IAR 使用指南

概述

IAR IDE 是一款较为通用的嵌入式开发 IDE，本文对其使用进行了较为全面的描述。

本手册适用范围：

适用范围	
1125 系列	SPC1125，SPC1128
1168 系列	SPC1155， SPC1156， SPC1158， SPC1168，SPD1148， SPD1178， SPD1188， SPD1163，SPM1173
2168 系列	SPC2168，SPC2165，SPC2166，SPC1198
1169 系列	SPC1169，SPD1179，SPD1176
2188 系列	SPC2188，SPC1185

目录

图片列表

表格列表

版本历史

版本	日期	作者	状态	变更
A/0	2023-09-01	X.He	Outdated	1. 首次发布。
C/0	2024-08-21	X.He	Released	1. 修改为全系列通用文档。

1 新建 IAR 工程

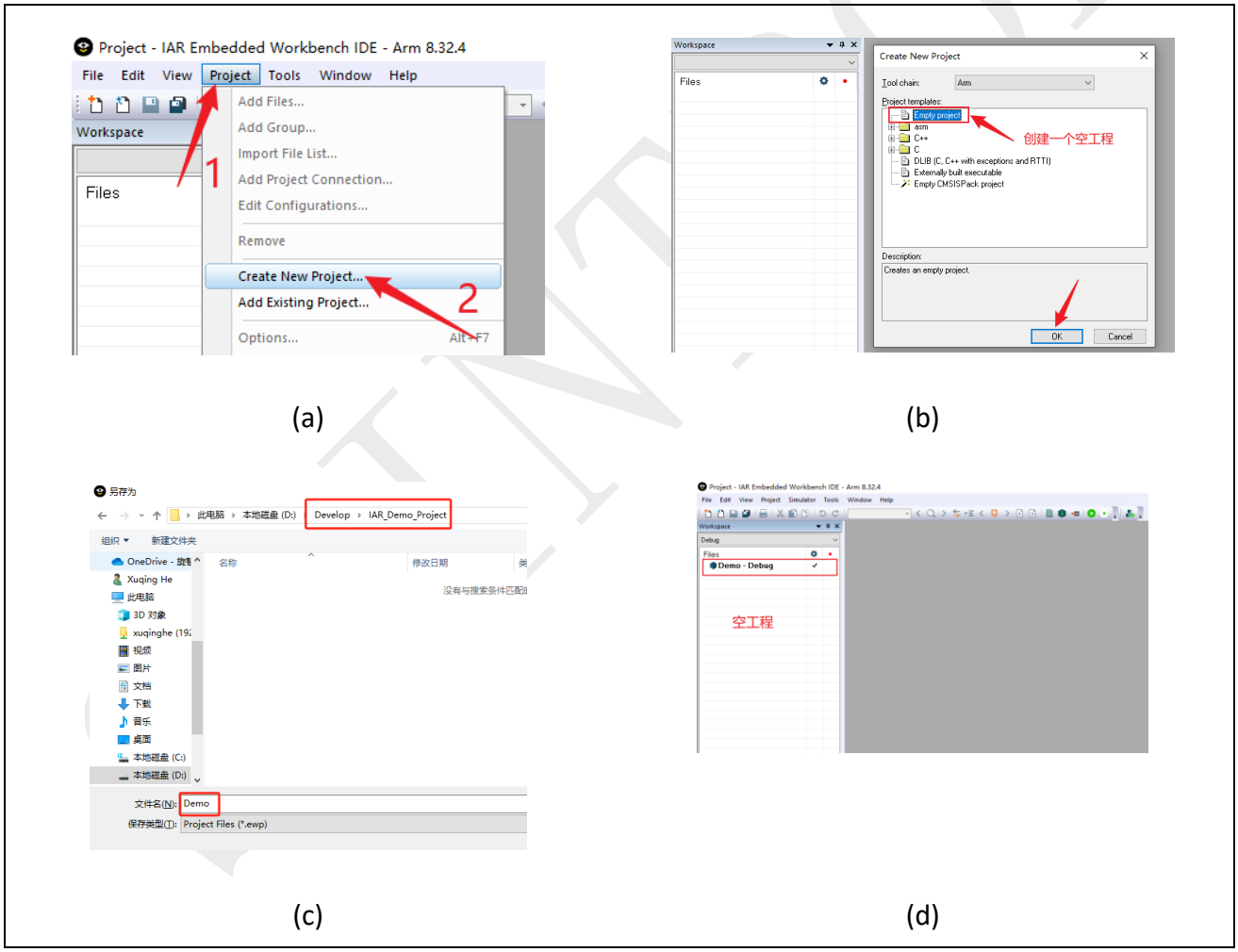
1.1 准备工作

在开始使用 IAR 软件新建工程前，首先需要安装 IAR EW for Arm 8.32.4，本文是基于此版本对 IAR 软件的使用进行介绍。IAR 软件可前往 IAR 官网（https://www.iar.com/）进行下载。

1.2 创建新工程

使用 IAR 软件创建新工程（Project —> Create New Project —> Empty project），具体操作如图 1-1 所示。

图 1-1：创建新工程



至此，一个空的基础工程就创建完成，接下来需要进一步添加文件到工程和配置工程。

1.3 添加源文件

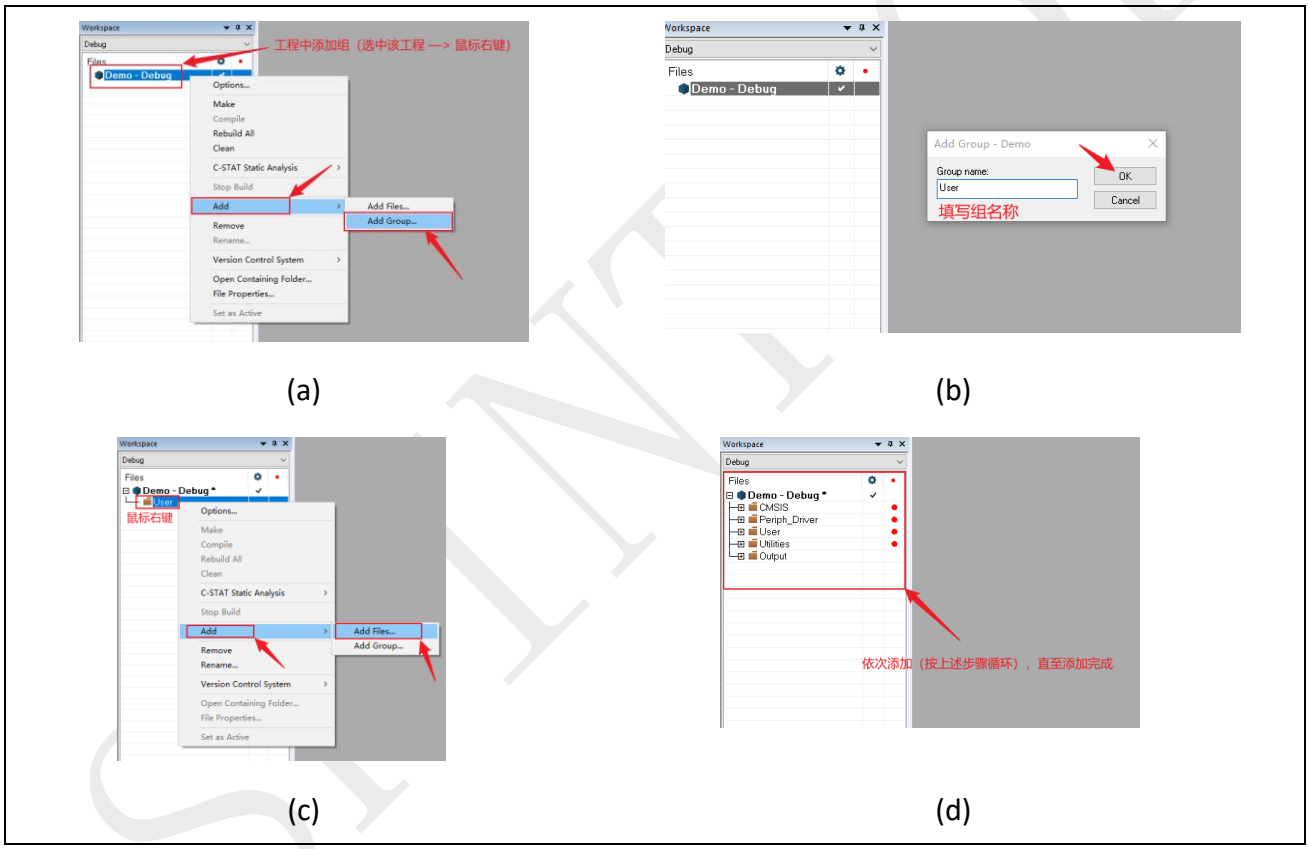
完成空的基础工程创建后，向该工程中添加组（文件夹）和添加源文件。也就是就将源代码（驱动库、新建的源文件等）添加到此工程中。此处的工程项目管理可根据用户自行定义（类似于自己分类、命名文件夹和文件），本文将按照常规的方式进行管理项目。

IAR 和 Keil 组管理的区别：

- IAR 可以添加多级组，类似于文件夹下可以再建文件夹，一直下去。
- Keil 只能添加单级组，类似于文件夹下面只能添加文件，而不能在添加文件夹。

为了简单、遵循 Keil 组结构，我们在 IAR 中分组方式也按照 Keil 方式分组，如图 1-2 所示进行操作，先在工程中添加组，再在组中添加文件，依次循环下去直到完成所有源文件的添加。

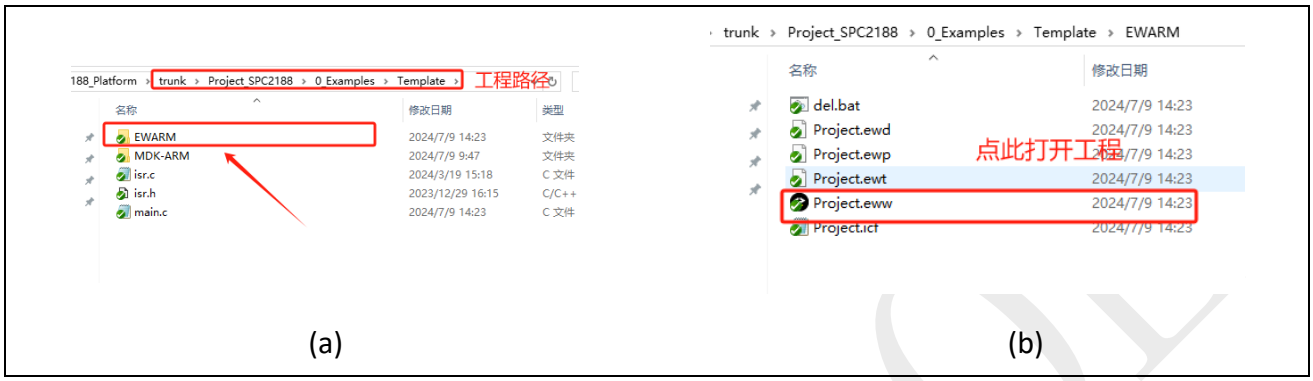
图 1-2：工程中添加组合源文件



2 加载 IAR 现有工程

首先找到如图 2-1 所示 Template 工程，双击打开此工程。

图 2-1：使用 IAR 打开已有工程

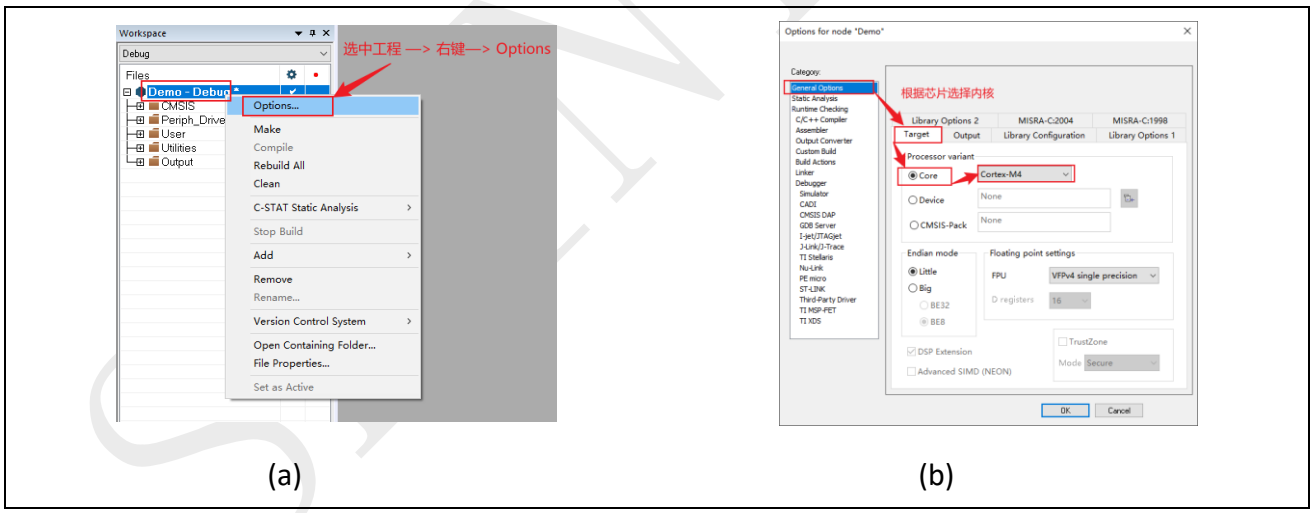


[1] 图中所示为 SPC2188 的 Template 实例。

2.1 配置工程

1. 首先根据如图 2-2 所示完成芯片内核选择。

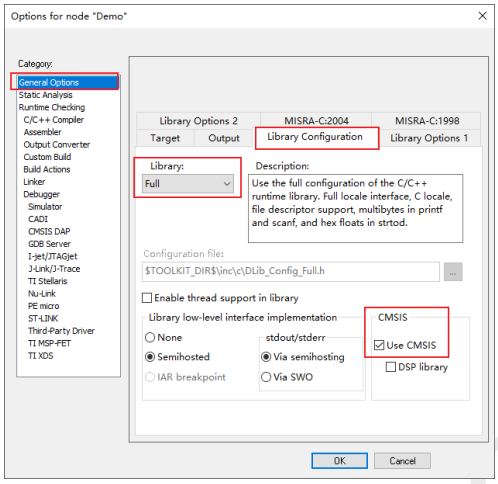
图 2-2：配置工程选择芯片内核



2. 根据如图 2-3 所示进行库配置 Library Configuration

- Library：如果需要使用某些标准的库函数接口（如 printf and scanf），就需要选择 Full。
- CMSIS：微控制器软件接口标准（Cortex Microcontroller Software Interface Standard），IAR for ARM 使用新版本 IAR 需要将其勾选。

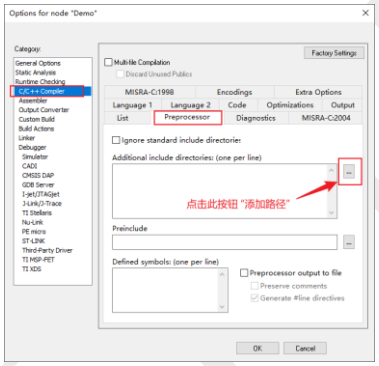
图 2-3：库配置



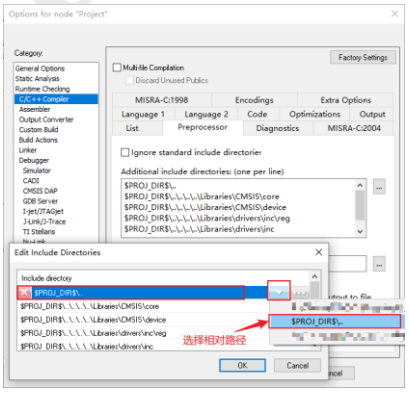
3. 预处理 Preprocessor - 添加路径

添加的路径最好是相对路径，而不是绝对路径。使用绝对路径工程位置改变之后就找不到文件，就会出错。可以点击按钮选择路径，也可以通过复制文件路径进行配置，如图 2-4 所示进行添加路径。

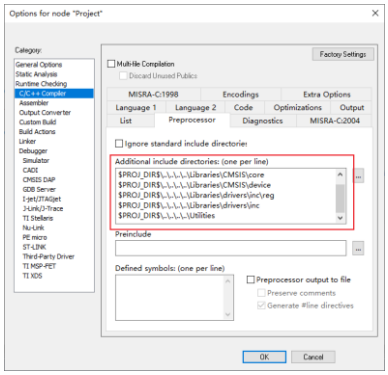
图 2-4：预处理 Preprocessor 添加路径



(a)



(b)

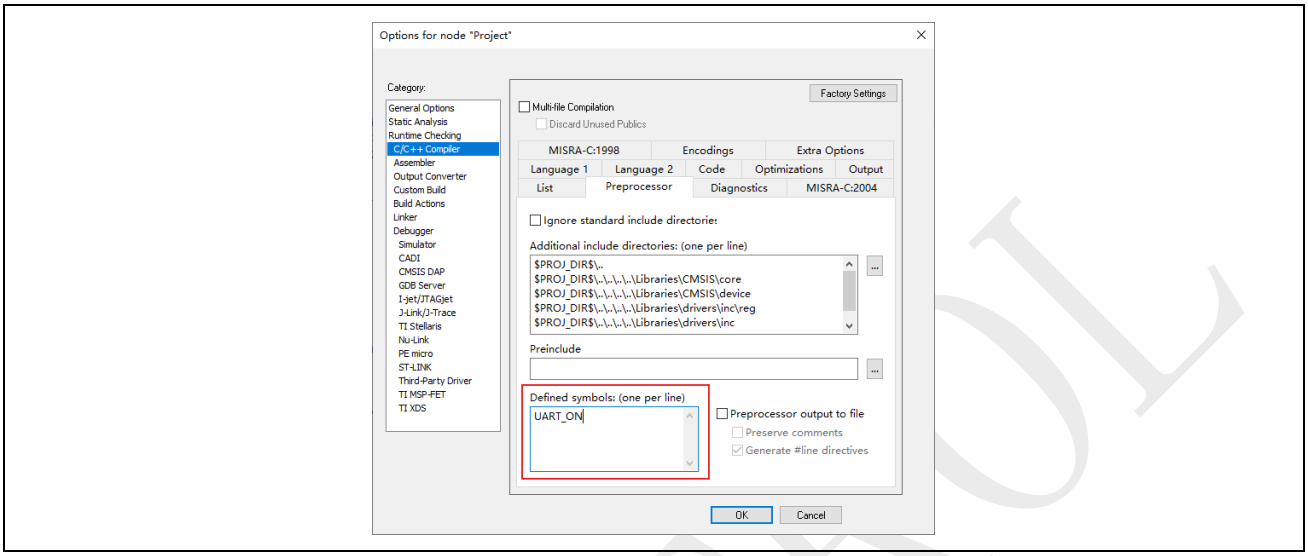


(c)

4. 预处理 Preprocessor - 预定义

这里的预定义类似于在源代码中的#define xxx 这种宏定义，如图 2-5 所示可进行设置。

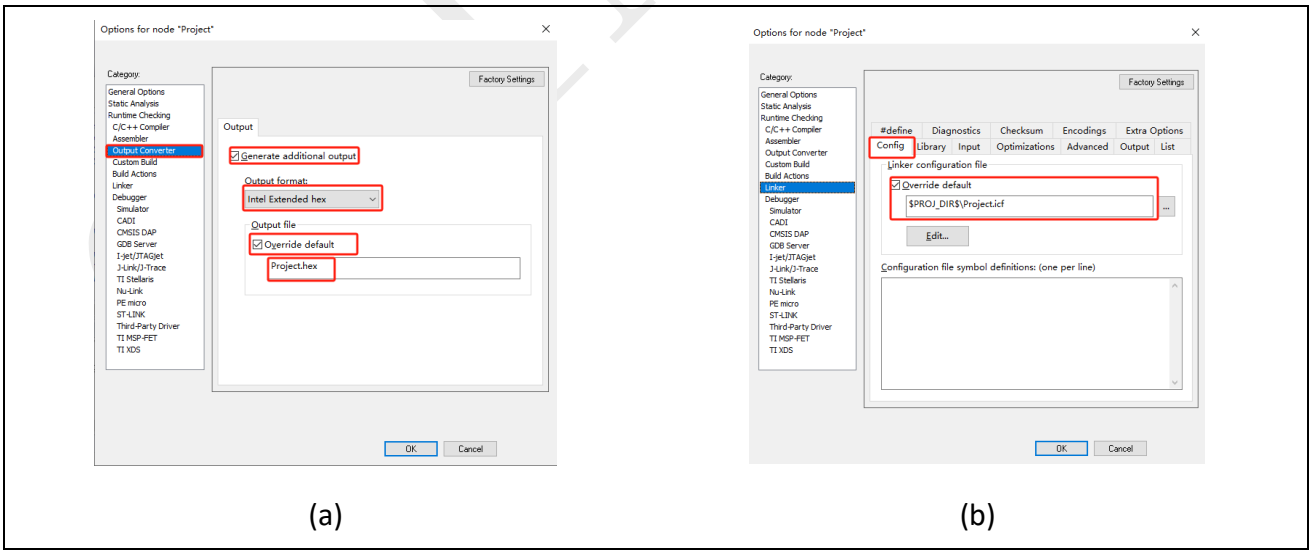
图 2-5：预处理 Preprocessor – 预定义



5. 输出 Hex 文件和链接配置文件

在编译完程序之后可以通过输出 Hex 文件进行程序烧录，可按照如图 2-6 所示完成配置即可输出 Hex 文件。也可通过链接配置文件查看链接程序时所产生的信息（定义内存位置、内存大小和堆栈大小等重要信息）。

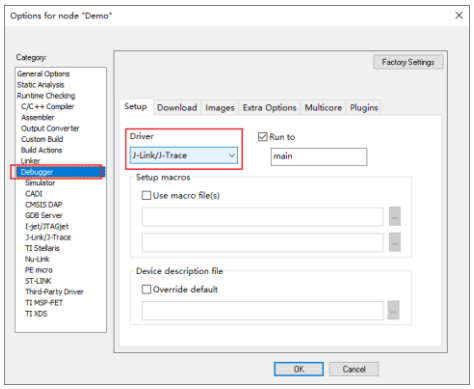
图 2-6：输出 Hex 文件和链接配置文件



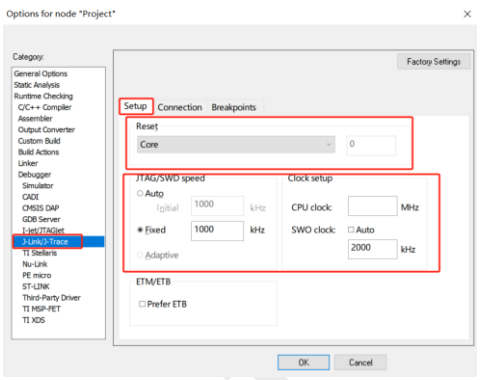
6. 选择下载调试工具

根据实际情况选择下载调试工具，本文选择 J-Link 作为下载调试工具，配置如图 2-7 所示。

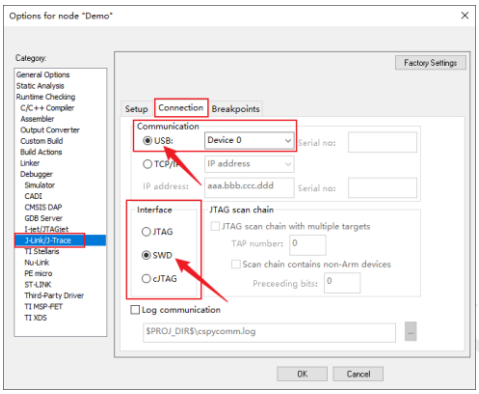
图 2-7：配置下载调试工具



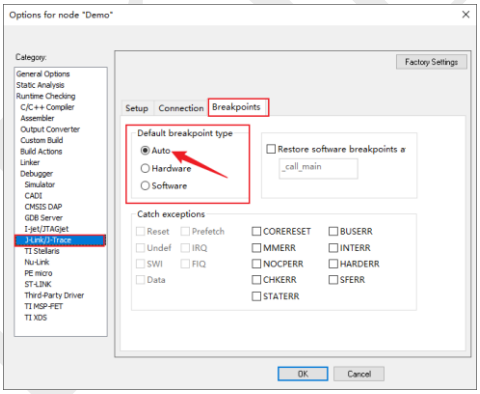
(a)



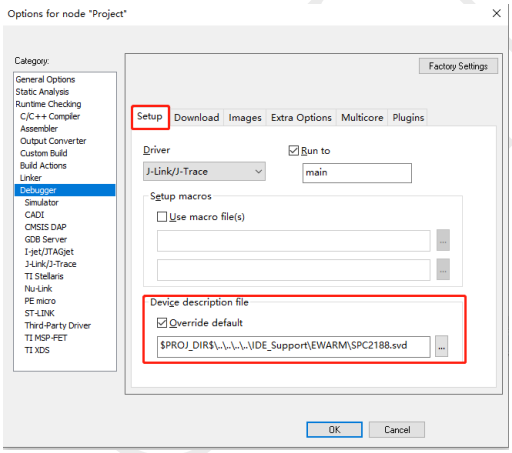
(b)



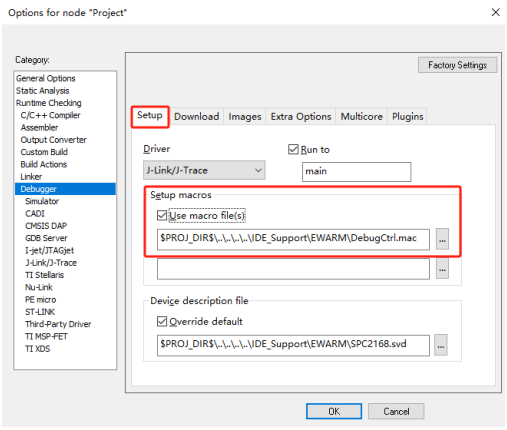
(c)



(d)



(e)



(f)

注意：SPC2188 系列可跳过此步骤，其他系列芯片都需要配置

在 flash 仿真时复位类型需要按照如表 2-1 所示进行选择，图 2-7 的（b）图选择复位 core 类型。在 RAM 仿真时 Core、Normal、Core and peripherals 复位所有平台都支持。

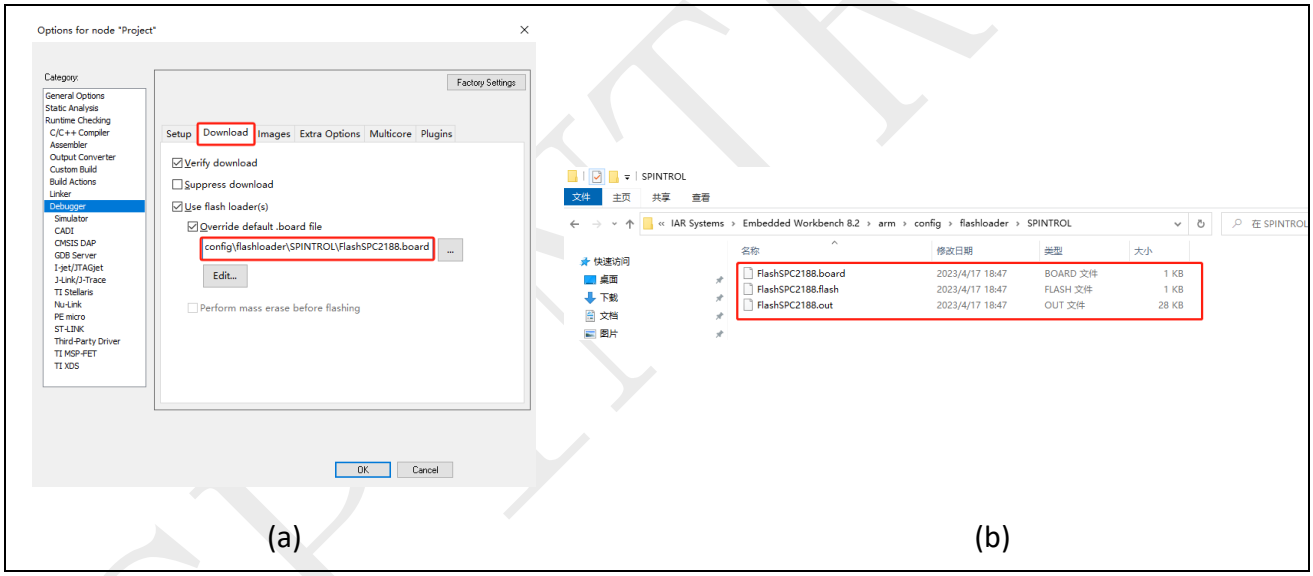
表 2-1：flash 仿真复位类型选择

芯片	复位类型选择
SPC1168 系列，SPC2168 系列，SPC1169 系列，SPC1125 系列	Core、Normal、Core and peripherals
SPC2188 系列	Core

在使用 J-Link 进行下载调试程序之前，还需要设置 Debugger Download 选项，如图 2-8 所示进行添加 board 文件。

注意：	在开始 Debug 之前，需要将 IDE_Support\EWARM\flashloader 目录下的算法文件复制到 IAR 软件安装目录下的 arm\config\flashloader\SPINTROL 目录下，否则会报错。

图 2-8：设置 Debugger Download



(a) (b)

[1] 图中所示为 SPC2188 的算法文件。

3 J-LINK 调试

3.1 J-LINK 与目标板硬件连接

J-LINK 适配器支持 2 种接口，如图 3-1 所示。推荐使用 SWD 接口，因为更省引脚而且调试功能不受影响，该接口如表 3-1 所示。

图 3-1：J-LINK 接口

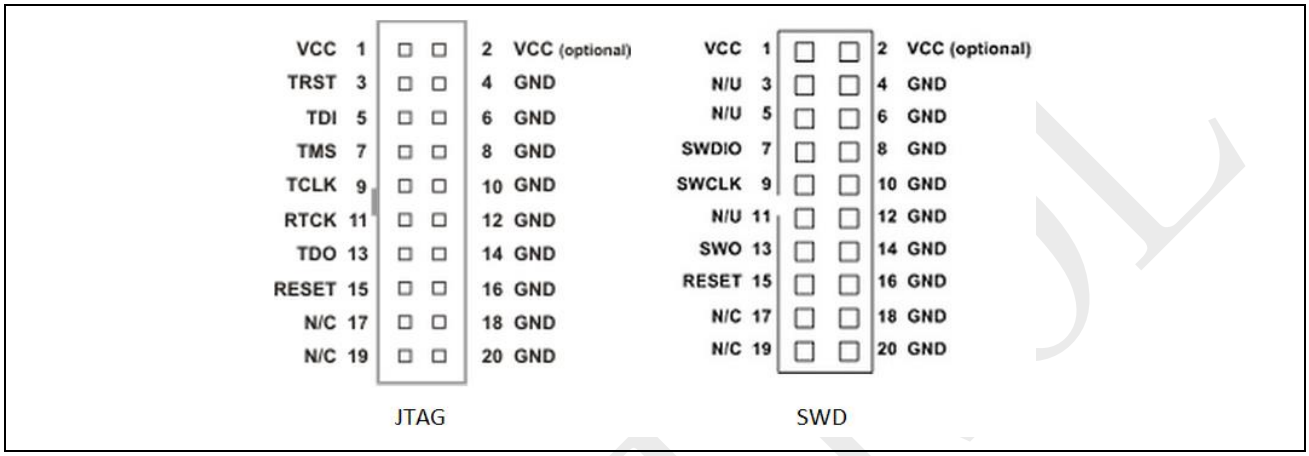


表 3-1：SW 接口信号定义

Signal	Connects to...
SWDIO	Data I/O pin
SWCLK	Clock pin
VCC	Positive Supply Voltage, the pin is optional.
GND	Digital ground
RESET	RSTIN pin, the pin is optional.
SWO	Serial data output, the pin is optional.

在使用芯片进行应用开发的过程的中，需要经常使用 J-LINK 进行程序的调试，各个芯片对应的 SWD 管脚如表 3-2 所示。

表 3-2：芯片与 SWD 管脚

芯片型号	SWD 引脚	
	SWDIO	SWCLK
SPC1169 系列	GPIO17	GPIO18
SPC1168 系列	GPIO38	GPIO39
SPC2168 系列（仅 CPU 核）	GPIO49	GPIO48
SPC2168 系列（仅 CAU 核）	GPIO51	GPIO50

芯片型号	SWD 引脚	
	SWDIO	SWCLK
SPC2188_CPU0/SPC2188_CPU1（单核 SWD 模式）或者 SPC2188_CPU0 （双核 SWD 模式）	GPIO80	GPIO81
SPC2188_CPU1（双核 SWD 模式）	GPIO78	GPIO79
SPC1185	GPIO80	GPIO81
SPC1125 系列	GPIO38	GPIO39

注意： J-LINK 调试时，芯片的 TRSTN 必须拉高，boot 引脚电平请参考 TRM 启动引导配置章节；

J-LINK 调试时，J-LINK 的 GND 必须连上调试板的 GND；

J-LINK 下载器端口电压需要与芯片端口电压一致。

根据前面的介绍，将 J-LINK 设备与 spintrol 芯片正确连接后，按照图 2-7、图 2-8 设置 Debug 的相关选项，就可以使用 J-LINK 设备调试程序了。

单击工具栏上的（进行下载和仿真）或者（仿真但不下载）进入 Debug 状态，如果是第一次进行调试会出现如图 3-2 弹框，点击“OK”进行选择调试设备选择如图 3-3 所示，选择进行仿真的设备。

图 3-2：未指定设备

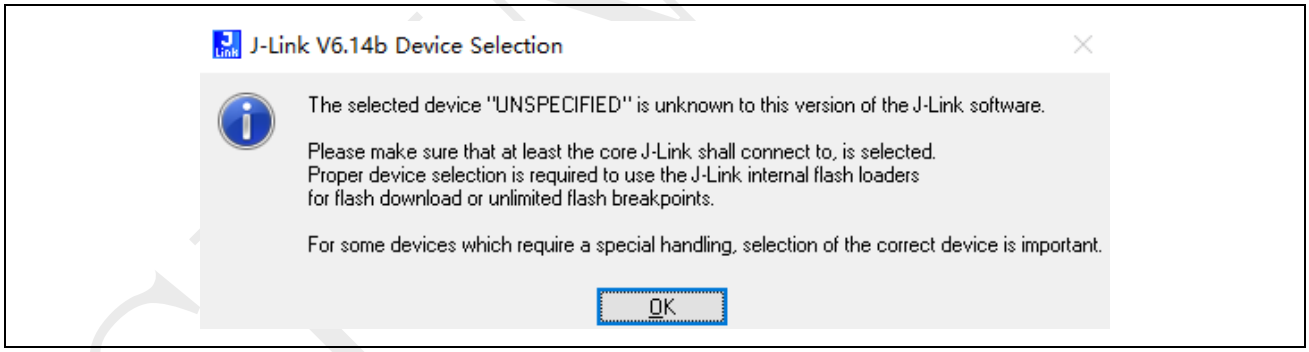
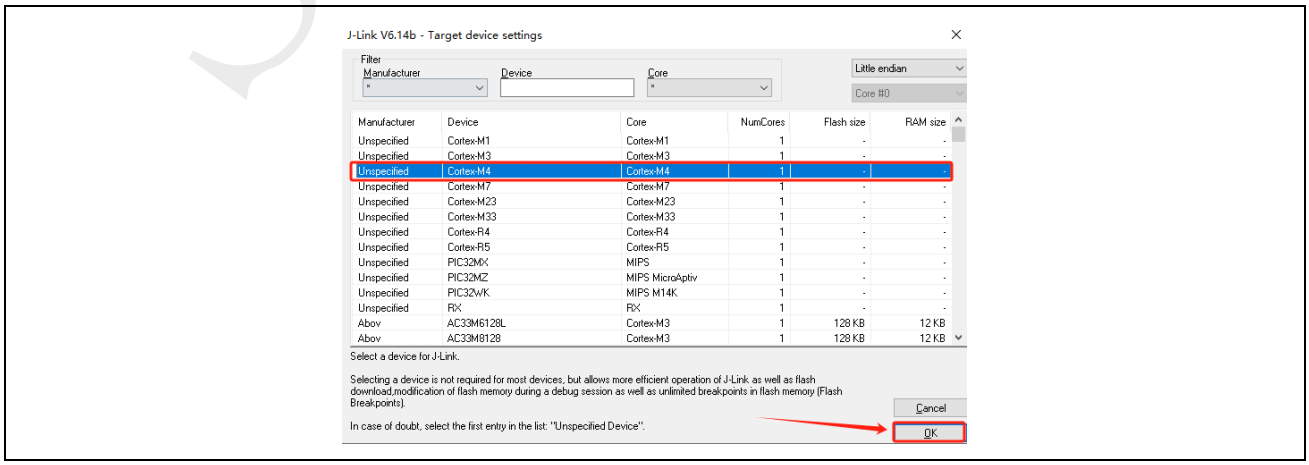


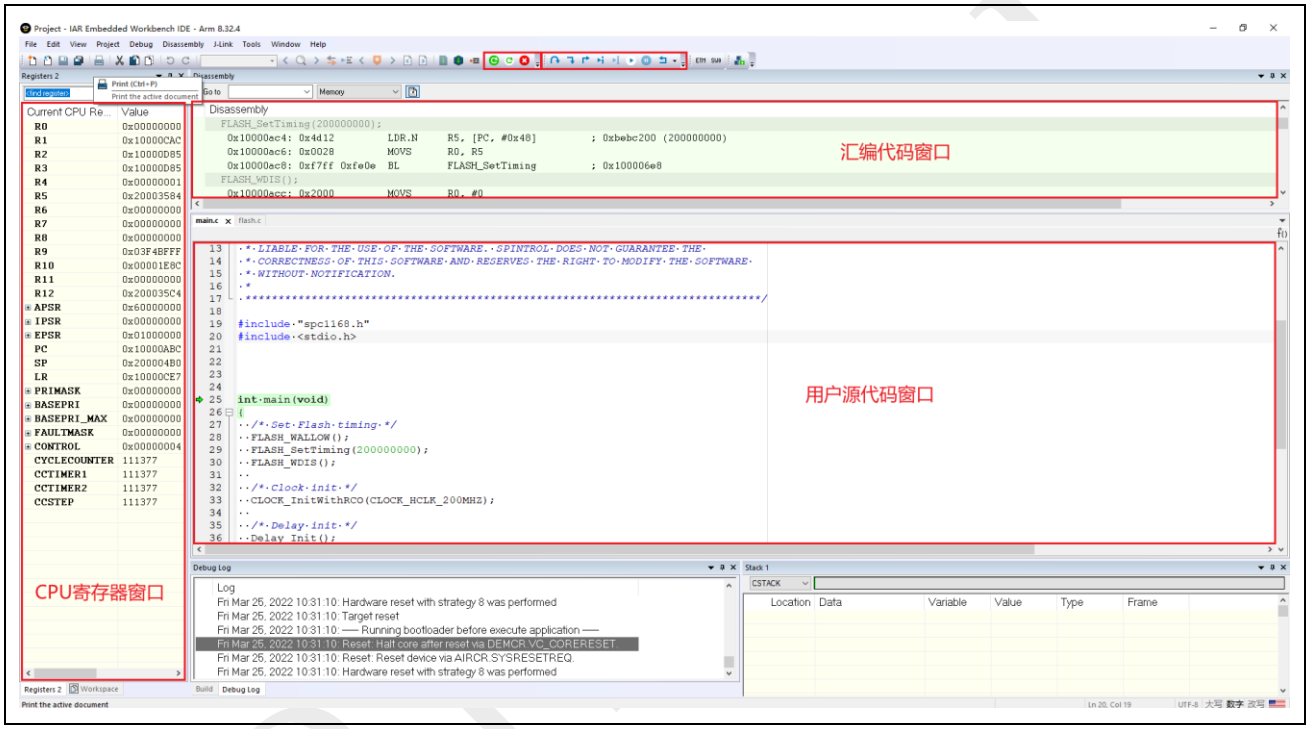
图 3-3：调试设备选择



当设备选择正确，调试界面将会进入如图 3-4 所示。程序执行到 main 函数入口处后停止，等待用户的进一步操作。此时，IAR 软件的界面也发生了变化：除了用户源代码窗口，还出现了汇编代码窗口和 CPU 寄存器窗口。在汇编代码窗口中，绿色底纹的汇编代码对应于用户代码窗口中的 C 代码；此外，菜单栏上也出现了一些与 Debug 相关的菜单选项。

注意：	在程序进入 Debug 状态后，如果想修改代码，可以通过单击按钮 重新下载程序并进入 Debug 模式。

图 3-4：启动 Debug 后的界面



3.2 单步调试

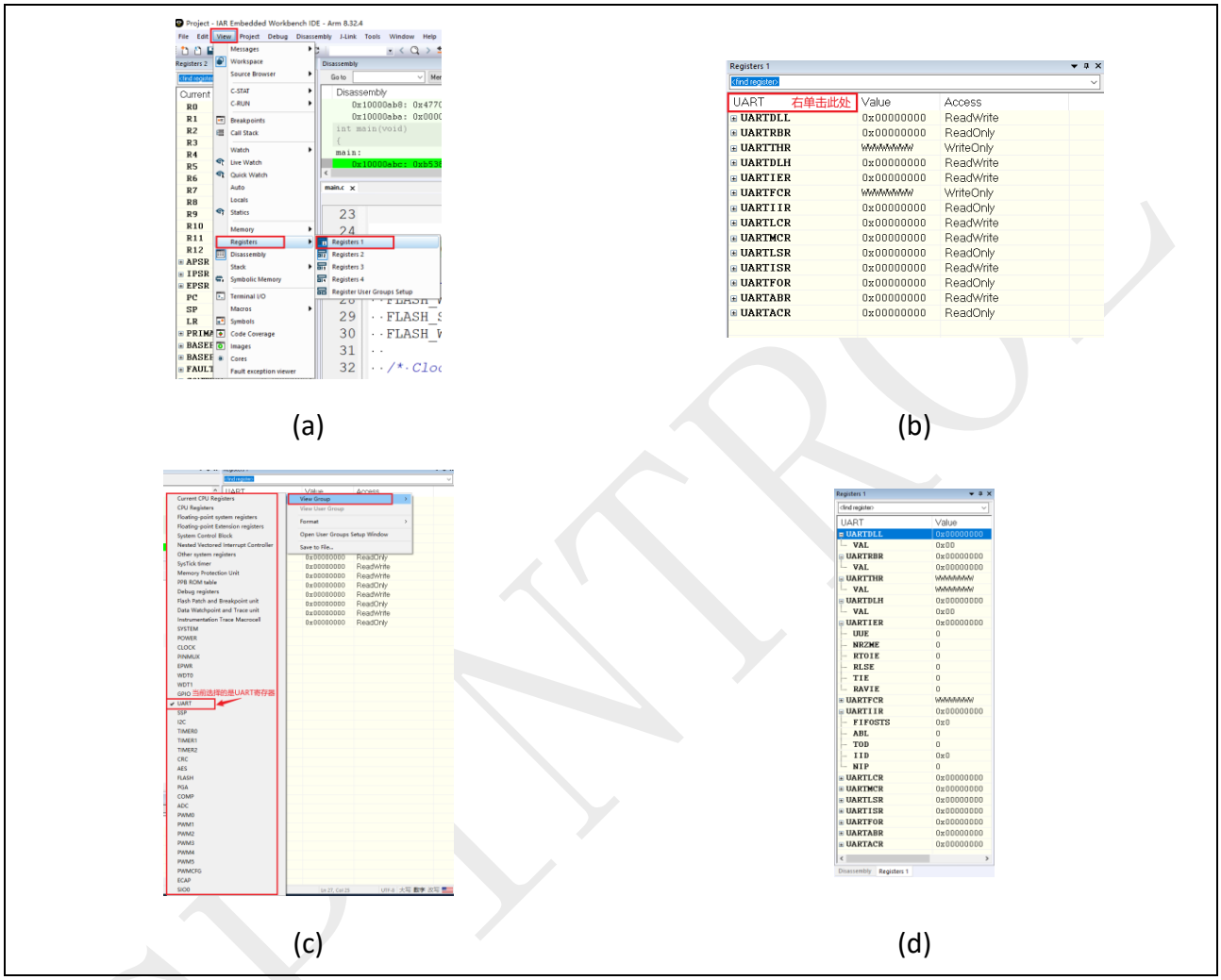
单击工具栏上的或者按钮后，程序进入 Debug 状态。此时单击工具栏上的按钮或者按下快捷键 F10 就可以单步执行程序。在单步调试的时候，用户代码窗口左侧边框处的绿色箭头表示当前位置的代码为下一次要执行的语句。因此，可以通过这个绿色箭头快速判断程序执行到了哪条语句。

有时候，我们希望程序能够快速地执行到某个位置，再进行单步调试。这时我们可以将光标定位到该位置，然后单击工具栏上的按钮，程序就会立即执行到当前光标处。此外，我们也可以通过设置断点的方式来实现上述功能。

3.3 观察外设寄存器

在调试程序的时候，当我们需要查看芯片外设 Register 的值时，可通过如图 3-5 所示方法进行查看。

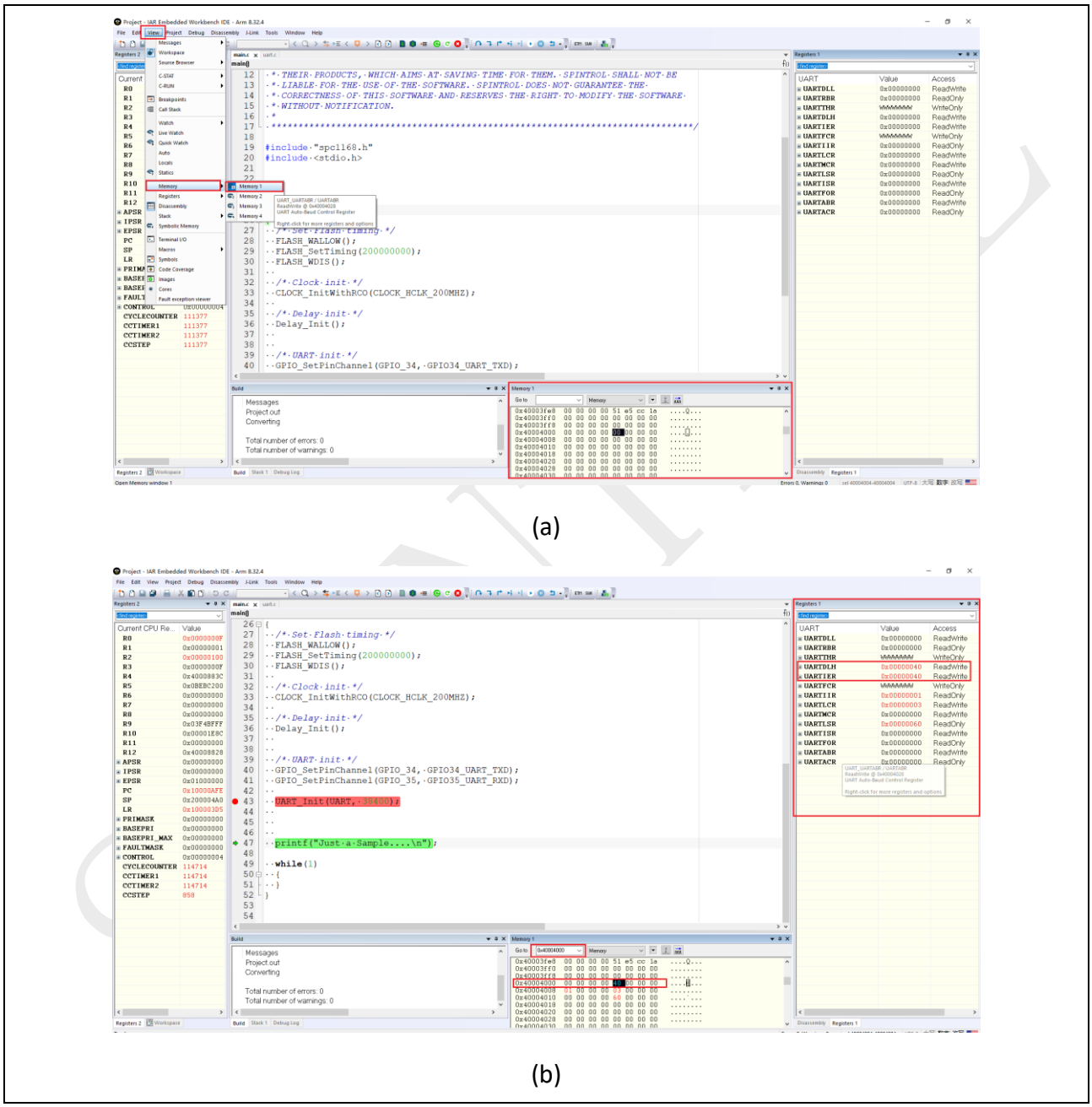
图 3-5：查看芯片外设寄存器



3.4 Memory 窗口

在 Debug 程序的过程中，我们还可以通过 Memory 窗口观察芯片内任一存储单元的地址。我们以芯片的 UART 模块为例，通过芯片技术参考手册可以得到 UARTDLH 寄存器和 UARTIER 寄存器的地址为 0x40004004。首先，打开一个 Memory 观察窗口（Memory1），如图 3-6 所示。

图 3-6：Memory 观察窗口



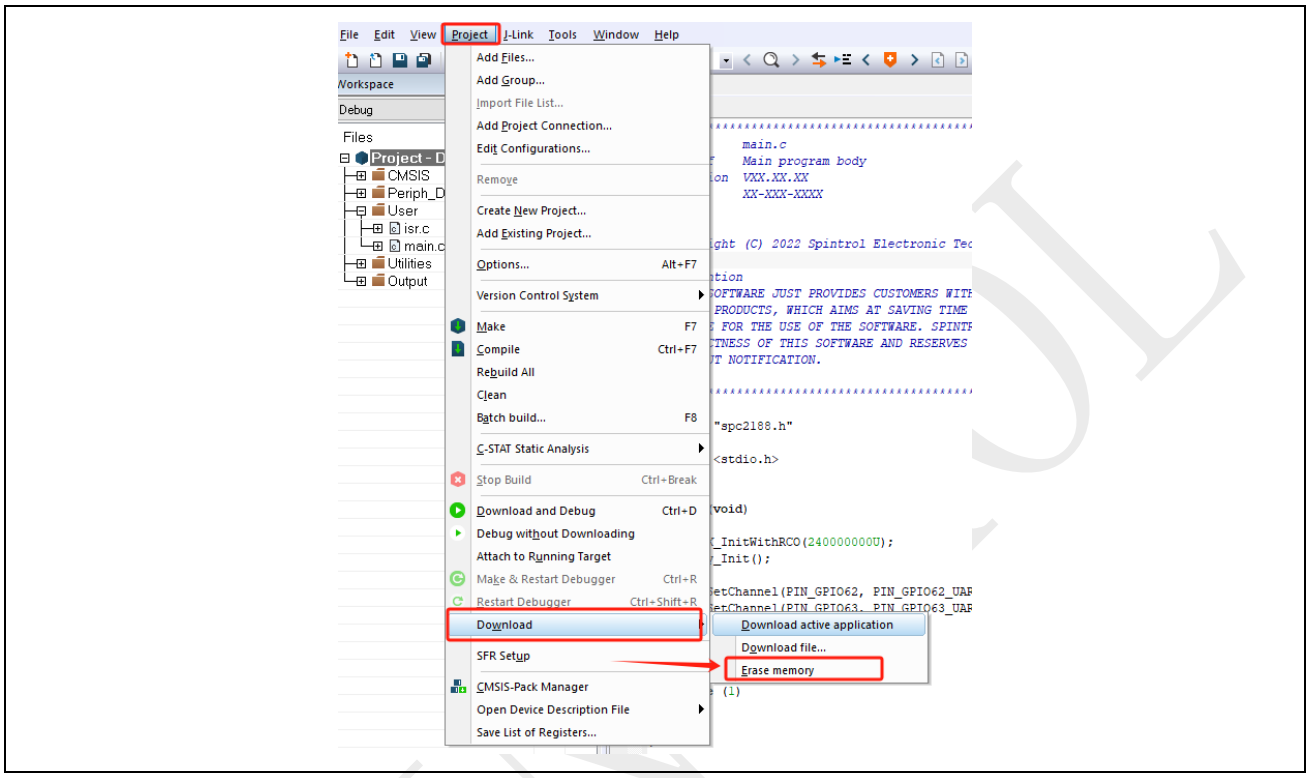
(a)

(b)

4 J-LINK 下载

使用 J-LINK 进行下载 Flash 之前需要先使用 J-LINK 对 Flash 进行擦除，如图 4-1 所示。

图 4-1：Flash 擦除



在擦除 Flash 之后可以对 Flash 进行写操作，下载 HEX 文件有两种方式，一种是选择如图 4-1 所示“Download active application”下载当前工程的 HEX 文件，另一种是选择“Download file…”指定某个 HEX 文件进行下载。

注意： 在开始下载之前，需要将 IDE_Support\EWARM\flashloader 目录下的算法文件复制到 IAR 软件安装目录下的 arm\config\flashloader\SPINTROL 目录下如图 2-8（b）图所示，否则会报错。

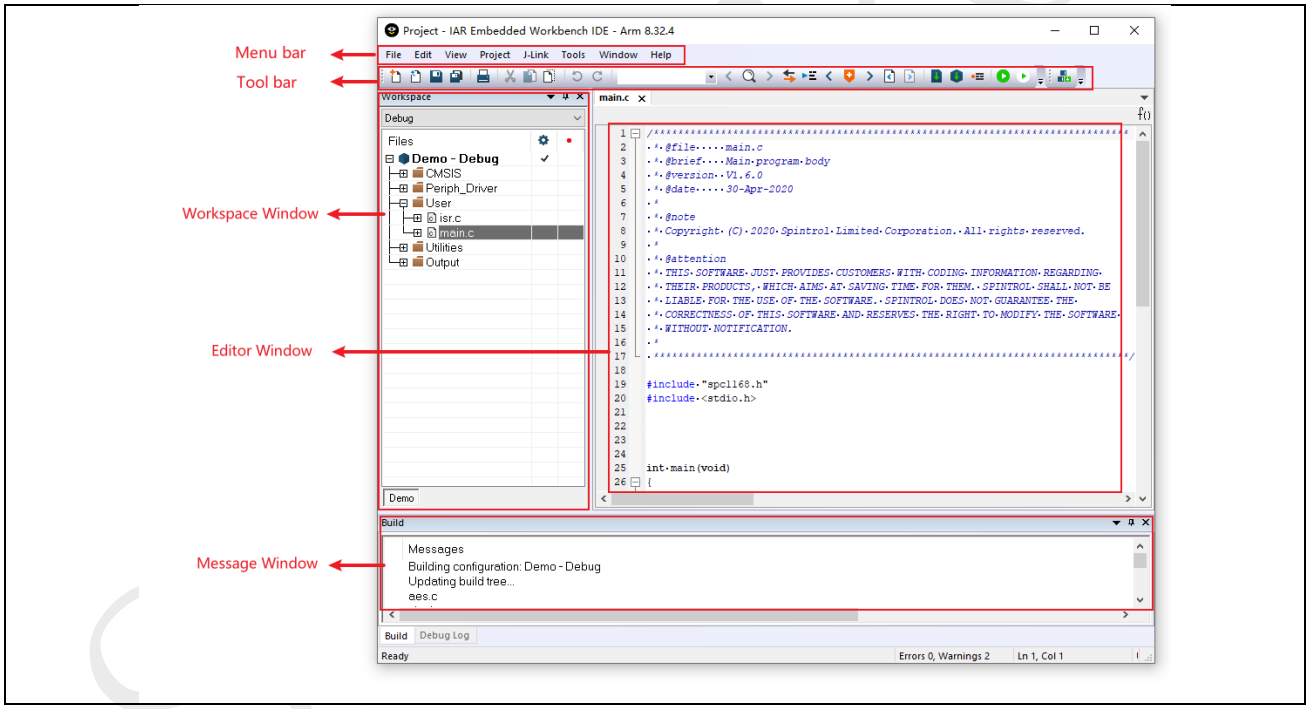
5 IAR 界面介绍

5.1 主窗口界面

这里简单介绍一下 IAR 软件主界面下的各个窗口，如图 5-1 所示。

- Menu Bar 菜单栏：该窗口是 IAR 比较重要的一个窗口，里面包含 IAR 所有操作及内容
- Tool Bar 工具栏：该窗口是一些常见的快捷按钮
- Workspace Window 工作空间窗口：一个工作空间可以包含多个工程，该窗口主要显示工作空间下面工程项目的内容
- Edit Window 编辑空间：代码编辑区域
- Message Window 信息窗口：该窗口包括编译信息、调试信息、查找信息等信息窗口
- Status Bar 状态栏：该窗口包含错误警告、光标行列等一些状态信息

图 5-1：主窗口界面

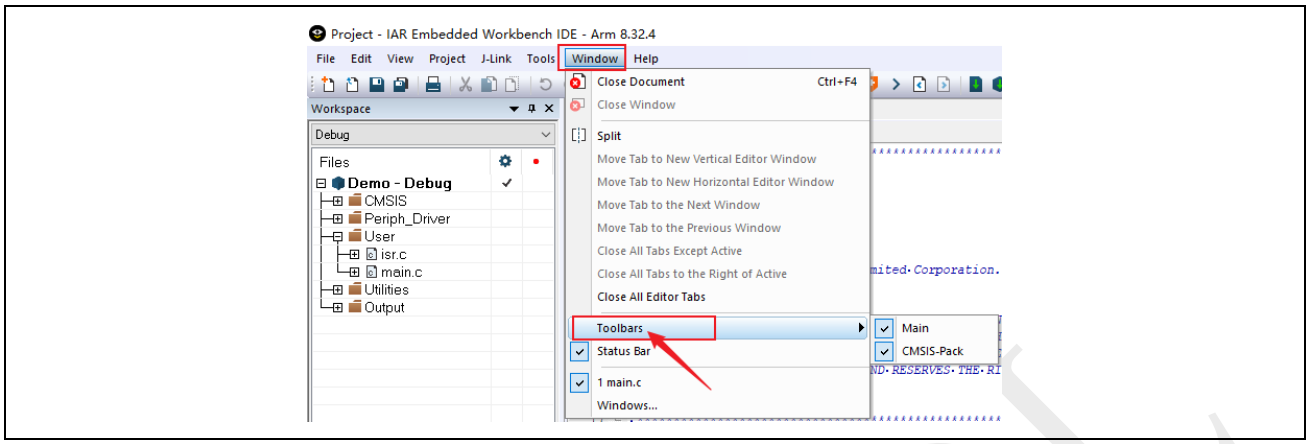


5.2 工具栏

IAR 软件中的 Tool Bar 工具栏一共有两个：Main 主工具栏和 Debug 调试工具栏。在编辑（默认）状态下只显示 Main 工具栏，在进入调试模式后才会显示 Debug 工具栏。

工具栏可以在通过菜单打开：Window --> Tool Bar，如图 5-2 所示。

图 5-2：工具栏



1. 主工具栏

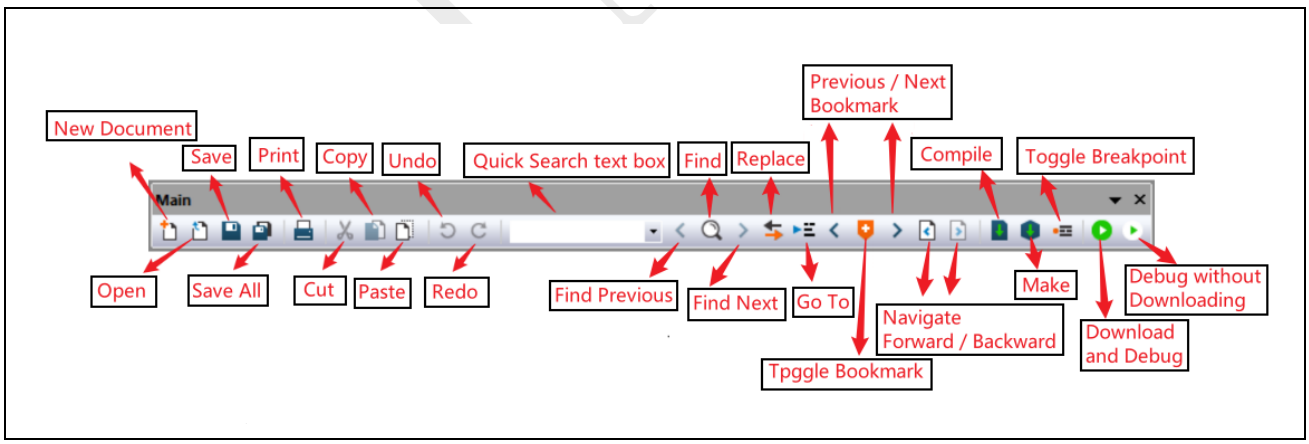
如图 5-3 所示，在编辑（默认）状态下，只有主工具栏，这个工具栏的内容也是在编辑状态下常用的快捷按钮。其中“Download and Debug”和“Debug without Downloading”这两个按钮的区别需要注意。

Download and Debug：是下载代码之后再进行调试。

Debug without Downloading：只调试不下载，即如果之前下载过了代码，只需要再点击该按钮即可，否则会出现错误。

这两个按钮图标在编辑和调试模式下略有差异，在调试模式下可以再次下载程序后继续调试。

图 5-3：主工具栏



2. 调试工具栏

调试工具栏则是在进行程序调试时才有效的一个快捷按钮，在编辑状态下，这些按钮是无效的。

如图 5-4 所示，以下是调试模式中常用的快捷按钮：

Reset 复位

Break 停止运行

Step Over 逐行运行 F10

Step Into 跳入运行 F11

Step Out 跳出运行 F11

Next Statement 运行到下一语句

Run to Cursor 运行到光标行

Go 全速运行 F5

Stop Debugging 停止调试 Ctrl + Shift + D

图 5-4：调试工具栏

6 IAR ICF 文件指令介绍

6.1 定义 symbol 指令

用法：

define [exported] symbol name = expr;

参数：

exported：导出该 symbol，使其对可执行镜像可用

name：符号名

expr：符号值

示例：

define symbol __ICFEDIT_region_ROM1_start__ = 0x10000000;

作用：

指定某个符号的值

6.2 定义 memory 指令

用法：

define memory [name] with size = size_expr;

参数：

Name：memory 的名称

size_expr：地址空间的大小

示例：

define memory mem with size = 4G;

作用：

定义一个可编址的存储地址空间

6.3 定义 region 指令

用法：

define region name = region-expr;

参数：

Name：region 的名称

region-expr：memory_name:[from expr to expr]，可以定义起止范围，也可以定义起始地址和 region 的大小。

示例：

define region ROM1_region = mem:[from __ICFEDIT_region_ROM1_start__ to __ICFEDIT_region_ROM1_end__];

define region RAM2_region = mem:[from __ICFEDIT_region_RAM2_start__ to __ICFEDIT_region_RAM2_end__];

作用：

定义一个存储地址区域（region）。一个区域可由一个或多个范围组成，每个范围内地址必须连续，但几个范围之间不必是连续的

6.4 block 指令

用法：

define block name[with param, param...]

参数：

Name：block 的名称

Param：

size =expr（块的大小）

maximum size = expr（块大小的上限）

alignment = expr（最小对齐字节数）

fixed order（按照固定顺序放置 sections）

示例：

define block CSTACK with alignment = 8, size = __ICFEDIT_size_cstack__ { };

作用：

定义一个地址块（block）；它可以是个空块，比如栈、堆；

6.5 定义 initialize 指令

用法：

initialize { by copy | manually } { section-selectors }

参数：

by copy：在程序启动时 IAR 软件进行自动拷贝。

Manually：在程序启动时 IAR 软件不进行自动拷贝。

示例：

initialize by copy { readwrite, };

作用：

初始化 section 将指定 section 从 ROM 拷贝到 RAM

6.6 定义 Do not initialize 指令

用法：

do not initialize { section-selectors }

参数：

section-selectors：section 选择器 [section-attribute][section sectionname] [object filename]

示例：

do not initialize { section .noinit };

作用：

规定在程序启动时不需要初始化的 sections。一般用于__no_init 声明的变量段（.noinit）

6.7 定义 place at 指令

用法：

place at { address [memory:] expr | start of region_expr | end of region_expr }

{

section-selectors

}

参数：

address [memory:] expr：特定内存中的特定地址。memory 地址必须在由 define memory 指令定义的提供的内存中可用。

start of region_expr：region 的起始地址。

end of region_expr：region 的结束地址。

section-selectors：section 选择器 [section-attribute][section sectionname] [object filename]

示例：

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

作用：

把一系列 sections 和 blocks 放置在某个具体的地址，或者一个 region 的开始或者结束处

6.8 定义 place in 指令

用法：

place in region-expr { section -selectors }

参数：

section-selectors: section 选择器 [section-attribute][section sectionname] [object filename]

示例：

place in ROM1_region { readonly };

作用：

把一系列 sections 和 blocks 放置在某个 region 中。sections 和 blocks 将按任意顺序放置。

6.9 概述

Section	描述
.bss	保存初始化为 0 的静态和全局变量
CSTACK	保存 C 或 C++ 程序使用的堆栈
.cstart	保存 start_up 代码
.data	保存静态和全局初始化变量包括初始值设定项
.data_init	保存 .data section 的初始值设定项。
.difunct	保存指向代码（通常 C++构造函数）的指针，这些代码应在调用 main 之前由系统启动代码执行
HEAP	保存用于动态分配数据的堆
.iar.dynexit	保存 atexit table
.intvec	保存复位和中断向量
IRQ_STACK	保存中断请求、IRQ 和异常的栈
.noinit	保存__no_init 静态和全局变量
.rodata	保存常量数据
.text	保存程序代码

如需查询指定的目标文件包含的 section 内容，可以使用 IAR 编译工具下面的 ielfdumparm.exe 可执行文件查看。ielfdumparm.exe 文件路径在安装 IAR 软件 IAR Systems\Embedded Workbench 8.2\arm\bin 目录下。

7 IAR ICF 文件使用示例

7.1 对单个函数进行重定向

7.1.1 使用__ramfunc 关键字进行重定向

使用__ramfunc 关键字会将关键字修饰的函数重定向到 ICF 文件中包含 readwrite 属性的 Region 包含的地址范围内，并且关键字修饰的函数不能调用没有被该关键字修饰的函数以及带有 const 修饰的全局变量。

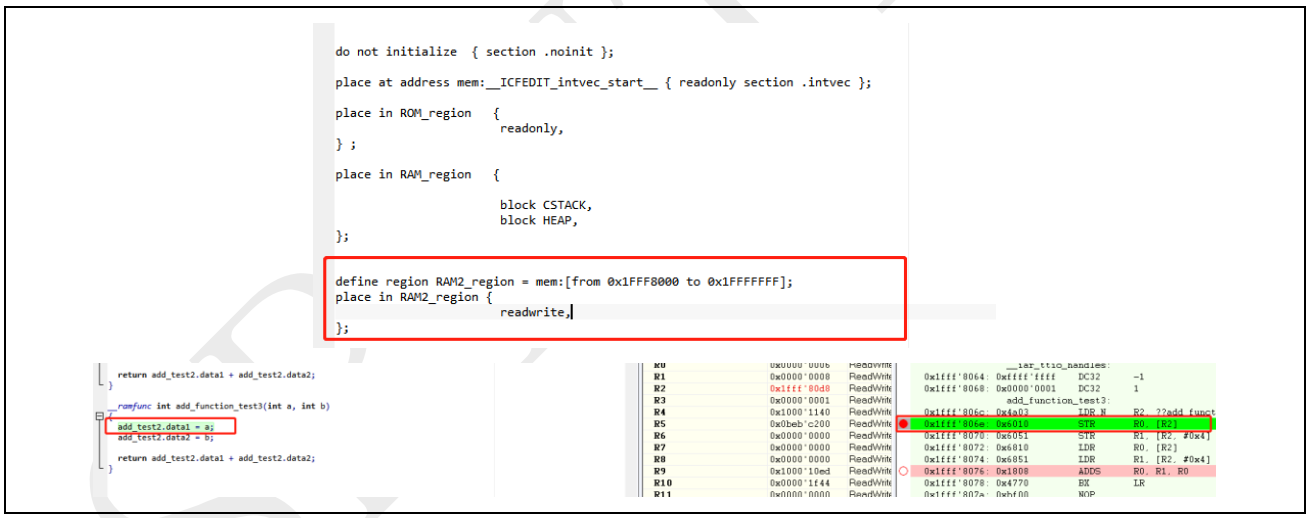
对函数代码而言，将会被重定向到 ICF 文件中包含有 readwrite 的 region；

对于全局变量而言，非 const 全局变量将会被重定向到 ICF 文件中包含有 readwrite 的 region；

对于局部变量而言，非 static 局部变量将会被重定向到 ICF 文件中包含有 block CSTACK 的 region；static 局部变量将会被重定向到 ICF 文件中包含有 readwrite 的 region；

如图 7-1 所示，add_function_test3 函数代码被重定向到 0x1FFF8000~0x1FFFFFFF RAM2_region 地址，是因为 ICF 文件中 RAM2_region 是包含有 readwrite 属性段的 region。形参 a 以及 b 将会被重定向到 RAM_region 中，因为这个 region 包含有 block CSTACK。非 const 且非 static 的全局变量 add_test2 将会被重定向到 RAM2_region 中，因为这个 region 包含有 readwrite。

图 7-1：ICF 文件与重定向内容



注意： 该方法的重定向适用所有函数包含中断服务函数及中断中调用的函数。

7.1.2 使用 section 修饰进行重定向

对某个函数进行重定向到 ICF 文件中 place in 指令指定的”section .add_test2_func_section” 的 Region 包含的地址范围内，在重定向的函数体中可以调用其他地址范围的函数。

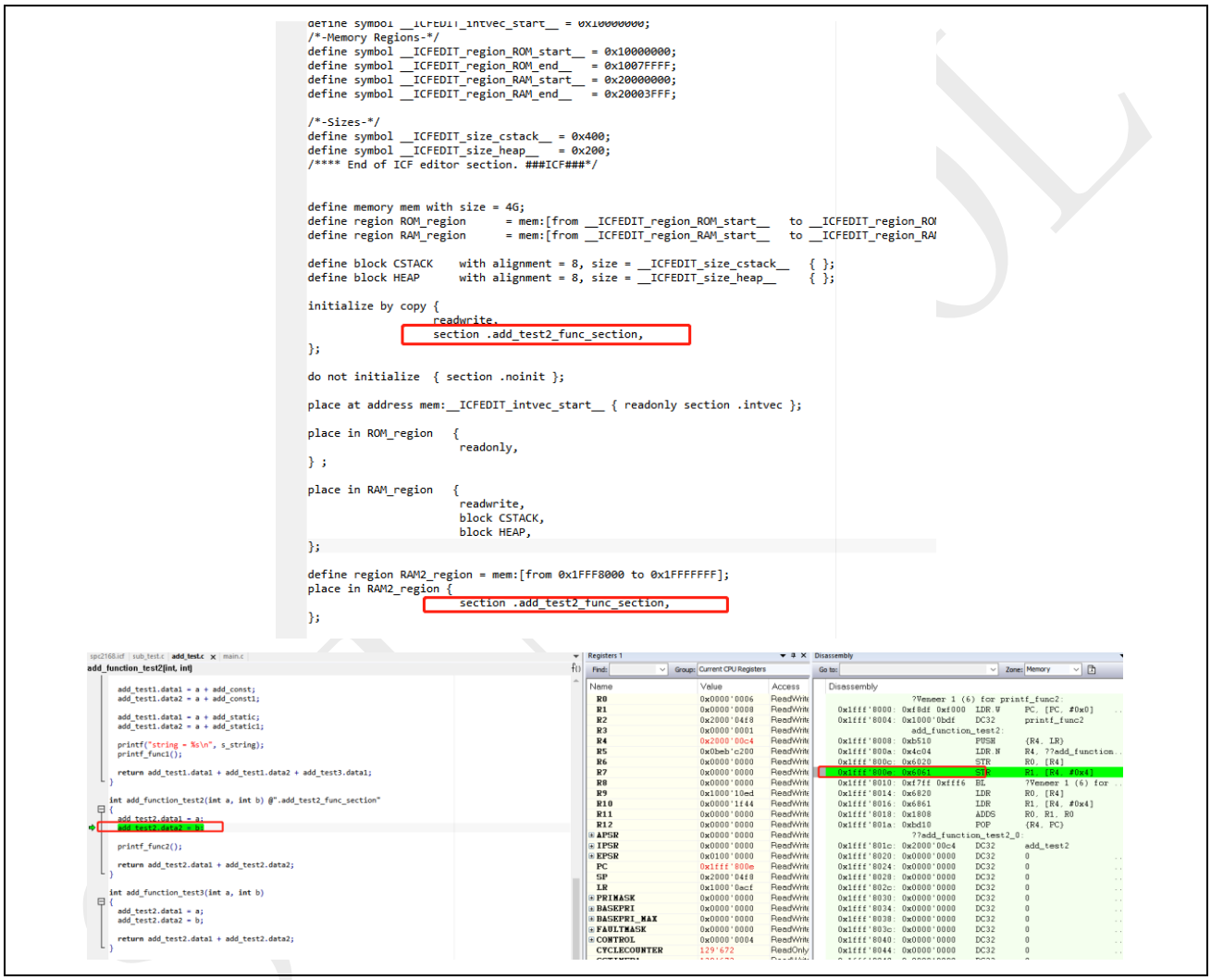
对函数代码而言，将被重定向到 ICF 文件中包含有”add_test2_func_sectio”标识符的 region。

对于全局变量而言，非 const 全局变量将会被重定向到 ICF 文件中包含有 readwrite 的 region；const 全局变量将会被重定向到 ICF 文件中包含有 readonly 的 region；

对于局部变量而言，非 static 局部变量将会被重定向到 ICF 文件中包含有 block CSTACK 的 region；static 局部变量将会被重定向到 ICF 文件中包含有 readwrite 的 region；

如图 7-2 所示，add_function_test2 函数被重定向到 0x1FFF8000~0x1FFFFFFF RAM2_region 地址，是因为 ICF 文件 RAM2_region 是包含有"add_test2_func_sectio"的 region。形参 a 以及 b 将会被重定向到 RAM_region 中，因为这个 region 包含有 block CSTACK。非 const 且非 static 的全局变量 add_test2 将会被重定向到 RAM2_region 中，因为这个 region 包含有 readwrite。

图 7-2：ICF 文件与重定向内容



注意： 该方法的重定向不适用中断服务函数及中断服务函数中调用的函数。

7.2 对多个函数进行重定向

对多个函数进行重定向到 ICF 文件中 place in 指令指定的”section .add_test1_func_section”的 Region 包含的地址范围内，在重定向的函数体中可以调用其他地址范围的函数，重定向多个函数时，重定向起始使用”#pragma default_function_attributes = @ "section_name"”，重定向结束使用”#pragma default_function_attributes =”。如果需要重定向变量可以用”default_variable_attributes”。

对函数代码而言，将被重定向到 ICF 文件中包含有”section_name”的 region。

对于全局变量而言，非 const 全局变量将会被重定向到 ICF 文件中包含有 readwrite 的 region；const 全局变量将会被重定向到 ICF 文件中包含有 readonly 的 region；

对于局部变量而言，非 static 局部变量将会被重定向到 ICF 文件中包含有 block CSTACK 的 region；static 局部变量将会被重定向到 ICF 文件中包含有 readwrite 的 region；

如图 7-3 所示，add_function_test2 和 add_function_test3 函数被重定向到 0x1FFF8000~0x1FFFFFFF RAM2_region 地址，是因为 ICF 文件 RAM2_region 是包含有”add_test1_func_section”的 region。形参 a 以及 b 将会被重定向到 RAM_region 中，因为这个 region 包含有 block CSTACK。非 const 且非 static 的全局变量 add_test2 将会被重定向到 RAM2_region 中，因为这个 region 包含有 readwrite。

图 7-3：ICF 文件与重定向内容

```
define symbol __ICFEDIT_intvec_start__ = 0x10000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x10000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x1007FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20003FFF;

/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x400;
define symbol __ICFEDIT_size_heap__   = 0x200;
/**** End of ICF editor section. ###ICF###*/


define memory mem with size = 4G;
define region ROM_region      = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region      = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy {
                      readwrite,
                      section .add_test1_func_section,
};

do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   {
                       readonly,
} ;

place in RAM_region   {
                       readwrite,
                       block CSTACK,
                       block HEAP,
};

define region RAM2_region = mem:[from 0x1FFF8000 to 0x1FFFFFFF];
place in RAM2_region {
                       section .add_test1_func_section,
};
```

```
#pragma default_function_attributes  = @ ".add_test1_func_section"
int add_function_test2(int a, int b)
{
  add_test2.data1 = a;
  add_test2.data2 = b;

  printf_func2();

  return add_test2.data1 + add_test2.data2;
}

int add_function_test3(int a, int b)
{
  add_test2.data1 = a;
  add_test2.data2 = b;

  return add_test2.data1 + add_test2.data2;
}

#pragma default_function_attributes  =
```

```
0x1fff'8004: 0x1000'0be7   DC32     printf_func2
                 add_function_test2:
0x1fff'8008: 0xb510        PUSH     {R4, LR}
0x1fff'800a: 0x4c50        LDR.N    R4, ??DataTable16
0x1fff'800c: 0x6020        STR      R0, [R4]
0x1fff'800e: 0x6061        STR      R1, [R4, #0x4]
0x1fff'8010: 0xf7ff 0xfff6 BL       ?Vemeer 2 (6) for ...
0x1fff'8014: 0x6820        LDR      R0, [R4]
0x1fff'8016: 0x6861        LDR      R1, [R4, #0x4]
0x1fff'8018: 0x1808        ADDS     R0, R1, R0
0x1fff'801a: 0xbd10        POP      {R4, PC}
                 add_function_test3:
0x1fff'801c: 0x4a03        LDR.N    R2, ??DataTable16 ...
0x1fff'801e: 0x6010        STR      R0, [R2]
0x1fff'8020: 0x6051        STR      R1, [R2, #0x4]
0x1fff'8022: 0x6810        LDR      R0, [R2]
0x1fff'8024: 0x6851        LDR      R1, [R2, #0x4]
0x1fff'8026: 0x1808        ADDS     R0, R1, R0
0x1fff'8028: 0x4770        BX       LR
0x1fff'802a: 0x0000        MOVS     R0, R0
                 ??DataTable16:
0x1fff'802c: 0x2000'00c4   DC32     add_test2
0x1fff'8030: 0x0000'0000   DC32     0
0x1fff'8034: 0x0000'0000   DC32     0
0x1fff'8038: 0x0000'0000   DC32     0
```

注意： 使用#pragma 方法无法重定向中断服务函数及中断服务函数调用的函数，仅__ramfunc 关键字可以重定向中断服务函数及中断服务函数调用的函数。

7.3 对整个文件进行重定向

假设目标文件为 test1.o、test2.o，将两个文件中的数据及代码进行重定向，此时将需要用到 place in 指令，并且在重定向的文件中可以调用其他地址范围的函数。

对代码而言，将被重定向到 ICF 文件中包含有”section .text object test1.o”和”section .text object test2.o”的 region。

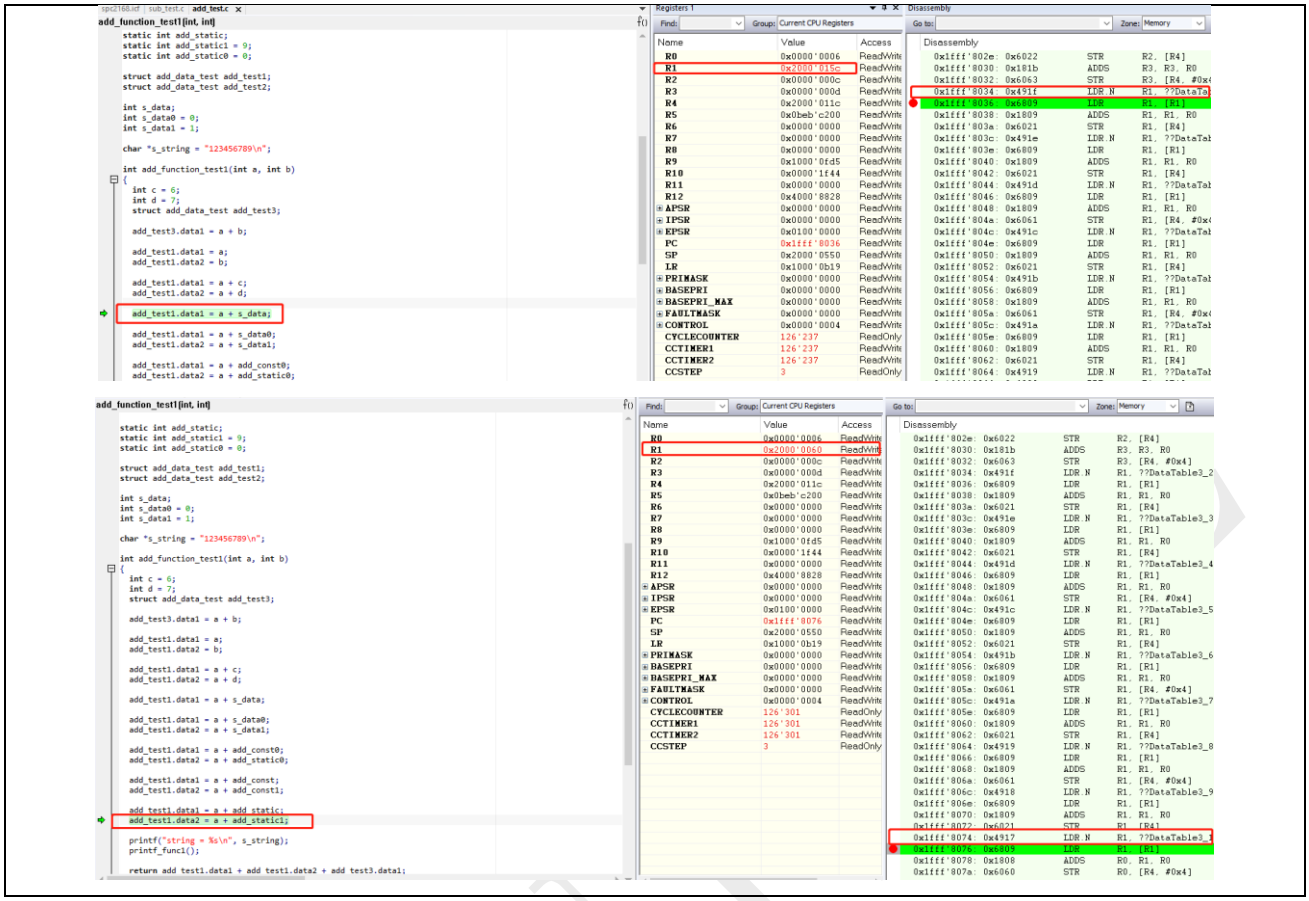
对于全局变量而言，非 const 全局变量将会被重定向到 ICF 文件中包含有 readwrite 的 region；const 全局变量将会被重定向到 ICF 文件中包含有”section .rodata object test1.o”和”section .rodata object test2.o”的 region；

对于局部变量而言，非 static 局部变量将会被重定向到 ICF 文件中包含有 block CSTACK 的 region；static 局部变量将会被重定向到 ICF 文件中包含有 readwrite 的 region；

如图 7-4 所示，程序代码被重定向到 0x1FFF8000~0x1FFFFFFF 的 RAM 地址，是因为 ICF 文件 RAM2_region 是包含有 section .text object add_test.o 和 section .text object sub_test.o 的 region。形参 a 以及 b 将会被重定向到 RAM_region 中，因为这个 region 包含有 block CSTACK。非 const 且非 static 和 static 的全局变量 s_data 和 add_static1 将会被重定向到 RAM2_region 中，因为这个 region 包含有 readwrite。const 的全局变量 add_const1 将会被重定向到 RAM2_region 中，因为这个 region 包含有”section .rodata object test1.o”。

图 7-4：ICF 文件与重定向内容

```
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x10000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x1007FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20003FFF;

/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x400;
define symbol __ICFEDIT_size_heap__   = 0x200;
/**** End of ICF editor section. ###ICF###*/


define memory mem with size = 4G;
define region ROM_region      = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region      = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy {
                readwrite,
                section .text object add_test.o,
                section .rodata object add_test.o,

                section .text object sub_test.o,
                section .rodata object sub_test.o,
};


do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   {
                        readonly,
} ;

place in RAM_region   {
                        readwrite,
                        block CSTACK,
                        block HEAP,
                        section .rodata object add_test.o,
                        section .rodata object sub_test.o,
};


define region RAM2_region = mem:[from 0x1FFF8000 to 0x1FFFFFFF];
place in RAM2_region {
                section .text object add_test.o,
                section .text object sub_test.o,
};
```

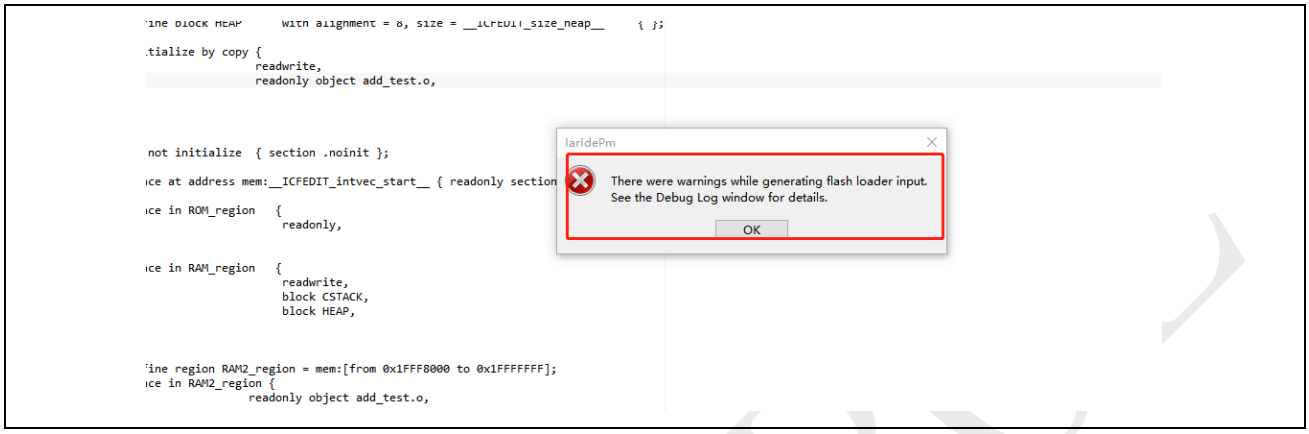


注意： 该方法的重定向不适用中断服务函数及中断服务函数中调用的函数。

7.4 重定向失败错误提示

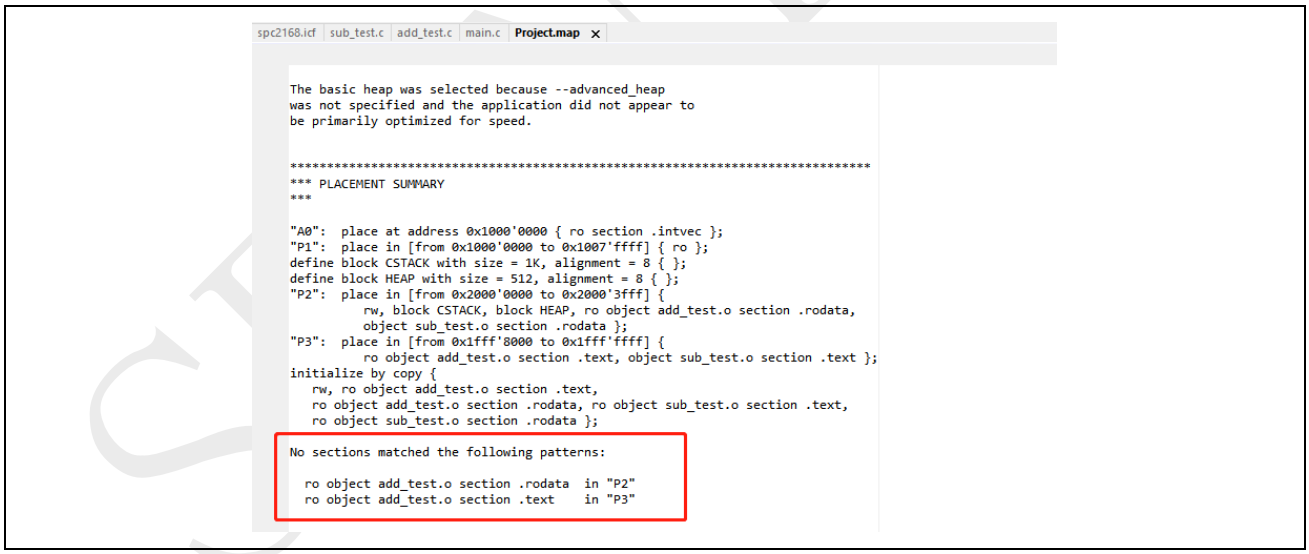
1. 编写 ICF 文件后点击 Download and Debug 时，如发生图 7-5 所示情况，有可能是因为 ICF 配置错误导致重定向失败，需要检查 ICF 文件。

图 7-5：IAR 错误弹框窗口



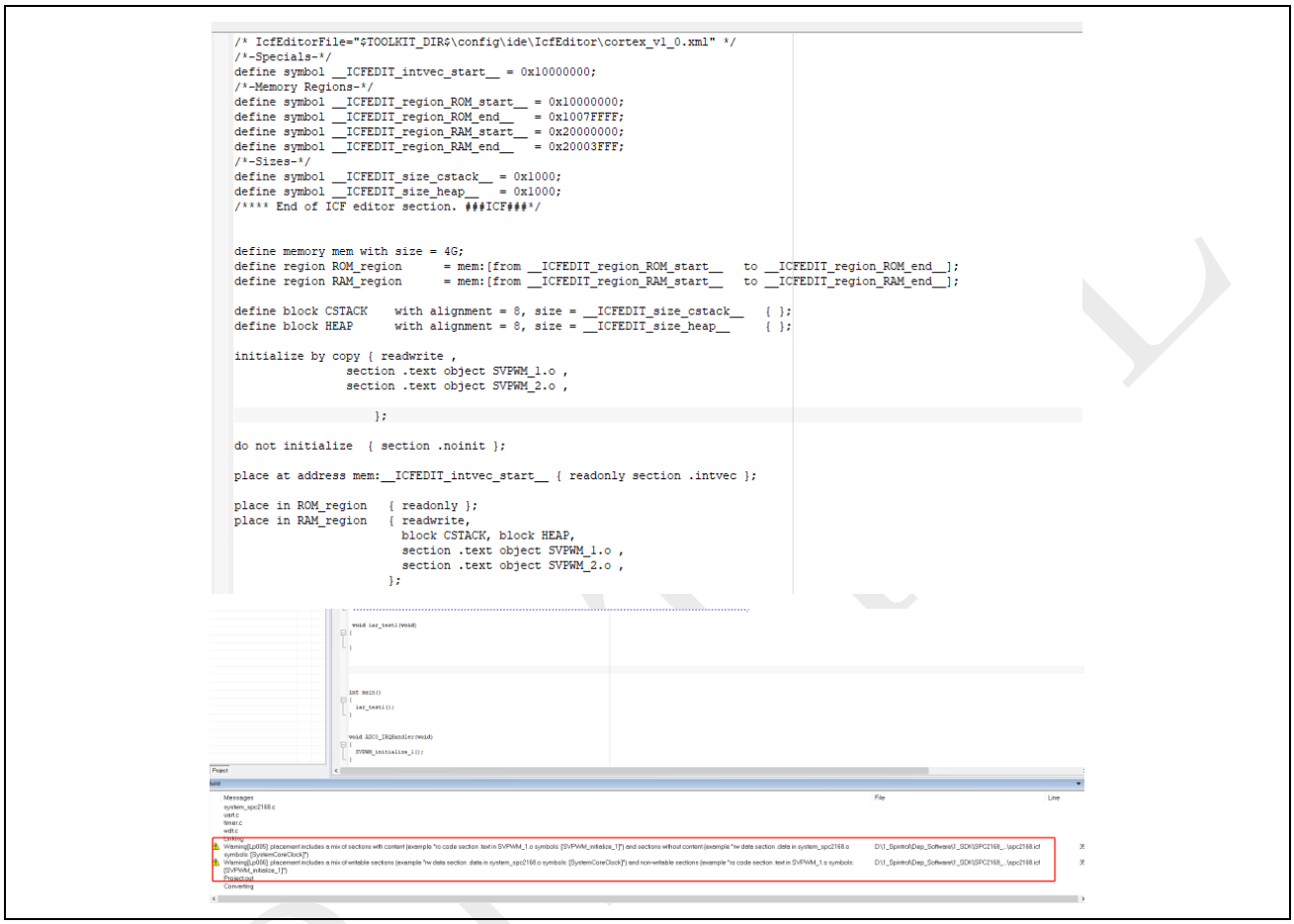
2. 如果点击 Download and Debug 没有产生错误弹框，重定向还是失败时，可以检查 Project.map 文件如果如图 7-6，需要重新检查 ICF 文件。

图 7-6：IAR 错误信息



3. 在编译程序时，如出现产生图 7-7 所示的 IAR 警告，可能是由于中断服务函数调用通过 ICF 文件配置的重定向函数导致 IAR Link 失败，则需要将中断服务函数调用在 ICF 重定向的函数前加上”__ramfunc”关键字，如图 7-8 所示。

图 7-7：IAR ICF 文件警告信息



```
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x10000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x10000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x1007FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20003FFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x1000;
define symbol __ICFEDIT_size_heap__   = 0x1000;
/**** End of ICF editor section. ###ICF###*/


define memory mem with size = 4G;
define region ROM_region      = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region      = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite ,
                     section .text object SVPWM_1.o ,
                     section .text object SVPWM_2.o ,

                   };

do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
                        block CSTACK, block HEAP,
                        section .text object SVPWM_1.o ,
                        section .text object SVPWM_2.o ,
                      };
```

图 7-8：IAR 示例

```
main.c | SVPWM_1.c | SVPWM_2.c | spc2168.icf

#include <spc2168.h>

#include "SVPWM_1.h"


/* Model initialize function */
__ramfunc void SVPWM_initialize_1(void)
{

}
```