FunctionalState state);
//是否使能 COMP 高速模式

void COMP_NegativeInputMUXSelect(comp *COMPx,COMP_MUX channel);
//选择 COMP 负极端输入的通道

void COMP_PositiveInputMUXSelect(comp *COMPx,COMP_MUX channel);
//选择 COMP 正极端输入的通道

void COMP_OutputDigitalFilterCmd(comp *COMPx,FunctionalState state);
//是否使能 COMP 输出数字滤波器

void COMP_OutputDigitalFilterPulseWidth(comp *COMPx,UINT8 width);
//设置 COMP 输出数字滤波脉冲宽度

©Levetop Semiconductor Co., Ltd.

```
int COMP_GetRisingEdgeStatus(comp *COMPx);                  //获取 COMP 上升沿状态

int COMP_GetFallingEdgeStatus(comp *COMPx);                 //获取 COMP 下降沿状态

int COMP_GetOutputCompareStatus(comp *COMPx);               //获取 COMP 输出比较结果

void COMP_ClearRisingEdgeInterruptPendingBit(comp *COMPx); //清除 COMP 上升沿中断标志位

void COMP_ClearFallingEdgeInterruptPendingBit(comp *COMPx);//清除 COMP 下降沿中断标志位
```

举例：初始化 COMP0 和设置比较中断

```
/*初始化 COMP0*/
void Init_COMP0(COMP_MUX chN,COMP_MUX chP)
{
 EIC_IRQChannelCmd(IE_COMP0,DISABLE);                      //关闭 COMP0 中断源
 COMP_HysteresisControl(COMP0,Hysteresis_Disabled);        //选择 COMP0 无电压滞后模式
 COMP_NegativeInputMUXSelect(COMP0, COMP_AIN7);            //选择 COMP0 负极端的输入通道为
                                                           // COMP_AIN7
 COMP_PositiveInputMUXSelect(COMP0, COMP_AIN3);            //选择 COMP0 正极端的输入通道为
                                                           // COMP_AIN3
 COMP_OutputToPadCmd(COMP0,ENABLE);                        //使能 COMP0 输出
 COMP_RisingEdgeInterruptCmd(COMP0,ENABLE);                //使能 COMP0 上升沿中断
 COMP_FallingEdgeInterruptCmd(COMP0,ENABLE);               //使能 COMP0 下降沿中断
 COMP_OutputDigitalFilterCmd(COMP0,ENABLE);                //使能 COMP0 输出数字滤波器
 COMP_OutputDigitalFilterPulseWidth(COMP0,30);             //设置 COMP0 数字滤波脉冲宽度为 30
 COMP_Cmd(COMP0,ENABLE);                                   //使能 COMP0
 EIC_IRQChannelCmd(IE_COMP0,ENABLE);                       //开启 COMP0 中断源
}
/* COMP0 中断服务函数*/
void COMP0_Handler(void)
{
 if(COMP_GetRisingEdgeStatus(COMP0))                       //获取 COMP0 上升沿的状态
 {
  COMP_ClearRisingEdgeInterruptPendingBit(COMP0);          //清除上升沿中断标志位
 }
 if(COMP_GetFallingEdgeStatus(COMP0))                      //获取 COMP0 下降沿的状态
 {
  COMP_ClearFallingEdgeInterruptPendingBit(COMP0);         //清除下降沿中断标志位
 }
}
```


©Levetop Semiconductor Co., Ltd.

2.11 I2C 硬件设置(I2C)

I2C 的时钟源是 IPG_clock。

功能封装函数档案名：iic_lib.c
功能封装函数声明档案名：iic _lib.h

```
void I2C_CLKTestCmd(FunctionalState state);                     //是否使能 I2C 时钟测试模式

void I2C_CLKPrescalerDivider(UINT8 div);                        //设置 I2C 时钟预分频值

void I2C_Cmd(FunctionalState state);                            //是否使能 I2C

void I2C_ModeSelect(I2C_MSMOD state);                           //选择从模式或主模式

void I2C_AcknowledgeCmd(FunctionalState state);                 //是否使能 I2C 应答

void I2C_GenerateRepeatSTARTCmd(FunctionalState state);         //是否允许产生重复开始位

void I2C_InterruptCmd(FunctionalState state);                   //是否使能 I2C 中断

void I2C_AddressMatchInterruptCmd(FunctionalState state);       //是否使能 I2C 地址匹配中断

void I2C_SlaveHighSpeedMode(FunctionalState state);             //是否使能 I2C 从机高速模式

void I2C_SlaveHighSpeedModeInterruptCmd(FunctionalState state);//是否使能 I2C 从机高速模式
                                                                //中断
void I2C_MasterHighSpeedModeCmd(FunctionalState state);         //是否使能主机高速模式

void I2C_SlaveAddress(UINT8 address);                           //设置从机地址

void I2C_SCLFilterCmd(FunctionalState state);                   //是否使能 SCL 滤波器

void I2C_SDAFilterCmd(FunctionalState state);                   //是否使能 SDA 滤波器

void I2C_SlaveSDALineHoldTime(UINT8 time);                      //设置 SCL 到达下降沿时从机
                                                                // SDA 稳定的时间
void I2C_SendData(UINT8 data);                                  //单字节发送函数

int I2C_ReceiveData(void);                                      //单字节接收函数
```


©Levetop Semiconductor Co., Ltd.

```
int I2C_GetFlagStatus(UINT8 I2C_FLAG);                              //获取 I2C 相应标志位状态

/*以下函数为 I2C 的 SCL、SDA 端口用作 GPIO 口的功能*/
void I2C_SDAConfiguredAsGPIOCmd(FunctionalState state); //是否使能 SDA 配置为 GPIO 口

void I2C_SCLConfiguredAsGPIOCmd(FunctionalState state); //是否使能 SCL 配置为 GPIO 口

void I2C_GPIODataOutputCmd(I2C_SDASCL state);            //选择 SCL 和 SDA 是否配置为输出

void I2C_GPIOOpendrainOutputCmd(I2C_SDASCL state);      //选择 SCL 和 SDA 是否配置为开漏输出

void I2C_GPIOPulldownCmd(I2C_SDASCL state);              //选择 SCL 和 SDA 是否配置为下拉

void I2C_GPIOPullupCmd(I2C_SDASCL state);                //选择 SCL 和 SDA 是否配置为上拉

void I2C_SDAGPIOData(I2C_GPIOValue state);              //配置 SDA 脚的电平

void I2C_SCLGPIOData(I2C_GPIOValue state);              //配置 SCL 脚的电平

int I2C_GetSDAGPIOStatus(void);                          //获取 SDA 的电平状态

int I2C_GetSCLGPIOStatus(void);                          //获取 SCL 的电平状态
```

具体使用可查阅 LT32U02_Test_I2C 工程的 iic.c 文件。

©Levetop Semiconductor Co., Ltd.

2.12 SPI 设置(SPI)

SPI 时钟源为 IPG _clock，包含两个通道。在 fixed 模式下，共用同一个时钟频率；在 variable 模式下，可独立设置分频值。

功能封装函数档案名：spi_lib.c
功能封装函数声明档案名：spi _lib.h

```
void SPI_VariableClockModeCmd(FunctionalState state);      //是否使能 SPI 可变频时钟模式

void SPI_FixedCLKBaudRatePrescaler(UINT16 div);            //设置 SPI 固定时钟预分频值

void SPI_CLK0BaudRatePrescaler(UINT16 div);                //设置 SPI clock0 时钟预分频值

void SPI_CLK1BaudRatePrescaler(UINT16 div);                //设置 SPI clock1 时钟预分频值

void SPI_ClockPolarity(SPI_CPOL cpol);                     //选择 SPI 时钟极性

void SPI_TransferDataWidth(SPI_DataWidth width);           //选择 SPI 传输数据的宽度

void SPI_LSBControlCmd(FunctionalState state);             //是否使能 SPI 最低字节先传输

void SPI_ByteReorderControlCmd(FunctionalState state);     //是否使能字节数据倒序传输

void SPI_StartTransmitControl(SPI_Channel SPI_CHx);        //选择在 SPI 先写入 FIFO 的通道

void SPI_InterruptCount(SPI_Channel SPI_CHx,SPI_TransferCount count);
//设置 SPI 传输要满足的中断次数

void SPI_SlaveSelect(SPI_Channel SPI_CHx,SPI_Level_Select state);
//控制 SPI 相关通道的片选信号

void SPI_WriteFifoEmptyInterruptCmd(SPI_Channel SPI_CHx,FunctionalState state);
//是否使能 SPI 写入 FIFO 空的中断

void SPI_WriteFifoOverrunInterruptCmd(SPI_Channel SPI_CHx,FunctionalState state);
//是否使能 SPI 写入 FIFO 溢出的中断

void SPI_ReadFifoOverrunInterruptCmd(SPI_Channel SPI_CHx,FunctionalState state);
//是否使能 SPI 读出 FIFO 溢出的中断
```


©Levetop Semiconductor Co., Ltd.

```
void SPI_TransferDoneInterruptCmd(SPI_Channel SPI_CHx,FunctionalState state);
//是否使能 SPI 传输完成的中断

void SPI_GuardTime(UINT8 gtime);
//设置启动 SPI 变频时钟时传输数据完成后的守候时间

void SPI_GuardTimerClockSelect(SPI_Channel SPI_CHx);          //选择作为变频时钟的 SPI 通道

void SPI_Cmd(SPI_Channel SPI_CHx,FunctionalState state);          //使能 SPI 相关的通道

int SPI_GetWriteFifoEmptyStatus(SPI_Channel SPI_CHx);          //获取 SPI 写入 FIFO 空的状态

int SPI_GetWriteFifoOverrunStatus(SPI_Channel SPI_CHx);          //获取 SPI 写入 FIFO 溢出的状
态

int SPI_GetWriteFifoFullStatus(SPI_Channel SPI_CHx);          //获取 SPI 写入 FIFO 满的状态

int SPI_GetReadFifoEmptyStatus(SPI_Channel SPI_CHx);          //获取 SPI 读出 FIFO 空的状态

int SPI_GetReadFifoOverrunStatus(SPI_Channel SPI_CHx);          //获取 SPI 读出 FIFO 溢出的状
态

int SPI_GetReadFifoFullStatus(SPI_Channel SPI_CHx);          //获取 SPI 读出 FIFO 满的状态

int SPI_GetTransferDoneStatus(SPI_Channel SPI_CHx);          //获取 SPI 传输完成的状态

void SPI_TransmitterDMARequestCmd(SPI_Channel SPI_CHx,FunctionalState state);
//是否使能发送器 DMA 的请求

void SPI_ReceiverDMARequestCmd(SPI_Channel SPI_CHx,FunctionalState state);
//是否使能接收器 DMA 的请求

void SPI_SendData(SPI_Channel SPI_CHx,UINT32 data);          //SPI 发送数据函数

int SPI_ReceiveData(SPI_Channel SPI_CHx);          //SPI 接收数据函数

/*以下函数为 SPI 的 SCK、SDI0、SDO0、SS0、SDI1、SDO1 和 SS1 端口用作 GPIO 口的功能*/

void SPI_GPIOFunctionControl(SPI_Port port);          //配置要作为 GPIO 口的管脚
```


©Levetop Semiconductor Co., Ltd.

```
void SPI_GPIOOutputControl(SPI_Port port);              //配置要作为输出方向的管脚

void SPI_GPIOInputControl(SPI_Port port);               //配置要作为输入方向的管脚

void SPI_GPIOHighLevelControl(SPI_Port port);           //配置要输出高电平的管脚

void SPI_GPIOLowLevelControl(SPI_Port port);            //配置要输出低电平的管脚

void SPI_GPIOPullupControl(SPI_Port port);              //配置要上拉的管脚

int SPI_GetGPIOStatus(SPI_Port port);                   //获取对应管脚的电平状态
```

举例：实现 SPI 初始化和发送接收功能

```
/* SPI 初始化*/
void SPI_Init()
{
 SPI_VariableClockModeCmd(DISABLE);                     //禁止 SPI 可变频时钟模式
 SPI_FixedCLKBaudRatePrescaler(8);                      //设置固定时钟为(2*(8+1))分频
 SPI_ClockPolarity(Msck_Idles_Low);                     //选择 SPI 空闲状态时 SCK 保持低电平
 SPI_TransferDataWidth(SPI_Width8);                     //设置 SPI 数据传输宽度为 8 位
 SPI_LSBControlCmd(DISABLE);                            //禁止 SPI 最低字节先传输
 SPI_ByteReorderControlCmd(DISABLE);                    //禁止 SPI 字节数据倒序传输
 SPI_StartTransmitControl(SPI_CH0);                     //选择 SPI 写入 FIFO 优先通道为
SPI_CH0
 SPI_InterruptCount(SPI_CH0,Every_Completed_Transfer);  //设置 SPI 传输要满足的中断次数为 1 次
 SPI_SlaveSelect(SPI_CH0,Low_State);                    //拉低 SPI_CH0 的片选信号
 SPI_WriteFifoEmptyInterruptCmd(SPI_CH0,DISABLE);       //禁止 SPI_CH0 写入 FIFO 空的中断
 SPI_WriteFifoOverrunInterruptCmd(SPI_CH0,DISABLE);     //禁止 SPI_CH0 写入 FIFO 溢出的中断
 SPI_ReadFifoOverrunInterruptCmd(SPI_CH0,DISABLE);      //禁止 SPI_CH0 读出 FIFO 溢出的中断
 SPI_TransferDoneInterruptCmd(SPI_CH0,DISABLE);         //禁止 SPI_CH0 传输完成的中断
 SPI_TransmitterDMARequestCmd(SPI_CH0,DISABLE);         //禁止 SPI_CH0 发送器 DMA 的请求
 SPI_ReceiverDMARequestCmd(SPI_CH0,DISABLE);            //禁止 SPI_CH0 接收器 DMA 的请求
 SPI_Cmd(SPI_CH0,ENABLE);                               //使能 SPI_CH0
}
```


©Levetop Semiconductor Co., Ltd.

```
int main(void)
{
 UINT8 j=0;
 UINT8 reV[2]={0};
 SPI_Init();
 while(1)
 {
  SPI_SlaveSelect(SPI_CH0,Low_State);              //拉低 SPI_CH0 的片选信号
  SPI_SendData(SPI_CH0,255-j);                      // SPI_CH0 发送数据
  while(!SPI_GetTransferDoneStatus(SPI_CH0));       //等待 SPI_CH0 传输完成状态
  while(SPI_GetReadFifoEmptyStatus(SPI_CH0));       //等待 SPI_CH0 读出 FIFO 空的状态
  reV[0] = SPI_ReceiveData(SPI_CH0);                //SPI_CH0 接收数据
  SPI_SlaveSelect(SPI_CH0,High_State);              //拉高 SPI_CH0 的片选信号
  LTPrintf("%d   SPI rec : %x %x\n\n",j ,reV[0]);
  DelayMs(200);
  j++;
  if(j>255) j = 0;
 }
}
/*该例程是将 SPI_CH0 通道的 SDI 和 SDO 端口短接，验证了 SPI 的发送接收功能*/
```


©Levetop Semiconductor Co., Ltd.

2.13 看门狗设置(WDTx)

WDT 的时钟源为 32KHZ 或者 32.768KHZ，包含了 WDT0 和 WDT1

功能封装函数档案名：wdt_lib.c
功能封装函数声明档案名：wdt _lib.h

void WDT_WaitModeControl(wdt *WDTx,WDT_Mode state); //选择 WDT 等待模式的状态

void WDT_DozeModeControl(wdt *WDTx,WDT_Mode state); //选择 WDT 休眠模式的状态

void WDT_StopModeControl(wdt *WDTx,WDT_Mode state); //选择 WDT 停止模式的状态

void WDT_DbgModeControl(wdt *WDTx,WDT_Mode state); //选择 WDT 调试模式的状态

void WDT_SetPrescaler(wdt *WDTx,WDT_Prescaler pre); //设置 WDT 的预分频值

void WDT_SetReload(wdt *WDTx,UINT16 reload); //设置 WDT 的计数重载值

void WDT_ChangeUpdateCmd(wdt *WDTx,FunctionalState update);//是否使能 WDT 重载值更新

void WDT_InterruptCmd(wdt *WDTx,FunctionalState state); //是否使能 WDT 中断

void WDT_Cmd(wdt *WDTx,FunctionalState state); //是否使能 WDT

int WDT_GetInterruptStatus(wdt *WDTx); //获取 WDT 中断的状态

int WDT_GetCLKDomainInterruptStatus(wdt *WDTx); //获取 WDT 时钟域中断的状态

void WDT_ClearInterruptPendingBit(wdt *WDTx); //清除 WDT 中断标志位

void WDT_FeedDog(wdt *WDTx); //WDT 喂狗函数

©Levetop Semiconductor Co., Ltd.

举例：利用 WDT1 实现 1s 的定时中断

```
/*WDT1 初始化*/
void WDT1_Init(void)
{
 CLOCK_SXOSCCrystalPortCmd(ENABLE);                 //使能 SXOSC 晶振引脚
 CLOCK_SXOSCStableTime(1000);                       //设置 SXOSC 时间为 1000 个 SIRC 时钟周
期
 CLOCK_WDTCLKSelectionControl(Clock_WDT32k768); //选择 32.768k 作为 WDT 时钟源
 CLOCK_WDT1CLKCmd(ENABLE);                          //使能 WDT1 时钟开启
 while(!CLOCK_GetSXOSCReadyStatus());               //等待 SXOSC 的准备状态

 WDT_WaitModeControl(WDT1,WDT_Normal);             //选择正常模式
 WDT_DozeModeControl(WDT1,WDT_Normal);             //选择正常模式
 WDT_StopModeControl(WDT1,WDT_Normal);             //选择正常模式
 WDT_DbgModeControl(WDT1,WDT_Normal);              //选择正常模式
 WDT_InterruptCmd(WDT1,ENABLE);                    //使能 WDT1 中断
 WDT_SetPrescaler(WDT1, WDT_PRE_DIV16);            //设置 WDT1 分频值为 16
 WDT_SetReload(WDT1,2048);                         //设置 WDT1 重载值为 2048
 WDT_Cmd(WDT1,ENABLE);                             //使能 WDT1
 WDT_ChangeUpdateCmd(WDT1,ENABLE);                 //使能 WDT1 允许重载值更新
 EIC_IRQChannelCmd(IE_WDT1,ENABLE);                //开启 WDT1 中断源
}

/*WDT1 中断服务函数*/
void WDT1_Handler(void)
{
 if(WDT_GetInterruptStatus(WDT1))                  //获取 WDT1 中断状态
{
  WDT_ClearInterruptPendingBit(WDT1);              //清除 WDT1 中断标志位
  WDT_InterruptCmd(WDT1,ENABLE);                   //使能 WDT1 中断
  WDT_FeedDog(WDT1);                               //WDT1 喂狗
 }
}
/*当要使用 32.768KHZ 晶振时，需要配置 clk_lib.c 中的部分函数，如上半部分例程所示。*/
```


©Levetop Semiconductor Co., Ltd.

2.14 Flash 操作

Main Flash 容量为 64K Byte，对应地址为 0 到 0xFFFF。

功能封装函数档案名：misc.c
功能封装函数声明档案名：misc.h

```
void EFM_EFlashInit(void);                              //初始化 FLASH

void EFM_EFlashSectorErase(UINT8 sector);               //对 FLASH 相关的扇区进行擦除

void EFM_EFlashProgram(UINT16 EFMaddr,UINT32 data);     //在 FLASH 相关的地址写入数据
```

举例:

```
int main(void)
{
 InitialSystem();
 LTPrintf("\nTest EFlash Write ready");
 EFM_EFlashInit();                                      //初始化 FLASH
 EFM_EFlashSectorErase(127);                            //擦除第 127 个 FLASH 扇区
 LTPrintf("\nErase EFlash end");
 for(i=0;i<5;i++)
 {
  EFM_EFlashProgram(0xFE00+4*i,0xAB345670+i);           //依次在所在地址上写入数据
 }
 LTPrintf("\nWrite EFlash end");
 while(1)
 {
  DelayMs(500);
  DelayMs(500);
  DelayMs(500);
  DelayMs(500);
  DelayMs(500);
  LTPrintf("\nRead 0xFE00 Main flash:\n");
  for(i=0;i<5;i++)
  {
   LTPrintf("0x%x ",IO_READ32(0xFE00+i*4));             //依次读取之前写入地址的值以做验证
  }
  LTPrintf("\n\n\n");
 }
}
```


©Levetop Semiconductor Co., Ltd.

2.15 其它选项设置(CACHE 、EFM、CCM)

功能封装函数档案名：misc.c
功能封装函数声明档案名：misc.h

void CHIP_PowerOnResetControl(EFM_ProgrammableVoltage val,FunctionalState state);
//是否使能上电复位及电压配置

void CHIP_CacheSwitchOn(void); //打开 CACHE

void CHIP_CacheShutDown(void); //关闭 CACHE

void CHIP_ExternalCrystalShutDownCmd(FunctionalState state); //是否使能关闭外部晶振管脚功能

void CHIP_SWDShutDownCmd(FunctionalState state); //是否使能关闭 SWD 管脚功能

void CHIP_CLKOUTShutDownCmd(FunctionalState state); //是否使能关闭 CLKOUT 管脚功能

void CHIP_RSTOUTShutDownCmd(FunctionalState state); //是否使能关闭 RSTOUT 管脚功能

void CHIP_STBShutDownCmd(FunctionalState state); //是否使能关闭 STB 功能

void CHIP_ClockModeControl(EFM_ClockMode state); //芯片时钟源选择

void CHIP_FIRCStableTimeConfiguration(UINT16 time); //开启和设置 FIRC 稳定时间

void CHIP_FXOSCtableTimeConfiguration(UINT16 time); //开启和设置 FXOSC 稳定时间

void CHIP_LDO3p3ShutDownCmd(void); //关闭 LDO3.3V

void CHIP_KernalRespondWaitControl(UINT8 count); //FLASH 响应等待次数选择

void CHIP_WakeupSourceFilterCmd(FunctionalState state); //是否使能唤醒源滤波器

void CHIP_SelectWKUPSource(UINT32 source); //选择相关的唤醒源

void CHIP_PinReduceDriveCmd(UINT32 pin,FunctionalState state);//选择降低相关的管脚驱动

void CHIP_VREFShutDownCmd(void); //关闭 VREF

©Levetop Semiconductor Co., Ltd.

举例：简单地初始化芯片

```
void InitialChip(void)
{
 CHIP_PowerOnResetControl(PVDC_2p48V,ENABLE);     //使能上电复位和设置电压为 2.48V
 CHIP_CacheSwitchOn();                            //打开 CACHE
 CHIP_LDO3p3ShutDownCmd();                        //关闭 LDO3.3V
 CHIP_KernalRespondWaitControl(2);                //设置 FLASH 响应等待次数为 2
 CHIP_FIRCStableTimeConfiguration(50);            //设置 FIRC 稳定时间为 50 个时钟周期

 CLOCK_SystemCLkDivider(2);                       //对 72MHZ 进行 2 分频
 CLOCK_IPGCLkDivider(2);                          //对系统时钟进行 2 分频
}
/*时钟设置调用于 clk_lib.c*/
```


©Levetop Semiconductor Co., Ltd.

2.16 函数库列表

表 2-1：函数库列表

No	函 数 名 称	说 明
有关时钟配置		
1	void CLOCK_SystemCLkDivider();	设置系统时钟分频值
2	void CLOCK_IPGCLkDivider();	设置外设时钟分频值
3	void CLOCK_CLKOutSelect();	选择要输出的时钟
4	void CLOCK_SIRCCmd();	是否使能 SIRC
5	void CLOCK_FXOSCCmd();	是否使能 FXOSC
6	void CLOCK_EnterLowPowerModeCmd();	是否允许低功耗模式
7	void CLOCK_SIRCLowPowerCmd();	是否允许 SIRC 低功耗
8	void CLOCK_FXOSCLowPowerCCmd();	是否允许 FXOSC 低功耗
9	void CLOCK_EFlashLowPowerCCmd();	是否允许 FLASH 低功耗
10	void CLOCK_SleepModeCmd();	是否进入休眠模式
11	void CLOCK_ADCCLKCmd();	是否使能 ADC 时钟
12	void CLOCK_StandbyOperationControl();	选择待机控制的模式
13	void CLOCK_SIRCStableTime();	设置 SIRC 稳定时间
14	void CLOCK_SXOSCStableTime();	设置 SXOSC 稳定时间
15	void CLOCK_SXOSCCrystalPortCmd();	是否使能 SXOSC 晶振管脚
16	void CLOCK_SXOSCPowerSupplyControl();	选择供应 SXOSC 的电源
17	void CLOCK_WDTCLKSelectionControl();	选择 WDT 的时钟源
18	void CLOCK_WDT0CLKCmd();	是否使能 WDT0
19	void CLOCK_WDT1CLKCmd();	是否使能 WDT1
20	int CLOCK_GetSIRCReadyStatus();	获取 SIRC 的准备状态
21	int CLOCK_GetSXOSCReadyStatus();	获取 SXOSC 的准备状态
22	int CLOCK_GetWDTCLKSwitchDoneStatus();	获取 WDT 的转换和工作状态
23	void CLOCK_ModuleStopCmd();	是否要停止相关的模块
24	void CLOCK_OpenEPTClockDivider();	使能 EPT 时钟和设置预分频值
中断控制		
25	void EIC_IRQChannelCmd()	是否开启相关的中断源
26	void EIC_PriorityLevelSelect();	设置相关的中断源的优先级
27	void EIC_SetInterruptPending();	保留相关的中断源挂起
28	void EIC_ClearInterruptPending();	清除相关的中断源挂起
EPT 定时器设置		
29	void EPT_SetReload();	设置 EPT 的计数重载值
30	void EPT_InterruptRequestCmd();	是否使能 EPT 中断请求
31	void EPT_CLKSourceSelect();	选择 EPT 的时钟源

©Levetop Semiconductor Co., Ltd.

No	函 数 名 称	说 明
32	void EPT_CounterCmd();	是否使能 EPT 计数器
33	int EPT_GetCountDownStatus();	获取计数到零的状态
PIT 定时器设置		
34	void PITCLKConfig();	配置时钟分频值和计数重载值
35	void PIT_InterruptCmd();	是否使能 PIT 中断
36	void PIT_ReloadCmd();	选择重装载的模式
37	void PIT_Cmd();	是否使能 PIT
38	int PIT_GetInterruptStatus();	获取 PIT 中断状态
39	void PIT_ClearInterruptPendingBit();	清除 PIT 中断标志
GPIO 口配置和外部中断设置		
40	void EFM_ADCChannelShutdown();	禁止对应的 ADC 管脚
41	void EPORT_PortDirectionControl();	配置 IO 口方向
42	void EPORT_OpenDrainOutputCmd();	是否使能 IO 口开漏输出
43	void EPORT_PullupCmd();	是否使能 IO 口上拉
44	void EPORT_SetPortData();	设置 IO 口输出的电平
45	int EPORT_GetPortDataStatus();	获取 IO 口的电平状态
46	void EPORT_PortTriggerConfigure();	选择外部中断边沿触发方式
47	void EPORT_PortLevelPolarityControl();	选择外部中断电平触发的电平高低
48	void EPORT_InterruptCmd();	是否使能外部中断
49	int EPORT_GetEdgePortStatus();	获取外部中断触发状态
50	void EPORT_ClearEdgePortPendingBit();	清除中断触发标志
51	void EPORT_DigitalFilterCmd();	是否开启数字滤波
52	void EPORT_SetDigitalFilterPulseWidth();	设置滤波器脉冲宽度
串口设置		
53	void UART_BaudRateConfigure();	设置波特率
54	void UART_DataFormatMode();	选择帧的格式
55	void UART_ParityControl();	选择奇偶检验的模式
56	void UART_TransmitterCmd();	是否使能发送器
57	void UART_ReceiverCmd();	是否使能接收器
58	void UART_LoopModeSelect();	选择传输模式
59	void UART_ITConfig();	是否使能相关的中断位
60	int UART_GetFlagStatus();	获取相关的中断标志位状态
61	void UART_SendData();	发送数据
62	UINT16 UART_ReceiveData();	接收数据
脉冲宽度调制		

©Levetop Semiconductor Co., Ltd.

No	函数名称	说明
63	void PWM_CLKSelectPrescale();	设置预分频值和选择分频时钟
64	void PWM_ModeSelect();	选择是单次触发或者自动重载的模式
65	void PWM_InverterTurnOnCmd();	是否使能输出反转
66	void PWM_DeadZoneGeneratorCmd();	是否使能死区发生器
67	void PWM_SetPeriod();	设置周期计数值
68	void PWM_SetDuty();	设置占空比较值
69	void PWM_InterruptCmd();	是否使能中断
70	int PWM_GetInterruptStatus();	获取中断的状态
71	void PWM_OutputCmd();	是否使能相关管脚为输出方向
72	void PWM_PullupCmd();	是否使能相关管脚为上拉
73	void PWM_PortDataConfigure();	设置相关管脚的电平
74	void PWM_GetPortDataStatus();	获取相关管脚的电平状态
75	void PWM_Cmd();	是否使能 PWM
76	void PWM_CaptureCmd();	是否使能 PWM 信道捕获功能
77	void PWM_CaptureFallingInterruptCmd();	是否使能下降沿捕获中断
78	void PWM_CaptureRisingInterruptCmd();	是否使能上升沿捕获中断
79	void PWM_CaptureInverterCmd();	是否使能捕获反转
80	int PWM_GetCaptureFallingStatus();	获取捕获下降沿标志的状态
81	int PWM_GetCaptureRisingStatus();	获取捕获上升沿标志的状态
82	int PWM_GetCaptureInterruptStatus();	获取捕获中断标志的状态
83	void PWM_ClearCaptureFallingPendingBit();	清除捕获下降沿的标志位
84	void PWM_ClearCaptureRisingPendingBit();	清除捕获上升沿的标志位
85	void PWM_ClearCaptureInterruptPendingBit();	清除捕获中断的标志位
ADC 设置		
86	void CLOCK_ADCCLKConfig();	配置 ADC 时钟
87	void ADC_SelectChannelLength();	选择转换序列的长度
88	void ADC_ContinuousConvModeCmd();	是否使能连续转换模式
89	void ADC_DataLeftAlignmentCmd();	是否使能数据左对齐
90	void ADC_Resolution();	选择分辨率
91	void ADC_SelectChannel();	选择转换序列的位置和对应的转换通道
92	void ADC_Samplingtime();	设置采样时间
93	void ADC_DMACmd();	是否使能 DMA 请求控制
94	void ADC_EnCmd();	是否使能 AD

©Levetop Semiconductor Co., Ltd.

No	函 数 名 称	说 明
95	void ADC_DisCmd();	是否禁止 ADC
96	void ADC_StartConvCmd();	是否使能开始转换 ADC
97	void ADC_StopConvCmd();	是否使能停止转换 ADC
98	void ADC_InterruptConfig();	是否使能相关的中断位
99	int ADC_GetInterruptStatus();	获取相关中断标志位状态
100	void ADC_ClearInterruptPendingBit();	清除相关中断标志位
101	int ADC_GetCRStatus();	获取相关控制标志位状态
102	void ADC_GetConversionValue();	获取相关通道的转换值
DMA 设置		
103	void DMA_PeripheralSelect();	选择相关通道对应的外设
104	void DMA_MemoryBaseAddr();	设置相关通道的内存基地址
105	void DMA_BufferSize_DataSize();	设置相关通道的数据宽度和需传输的总字节数
106	void DMA_TransferDirection();	选择相关通道的传输方向
107	void DMA_LeftByteTransferFirstCmd();	是否使能相关通道的左边字节先传输
108	void DMA_MemoryChangeMode();	选择相关通道的内存变化模式
109	void DMA_CircularModeCmd();	是否使能相关通道的循环模式
110	void DMA_HardwareTriggerModeCmd();	是否使能相关通道的循环模式
111	void DMA_CompleteTransferInterruptCmd();	是否使能相关通道的传输完成中断
112	void DMA_HalfTransferInterruptCmd();	是否使能相关通道的半传输完成中断
113	void DMA_Cmd();	是否使能相关通道
114	int DMA_GetConfigurationErrorStatus();	获取相关通道的配置错误状态
115	int DMA_GetTransferBusyStatus();	获取相关通道的传输忙状态
116	int DMA_GetCompleteTransferStatus();	获取相关通道的传输完成状态
117	int DMA_GetHalfTransferStatus();	获取相关通道的半传输完成状态
118	void DMA_ClearCompleteTransferInterruptPendingBit();	清除相关通道的传输完成中断标志位
119	void DMA_ClearHalfTransferInterruptPendingBit();	清除相关通道的半传输完成中断标志位
COMP 比较器设置		

©Levetop Semiconductor Co., Ltd.

No	函 数 名 称	说 明
120	void COMP_Cmd();	是否使能 COMP
121	void COMP_HysteresisControl();	选择滞后电压
122	void COMP_OutputToPadCmd();	是否使能输出
123	void COMP_RisingEdgeInterruptCmd();	是否使能上升沿中断
124	void COMP_FallingEdgeInterruptCmd();	是否使能下降沿中断
125	void COMP_SelectWakeUpMode();	选择唤醒的模式
126	void COMP_HighSpeedModeCmd();	是否使能高速模式
127	void COMP_NegativeInputMUXSelect();	选择负极端输入的通道
128	void COMP_PositiveInputMUXSelect();	选择正极端输入的通道
129	void COMP_OutputDigitalFilterCmd();	是否使能输出数字滤波器
130	void COMP_OutputDigitalFilterPulseWidth();	设置输出数字滤波脉冲宽度
131	int COMP_GetRisingEdgeStatus();	获取上升沿标志位状态
132	int COMP_GetFallingEdgeStatus();	获取下降沿标志位状态
133	int COMP_GetOutputCompareStatus();	获取输出比较结果
134	void COMP_ClearRisingEdgeInterruptPendingBit();	清除上升沿中断标志位
135	void COMP_ClearFallingEdgeInterruptPendingBit();	清除下降沿中断标志位
I2C 硬件设置		
136	void I2C_CLKTestCmd();	是否使能时钟测试模式
137	void I2C_CLKPrescalerDivider();	设置时钟预分频值
138	void I2C_Cmd();	是否使能 I2C
139	void I2C_ModeSelect();	选择从模式或主模式
140	void I2C_AcknowledgeCmd();	是否使能应答
141	void I2C_GenerateRepeatSTARTCmd();	是否允许产生重复开始位
142	void I2C_InterruptCmd();	是否使能 I2C 中断
143	void I2C_AddressMatchInterruptCmd();	是否使能地址匹配中断
144	void I2C_SlaveHighSpeedMode();	是否使能从机高速模式
145	void I2C_SlaveHighSpeedModeInterruptCmd();	是否使能从机高速模式中断
146	void I2C_MasterHighSpeedModeCmd();	是否使能主机高速模式
147	void I2C_SlaveAddress();	设置从机地址
148	void I2C_SCLFilterCmd();	是否使能 SCL 滤波器
149	void I2C_SDAFilterCmd();	是否使能 SDA 滤波器
150	void I2C_SlaveSDALineHoldTime();	设置 SCL 到达下降沿时从机 SDA 稳定的时间
151	void I2C_SendData();	单字节发送
152	int I2C_ReceiveData();	单字节接收
153	int I2C_GetFlagStatus();	获取相关的标志位状态
154	void I2C_SDAConfiguredAsGPIOCmd();	是否使能 SDA 配置为 GPIO 口

©Levetop Semiconductor Co., Ltd.

No	函 数 名 称	说 明
155	void I2C_SCLConfiguredAsGPIOCmd();	是否使能 SCL 配置为 GPIO 口
156	void I2C_GPIODataOutputCmd();	选择 SCL 和 SDA 是否配置为输出
157	void I2C_GPIOOpendrainOutputCmd();	选择 SCL 和 SDA 是否配置为开漏输出
158	void I2C_GPIOPulldownCmd();	选择 SCL 和 SDA 是否配置为下拉
159	void I2C_GPIOPullupCmd();	选择 SCL 和 SDA 是否配置为上拉
160	void I2C_SDAGPIOData();	配置 SDA 管脚的电平
161	void I2C_SCLGPIOData();	配置 SCL 管脚的电平
162	int I2C_GetSDAGPIOStatus();	获取 SDA 的电平状态
163	int I2C_GetSCLGPIOStatus();	获取 SCL 的电平状态
SPI 设置		
164	void SPI_VariableClockModeCmd();	是否使能变频时钟模式
165	void SPI_FixedCLKBaudRatePrescaler();	设置固定时钟预分频值
166	void SPI_CLK0BaudRatePrescaler();	变频时钟模式下，设置 clock0 预分频值
167	void SPI_CLK1BaudRatePrescaler();	变频时钟模式下，设置 clock1 预分频值
168	void SPI_ClockPolarity();	选择时钟极性
169	void SPI_TransferDataWidth();	选择传输数据的宽度
170	void SPI_LSBControlCmd();	是否使能最低字节先传输
171	void SPI_ByteReorderControlCmd();	是否使能字节数据倒序传输
172	void SPI_StartTransmitControl();	选择先写入 FIFO 的通道
173	void SPI_InterruptCount();	设置传输要满足的中断次数
174	void SPI_SlaveSelect();	控制相关通道的片选信号
175	void SPI_WriteFifoEmptyInterruptCmd();	是否使能写入 FIFO 空的中断
176	void SPI_WriteFifoOverrunInterruptCmd();	是否使能写入 FIFO 溢出的中断
177	void SPI_ReadFifoOverrunInterruptCmd();	是否使能读出 FIFO 溢出的中断
178	void SPI_TransferDoneInterruptCmd();	是否使能传输完成的中断
179	void SPI_GuardTime();	设置启动 SPI 变频时钟时传输数据完成后的守候时间
180	void SPI_GuardTimerClockSelect();	选择作为变频时钟的通道
181	void SPI_Cmd();	使能相关的通道
182	int SPI_GetWriteFifoEmptyStatus();	获取写入 FIFO 空的状态

©Levetop Semiconductor Co., Ltd.

No	函 数 名 称	说 明
183	int SPI_GetWriteFifoOverrunStatus();	获取写入 FIFO 溢出的状态
184	int SPI_GetWriteFifoFullStatus();	获取写入 FIFO 满的状态
185	int SPI_GetReadFifoEmptyStatus();	获取读出 FIFO 空的状态
186	int SPI_GetReadFifoOverrunStatus();	获取读出 FIFO 溢出的状态
187	int SPI_GetReadFifoFullStatus();	获取读出 FIFO 满的状态
188	int SPI_GetTransferDoneStatus();	获取传输完成的状态
189	void SPI_TransmitterDMARequestCmd();	是否使能发送器 DMA 的请求
190	void SPI_ReceiverDMARequestCmd();	是否使能接收器 DMA 的请求
191	void SPI_SendData();	发送数据
192	int SPI_ReceiveData();	接收数据
193	void SPI_GPIOFunctionControl();	配置作为 GPIO 口的管脚
194	void SPI_GPIOOutputControl();	配置要作为输出的管脚
195	void SPI_GPIOInputControl();	配置要作为输入的管脚
196	void SPI_GPIOHighLevelControl();	配置输出高电平的管脚
197	void SPI_GPIOLowLevelControl();	配置输出低电平的管脚
198	void SPI_GPIOPullupControl();	配置要上拉的管脚
199	int SPI_GetGPIOStatus();	获取相关管脚的电平状态
看门狗设置		
200	void WDT_WaitModeControl();	选择等待模式的状态
201	void WDT_DozeModeControl();	选择休眠模式的状态
202	void WDT_StopModeControl();	选择停止模式的状态
203	void WDT_DbgModeControl();	选择调试模式的状态
204	void WDT_SetPrescaler();	设置预分频值
205	void WDT_SetReload();	设置计数重载值
206	void WDT_ChangeUpdateCmd();	是否使能重载值更新
207	void WDT_InterruptCmd();	是否使能 WDT 中断
208	void WDT_Cmd();	是否使能 WDT
209	int WDT_GetInterruptStatus();	获取 WDT 中断的状态
210	int WDT_GetCLKDomainInterruptStatus();	获取 WDT 时钟域中断的状态
211	void WDT_ClearInterruptPendingBit();	清除 WDT 中断标志位
212	void WDT_FeedDog();	WDT 喂狗
FLASH 操作		
213	void EFM_EFlashInit();	初始化 FLASH
214	void EFM_EFlashSectorErase();	对 FLASH 相关的扇区进行擦除

©Levetop Semiconductor Co., Ltd.

No	函 数 名 称	说 明
215	void EFM_EFlashProgram();	在FLASH相关的地址写入数据
其它选项设置		
216	void CHIP_PowerOnResetControl();	是否使能上电复位及电压配置
217	void CHIP_CacheSwitchOn();	打开 CACHE
218	void CHIP_CacheShutDown();	关闭 CACHE
219	void CHIP_ExternalCrystalShutDownCmd();	是否使能关闭外部晶振管脚功能
220	void CHIP_SWDShutDownCmd();	是否使能关闭 SWD 管脚功能
221	void CHIP_CLKOUTShutDownCmd();	是否使能关闭 CLKOUT 管脚功能
222	void CHIP_RSTOUTShutDownCmd();	是否使能关闭 RSTOUT 管脚功能
223	void CHIP_STBShutDownCmd();	是否使能关闭 STB 功能
224	void CHIP_ClockModeControl();	选择芯片时钟源
225	void CHIP_FIRCStableTimeConfiguration();	设置 FIRC 稳定时间
226	void CHIP_FXOSCtableTimeConfiguration();	设置 FXOSC 稳定时间
227	void CHIP_LDO3p3ShutDownCmd();	关闭 LDO3.3V
228	void CHIP_KernalRespondWaitControl();	选择 FLASH 响应等待次数
229	void CHIP_WakeupSourceFilterCmd();	是否使能唤醒源滤波器
230	void CHIP_SelectWKUPSource();	选择相关的唤醒源
231	void CHIP_PinReduceDriveCmd();	选择降低相关的管脚驱动
232	void CHIP_VREFShutDownCmd();	关闭 VREF

©Levetop Semiconductor Co., Ltd.

3. 开发板介绍

3.1 LT32U02 开发板

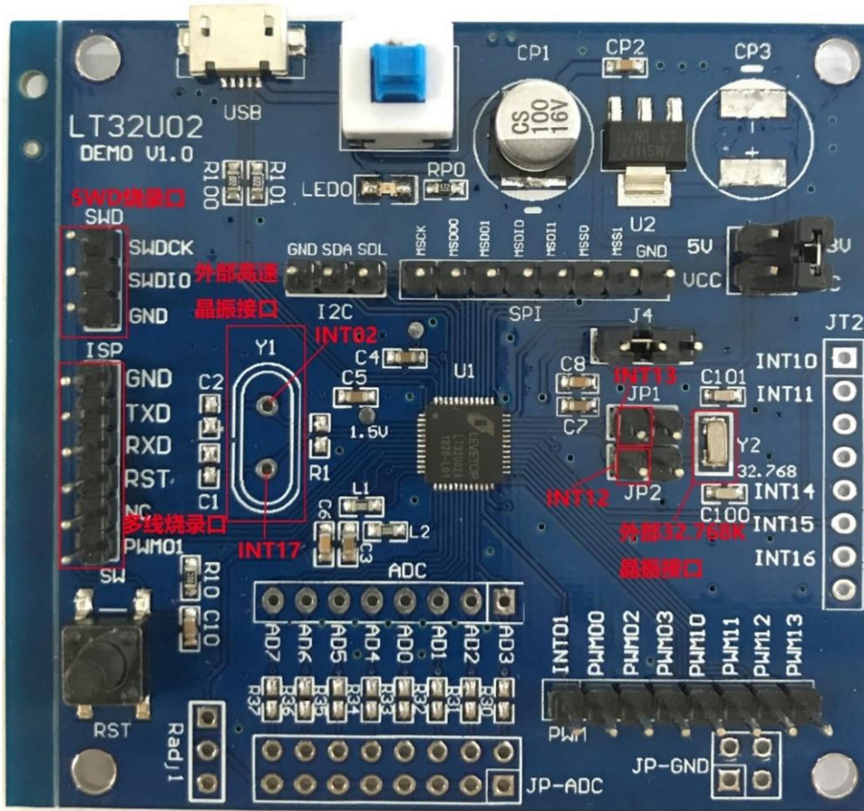


图 3–1: LT32U02 開發板

3.2 I2C

I2C 称为同步串行半双工通信总线，总线只用 SDA 和 SCL 两根线控制。总线必须由主机控制，主机产生串行时钟控制传输方向和产生起始与停止条件。SDA 线上的数据状态仅在 SCL 为低电平的期间才能改变。

该总线提供了标准模式、快速模式和高速模式的传输模式的选择，传输速率分别达到 100Kbits/s、400Kbits/s 和 3.4Mbits/s。支持从机 7 位寻址。I2C 通信协议由 6 个部分组成：开始位，数据源/宿主位，数据方向位，从机应答位，数据位，数据应答和停止位。I2C 时钟模式分为：时钟测试模式和正常模式。

- 时钟测试模式

  IPG_clock 为 72MHZ，I2C 正常运行时钟频率范围：140KHZ~1.47MHZ
  IPG_clock 为 36MHZ，I2C 正常运行时钟频率范围：70.1KHZ~1.44MHZ
  IPG_clock 为 18MHZ，I2C 正常运行时钟频率范围：35.0KHZ~1.80MHZ

©Levetop Semiconductor Co., Ltd.

■ 正常模式

IPG_clock 为 72MHZ，I2C 正常运行时钟频率范围：2.8KHZ~181.3KHZ
IPG_clock 为 36MHZ，I2C 正常运行时钟频率范围：1.4KHZ~ 90.7KHZ
IPG_clock 为 18MHZ，I2C 正常运行时钟频率范围：0.7KHZ~45.3KHZ

3.3 SPI

在 LT32U02 中，SPI 模块包含了双通道的接口，两个独立的接收缓冲区和两个发送缓冲区。SPI 称为同步串行半双工通信总线，主机可同时与两个 SPI 从设备进行通信，不过共享同一个的 SPICLK 端口。可选择的传输数据宽度有 8、16、24 和 32 位，使用双通道时可选择主导的通道。根据波特率，有固定模式和可变模式供选择，可变模式是在 SPI 通信期间波特率设置为可变，可对其设置成自己所需的波特率。

IPG_clock 为 72MHZ，SPI 正常运行时钟频率范围：549HZ~36MHZ
IPG_clock 为 36MHZ，SPI 正常运行时钟频率范围：274HZ~18MHZ
IPG_clock 为 18MHZ，SPI 正常运行时钟频率范围：137HZ~ 9MHZ

3.4 UART

UART 称为异步串行全双工总线，主要由波特率发生器、发送器和接收器组成，由 TX 和 RX 两根线来实现通信。传输帧的数据格式可选择 8 位或 9 位，波特率因子由 16 位的整数和 6 位的小数组成，同时支持红外接口，兼容 IrDA 协议。

IPG_clock 为 72MHZ，UART 正常运行波特率范围：4800~3090000
IPG_clock 为 36MHZ，UART 正常运行波特率范围：4800~1545000
IPG_clock 为 18MHZ，UART 正常运行波特率范围：4800~1167315

3.5 ADC

图 3-1 中，AD2 默认为 SWDIO 功能，要想作为 ADC 管脚需先禁止 SWDIO 功能。它具有多达 9 个通道，可以测量来自 8 个外部和 2 个内部信号源的信号。ADC 的分辨率可配置为 12 位、10 位、8 位或 6 位，设置为 12 位分辨率时转换时间可达到 1us，在低功耗模式下，ADC 仍能保持出色的转换性能。同时支持 DMA 的传输。

3.6 PWM

PWM 模块包含两组 PWM。1 个 PWM 中包含 2 个死区发生器和 4 个 PWM 定时器，每个定时器都可以用作计时器并独立发出中断。每个定时器都包含一个捕获通道，捕获和 PWM 共享一个定时器，如 PWM01 和 Capture 01 共享相同的中断。因此同一信道中的 PWM 功能和捕获功能不能同时使用。

©Levetop Semiconductor Co., Ltd.

3.7 EPORT

图 3-1 中，ADC 管脚处从左到右 INT0 组管脚依次对应为为 INT07(AD7)、INT06(AD6)、INT05(AD5)、INT04(AD4)、INT03(AD3)，该管脚默认为 ADC 功能，要使能 EPORT 功能需禁止对应的 ADC 管脚，其余的分别为 INT01、INT00(SWDCK)和 INT02(EXTAL)，要想使能 EPORT 功能也需禁止对应的复用功能。

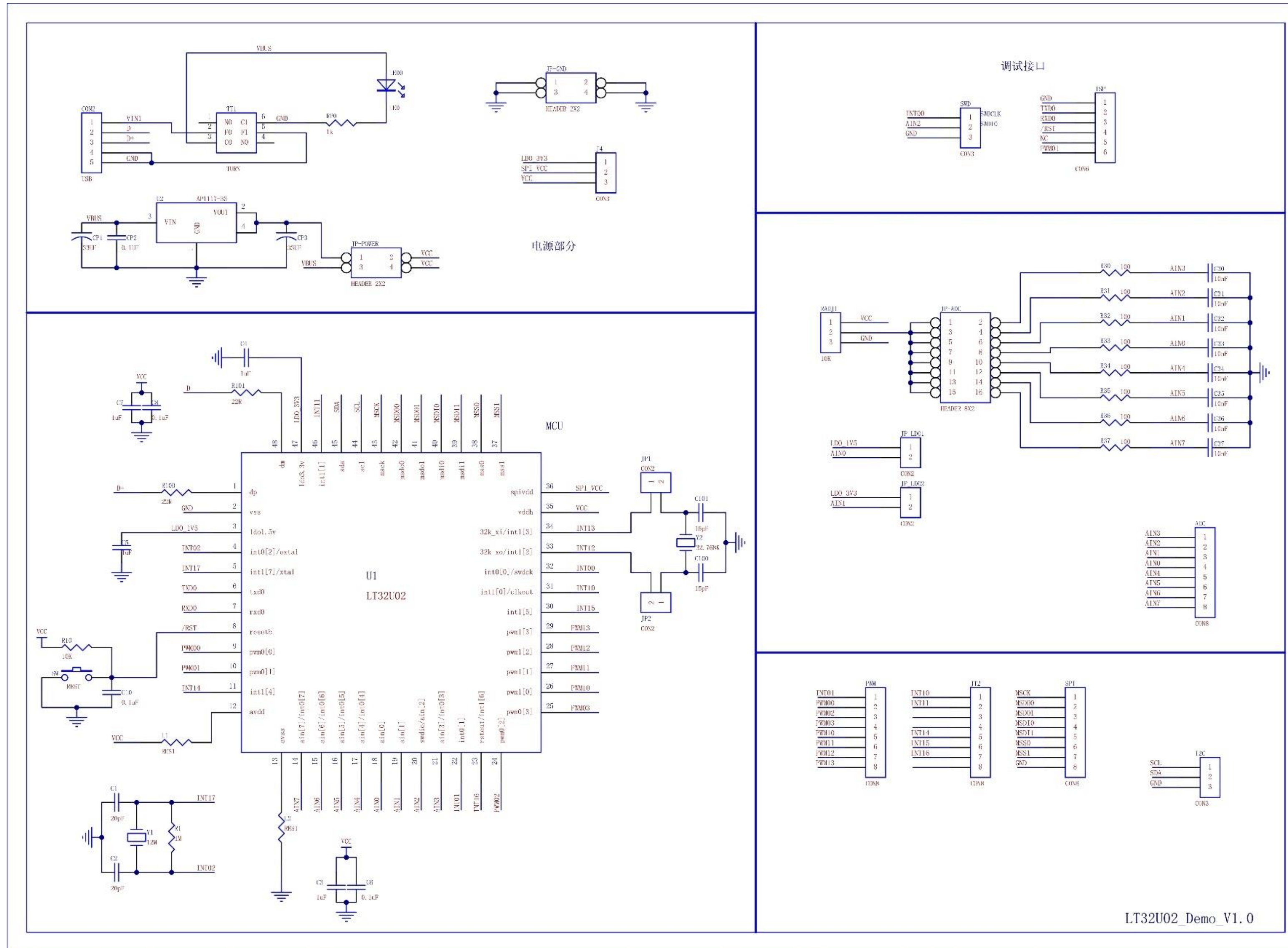
INT1 组管脚中 INT11、INT12、INT13、INT14 和 INT15 默认为 EPORT 功能，其余的 INT10(CLKOUT)、INT16(RSTOUT)、INT17(XTAL)要想使能 EPORT 功能需要禁止对应的复用功能。EPORT 功能不仅包含普通 IO 口的功能，也包含外部中断的功能。

3.8 其它

图 3-1 中，电源电路部分，注意选择给 VCC 提供的电压；与 AD2 引脚串联的 C31 电容需要去掉，否则可能会导致仿真的失败；需要用到外部 32.768K 晶振时要注意图 3-1 中的 JP1 和 JP2 的是否有直连，同时还要使能 32K_XO 和 32K_XI 功能。

©Levetop Semiconductor Co., Ltd.

3.9 开发板原理图



LT32U02_APNote / V1.2

4. 烧录说明

我司提供的 LT32U02 烧录器，可以实现多线和 SWD 两种烧录方式。

4.1 多线烧录

此方式不需要给 LT32U02 Demo 板提供电源，通过 5 条线来更新，请注意连接定义一一对应，其中烧录器中的 PWM00 的管脚是不需要连接的。

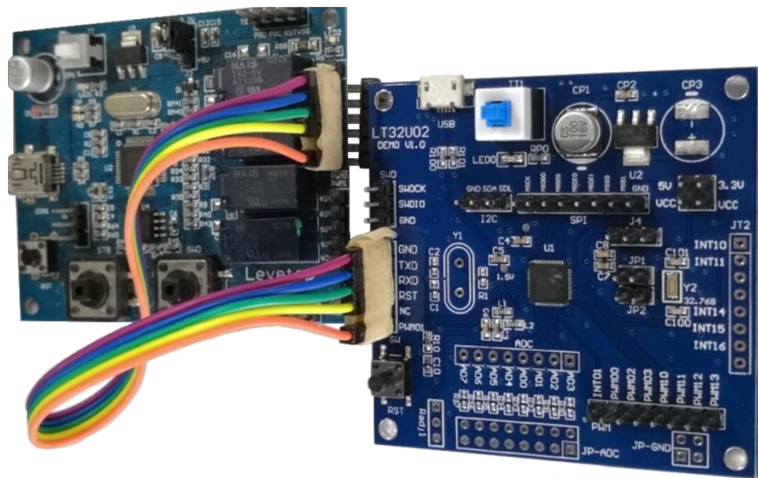


图 4–1

连接好硬件后，点击烧录软件 LT32xx_PMGER_V3.2.exe，正确连接后提示如下：

图 4–2

©Levetop Semiconductor Co., Ltd.

烧录口选择“多线烧录”，MCU 型号选择“32U02”，点击“导入文件”，选择需要下载的程序，如 LT32U02_Sample_ADC.bin。

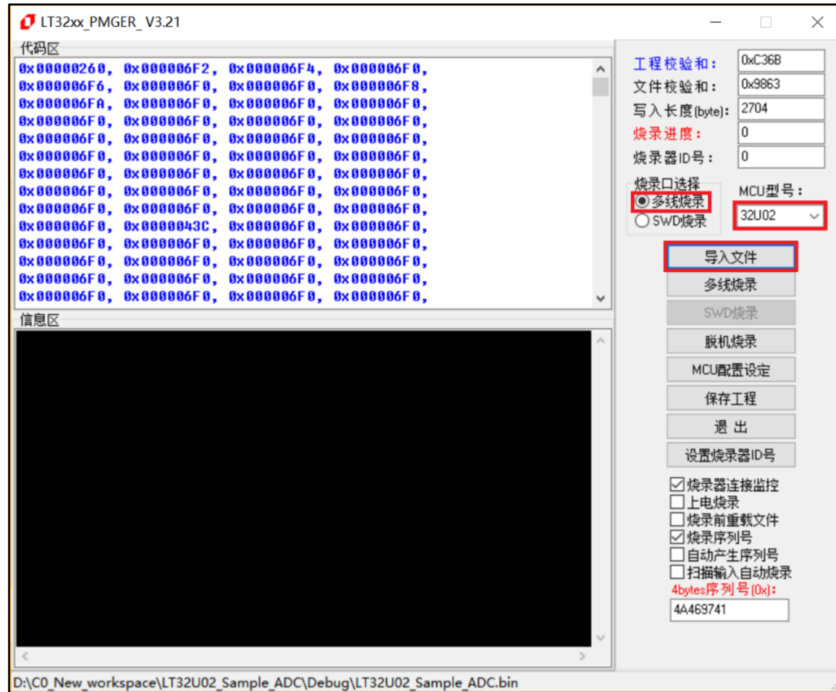


图 4–3

烧录前 LT32U02 Demo 板最好先断电，取消“上电烧录”，再点击“多线烧录”，完成烧录将出现下图：

©Levetop Semiconductor Co., Ltd.

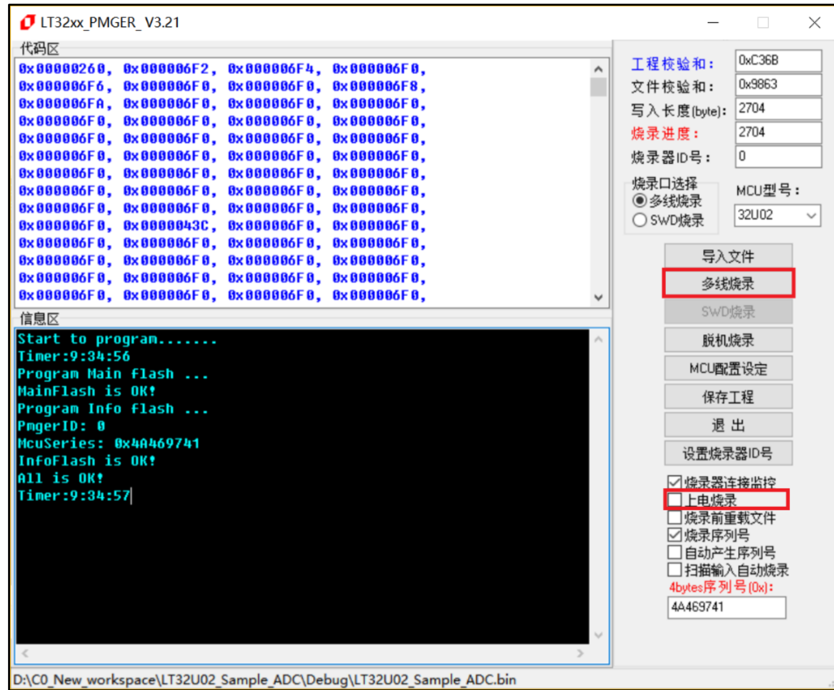


图 4–4

©Levetop Semiconductor Co., Ltd.

4.2 SWD 烧录

按如图三条线连接，注意采用此方式烧录，必须给 LT32U02 Demo 板提供电源。

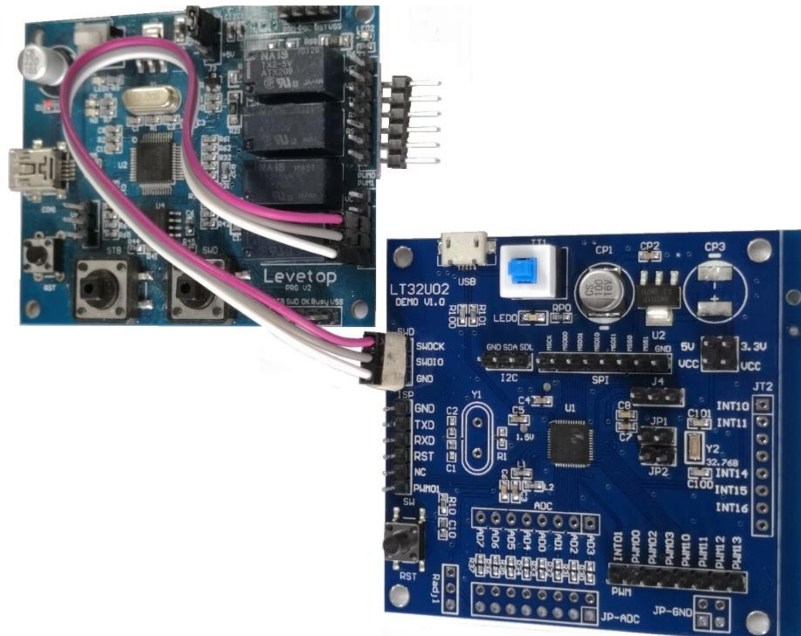


图 4-5

同多线烧录一样，只需选择 SWD 烧录，再点击 SWD 烧录即可。

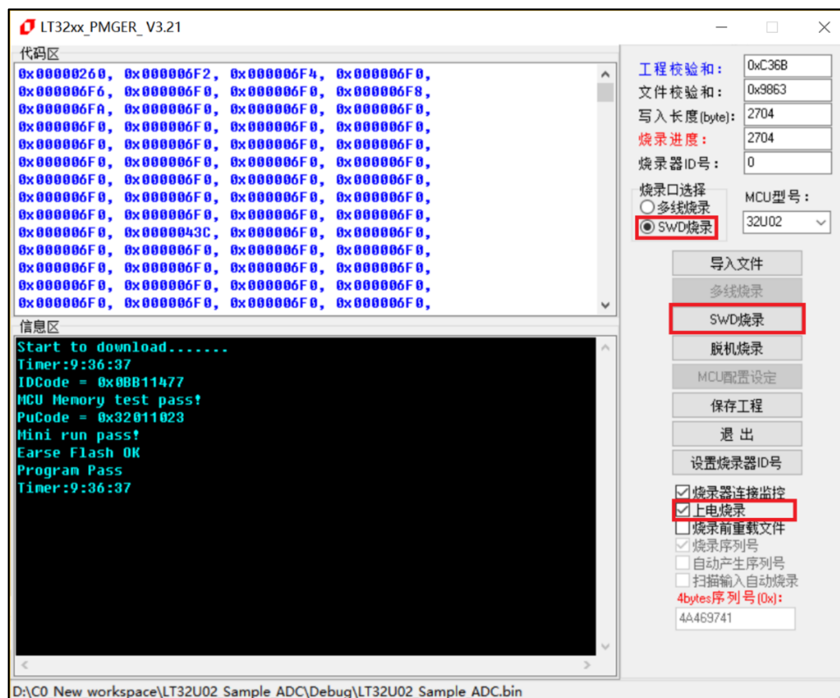


图 4-6

©Levetop Semiconductor Co., Ltd.

4.3 脱机烧录

点击“脱机烧录”后，烧录的程序将固化到烧录器，固化完成后，客户可选择“STB”或“SWD”按键通过相应的接口来实现一键下载。

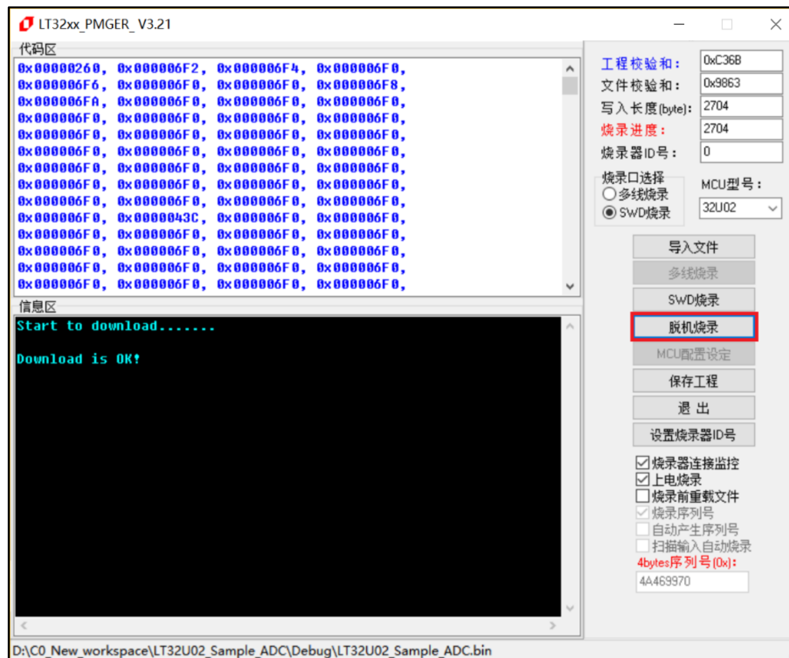


图 4–7

©Levetop Semiconductor Co., Ltd.

4.4 拓展功能

序列号产生有两种方式可选择：自动产生序列号和扫描枪扫码输入。

勾选了“自动产生序列号”，每次烧录时序列号都会变化。

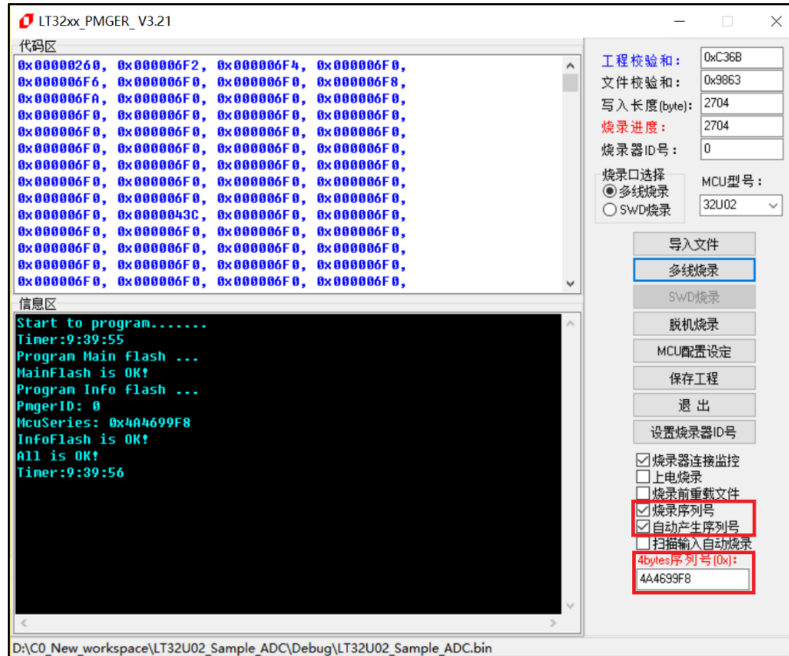


图 4–8

勾选了“扫描输入自动烧录”，使用扫描枪扫描对应的序列号产生的二维码或条形码，此时会将程序和序列号自动烧录进去。注意：扫描时需要选中“4bytes 序列号(0x)”输入框。

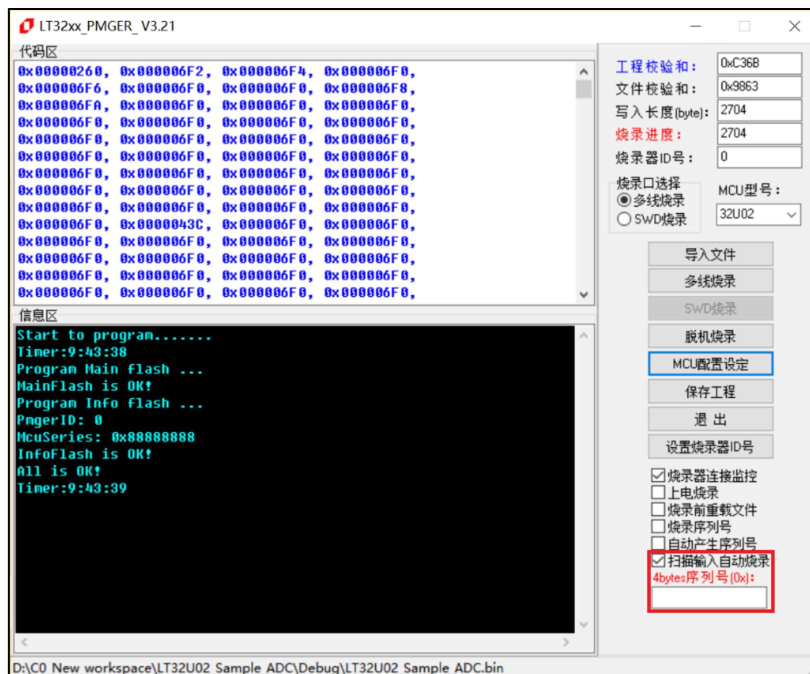


图 4–9

©Levetop Semiconductor Co., Ltd.

需要用到 MCU 配置功能时，点击“MCU 配置设定”

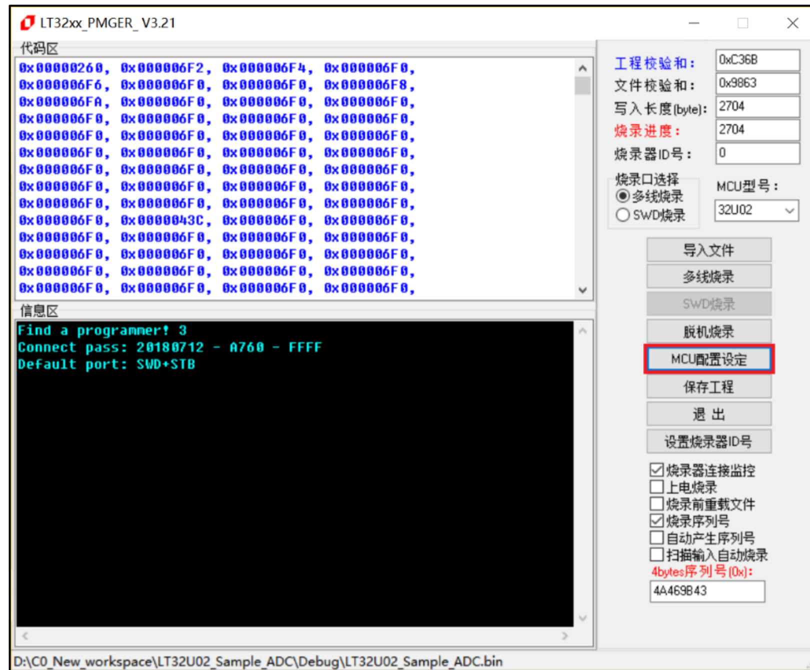


图 4–10

根据需要，选择禁止或使能相应的功能。

图 4–11

注意：如要使能或取消“低电复位使能”功能，重新烧录时需要上电，否则可能会出现烧录失败的情况。

©Levetop Semiconductor Co., Ltd.

若烧录与 USB 相关的程序代码时，烧录前最好先取消“烧录器连接监控”，以便正常烧录，如图 4-12

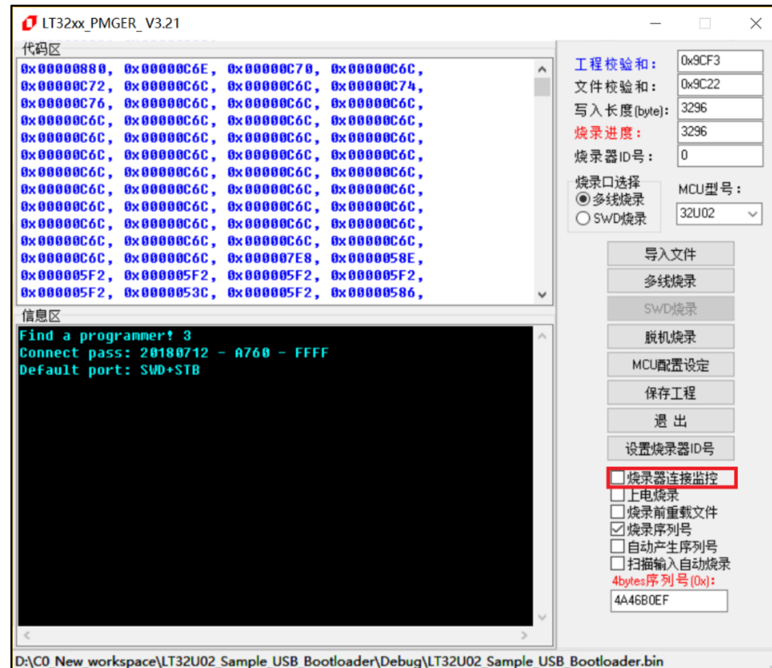


图 4-12

每个烧录器的 ID 号可根据实际情况而修改，详情可查阅“修改烧录器 ID 号说明.pdf”。

©Levetop Semiconductor Co., Ltd.

5. IDE 工具使用说明

新的 IDE 安装注意以下几点：

1) 安装路径不要用旧的 C0_IDE 文件夹，不能用旧版的 workspace；
2) 注意 C0file 拷贝方法；
3) Flash 烧录 LT32U02 程序要选择 LT32U02_Eflash_pmg_NewIDE.elf 文件。

5.1 软件安装

IDE 下载地址 http://www.levetop.cn/ch/download1-MCU.html
打开新版程序 setup_CCore_20180222.exe（win8，win10 最好用管理员权限运行），安装前建议关闭杀毒软件或 360 卫士，双击安装包程序进入安装界面:

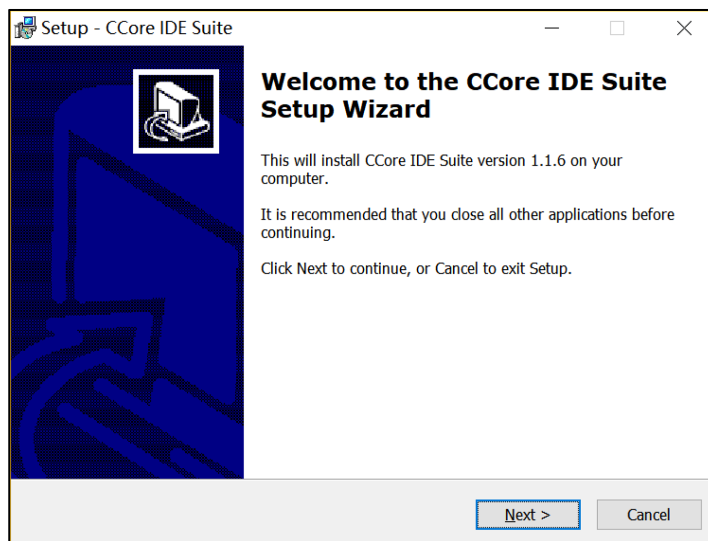


图 5–1

©Levetop Semiconductor Co., Ltd.

这里说明安装到 d:\C0_IDE 目录（其他目录也行，但不能中文路径），

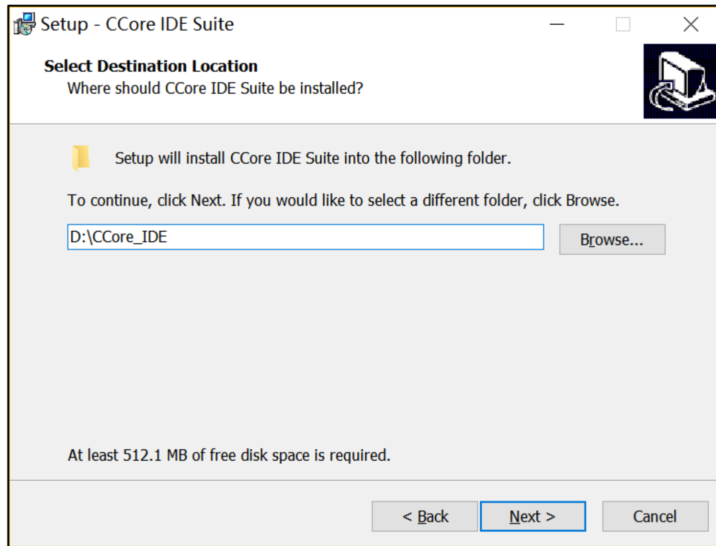


图 5–2

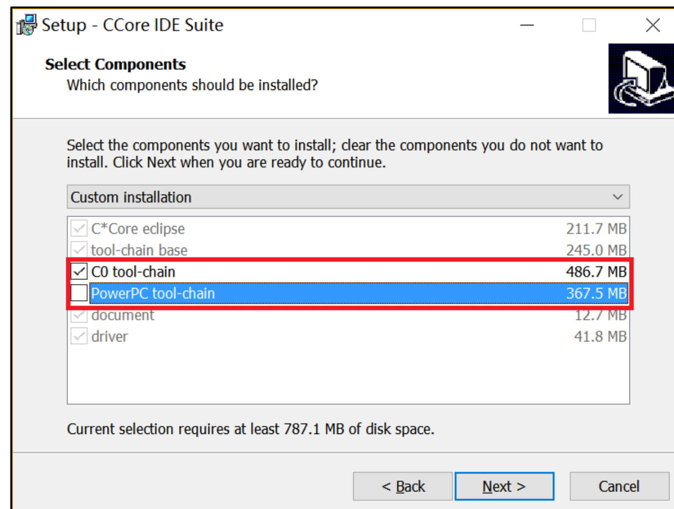


图 5–3

©Levetop Semiconductor Co., Ltd.

接着按默认选项安装，安装完毕。将“\新版 IDE 说明资料\C0File”文件夹替换“\CCore_IDE\ccore-eclipse\C0File”文件夹,注意 C0File 文件夹在\CCore_IDE\ccore-eclipse 目录下，该文件夹内容如下，

图 5–4

提示：C0File 文件夹内容有误，会导致 IDE 创建新工程有问题。

第一次打开 IDE 工具，可以用默认文件夹，也可以创建一个新的 workspace 文件夹，需要注意的是不能选择旧版原有的 workspace 文件夹，这里创建一个新的文件夹 C0_New_workspace（可自由命名，只要不是选择旧版 workspace 文件夹即可），如下图：

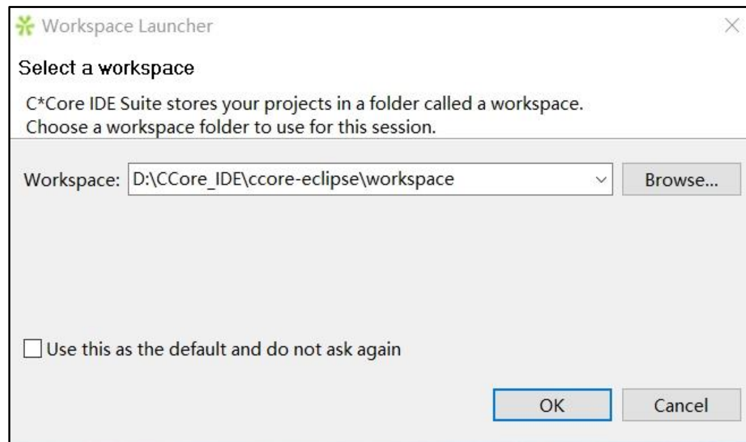


图 5–5

©Levetop Semiconductor Co., Ltd.

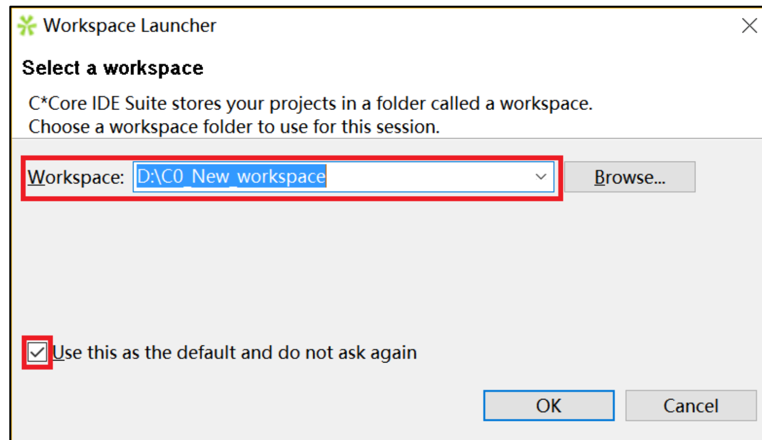


图 5-6

按“OK”进入界面，这时 D 盘下已经有了 C0_New_workspace 这个文件夹，在 C0_New_workspace 文件夹里创建 eFlash 文件夹，然后将 LT32U02_Eflash_pmg_NewIDE.elf 文件拷贝到这个目录下。

©Levetop Semiconductor Co., Ltd.

5.2 Stlink 驱动安装

第一次使用需要安装 stlink 驱动，执行 stlink-dirver 中的 st-link_v2_usbdriver.exe，安装成功后。在设备管理器中可以检查到 stlink 设备。（win10 用 STM32 ST-LINK Utility_v3.4.0 安装 stlink 驱动）

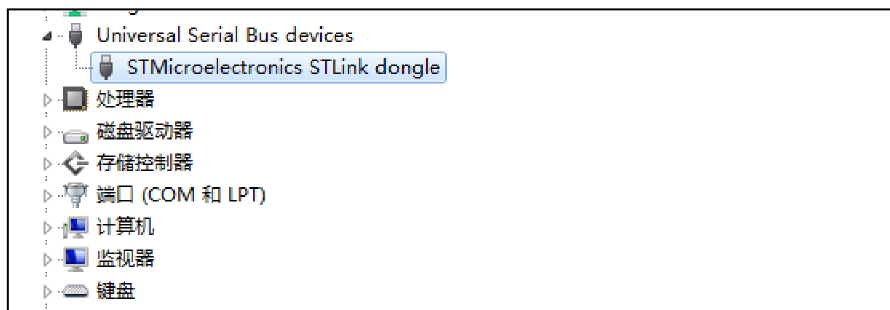


图 5–7

驱动安装成功后，建议更新 stlink 的驱动到 V23。更新程序在文件夹 CCore_IDE\stlink-driver\STLinkUpgradeV2J23 中。

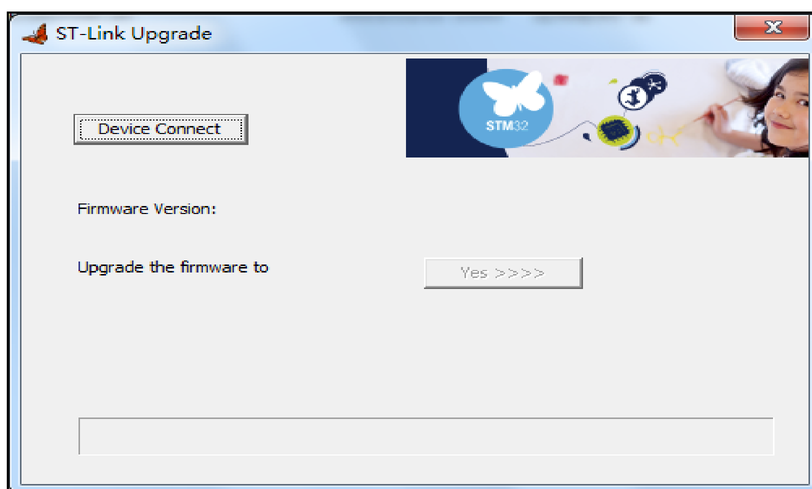


图 5–8

©Levetop Semiconductor Co., Ltd.

5.3 创建工程及配置

打开 CCore_IDE.exe 之后可以创建新工程。

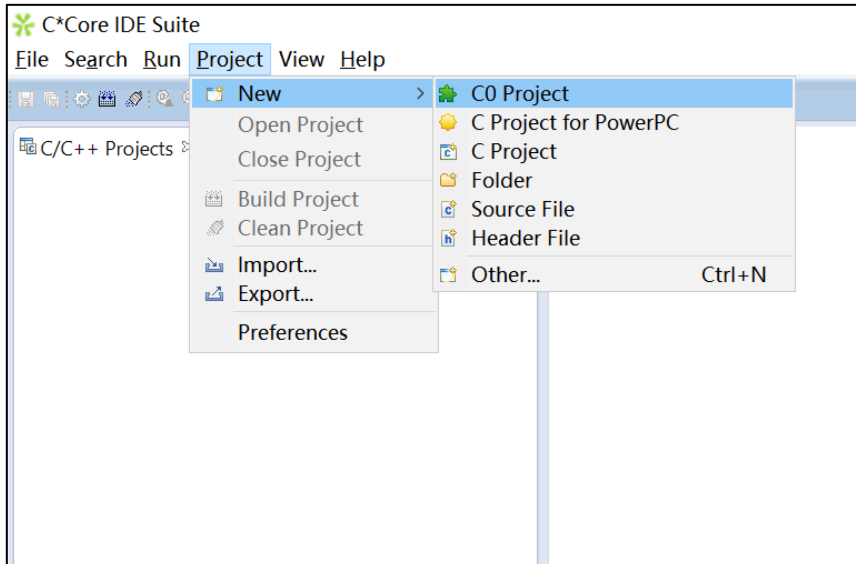


图 5–9

在 project 目录下选择新建 C0 Project

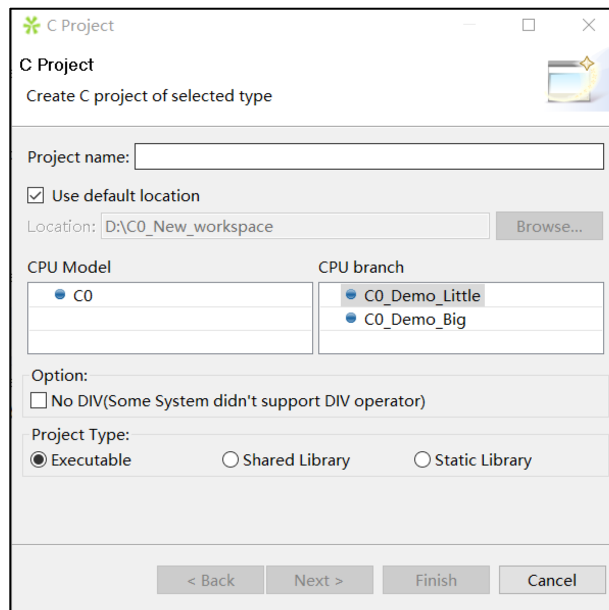


图 5–10

©Levetop Semiconductor Co., Ltd.

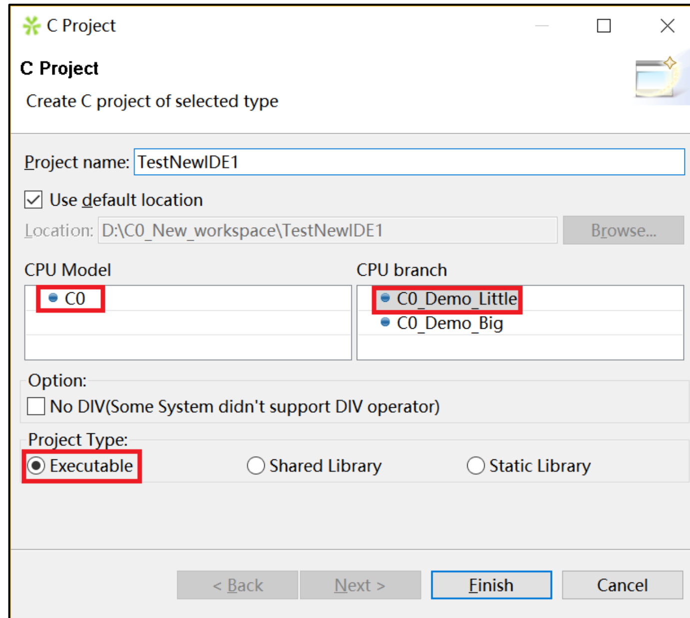


图 5-11

点击“Finish”按键创建新的工程 testNewIDE1。

工程下的 linkmap 文件里 3 个 rom 代表新工程默认运行在 Flash 上，打开 linkmap 如下图所示：

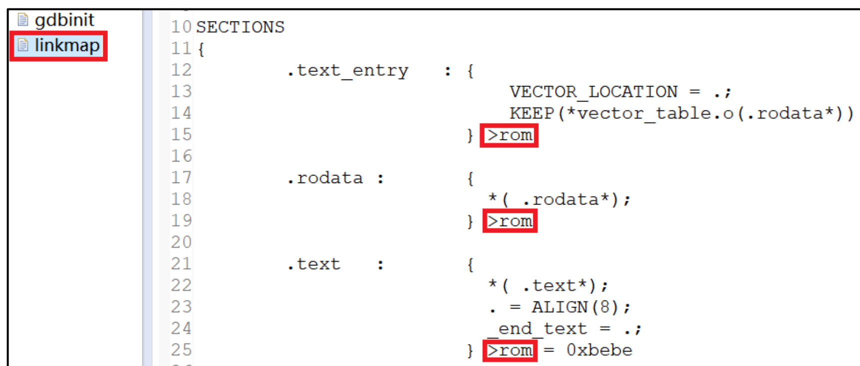


图 5-12

©Levetop Semiconductor Co., Ltd.

如果想运行在 SRAM 里，改成如下：

```
gdbinit
linkmap

10 SECTIONS
11 {
12         .text_entry    : {
13                     VECTOR_LOCATION = .;
14                     KEEP(*vector_table.o(.rodata*));
15                 } >ram
16
17         .rodata  :          {
18                     *( .rodata*);
19                 } >ram
20
21         .text    :          {
22                     *( .text*);
23                     . = ALIGN(8);
24                     _end_text = .;
25                 } >ram = 0xbebe
```

图 5-13

在 SRAM 里运行的程序，可以直接仿真。按下图所示连接好之后，首先运行 SWD 程序，

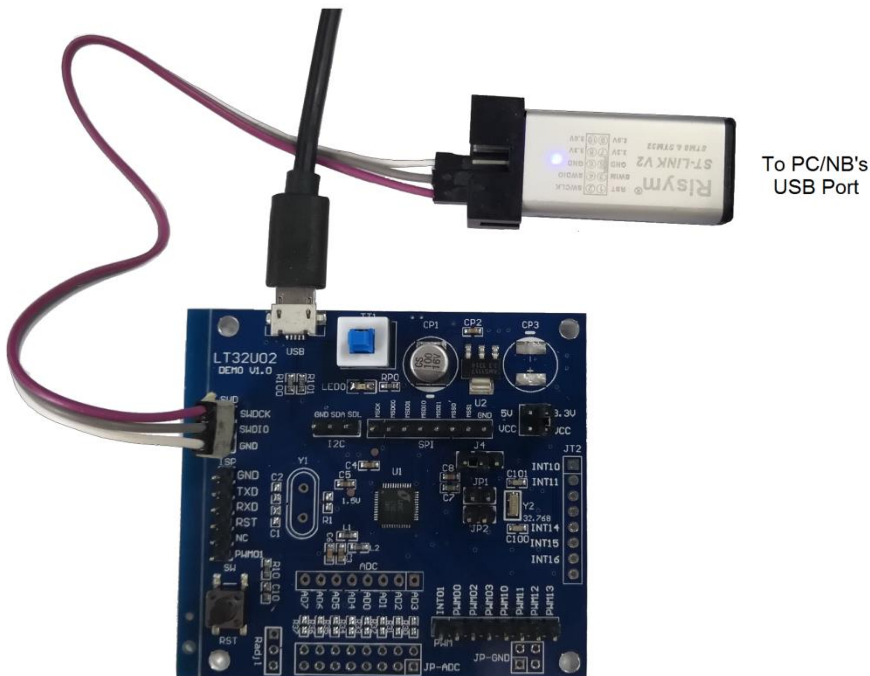


图 5-14

©Levetop Semiconductor Co., Ltd.

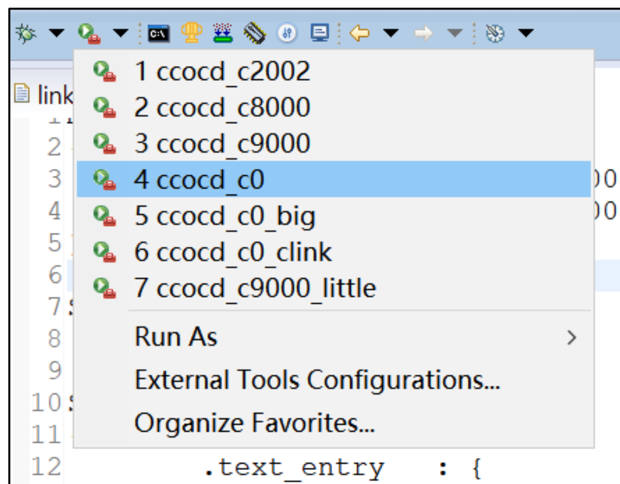


图 5–15

©Levetop Semiconductor Co., Ltd.

SWD 找到 MCU，显示如下：

```
Problems  Console  Properties
ccocd_c0 [Program] d:\CCore_IDE\tool-chain\gdb-server\ccocd_swd.exe
C*CORE On-Chip Debugger 0.9.7 (2017-07-18-10:58)
Licensed under GNU GPL v2
adapter speed: 4000 kHz
debug_level: 2
srst_only separate srst_gates_jtag srst_open_drain connect_deassert_srst
Info : Debug adapter  support 'hla_swd'

C*core-C0.cpu
Info : core.c clock speed 4000 kHz
Info : STLINK v2 JTAG v23 API v2 SWIM v4 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 3.220053
Info : Target versin: 2 jtag : 23

Info : C*core-C0.cpu: hardware has 4 breakpoints, 0 watchpoints
Info : adapter_halt
Warn : target was in unknown state when halt was requested
Info : >>C0 little endian support
Info : >>>>>>>>>>>>>>SWD examine OK,psr: 0x80000140
```

图 5-16

新工程还没有仿真连接，需要创建一下，先要“Build Project”编译一下工程

```
(no launch history)
Debug As
Debug Configurations...
Organize Favorites...
5 }
6
7 STACK_LOCATION = 0x00800ffc ;
8
9
```

图 5-17

©Levetop Semiconductor Co., Ltd.

点击 “debug Configuration”，进入设置界面：
这里还没有 testNewIDE1 Debug,鼠标光标在 C*Core C0 Application 上双击一下，

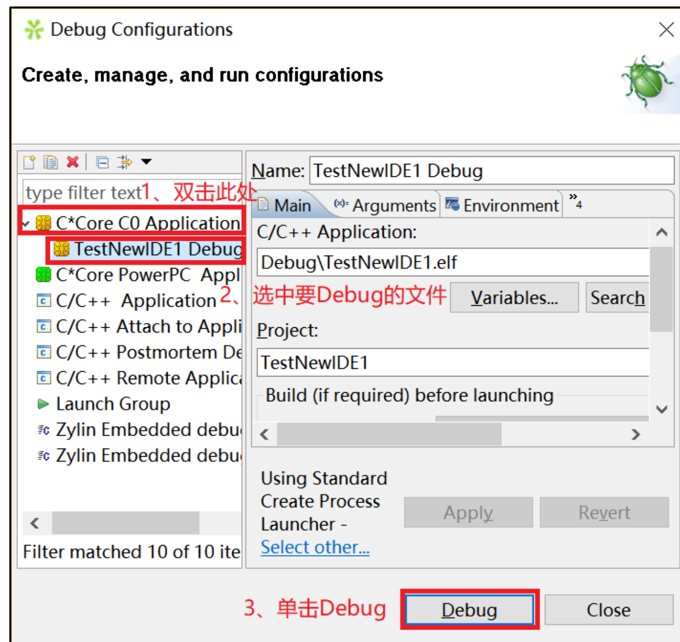


图 5–18

如果没有生成 testNewIDE1 Debug，就退出这个界面，重新编译一下 TestNewIDE 工程。。
这时可以点击 “Debug” 按键进行仿真。

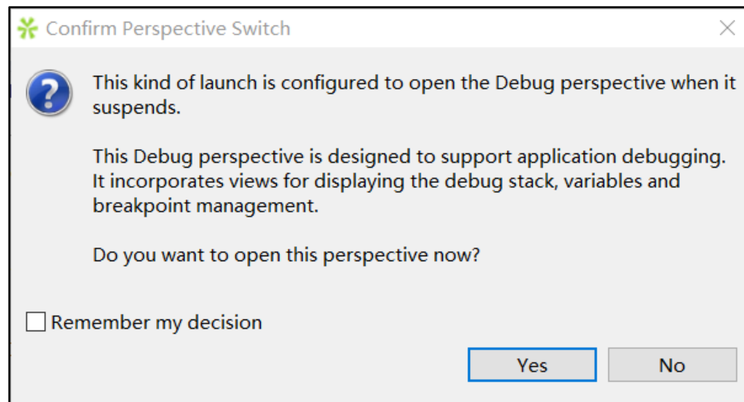


图 5–19

©Levetop Semiconductor Co., Ltd.

这时选择 “Yes”，之后就进入了仿真界面。

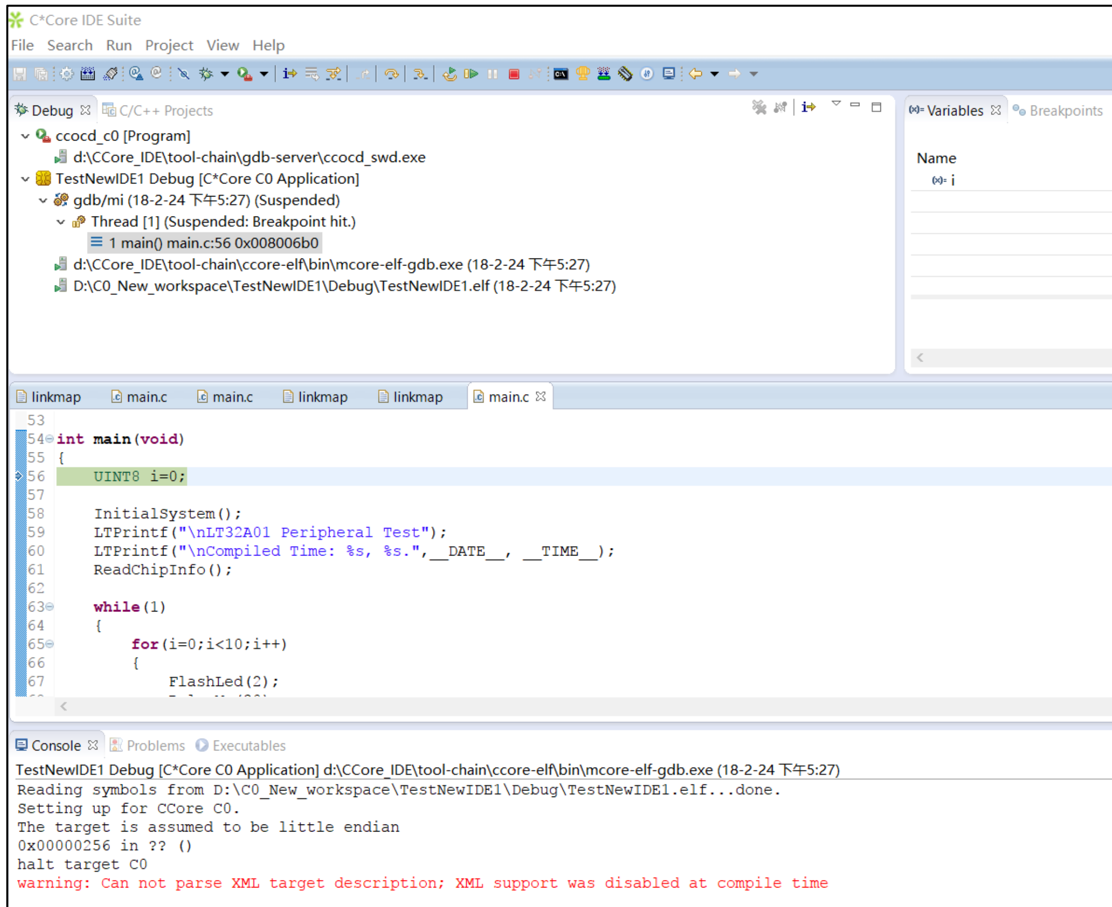


图 5–20

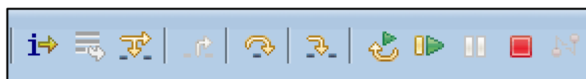


图 5–21

此时就能在 SRAM 中进行调试仿真。

©Levetop Semiconductor Co., Ltd.

如果想程序运行在 Flash 上以及仿真，linkmap 设置如下，编译

```
gdbinit
linkmap
10 SECTIONS
11 {
12         .text_entry   : {
13                          VECTOR_LOCATION = .;
14                          KEEP(*vector_table.o(.rodata*));
15                       } >rom
16
17         .rodata :       {
18                          *( .rodata*);
19                       } >rom
20
21         .text   :       {
22                          *( .text*);
23                          . = ALIGN(8);
24                          _end_text = .;
25                       } >rom = 0xbebe
```

图 5–22

同样，先要运行 SWD，

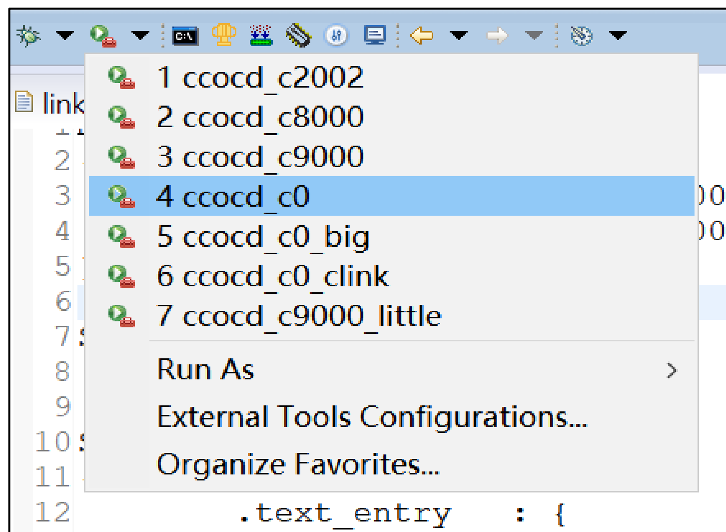
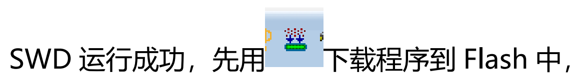


图 5–23

SWD 运行成功，先用下载程序到 Flash 中，

©Levetop Semiconductor Co., Ltd.

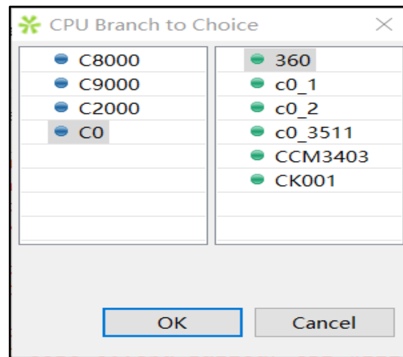


图 5–24

点击“OK”按钮，进入配置界面，按如下配置：

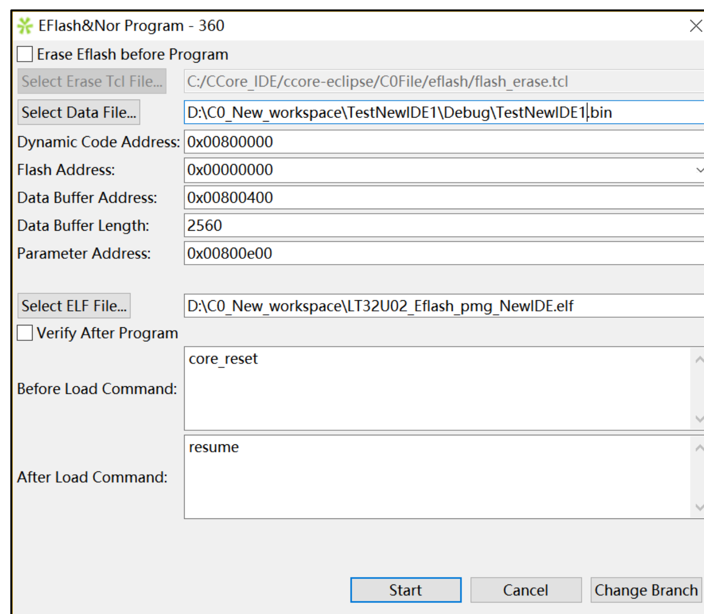


图 5–25

点击“Start”开始烧录，如下图：

©Levetop Semiconductor Co., Ltd.

```
ccocd_c0 [Program] d:\CCore_IDE\tool-chain\gdb-server\ccocd_swd.exe
Polling target C*core-C0.cpu failed, GDB will be halted. Polling again in 100ms
Info : Previous state query failed, trying to reconnect
Error: jtag status contains invalid mode value - communication failure
Polling target C*core-C0.cpu failed, GDB will be halted. Polling again in 300ms
Info : Previous state query failed, trying to reconnect
Polling target C*core-C0.cpu succeeded again, trying to reexamine
Info : C*core-C0.cpu: hardware has 4 breakpoints, 0 watchpoints
Info : adapter_halt
Info : >>C0 little endian support
Info : >>>>>>>>>>>>>>SWD examine OK,psr: 0x80000140

Info : adapter_halt
Info : dropped 'gdb' connection
Info : accepting 'telnet' connection on tcp/4444
background polling: on
TAP: C*core-C0.cpu (enabled)
target state: halted
target halted due to debug-request PSR 0x80000141 PC 0x800738
Info : adapter_halt
Info : adapter_halt
Info : adapter_halt
Info : core_reset -- PSR: 0x80000140,PC: 0x140

976 bytes written at address 0x00800000
downloaded 976 bytes in 0.007018s (135.811 KiB/s)
Info : buffer addr :0x800400, flash_addr:0x0, data_len :0x948,param_addr: 0x800e00

Info : Programing..., addr=00000000, length=2376
Info : adapter_halt
Info : Program data end! set pc: 0x1b0
```

图 5–26

烧录成功，程序应该已经运行。如果想进行仿真，再参考 SRAM 仿真部分。

5.4 导入

选择已有工程，旧的工程需要拷贝到新的 workspace（C0_New_workspace）里，导入即可，Project-->Import 选择 Existing Projects into Workspace，然后从 Browse 导入，如下如所示：

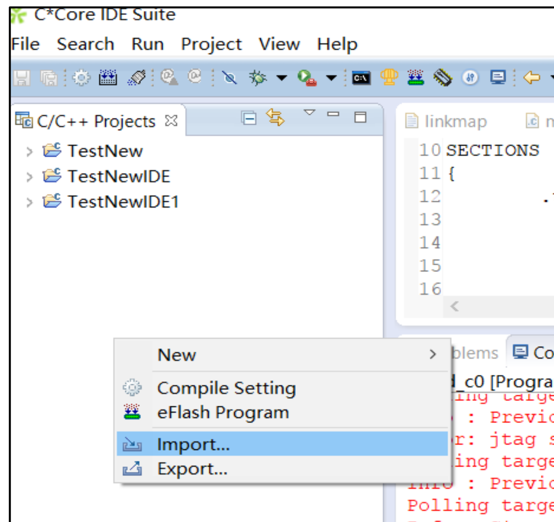


图 5–27

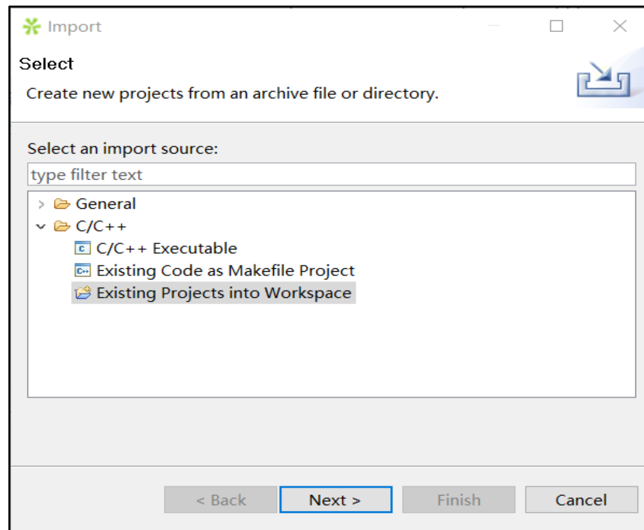


图 5–28

©Levetop Semiconductor Co., Ltd.

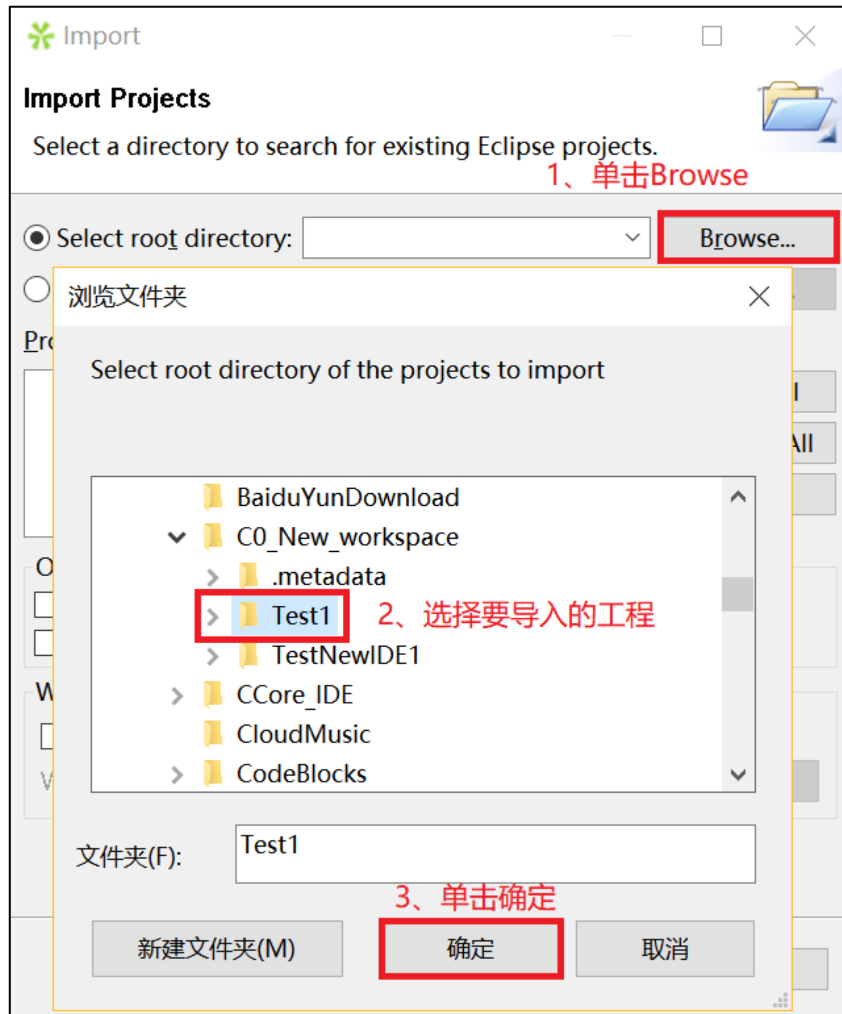


图 5–29

如果需要仿真和下载，参考新工程的仿真和下载。

©Levetop Semiconductor Co., Ltd.

5.5 工程配置

注意：工程默认为小端配置，我们使用的也是小端配置，请不要配置为大端模式，否则会造成寻址错误。
选择编译器优化等级：

- Tool Settings　Cross GCC Compiler　Optiization 选择优化等级，一般选择-O1,-Os 为最高优化等级。
- Tool Settings　Cross G++ Compiler　Optiization 选择优化等级，一般选择-O1,-Os 为最高优化等级。

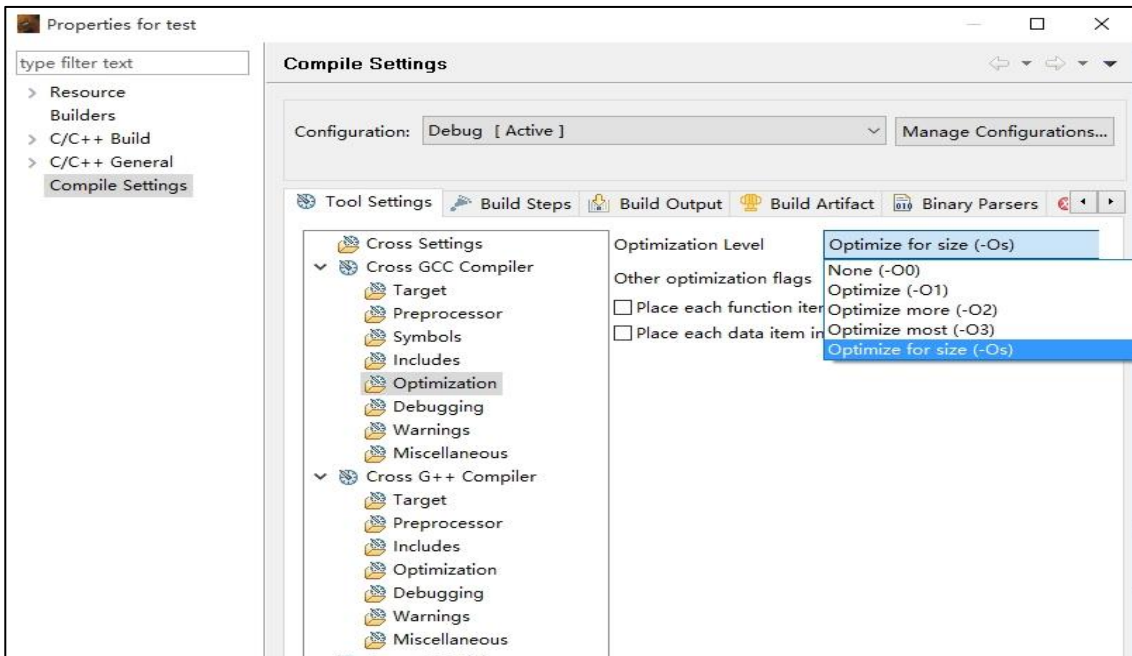


图 5–30

©Levetop Semiconductor Co., Ltd.

5.6 调试注意事项

进入调试界面如下：

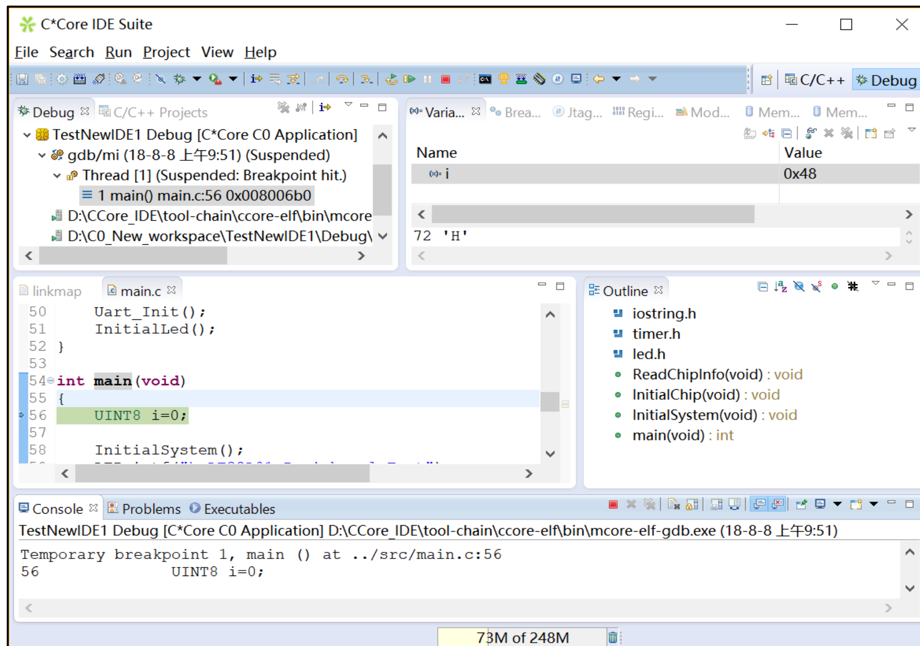


图 5-31

注意：关闭仿真时需要先暂停 gdb/mi，再选择 Terminate/Disconnect ALL 关闭所有链接。

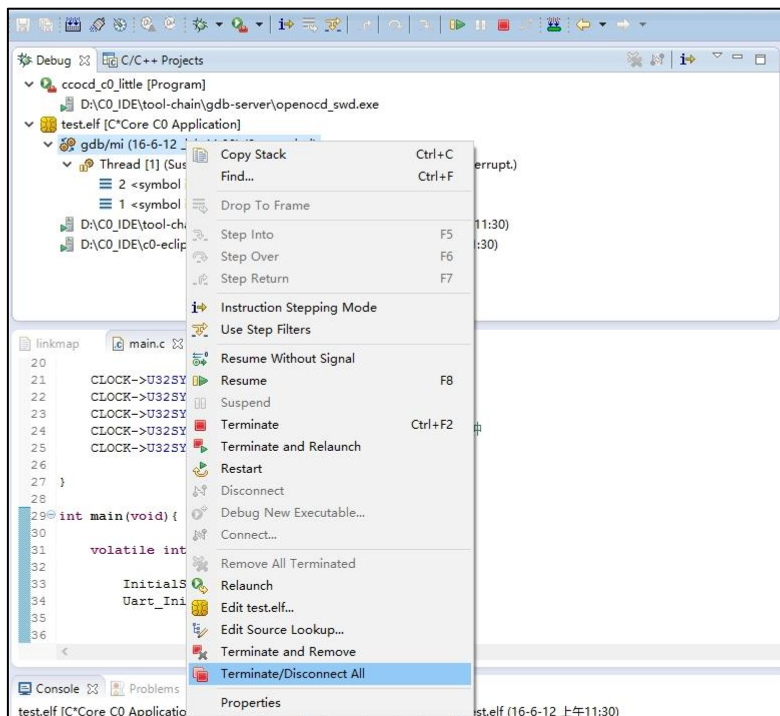


图 5-32

©Levetop Semiconductor Co., Ltd.

5.7 Telnet 工具使用

调试服务程序除了提供 GDB 访问接口外，还提供了 telnet 访问接口。可以在 cmd console 中 输入命令：telnet localhost 4444。

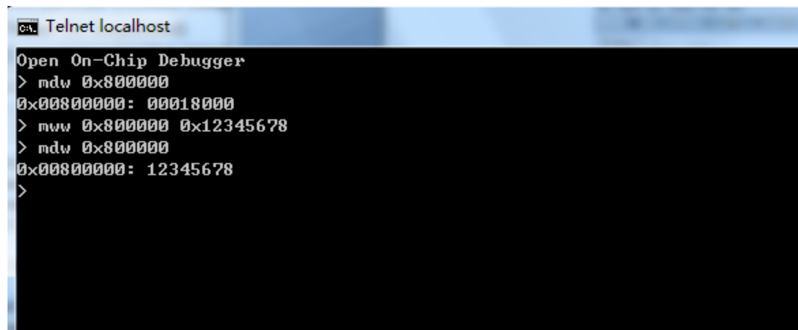


图 5-33

eclipse 也集成了图形化的操作接口。可以通过图形化来访问寄存器、内存、flash 等操作。

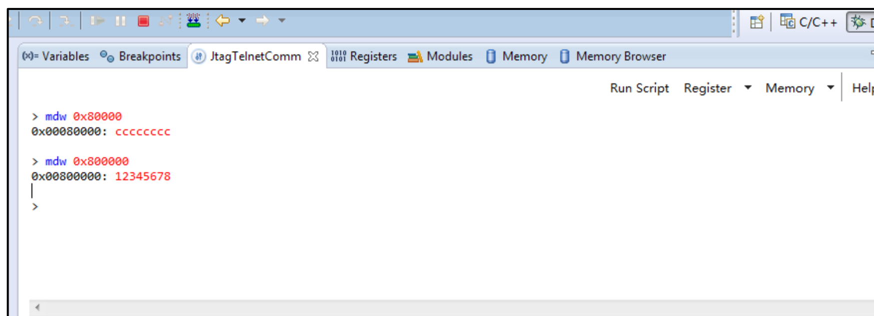


图 5-34

©Levetop Semiconductor Co., Ltd.

5.8 常见的调试命令和调试菜单

■ 步进、步越、步出：

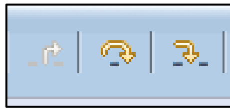


图 5-35

■ 选择汇编单步和程序单步，汇编单步在反汇编窗口执行：

图 5-36

■ 程序全速执行、暂停、终止：

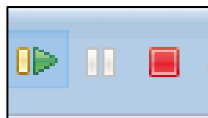


图 5-37

■ 设置断点（在源代码或反汇编文件左侧双击）：

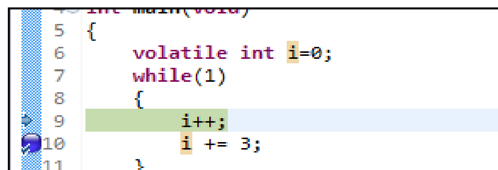


图 5-38

©Levetop Semiconductor Co., Ltd.

■ 手动修改程序执行序列（在调试位置右键）：

Run to Line： 运行到当前位置
Move to Line: 设置运行指针到当前行
Resume At Line: 从当前位置执行程序

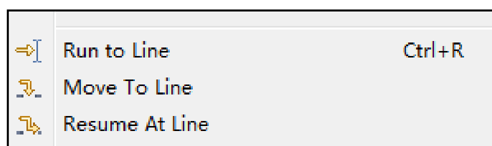


图 5-39

■ 局部变量窗口：

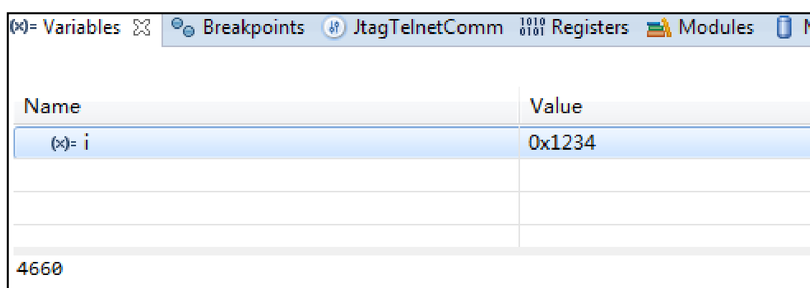


图 5-40

■ 断点管理窗口：

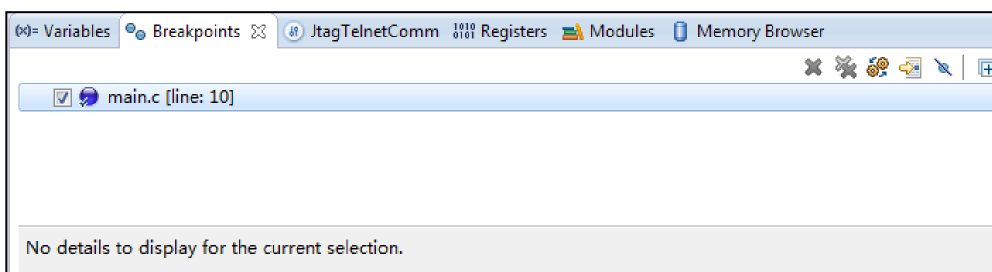


图 5-41

©Levetop Semiconductor Co., Ltd.

■ 寄存器观察窗口：

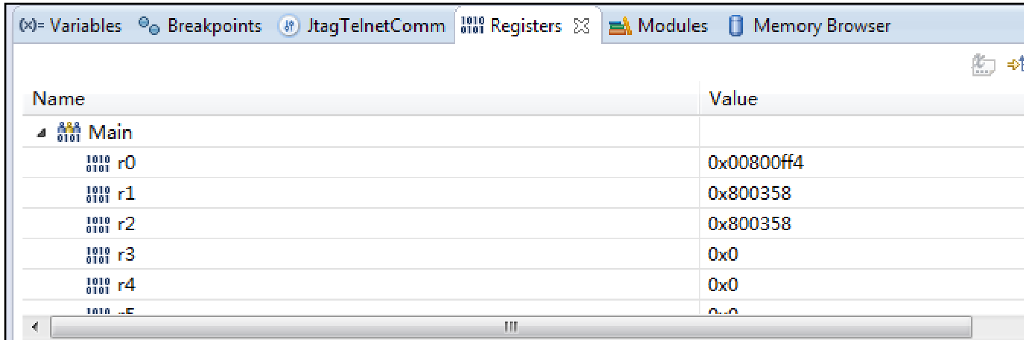


图 5–42

■ 内存观察窗口：

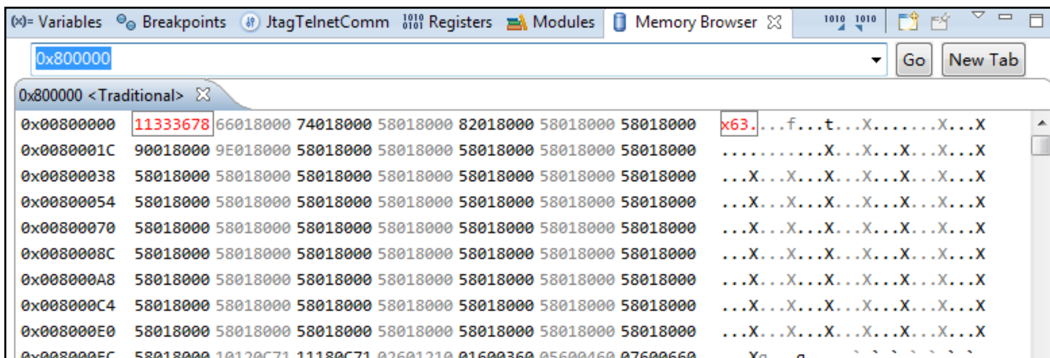


图 5–43

■ 调试服务程序和程序进程调用窗口：

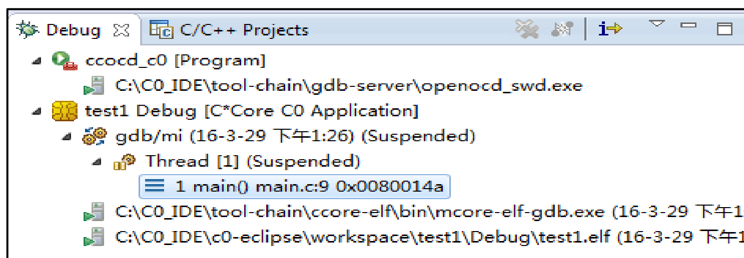


图 5–44

©Levetop Semiconductor Co., Ltd.

5.9 常见问题及原因

(1) 调试服务程序打开失败：

```
Console ☒ Problems Executables
<terminated> ccocd_c0 [Program] C:\C0_IDE\tool-chain\gdb-server\openocd_swd.exe
Open On-Chip Debugger 0.9.0-dev-00223-g1567cae-dirty (2016-03-23-10:57)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.sourceforge.net/doc/doxygen/bugs.html
adapter speed: 4000 kHz
srst_only separate srst_gates_jtag srst_open_drain connect_deassert_srst
Info : Debug adapter  support 'hla_swd'

Info : The selected transport took over low-level target control. The results might differ compared to
C*core-C0.cpu
Info : clock speed 4000 kHz
Info : STLINK v2 JTAG v16 API v2 SWIM v4 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 1.596832
Error: init mode failed
in procedure 'init'
in procedure 'ocd_bouncer'
in procedure 'transport'
in procedure 'init'
```

图 5–45

可能原因：

- 开发板未上电；
- SWD 连线错误；
- 系统异常；
- Stlink 连接异常，重新插拔 stlink。

(2) 连接中断

```
Console ☒ Problems Executables
ccocd_c0 [Program] C:\C0_IDE\tool-chain\gdb-server\openocd_swd.exe
Info : clock speed 4000 kHz
Info : STLINK v2 JTAG v16 API v2 SWIM v4 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 4.967700
Info : C*core-C0.cpu: hardware has 4 breakpoints, 0 watchpoints
Info : adapter_halt
Warn : target was in unknown state when halt was requested
Info : >>>>>>>>>>>>>>SWD examine OK,psr: 0x40010080

Info : >>C0 big endian support
Error: jtag status contains invalid mode value - communication failure
Polling target C*core-C0.cpu failed, GDB will be halted. Polling again in 100ms
Info : Previous state query failed, trying to reconnect
Error: jtag status contains invalid mode value - communication failure
Polling target C*core-C0.cpu failed, GDB will be halted. Polling again in 300ms
Info : Previous state query failed, trying to reconnect
Error: jtag status contains invalid mode value - communication failure
Polling target C*core-C0.cpu failed, GDB will be halted. Polling again in 700ms
```

图 5–46

可能原因：

- 开发板断电；
- 系统异常，无法连接 CPU。

©Levetop Semiconductor Co., Ltd.

6. 版本记录

表 6-1

版 别	发 布 日 期	改 版 说 明
V1.0	2018/08/17	Preliminary version (初版)
V1.1	2018/08/30	配合新版 V3.2 烧录软件(LT32xx_PMGER_V3.2.exe) 修改第四章（烧录说明）
V1.2	2022/05/30	移除 LT32A02 信息

7. 版权说明

本文件之版权属于 深圳市乐升半导体 所有，若需要复制或复印请事先得到 乐升半导体 的许可。本文件记载之信息虽然都有经过校对，但是 乐升半导体 对文件使用说明的规格不承担任何责任，文件内提到的应用程序仅用于参考，乐升半导体 不保证此类应用程序不需要进一步修改。乐升半导体 保留在不事先通知的情况下更改其产品规格或文件的权利。有关最新产品信息，请访问我们的网站 Http://www.levetop.cn 。