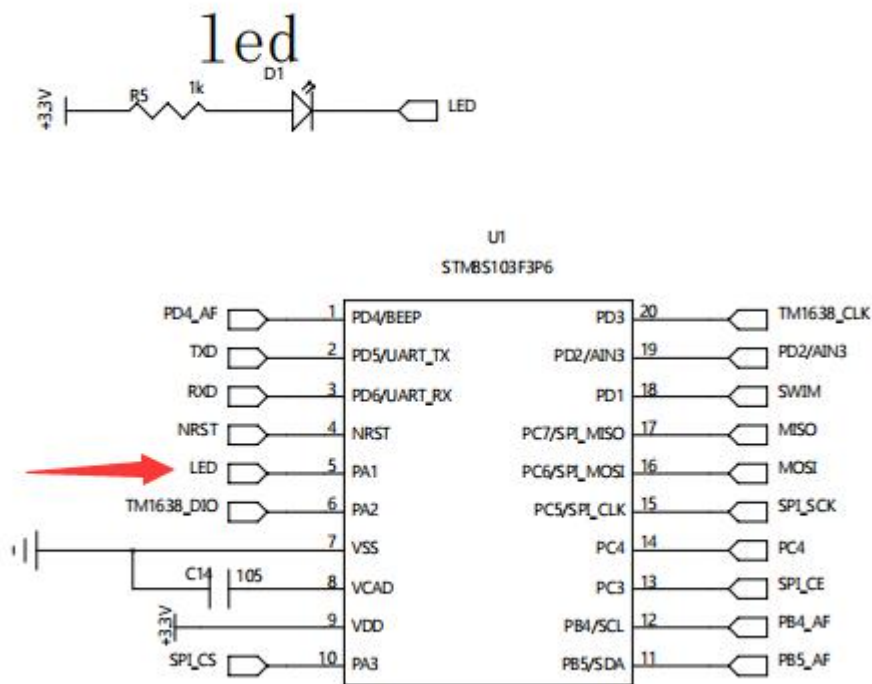


GPIO 之点亮 led

实验目的：

- 1, 学习编写一个最简单的点亮 led 程序
- 2, 学习使用 stvd 的 debug 调试程序。

1, 查看原理图，根据原理图可知：IO 口 PA1 输出低电平可点亮 led，输出高电平则 led 熄灭。



2, 本实验用到 stm8 的通用 io 口，所以需要用到库文件 `stm8s_gpio.c` 中的库函数。在 stvd 中将库文件 `stm8s_gpio.c` 添加进工程的 `lib/src` 目录中，再添加对应的 `stm8s_gpio.h` 文件到 `lib/inc`,

3, 根据设计思路敲代码。

主要代码如下：

```

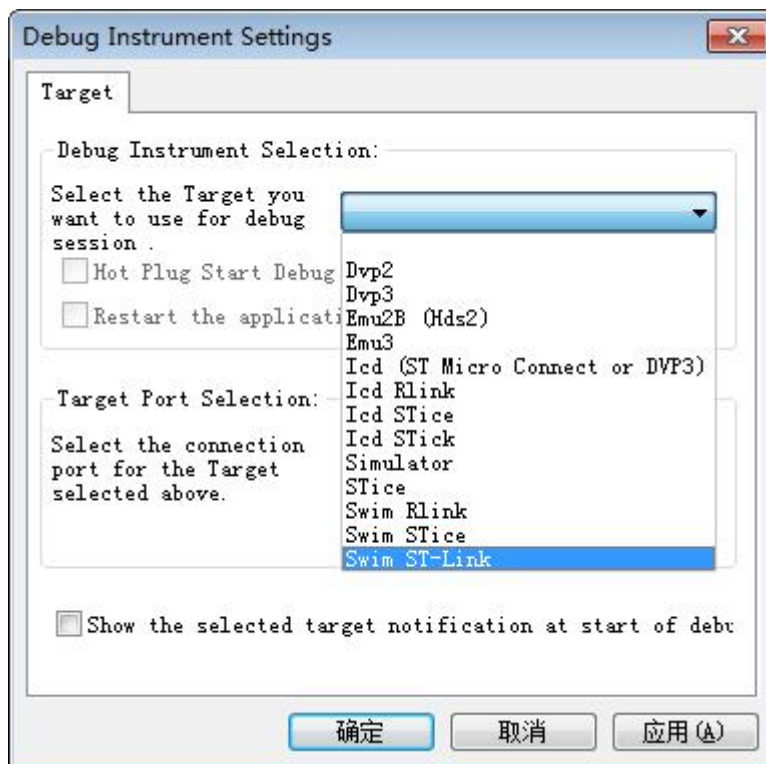
/**
 * author : tianyx
 * email  : zzztyx55@sina.com
 * qq     : 609421258
 * github : https://github.com/zzztyx55
 */
void main(void)
{
    // 初始化 PA1 为推挽输出 // 库文件中提供的函数
    GPIO_DeInit(GPIOA);
    GPIO_Init(GPIOA, GPIO_PIN_1, GPIO_MODE_OUT_PP_HIGH_SLOW);
    // 控制 PA1 输出低电平 // 库文件中提供的函数
    GPIO_WriteLow(GPIOA, GPIO_PIN_1);
    /* Infinite loop */
    while (1);
}

```


编译工程，通过。

4，在开发板上跑一下

4.1 需要先对 stvd 进行一些配置， 打开菜单 “Debug instrument” 选择 “Target Settings” 选项，在下拉菜单中选择使用的 debug 烧写器，本人使用的是 st link v2,所以选择 Swim ST-Link,如下图，然后确认。



4.2 开发板通过 st link 连接电脑


4.3 选择 “Debug” 菜单下的 “Start Debugging” 选项，或点击工具栏  按钮。等待程序下载到 flash 中


4.4 现在可以使用 工具栏上的 快捷按钮 很方便的 进行 调试，




下面介绍几个我常用的项:

 : 程序复位, 即跳转到 `main()` 入口处

 : 继续运行

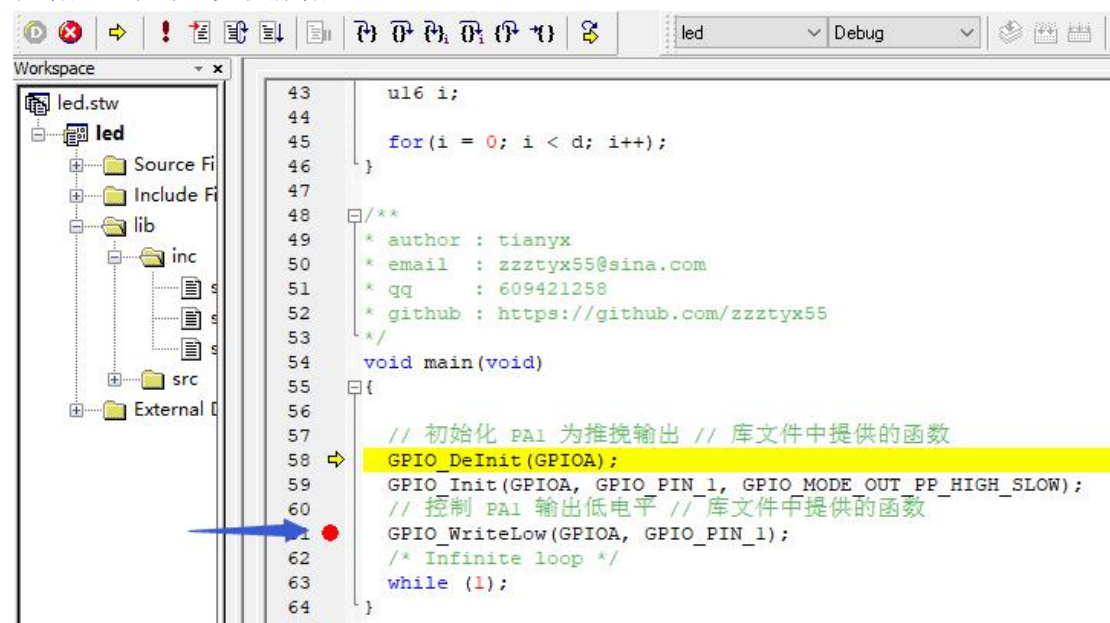
 : 单步运行, 不进入函数内部

 : 单步运行, 进入函数内部

 : 退出调试

另外还可以设置断点, 如下图箭头标示处双击, 即可放置一个断点, 然后调试程序时程序运行到断点处就会自动停下来。

在断点处单击可以取消断点。



4.5 然而调试不是让程序执行一步或者 N 步, 而是执行到某个需要的地方, 然后查看程序是否执行正确, 同时进行分析程序下一步的走向, 然后在后面适当的地方再设置断点进行调试。那么怎么查看程序是否执行正确以及判断之后走向呢?

对于本实验很简单, 如上图处设置好断点之后, 从头开始运行程序, 然后程序会停在断点处, 这时查看 led 是熄灭的, 然后单步运行(不进入函数内部), 程序执行将会跳过断点, 这时可以看到 led 被点亮了, 说明 `GPIO_WriteLow(GPIOA, GPIO_PIN_1);` 确实控制 PA1 输出低电平了。

上面的方法确实可以进行调试, 但是有很大的局限性, 因为很多时候我们并不是让程序点灯, 比如通过 AD 采集获取到了转换后的数字数据, 我们这时需要查看转换结果, 怎么办呢? 玩

过单片机的应该都知道，像这种 debug 都会有一个功能，那就是查看 ram 中的数据，具体怎么做呢？

下面就给大家讲解一下。

5, debug 调试时让程序暂停，查看运行数据是否正确

5.1 为了进行演示，需要将程序修改一下，修改后的代码如下：

```
/**
 * author : tianyx
 * email  : zzztyx55@sina.com
 * qq     : 609421258
 * github : https://github.com/zzztyx55
 */
void main(void)
{
    volatile int i;

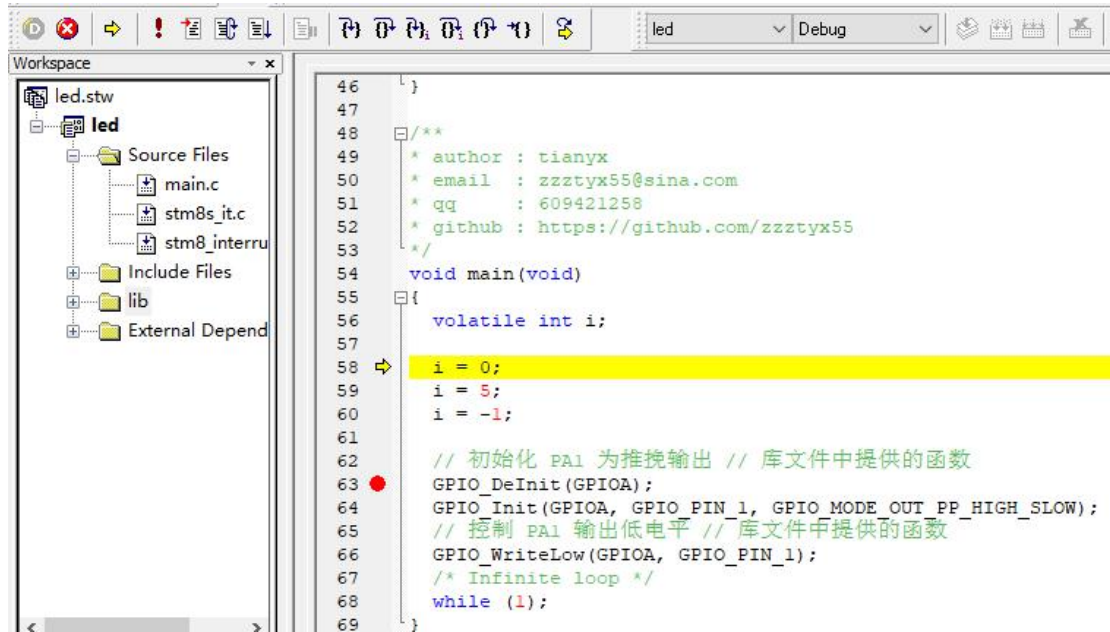
    i = 0;
    i = 5;
    i = -1;

    // 初始化 PA1 为推挽输出 // 库文件中提供的函数
    GPIO_DeInit(GPIOA);
    GPIO_Init(GPIOA, GPIO_PIN_1, GPIO_MODE_OUT_PP_HIGH_SLOW);
    // 控制 PA1 输出低电平 // 库文件中提供的函数
    GPIO_WriteLow(GPIOA, GPIO_PIN_1);
    /* Infinite loop */
    while (1);
}
```

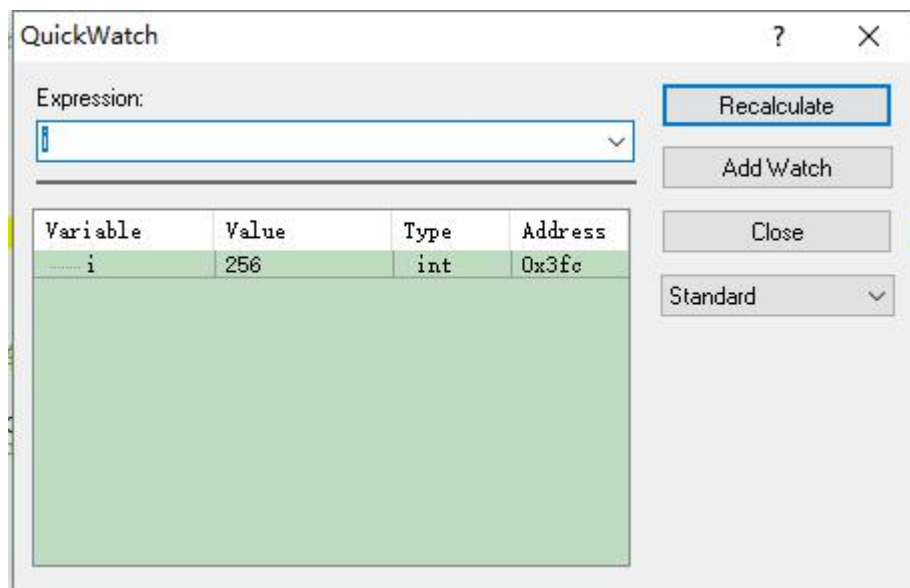
就是在程序的开头处定义了一个 int 型的变量 i，然后对 i 分别赋值 0, 5, -1.

Volatile 是一个关键字，用于确保对 i 的三条赋值语句不会被优化，初学者如果不理解暂时不必深究。

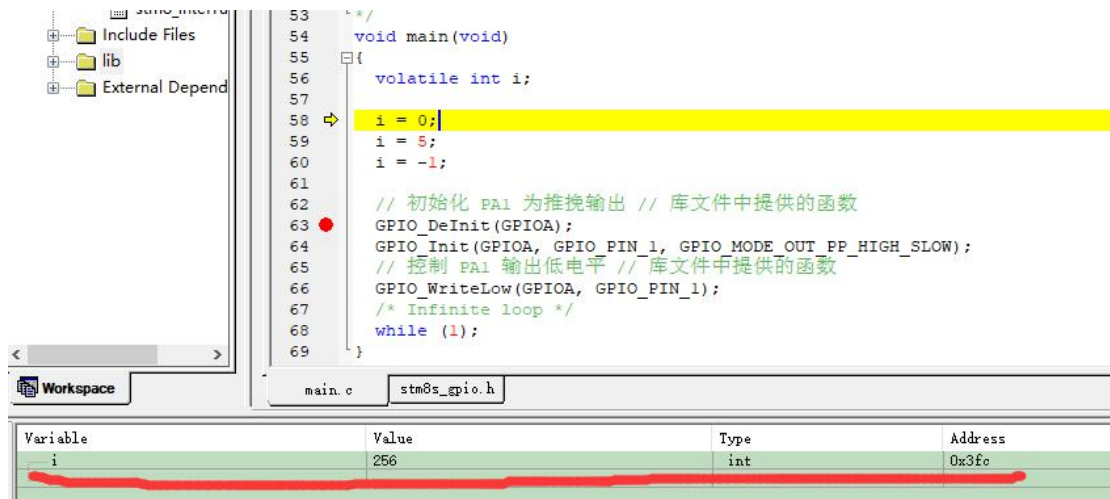
5.2 然后我们来编译，进入 debug 模式，然后让程序复位，跳转到 main() 入口，如下图：



5.3 双击选中变量 `i`，然后右键选中 `quick watch`，弹出一个对话框，可以看到变量 `i` 的值，类型，地址。同时右侧有个下拉按钮，可以选择变量值的显示方式(标准/ 十六进制/ 十进制/ 无符号型/二进制).如下图，



实际调试时根据自己的方便选择显示方式，这里我用默认的 `standard`。然后点击 `Add Watch` 将变量 `i` 添加到监视窗口如下图所示。



5.4 此时看到变量 i 是一个随机值 256，
单步运行后，i 的值变为 0，如下图所示。
再单步运行后，i 的值变为 5



这种调试方法对于 stm8 和 stm32 都非常简单有效，请大家务必掌握。

练习：

对于变量 i 不加 volatile 修饰符，再来单步调试，会是什么样的效果？

6，前面的测试 led 灯的代码非常简单，上电启动就点亮 led 灯，现在我们加点“炫”的效果：
让 led 灯闪烁起来，代码如下：


```

void main(void)
{
    volatile int i;

    i = 0;
    i = 5;
    i = -1;

    // 初始化 PA1 为推挽输出 // 库文件中提供的函数
    GPIO_DeInit(GPIOA);
    GPIO_Init(GPIOA, GPIO_PIN_1, GPIO_MODE_OUT_PP_HIGH_SLOW);
    // 控制 PA1 输出高电平, 关led // 库文件中提供的函数
    GPIO_WriteHigh(GPIOA, GPIO_PIN_1);

    // 延时, 等待系统稳定
    delay(100);

    /* Infinite loop */
    while (1)
    {
        delay(20000);
        GPIO_WriteHigh(GPIOA, GPIO_PIN_1);    // PA1输出高, 灯灭
        delay(20000);
        GPIO_WriteLow(GPIOA, GPIO_PIN_1);      // PA1输出低, 灯亮
    }
}

```

在 while 循环中, 我们延时, 拉高 PA1, 延时, 拉低 PA1, 循环下去, 实际执行效果就是 led 灯闪烁效果。

试想, 如果把这里的 delay() 函数去掉, 会产生什么效果?

```

/* Infinite loop */
while (1)
{
    //delay(20000);
    GPIO_WriteHigh(GPIOA, GPIO_PIN_1);    // PA1输出高, 灯灭
    //delay(20000);
    GPIO_WriteLow(GPIOA, GPIO_PIN_1);      // PA1输出低, 灯亮
}

```

测试发现 led 灯是一直亮着的, 为什么会这样? 明明代码里面有执行灭灯动作, 却是常亮?! 解密一下, 实际上灯并不是常亮, 去掉 delay() 后的程序依然是亮灭亮灭循环的, 只不过由于程序执行太快, 人眼观察不到灯有灭掉过, 所以以为是常亮, 如果认真观察, 会发现这时灯的亮度没有之前点亮的亮度强。

这就是 PWM 点灯/驱动电机等的原理, 使用 PWM 点灯, 可以控制灯的亮度, 屏幕的背光调节就是这么设计的; 使用 PWM 驱动电机, 可以调节电机的转速。

上面使用 4 行代码控制了 led 的灯亮灭, 现在我们尝试使用 2 行代码来实现同样的功能:

```

    /* Infinite loop */
    while (1)
    {
    #if 0
        //delay(20000);
        GPIO_WriteHigh(GPIOA, GPIO_PIN_1);    // PA1输出高，灯灭
        //delay(20000);
        GPIO_WriteLow(GPIOA, GPIO_PIN_1);      // PA1输出低，灯亮
    #else
        delay(20000);
        GPIO_WriteReverse(GPIOA, GPIO_PIN_1);  // PA1 输出电平翻转
    #endif
    }

```

函数 GPIO_WriteReverse(), 控制指定 io 口电平翻转，如执行前是高电平，执行后输出低电平；如执行前是低电平，执行后输出高电平。这个有点像位运算中的异或运算，然后看看函数 GPIO_WriteReverse()的实现方式

```

^/
void GPIO_WriteReverse(GPIO_TypeDef* GPIOx, GPIO_Pin_TypeDef PortPins)
{
    GPIOx->ODR ^= (uint8_t)PortPins;
}

```

发现这个函数确实使用异或实现的。