SPC11XX 环境调试指南

2022 年 1 月 – 版本 1

概述

SPC11X8/SPD11X8 是旋智第三代 MCU，使用 ARM 的 Cortex-M4F 核心,支持 JLINK 下载调试，开发者可以通过 Keil 或者 IAR 工具编码进行开发或者在线调试，可以使用 Jscope 或者 SeggerRTT 等调试手段，并且支持 JFLASH，ISP 等下载工具烧录代码。本文主要介绍如何利用 Keil 或者 IAR 搭配 JLINK 进行烧录代码并且利用 Jscope 或者 RTT 进行调试等功能。

1 目录

表格列表

版本历史

表 7-1：文档修订记录

日期	版本	修改内容	作者
2022-3-10	1	初始版本	Bohuang（bo.huang@spintrol.com）

1 J-LINK 与 SPC11X8/SPD11X8 连接

1.1 JLINK 介绍

J-Link 是 SEGGER 公司为支持仿真 ARM 内核芯片推出的 JTAG 仿真器。简单地说，是给一个 JTAG 协议转换盒。其连接到计算机用的是 USB 接口，而到目标板内部用的还是 jtag 协议。它完成了一个从软件到硬件转换的工作。配合 IAR EWAR， ADS， KEIL， WINARM， RealView 等集成开发环境支持所有 ARM7/ARM9/ARM11，Cortex M0/M1/M3/M4， Cortex A5/A8/A9 等内核芯片的仿真，与 IAR，Keil 等编译环境无缝连接，操作方便、连接方便、简单易学，是学习开发 ARM 最好最实用的开发工具。JLINK 仿真器目前已经升级到 V9.1 版本，其仿真速度和功能远非简易的并口 WIGGLER 调试器可比。J-LINK 支持 ARM7/ARM9/ARM11，Cortex M0/M1/M3/M4， Cortex A4/A8/A9 等内核芯片，支持 ADS、IAR、KEIL 开发环境。V9.3 版本较 V8.0 版本进一步提升了下载速度，最大下载速度提升到 1 MByte/s.

图 1- 1. JLINK

LINK 适配器支持 2 种接口，如图 1- 2 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 1- 1 所示。

官网地址为：https://www.segger.com/downloads/jlink/#J-LinkSoftwareAndDocumentationPack

图 1- 2. J-LINK 接口

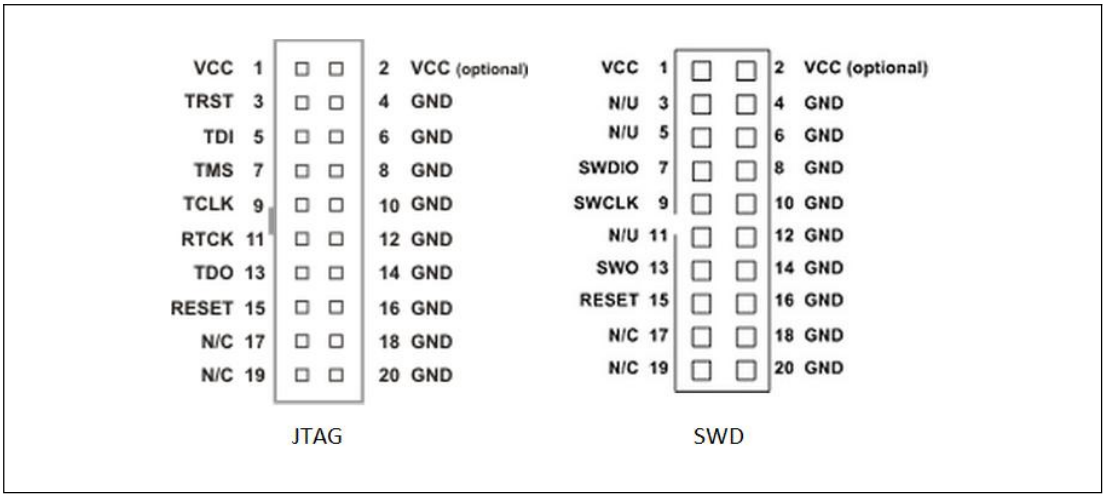


表 1-1. SW 接口信号定义

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.

1.2 JLINK 连接

在采用 SPC11X8/SPD11X8 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。以 SPC1168 开发板为例，J-LINK 与 SPC11X8/SPD11X8 的硬件连接如**错误!未找到引用源。**所示。表 1-2 中为具体的 PIN 脚连接关系。

图 1-3. J-LINK 与 SPC11X8/SPD11X8 实物连接（以 SPC1168 开发板为例）

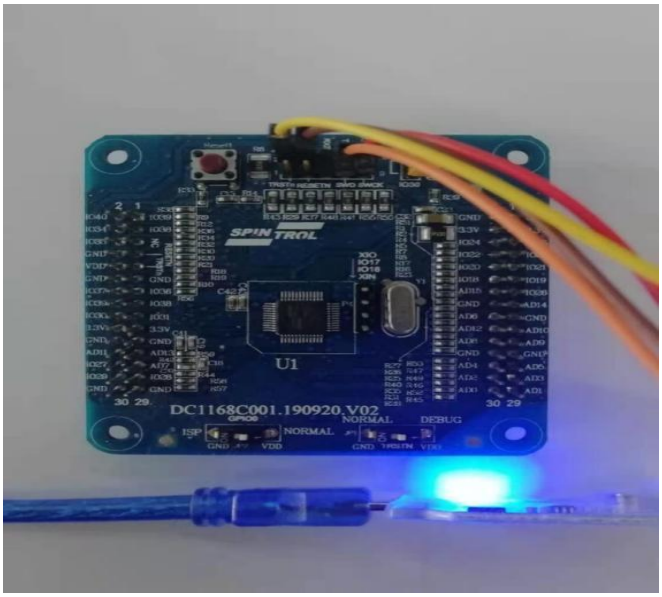


表 1- 2. J-LINK 与 SPC11X8/SPD11X8 管脚连接

J-LINK	SPC11X8/SPD11X8
SWDIO	SWDIO(GPIO38)
SWCLK	SWCLK(GPIO39)
VCC	VDD(3.3V)
GND	GND
Vbat	使用 SPD11XX 系列的 MCU（SPD1178 除外），Vbat > 5.5V， 才能正常下载
TRSTn	高电平（3.3V）
Boot	高电平（3.3V）

注意：（见《SPC11XX datasheet》的启动模式章节介绍）

启动程序位于片上 ROM。复位后，ARM 处理器从 ROM 开始执行程序。通过 BOOT 引脚和 TRSTn 引脚来选择两种启动模式：

- Flash 启动（BOOT 引脚 = 1，TRSTn 引脚 = X）：启动加载器跳转至嵌入式 Flash 并从地址 0x1000 0000 开始执行。
- ISP 启动（BOOT 引脚 = 0，TRSTn 引脚 = 0）：启动加载器通过 UART 对嵌入式 Flash 进行重新编程。在这个过程中，GPIO34 被配置为 UART_TXD 功能；GPIO35 被配置为 UART_RXD 功能。

注意 1：Boot 引脚可以被配置作为 GPIO 使用，在客户应用中仅可作为输出使用。当器件复位时，需确保该引脚为高电平，否则器件将进入 ISP 模式。

注意 2：无论何时，当 Boot 引脚为低时，务必保证 TRSTn 引脚为低，否则芯片会进入工程测试模式，无法正常工作。

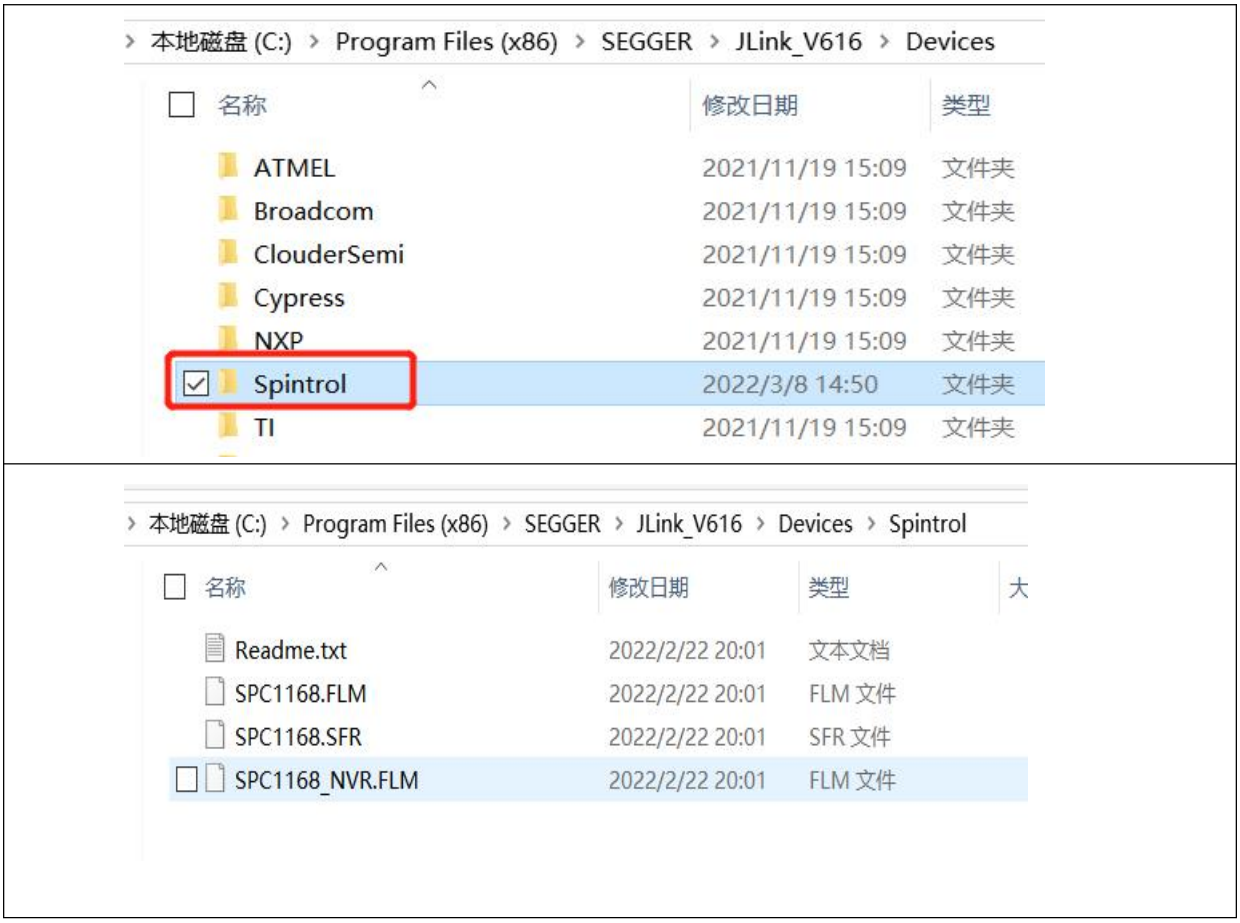
1.3 JLINK 配置 Device

如果需要使用到 JFLASH, RTT , J-Scope 等工具，则需要进行此配置。

1.1.1 JLINK 添加 Device 文件

在安装 JLINK 时候时，软件会默认安装在如下目录：C:\Program Files (x86)\SEGGER\JLink_VXX，打开 C:\Program Files (x86)\SEGGER\JLink_VXX\Devices 目录，需要将旋智开发库 V1_x\IDE_Support\MDK-ARM 文件夹复制到此目录下，并重命名为 Spintrol。此文件夹包含下载所需要算法文件， 即为 SPC1XXX.FLM /SPD1XXX.FLM 文件以及 SPC11XX_NVR.FLM/SPD11XX_NVR.FLM 文件。

图 1- 4. JLink Devices 文件



1.1.2 JLINK 配置修改 JLinkDevices.xml

打开 C:\Program Files (x86)\SEGGER\JLink_V616\JLinkDevices.xml 文件夹。添加如图 1-5 Device 节点到 JLinkDevices.xml 的末尾。

注意要在 JLinkDevices.xml 的末尾的</Device>与</DataBase>之间添加内容。

图 1-5. JLinkDevices.xml 配置 Device 节点

```
</Device>
<Device>
<ChipInfo Vendor="Spintrol" Name="SPC1158" WorkRAMAddr="0x20000000"
    WorkRAMSize="0x4000" Core="JLINK_CORE_CORTEX_M4"/>
<FlashBankInfo Name="Internal Flash(NVR)" BaseAddr="0x11000400" MaxSize="0x400"
    Loader="Devices/Spintrol/SPC1168_NVR.FLM" LoaderType="FLASH_ALGO_TYPE_OPEN"
    AlwaysPresent="1"/>
<FlashBankInfo Name="Internal Flash(Main)" BaseAddr="0x10000000" MaxSize="0x10000"
    Loader="Devices/Spintrol/SPC1168.FLM" LoaderType="FLASH_ALGO_TYPE_OPEN"
    AlwaysPresent="1"/>
</Device>
<Device>
<ChipInfo Vendor="Spintrol" Name="SPC1168" WorkRAMAddr="0x20000000"
    WorkRAMSize="0x4000" Core="JLINK_CORE_CORTEX_M4"/>
<FlashBankInfo Name="Internal Flash(NVR)" BaseAddr="0x11000400" MaxSize="0x400"
```

```
        Loader="Devices/Spintrol/SPC1168_NVR.FLM"          LoaderType="FLASH_ALGO_TYPE_OPEN"
        AlwaysPresent="1"/>
<FlashBankInfo    Name="Internal    Flash(Main)"    BaseAddr="0x10000000"    MaxSize="0x10000"
        Loader="Devices/Spintrol/SPC1168.FLM"              LoaderType="FLASH_ALGO_TYPE_OPEN"
        AlwaysPresent="1"/>
</Device>
<Device>
<ChipInfo          Vendor="Spintrol"          Name="SPC2168"          WorkRAMAddr="0x20000000"
        WorkRAMSize="0x4000" Core="JLINK_CORE_CORTEX_M4"/>
<FlashBankInfo    Name="Internal    Flash(NVR)"          BaseAddr="0x11000400"    MaxSize="0x400"
        Loader="Devices/Spintrol/SPC2168.FLM"              LoaderType="FLASH_ALGO_TYPE_OPEN"
        AlwaysPresent="1"/>
<FlashBankInfo    Name="Internal    Flash(Main)"    BaseAddr="0x10000000"    MaxSize="0x80000"
        Loader="Devices/Spintrol/SPC2168.FLM"              LoaderType="FLASH_ALGO_TYPE_OPEN"
        AlwaysPresent="1"/>
</Device>
</DataBase>
```

注：

- <Device> ... </Device> 表示一个节点，对应一款 MCU
- <ChipInfo ... />表示 MCU 信息 <FlashBankInfo />表示 flash 的 bank 信息
- Vendor 制造厂为 spintrol
- Name 为 MCU 名称
- WorkRAMAddr 为 mcu ram 起始地址，WorkRAMSize 为 MCU ram 大小
- Core 为 MCU 内核类型

  SPC11XX/SPD11XX 系列 MCU 为 JLINK_CORE_CORTEX_M4 类型。
- BaseAddr 为 flash 基址，MaxSize 为 flash 大小，

  SPC11XX_NVR.FLM 的 fash 基址为 0x11000400，MaxSize 为 0x400
- Loader 为 JLINK 配置烧录文件路径，LoaderType 为 FLASH_ALGO_TYPE_OPEN 类型
- alwaysPresent 默认为 1.

最终配置完成如图 1- 6 显示如下：

图 1- 6.JLinkDevices.xml 末尾图

2 Keil 环境搭建

2.1 KEIL 环境下 J-LINK 配置

在安装 KEIL MDK 时，软件会默认安装 J-LINK 设备的驱动。按照表 1-1 将 J-LINK 与 SPC11X8/SPD11X8 连接，然后给芯片上电。这时打开 KEIL 软件，鼠标左键单击图标，弹出界面如下：

Target 的 IROM 与 IRAM 地址大小请根据芯片手册的内存映射来进行配置。

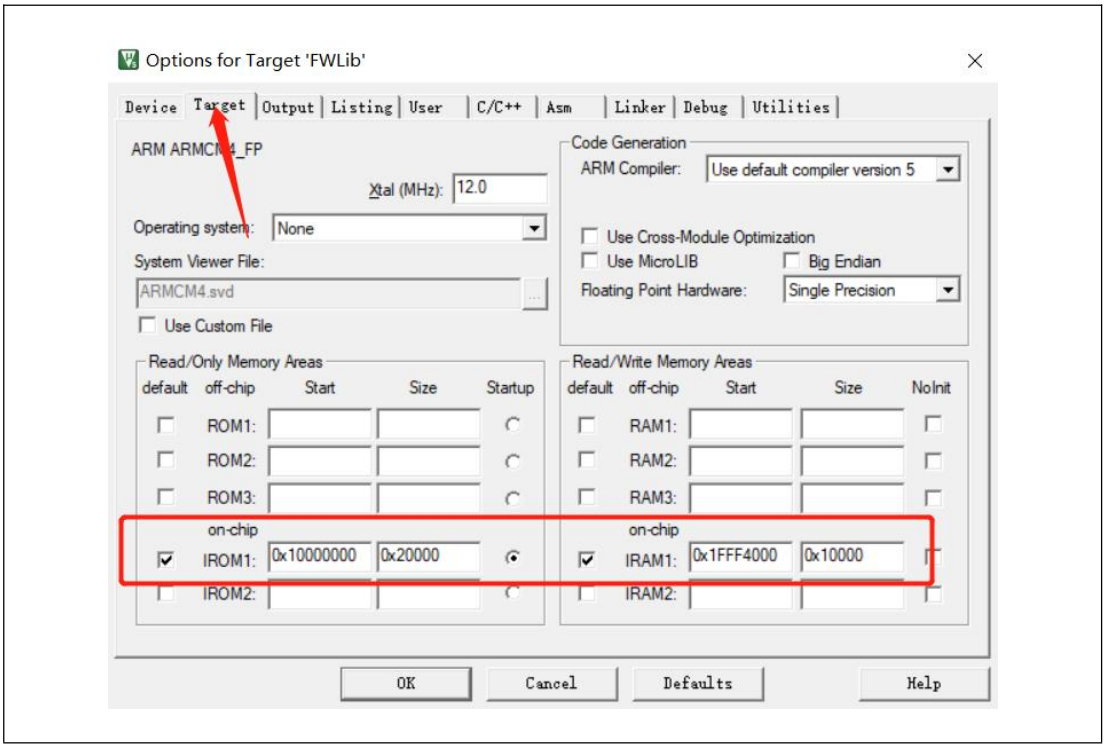
比如内存映射图 2-25 所示:

IROM（既 flash 地址）地址 0x1000 0000, SIZE: 0x20000(128K)

IRAM 地址 0x1FFF 4000, SIZE: 0x10000 (64K)

则需要按如下所配置:

图 2-1. Options for Target 对话框



选择 Debug 选项卡，会看到如图 2-2 所示的界面。红色矩形框标记的内容是 Debug 时需要设置的选项。

图 2-2. Debug 配置界面

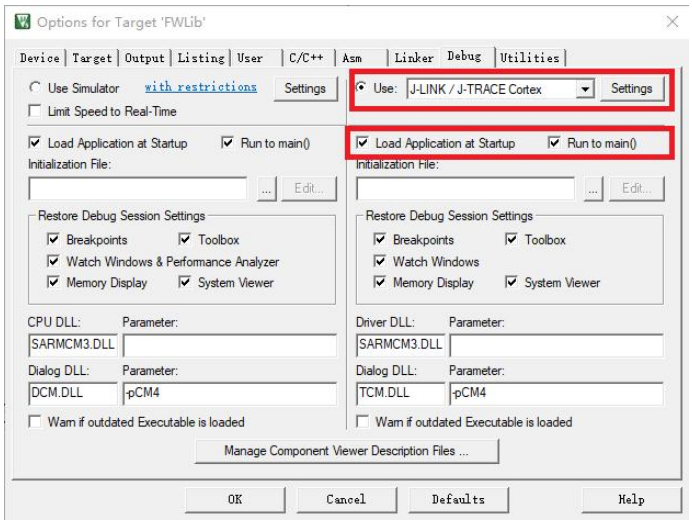
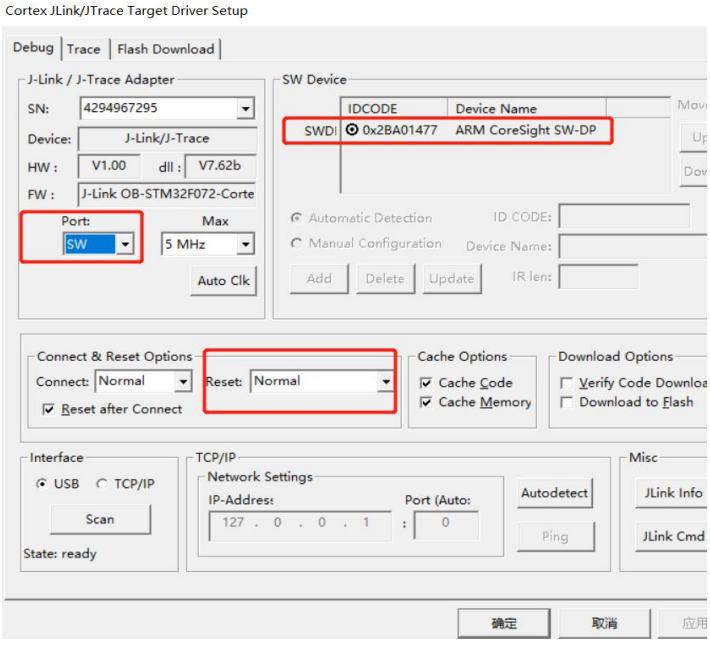


图 2-2 所示界面中，左侧是仿真调试相关的配置选项，右侧则是与硬件调试相关的选项。根据实际情形，选择使用 J-LINK/J TRACE Cortex 选项。单击 Settings 按钮，会弹出与 J-LINK 相关的设置，如图 2-3 所示。可以看到，红色矩形框中出现 Debug targets 的信息，表明 J-LINK 设备此时是正常工作的；否则，则表明 J-LINK 设备不可用。因此，在用 J-LINK 调试程序时，常常用此方法检查 J-LINK 设备是否正常。此外，建议用户按照图 2-3 配置 Connect & Reset Options，Reset 方式选择 Core and Peripheral。此外，SPC11X8/SPD11X8 芯片支持 JTAG 和 SWD 两种 Debug 协议，用户可以根据需要进行配置。

图 2-3. J-LINK 设置对话框



在使用 J-LINK 调试程序之前，还需要设置 Flash Download 选项，如图 2-4 所示。其中，SPC11X8/SPD11X8 Programming Algorithm 可以通过点击 Add 按钮来添加，如图 2-5 所示。

（注意：需要将 V1_x\IDE_Support\MDK-ARM 目录下的 SPC1168.FLM 文件复制到 KEIL 软件安装路径下的目录 Keil_v5\ARM\Flash\）

图 2-4. Flash Download 设置

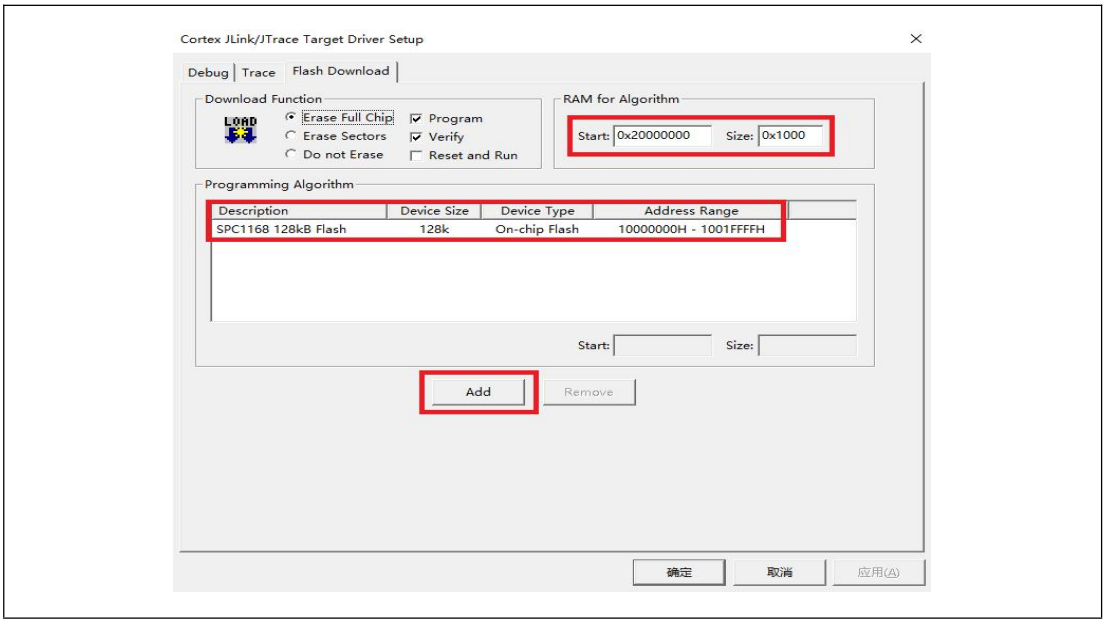
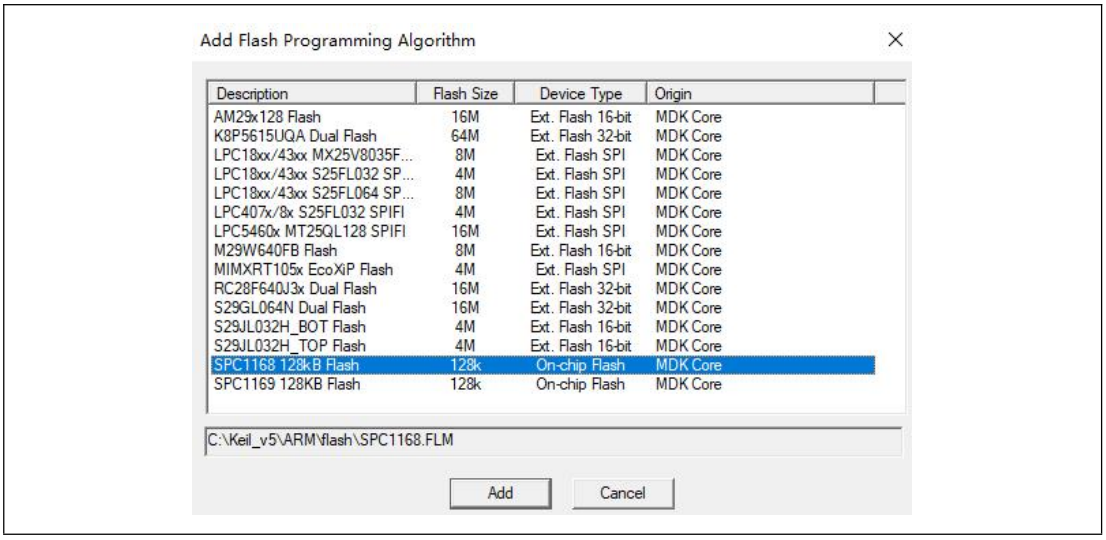
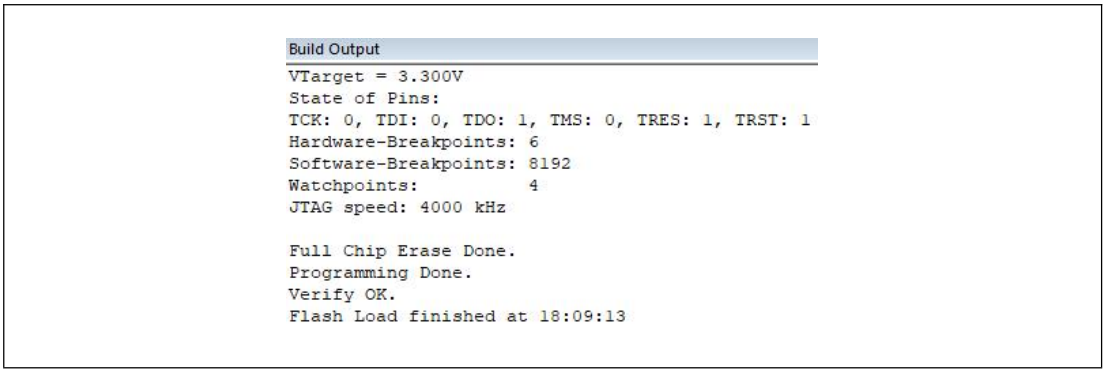


图 2-5. Add Flash Programming Algorithm



Flash Download 设置完成之后，将应用程序编译，然后点击 KEIL 软件工具栏上的LOAD按钮，就可以将应用程序下载到芯片中。用户可以在 Build Output 窗口中查看具体的 Download 过程信息，如图 2-6 所示。

图 2-6. Build Output 窗口信息

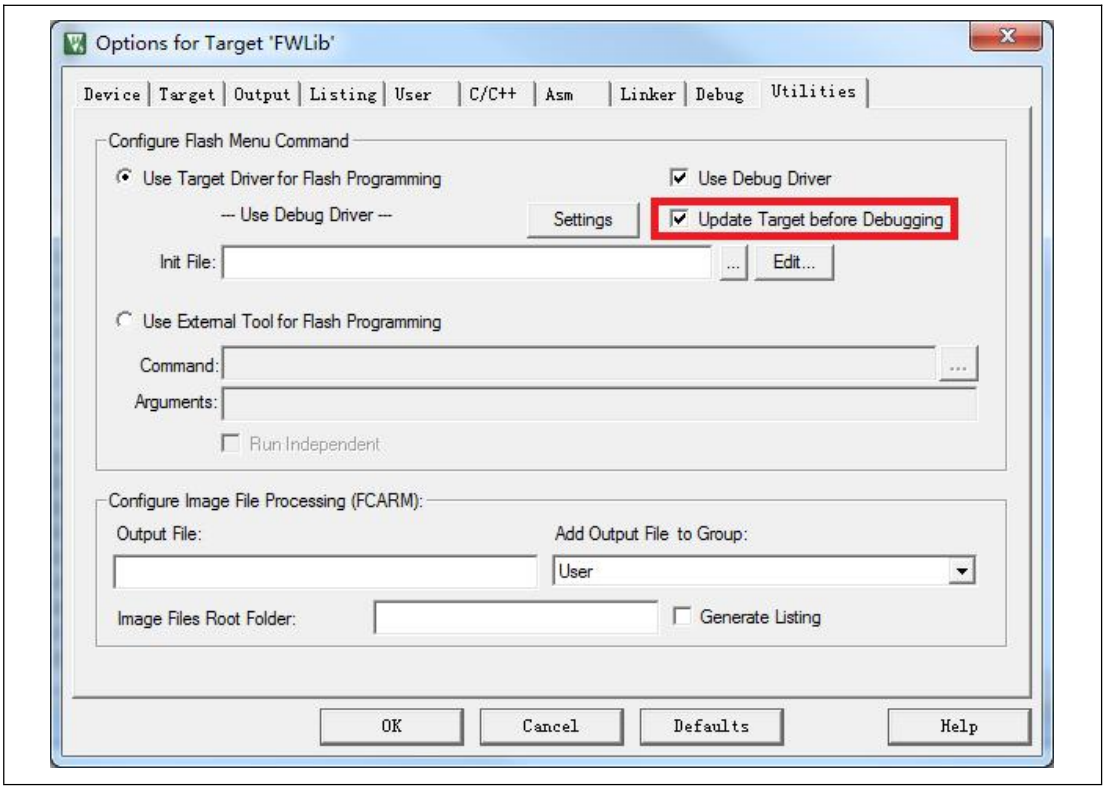


接下来，在图 2-2 中，我们看到有两个选项：Load Application at Startup 和 Run to main()。其中 Load Application at Startup 选项是必须要勾选的，Run to main()选项根据需要决定要不要勾选。

2.2 KEIL 环境下使用 J-LINK 调试

根据前面的介绍，将 J-LINK 设备与 SPC11X8/SPD11X8 正确连接后，按照图 2-7 以及图 2-8 设置 Debug 的相关选项，就可以使用 J-LINK 设备调试程序了。使用 J-LINK 调试程序时，必须保证 Flash 存储器中的程序与当前程序一致。这就需要用户每次修改代码后，都要点击  按钮将程序下载到 Flash 存储器中。值得一提的是，KEIL 软件提供了一个功能，可以自动上述动作，如图 2-7 所示。用户只需勾选 Update Target before Debugging 选项，那么在每次启动 Debug 会话时，KEIL 软件会自动通过 J-LINK 设备将程序下载到 Flash 中，从而保证了 Flash 中的程序与当前调试的程序一致。

图 2-7. Update Target before Debugging 设置



单击工具栏上的  按钮进入 Debug 状态，程序界面如**错误!未找到引用源。**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，KEIL 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，黄色底纹的汇编代码对应于用户代码窗口中光标所在位置的 C 代码；此外，菜单栏上也出现了一些与 Debug 相关的菜单选项，如表 2-1 所示。

注意：在程序进入 Debug 状态后，代码是不可以修改的。如果想修改代码，需要单击按钮  退出 Debug 模式，然后才能修改代码。修改后的代码编译通过后，将代码重新下载到 Flash 中，用户可以继续单击按钮  进行 Debug。

图 2-8. 启动 Debug 后的界面

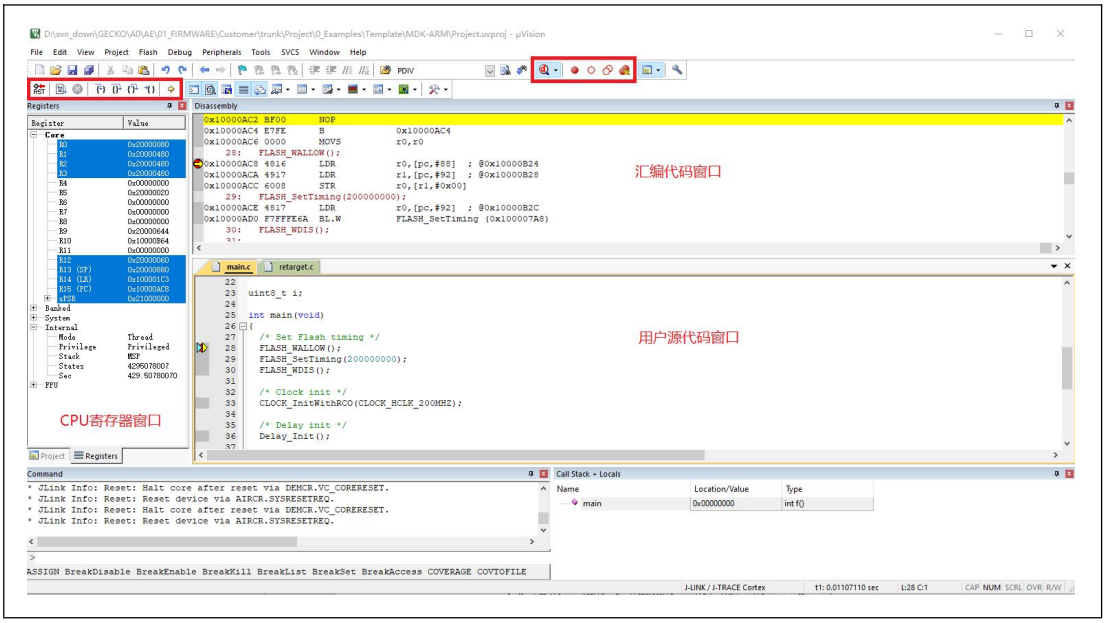


表 2-1. Debug Menu and Commands

Debug Menu	Toolbar	Shortcut	Description
Start/Stop Debug Session		**Ctrl+F5**	Starts or stops a debugging session.
Reset CPU			Sets the CPU to RESET state.
Run		**F5**	Continues executing the program until the next active breakpoint is reached.
Stop			Stops the program execution immediately.
Step		**F11**	Executes a single-step into a function; Executes the current instruction line.
Step Over		**F10**	Executes a single-step over a function.
Step Out		**Ctrl+F11**	Finishes executing the current function and stops afterwards.
Run to Cursor Line		**Ctrl+F10**	Executes the program until the current cursor line is reached.
Show Next Statement			Shows the next executable statement/instruction.
Breakpoints		**Ctrl+B**	Opens the dialog Breakpoints.
Insert/Remove Breakpoint		**F9**	Toggles the breakpoint on the current line.
Enable/Disable Breakpoint		**Ctrl+F9**	Enables/disables the breakpoint on the current line.
Disable All Breakpoints			Disables all breakpoints in the program.
Kill All Breakpoints		**Ctrl+Shift+F9**	Removes all breakpoints in the program.

2.2.1 单步调试

单击工具栏上的按钮后，程序进入 Debug 会话状态。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有两个三角箭头：表示当前光标所在的位置；表示当前位置的代码为下一次要执行的语句。因此，可以通过这两个三角箭头快速判断程序执行到哪条语句以及光标的位置。

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。

2.2.2 断点设置

当启动 Debug 会话后，在用户代码所在行左侧边框处单击鼠标左键，即可快速地设置断点，此时左侧边框会出现一个红色的圆形标记，如图 2-9 所示。当然，也可以通过工具栏上的按钮设置断点，具体做法是：将光标定位到欲设置断点的代码行，然后单击按钮，即可设置断点。此时，单击工具栏上的按钮或者按下快捷键 F5，程序就会执行起来，一直执行到断点处停下来，如 图 2-10 所示。

图 2-9. 设置断点

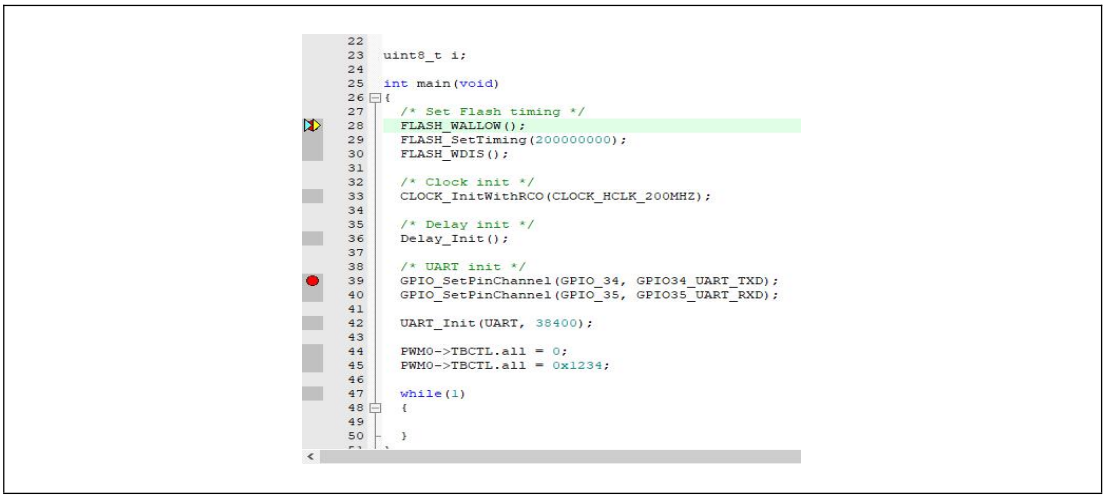
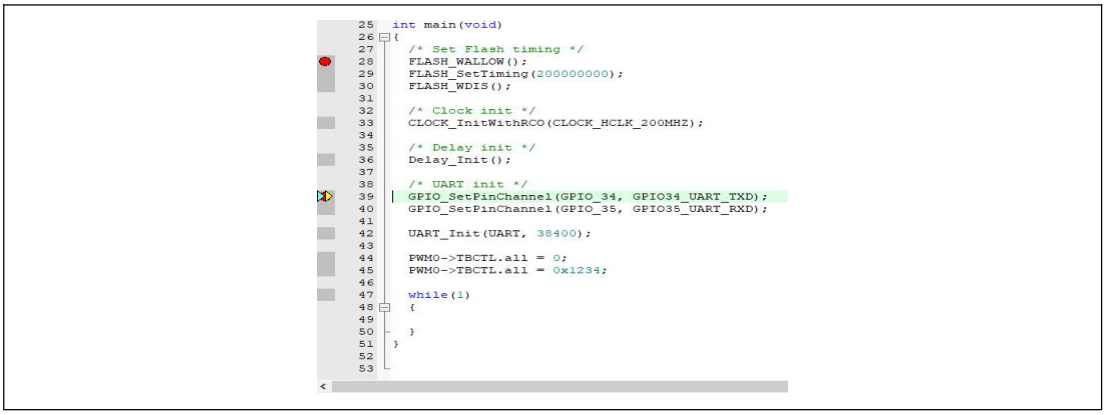


图 2-10. 程序执行到断点



设置断点除了可以让程序快速的执行到某个位置外，在 Debug 程序时也非常有用。例如，可以

在中断服务函数中设置断点，用来判断相应的中断是否发生。

2.2.3 观察变量值

在调试程序的时候，常常需要观察变量的值。这个可以通过 KEIL 软件提供的 Watch Window 来实现。具体实现过程如下：将光标定位到要观察的变量（光标移到变量名左侧），单击鼠标右键，在弹出的快捷菜单中即可将变量添加到观察窗口中，如图 2-11 所示。

从图 2-11 可以看出，我们将变量 i 添加到观察窗口 Watch1 里，结果如图 2-12 所示。从图中可以看出在代码区下方出现了 Watch1 窗口，在 Watch1 中可以看到刚刚添加的变量 i。

图 2-11. 添加变量到观察窗口

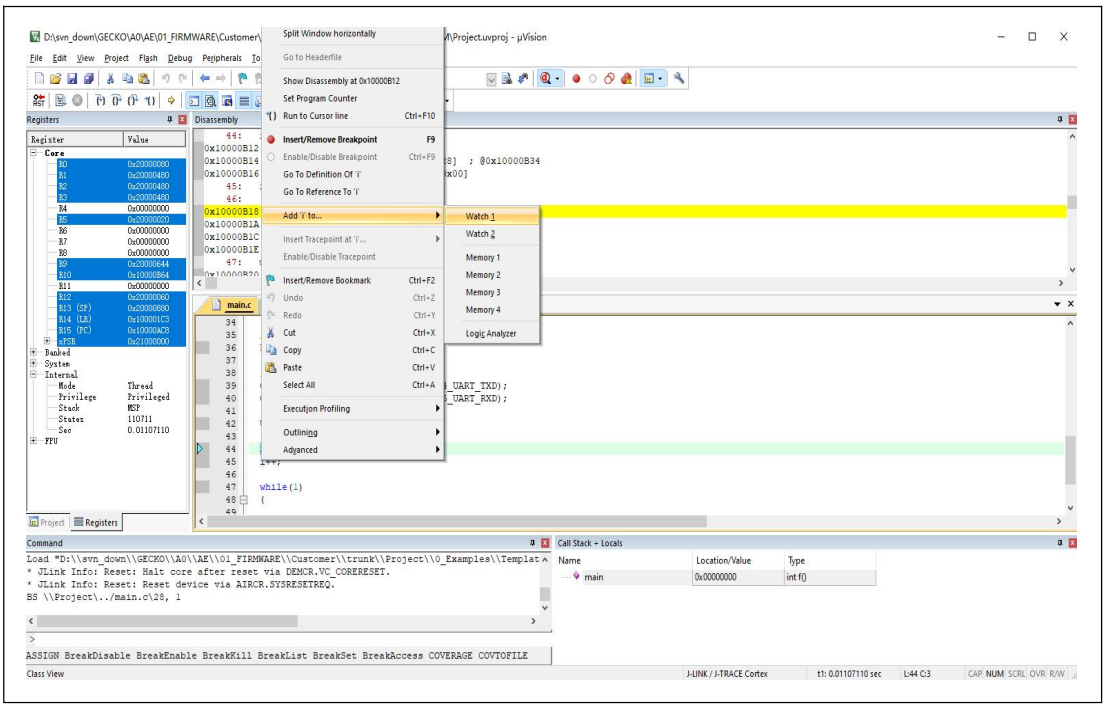
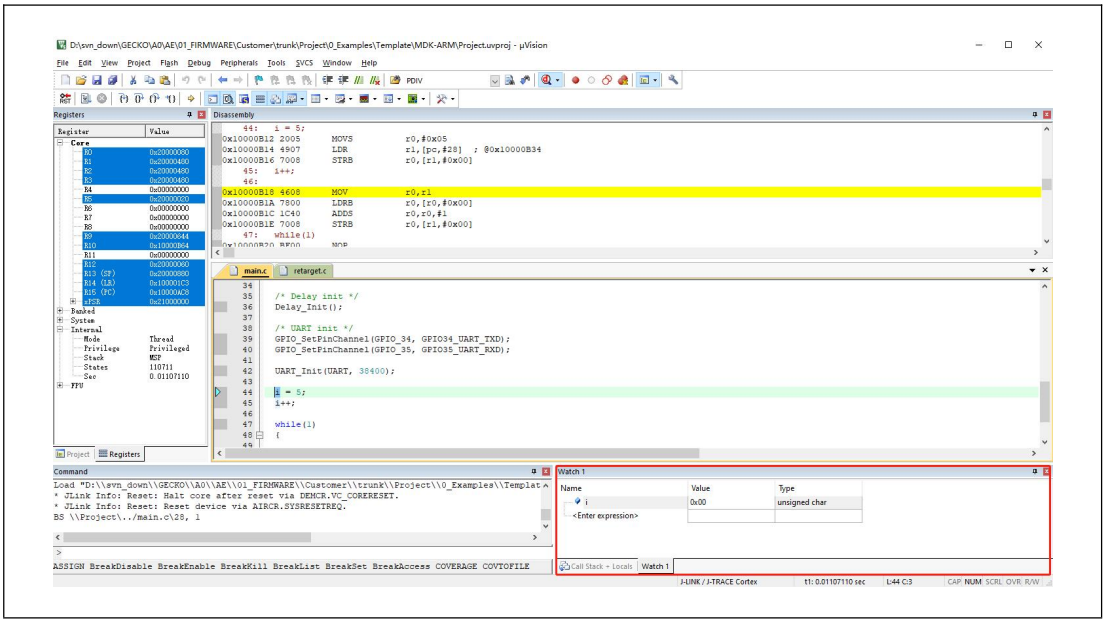


图 2-12. 添加变量到观察窗口的结果



接下来，我们就通过单步执行观察变量 i 的值。

第一步：单步执行语句 i = 5，执行结果如图 2-13 所示，可以看出变量 i 的值变为 5；

第二步：单步执行语句 i++，执行结果如图 2-14 所示，可以看出变量 i 的值变为 6。

图 2-13. i=5 执行结果

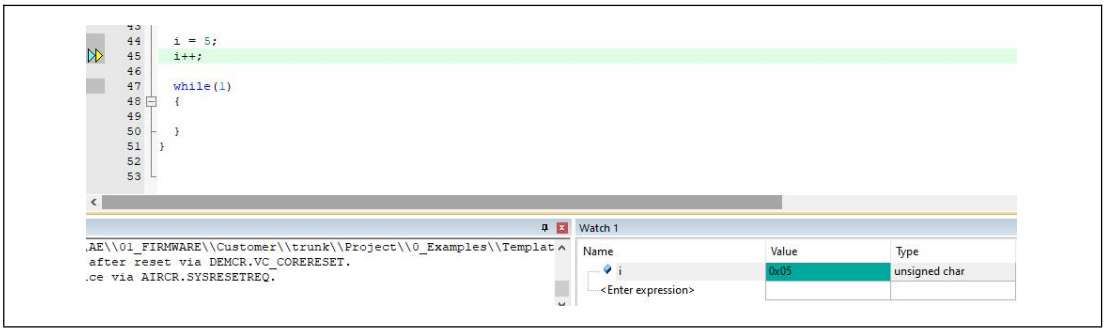
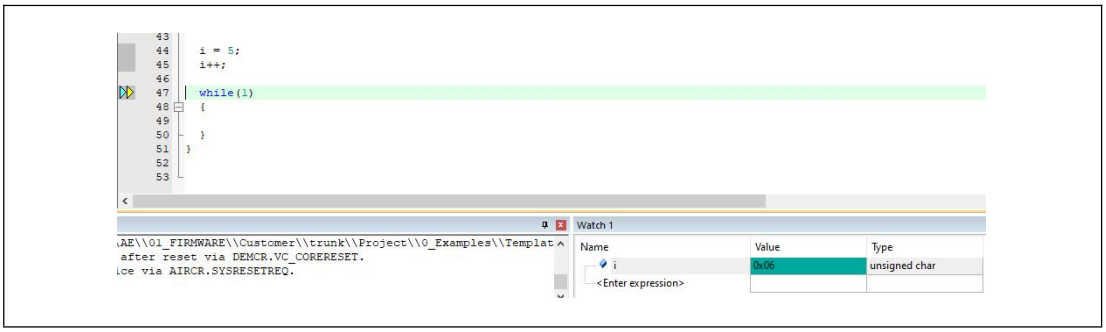


图 2-14. i++执行结果



2.2.4 观察外设寄存器

在调试程序的时候，我们不仅需要观察变量的值，也需要查看芯片外设 Register 的值。本节以芯片 SPC11X8/SPD11X8 外设模块 PWM0 为例介绍实现过程。

（1）通过 KEIL 软件添加芯片 System Viewer File。单击图标，在弹出的界面中勾选 Use Custom File 选项，然后单击图标，在弹出的对话框中选中 SPC11X8/SPD11X8.SFR 文件，位于目录 V1_x\IDE_Support\MDK-ARM 中。设置结果如图 2-15 所示。

（2）单击按钮，进入 Debug 模式，将芯片外设 PWM0 添加到 System Viewer 窗口，如图 2-16 所示。添加后的结果如图 2-17 所示。从图 2-17 可以看出，在 System Viewer 窗口中，不仅可以看到 PWM0 模块各个 Register 的值，而且还可以看到 Register 各个位段的值。

按照上面的步骤将 PWM0 添加到 System Viewer 窗口后，就可以在 Debug 程序的过程中观察到 PWM0 各个寄存器的值。我们单步执行图 2-17 中 main 函数的程序，分别将 TBCTL 寄存器赋值为 0 和 0x1234，结果分别如图 2-18 和图 2-19 所示。

图 2- 15. System Viewer File 设置界面

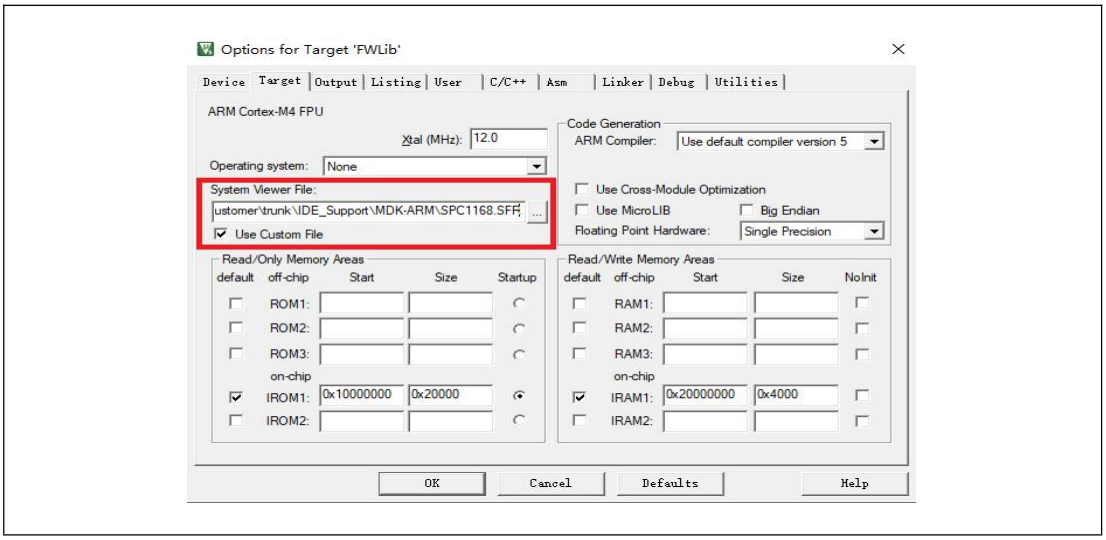


图 2- 16. 添加 PWM0 到 System Viewer 窗口

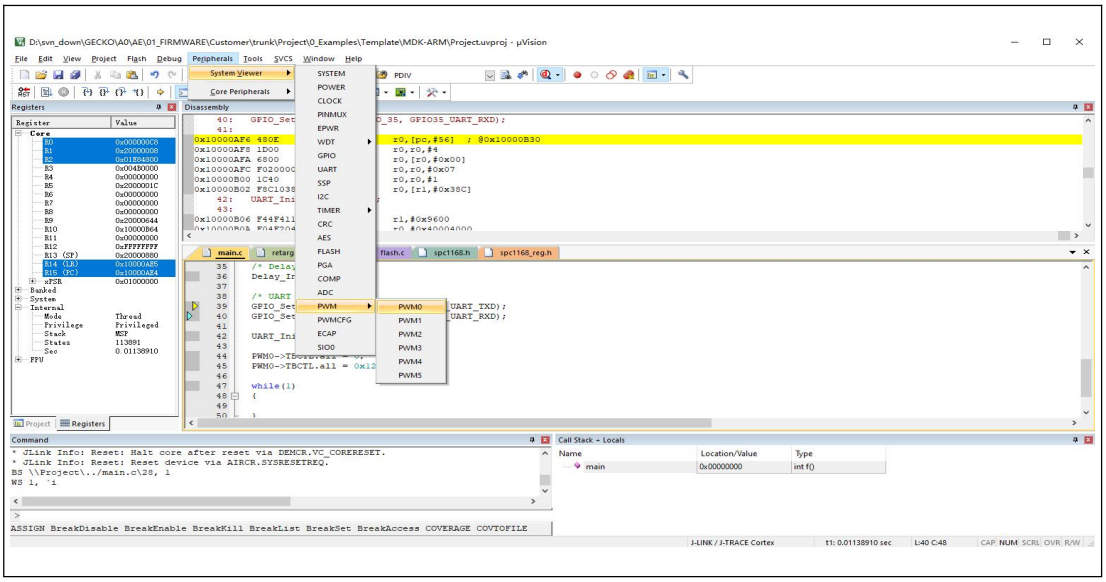


图 2- 17. 添加 PWM0 到 System Viewer 窗口的结果

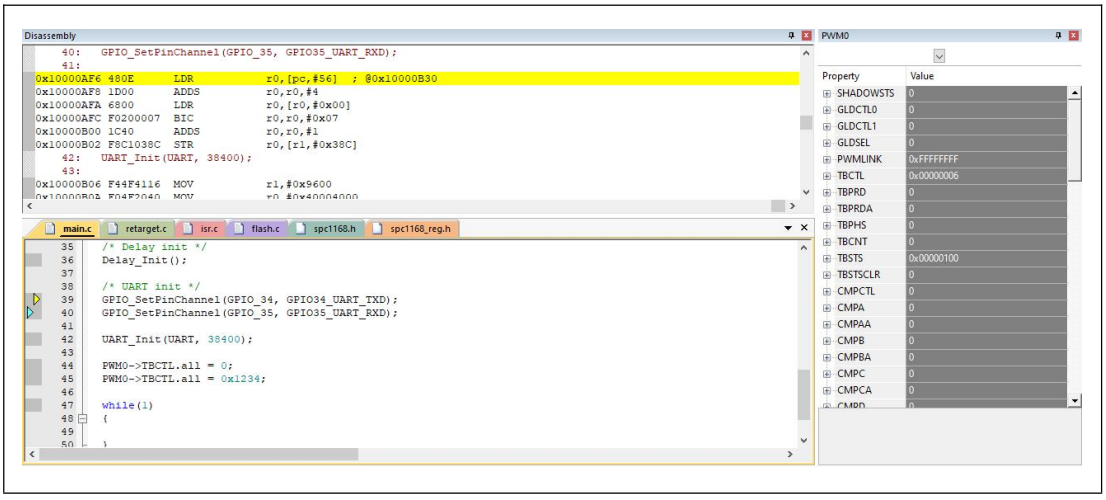


图 2- 18. TBCTL=0 执行结果

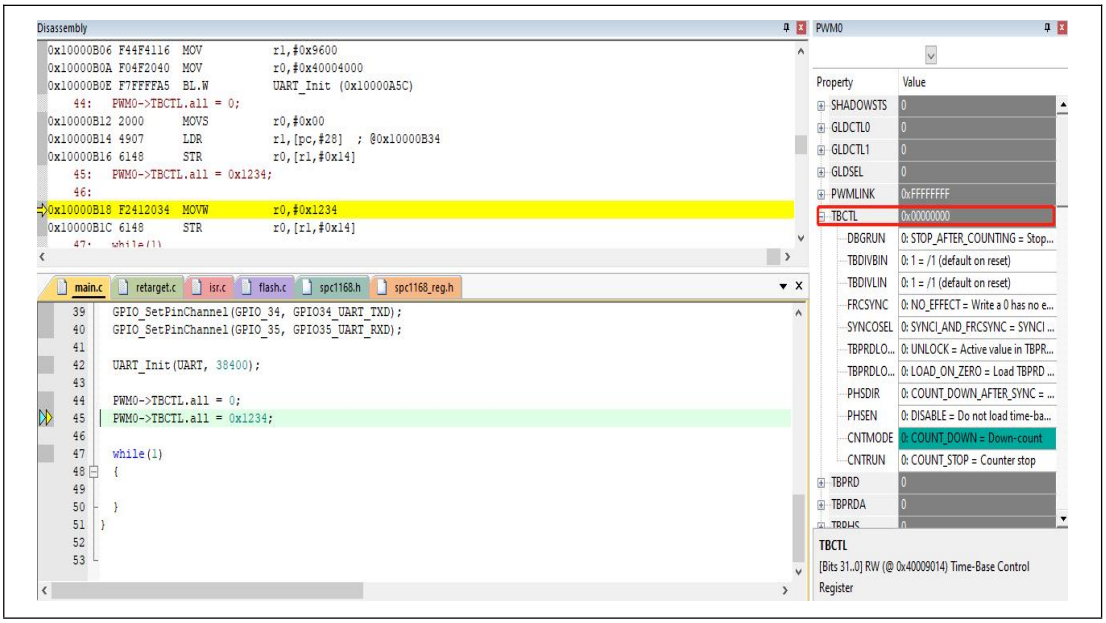
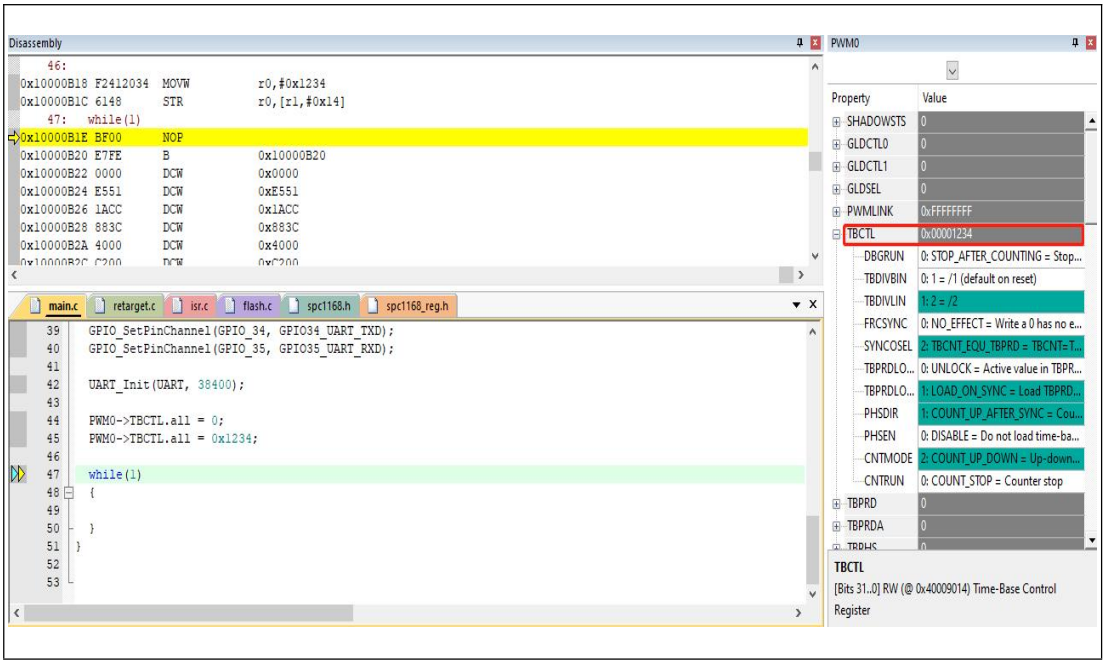


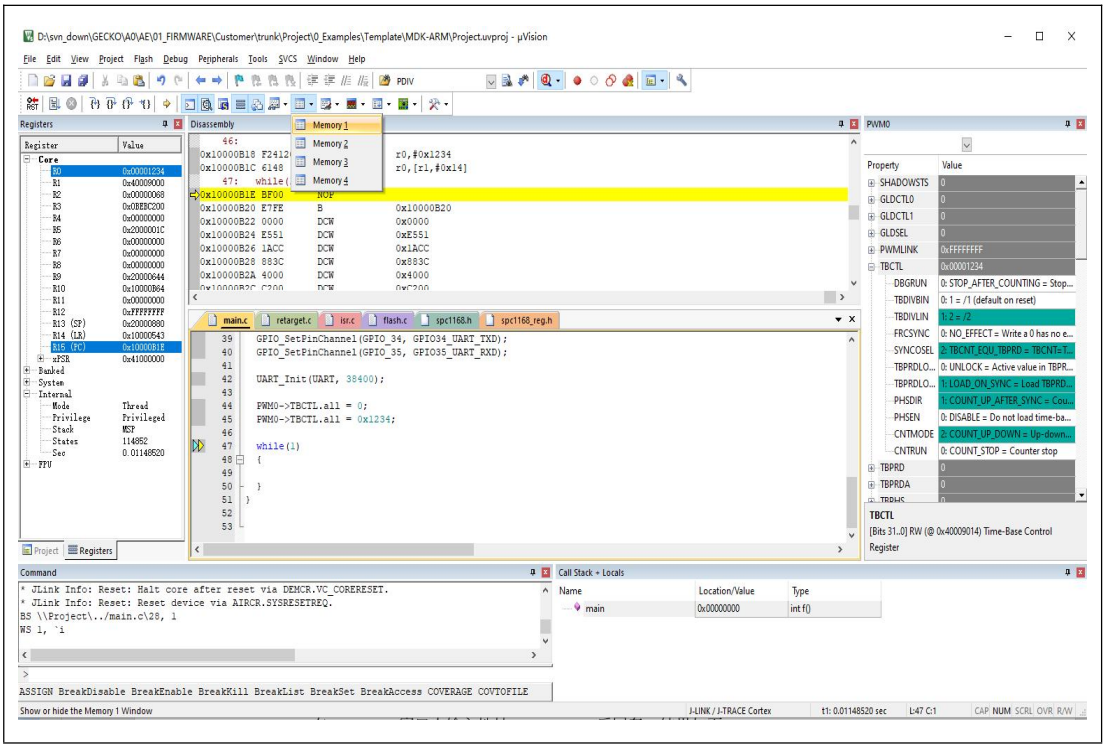
图 2- 19. TBCTL=0x1234 执行结果



2.2.5 Memory 窗口

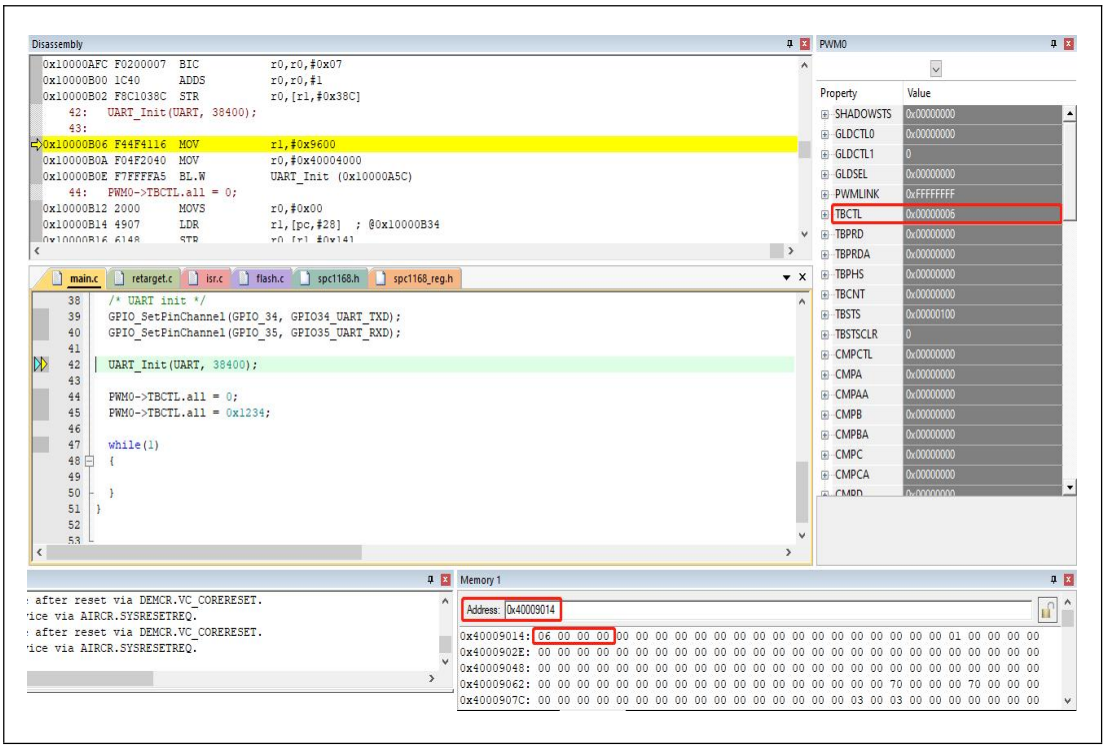
在 Debug 程序的过程中，我们还可以通过 Memory 窗口观察芯片内任一存储单元的地址。我们以章节 2.2.4 中的程序为例，其中 SPC11X8/SPD11X8 芯片 PWM0 模块 TBCTL 寄存器的地址为 0x40009014。首先，打开一个 Memory 观察窗口（Memory1），如图 2- 21 所示。

图 2- 20. 打开 Memory 观察窗口



在 Memory1 窗口中输入地址 0x40009014 后回车，结果如下：

图 2- 21. Memory 窗口中观察到的 TBCTL 初始值



分别单步执行图 2- 21 中 main 函数的程序，分别将 TBCTL 寄存器赋值为 0 和 0x1234，结果分别如图 2- 22 和图 2- 23 所示。

图 2- 22. Memory 窗口结果（TBCTL=0）

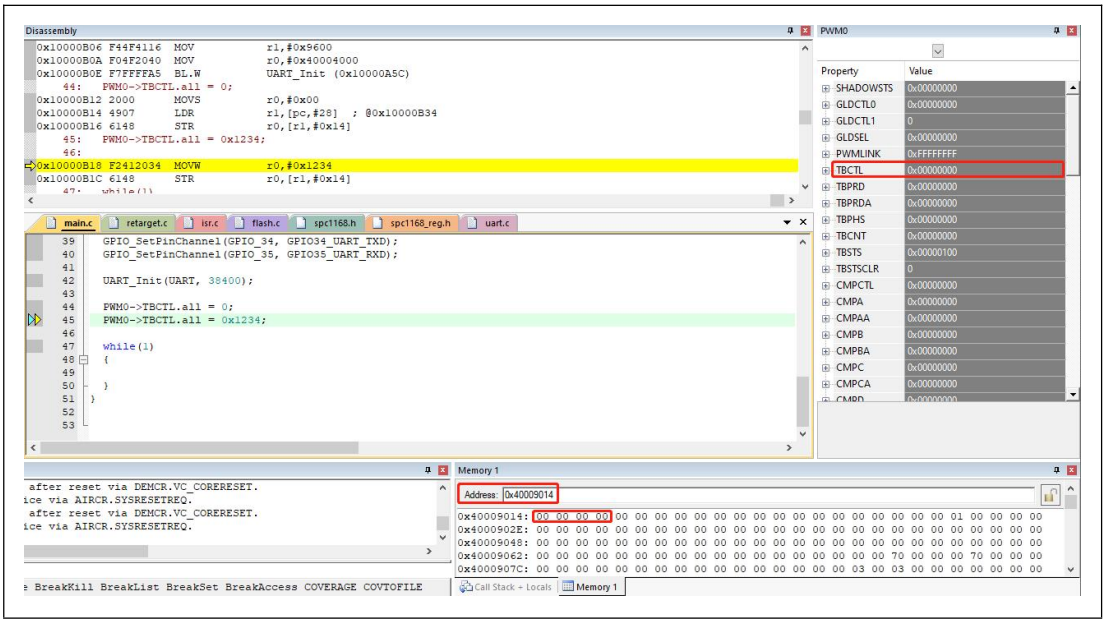
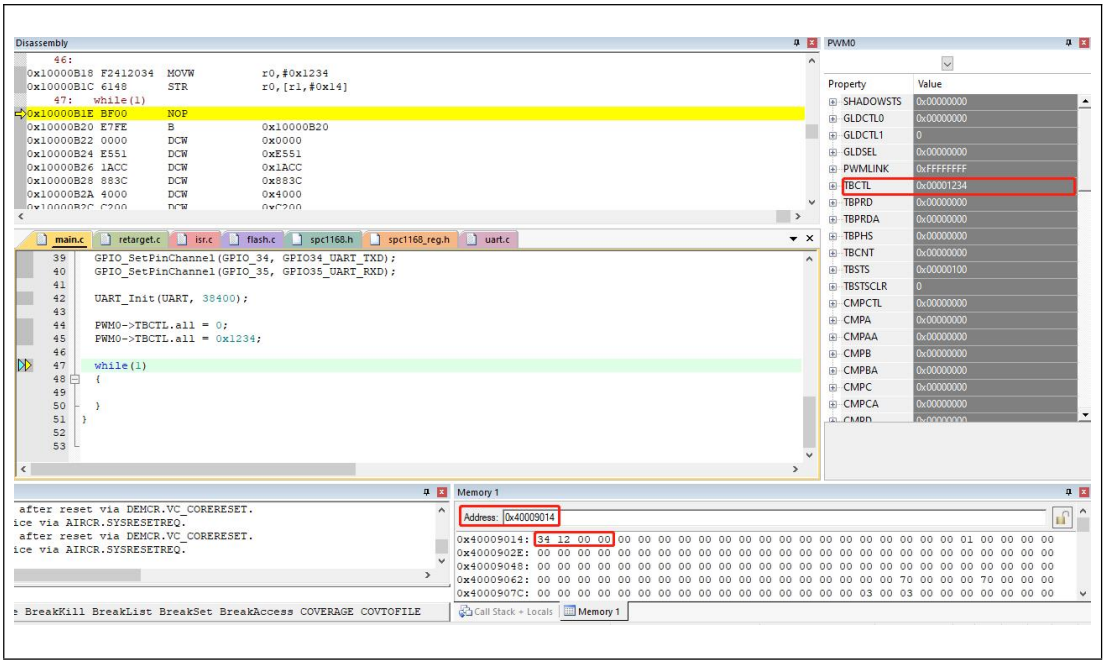


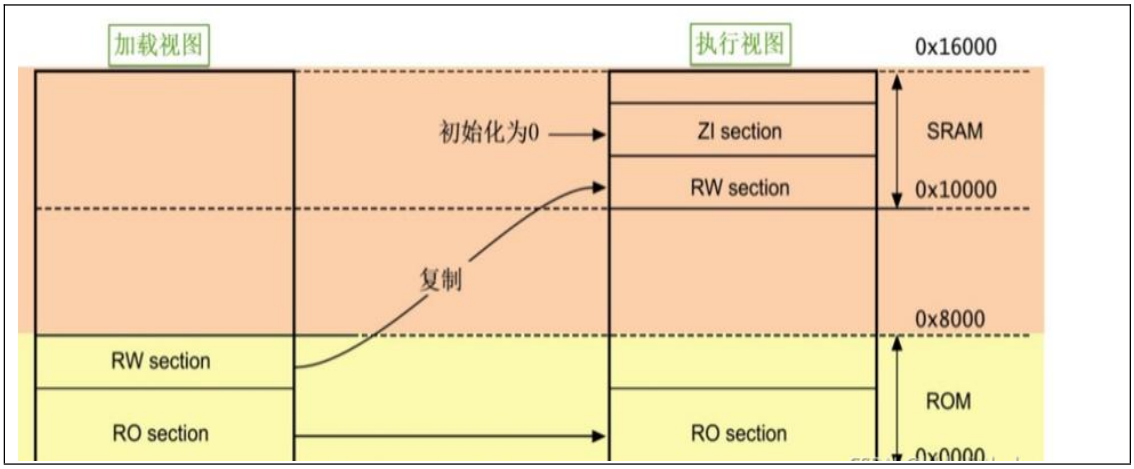
图 2- 23. Memory 窗口结果（TBCTL=0x1234）



2.3 KEIL 环境下分散加载文件

分散加载文件（即 scatter file，后缀为.scf）是一个文本文件，通过此文件指定 ARM 连接器再生成映像文件时如何分配 RO,RW,ZI 等数据的存放地址。 编译器在链接的时候，是根据分散加载(.scf 后缀的文件)来确定程序的加载域和运行域的。加载域就是程序运行前在 flash 中具体分区情况，执行域就是程序运行后，程序在 flash 和 ram 中的分区情况。

图 2- 24. 加载域余执行域



左边是加载视图，RW 段和 RO 段对应的存储空间我们称为加载域。当程序运行后，RW 段中的数据会被复制到 RAM 中，同时还会初始化一个 ZI 段用来存放没有初始化和被初始化为零的相关变量。因此右边的相关储存空间我们称为执行域。

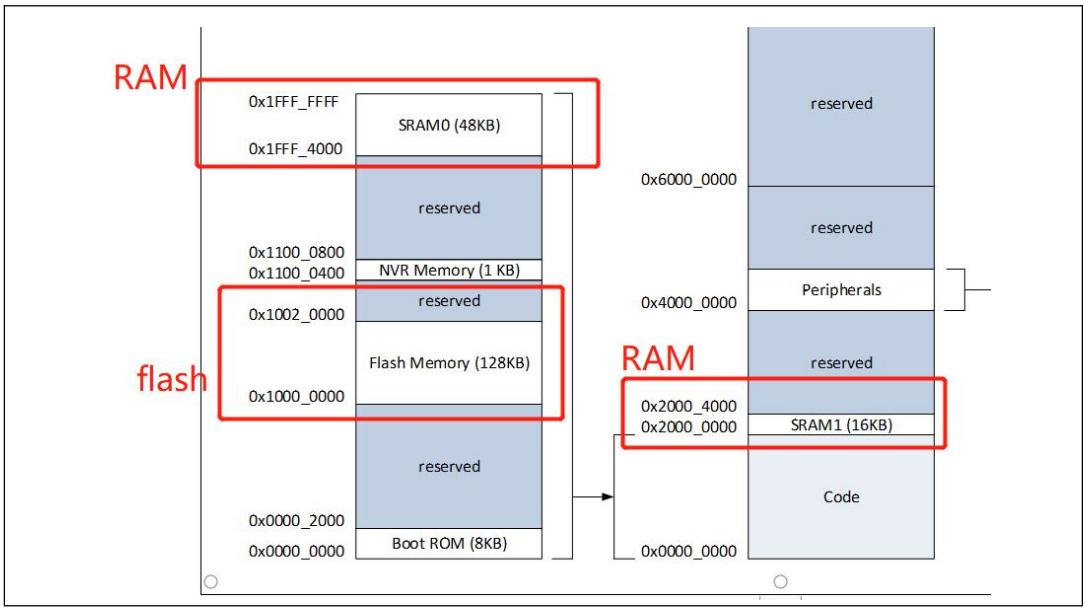
2.3.1 查找 MCU 的内存映射关系

先查找 MCU 的内存映射图，从 MCU 手册 可以查找如图 2- 25 所示。 则

SPC11XX 的 flash 起始地址为 0x10000000 大小为 128k,即 0x20000。

SPC11XX 的 RAM 地址起始地址为 0x1FFF4000 大小为 64k（48k+16k），即 0x10000。

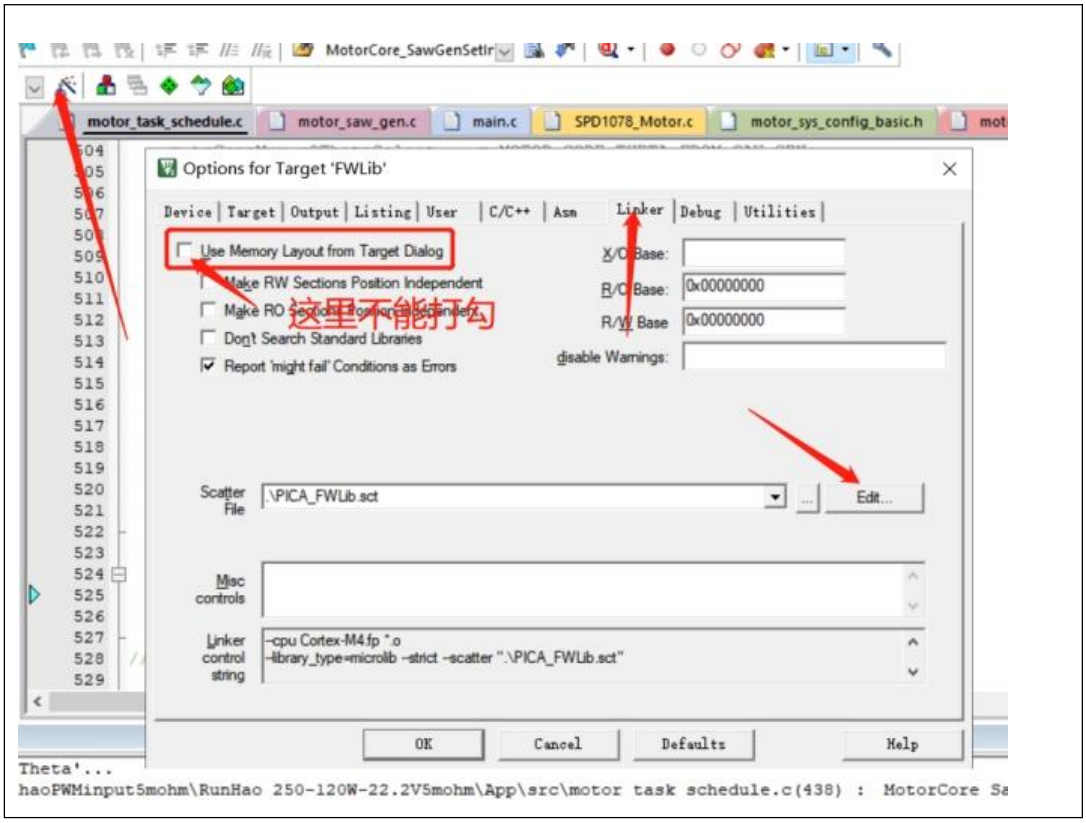
图 2- 25. 内存映射



2.3.2 Keil 配置添加 scatter 文件

按照如图配置，则点击 Edit 的时候会打开 scatter 文件，后缀为 sct。如果没有可以自行创建。

图 2- 26. Keil 配置使用 scatter.



2.3.3 Scatter 文件配置

Scatter 文件内容如下所示：

图 2- 27. Scatter 文件内容

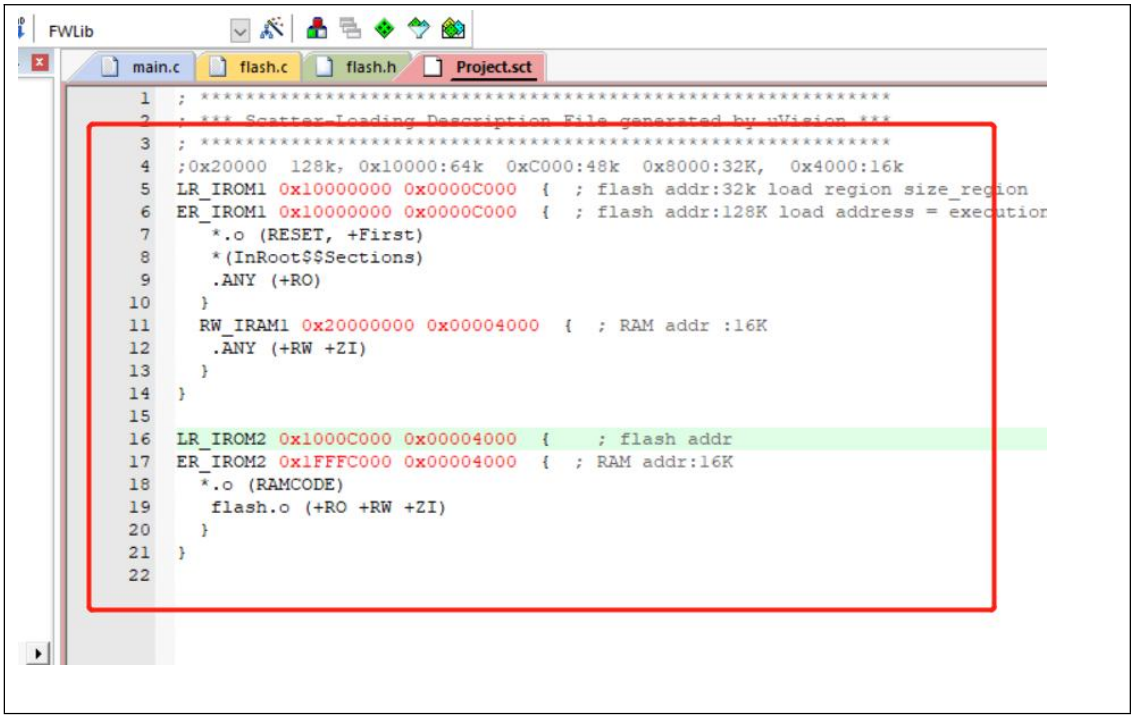
flash 也可以配置为两个 IROM1,IROM2，假设：

IROM1(0x10000000) 48k， 包含 ER_IROM1(0x10000000) 以及 RW_IRAM1(0x20000000, 16k)

IROM2(0x1000C000) 16k， 包含 ER_IROM2(0x1FFFC000, 16k)。

注意：两个 IROM 的 flash 地址与 ram 地址不要冲突越界。

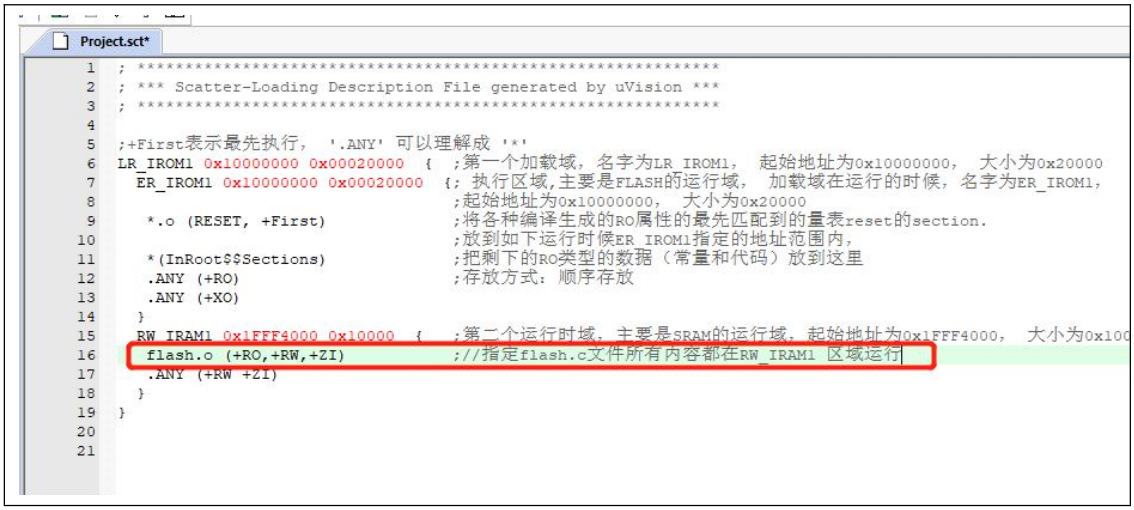
图 2- 28. Scatter 两个 IROM 配置



2.3.4 Scatter 指定相关函数文件在 ram 区域运行

如果需要某个文件全部函数在 ram 上 run，如 flash.c 文件，则可以直接把 flash.o 写到 RW_IRAM1 里面。如图 2-29 所示。

图 2-29. 指定文件到 ram 区域运行



也可以设置相关的 section, section 会在这个 RAM 区域，让特定的函数在这个 sction 区域里面运行，也就是在 RAM 运行。如图 2-30 所示。代码再按照如图 2-31 所示。则 myfuntion 函数就会运行再相关的区域。

图 2-30. 指定函数在 RAM 空间运行

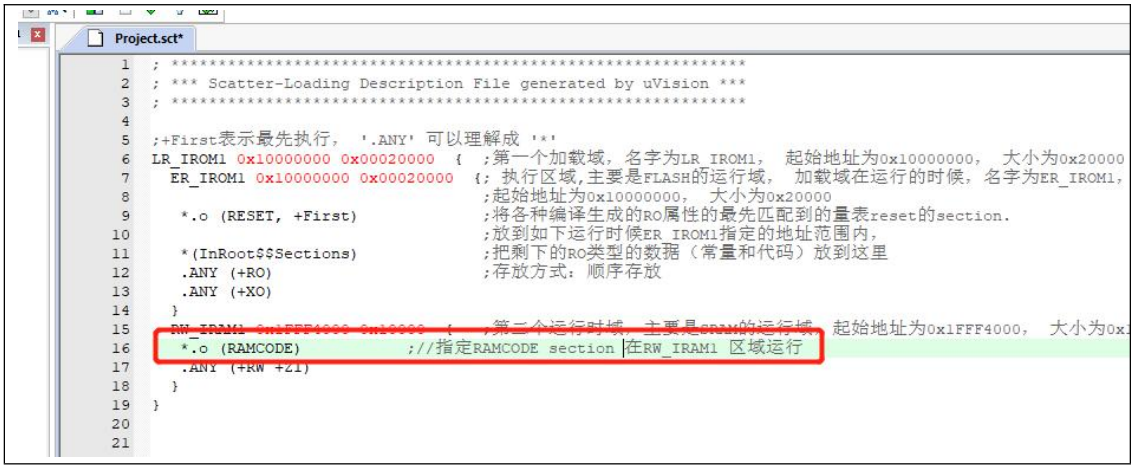


图 2- 31. 指定函数加载到 RAMCODE 里面

```
#pragma arm section code = "RAMCODE"
void myfuntion()
 {
 ...
 }
 #pragma arm_section
```

2.3.5 Scatter 配置软件复位后变量不丢失数据

RAM 分为 DRAM 与 IRAM ，在 Keil 中使能自定义的 sct 文件，从而手动分配地址空间，在 sct 文件中填入自定义地址空间并指定 NO_INIT 区域， NO_INIT 区间的地址必须是没有被 boot 代码使用的区间，如果正好选在这一空间，RAM 值会在 boot 代码中被覆盖。

SPC11XX 芯片 RAM 要避开这一区间为 0x20003000 ~ 0x20004000。

图 2- 32. 配置不初始化 NO_INIT 区域

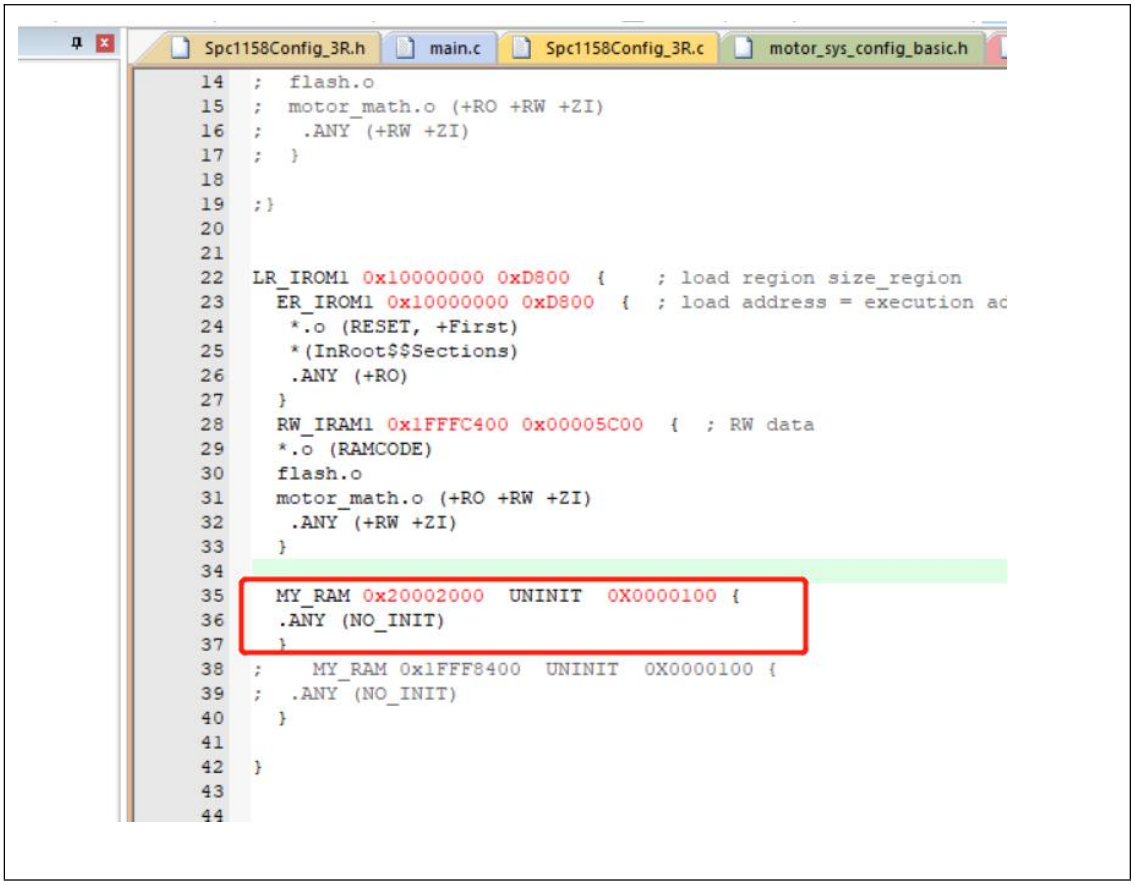


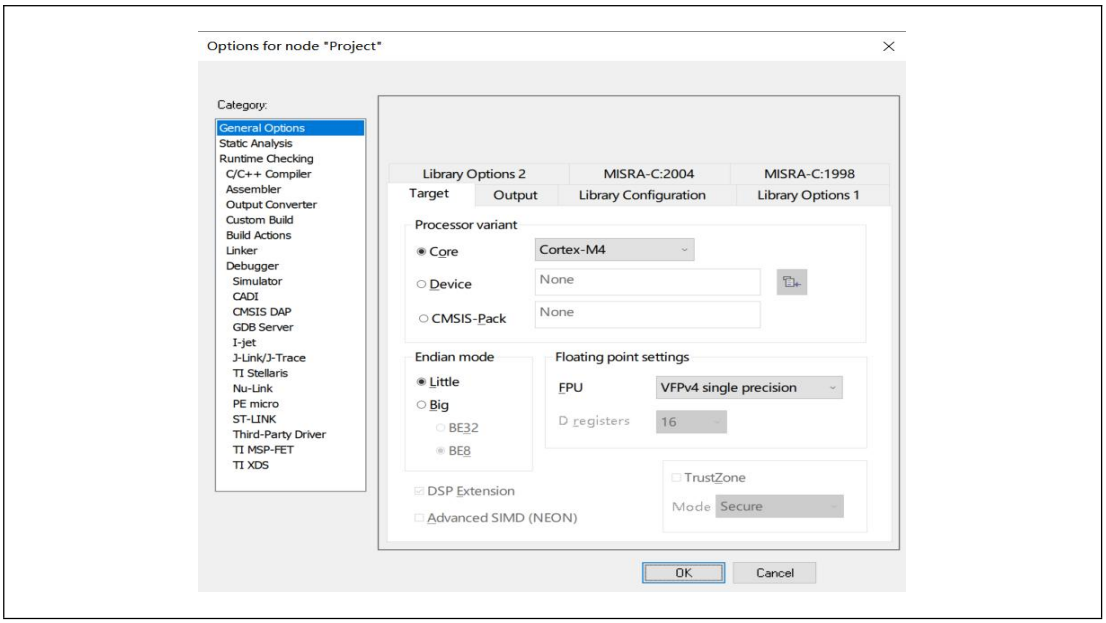
图 2- 33. 变量链接到 NO_INIT 区域

3 IAR 环境搭建

3.1 IAR 环境下 J-LINK 配置

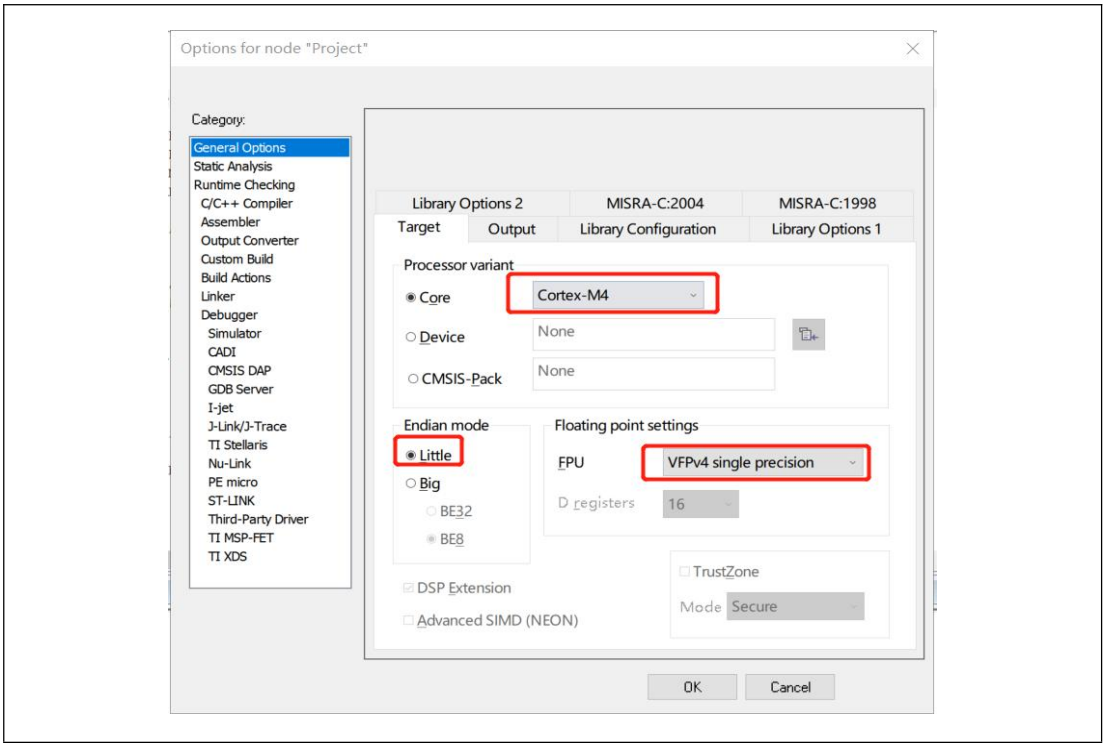
在安装 IAR Embedded workbench 时，软件会默认安装 J-LINK 设备的驱动。按照表 1-1 将 J-LINK 与 SPC11X8/SPD11X8 连接，然后给芯片上电。这时打开 IAR Embedded workbench 软件，鼠标右键项目名称，选择 Options，弹出界面如下：

图 3-1. Options for project 对话框



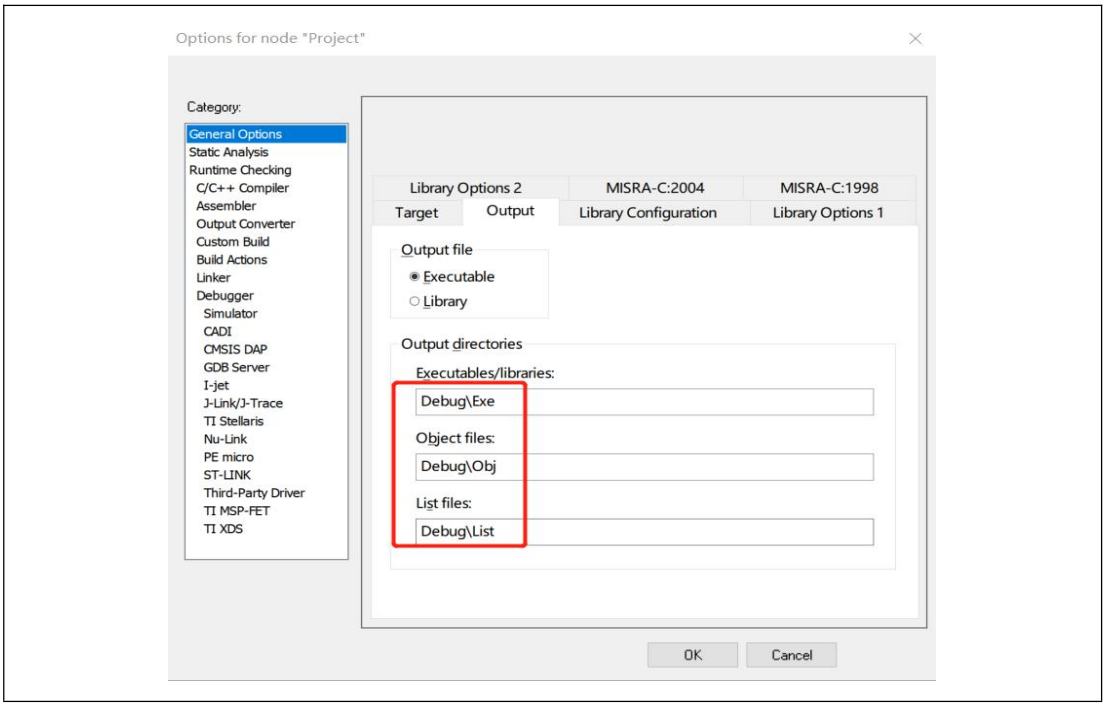
选择 **General Options->Target** 选项卡，会看到如图 3-2 所示的界面。红色矩形框标记的内容是需要设置的选项。

图 3-2. Target 配置界面



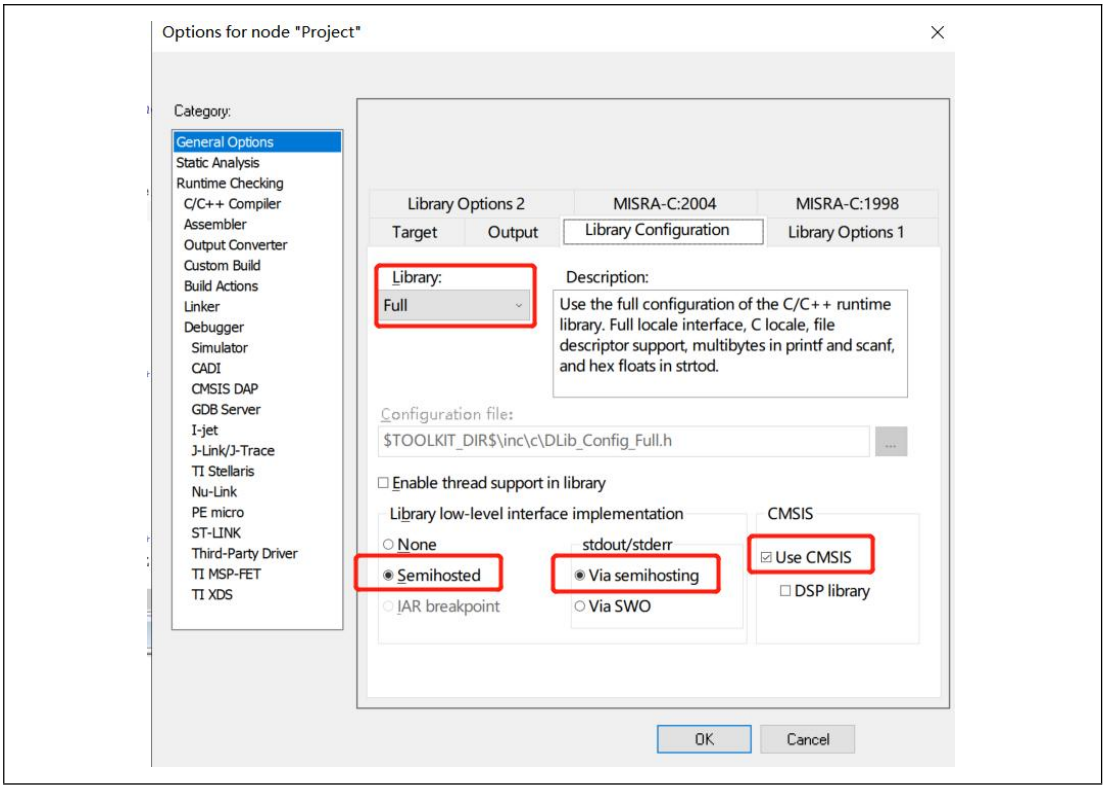
选择 **General Options->Output** 选项卡，会看到如图 3-3 所示的界面。红色矩形框标记的内容是需要设置的选项。

图 3-3. Output 配置界面



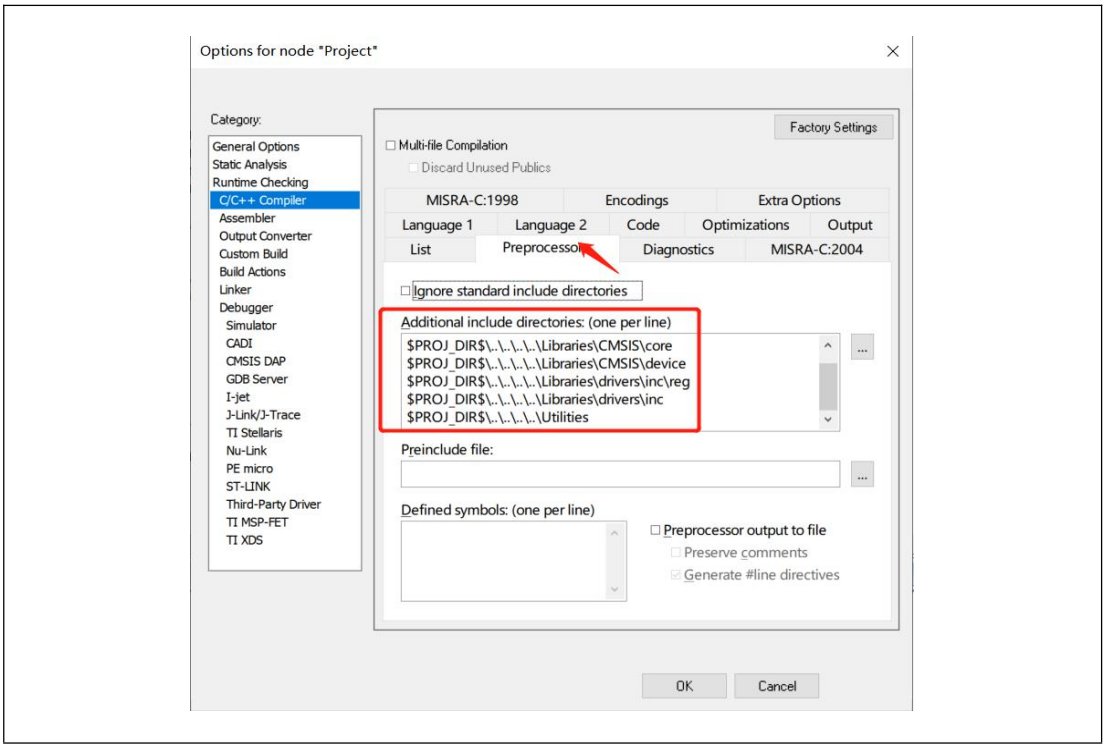
选择 **General Options->Library Configuration** 选项卡，会看到如图 3-4 所示的界面。红色矩形框标记的内容是需要设置的选项。

图 3-4. Library Configuration 配置界面



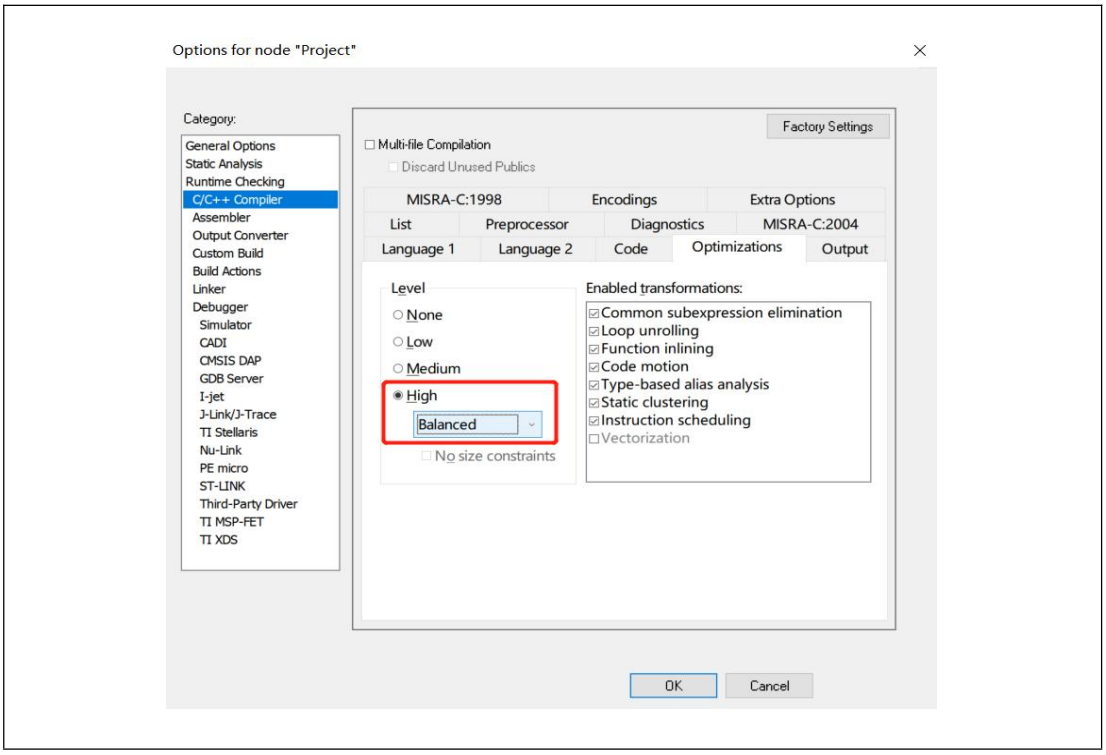
选择 **CC++ Compler　Options->Preprocessor** 选项卡，会看到如图 3-5 所示的界面。添加项目的头文件路径。红色矩形框标记的内容是需要设置的选项。

图 3-5. 预处理器配置界面



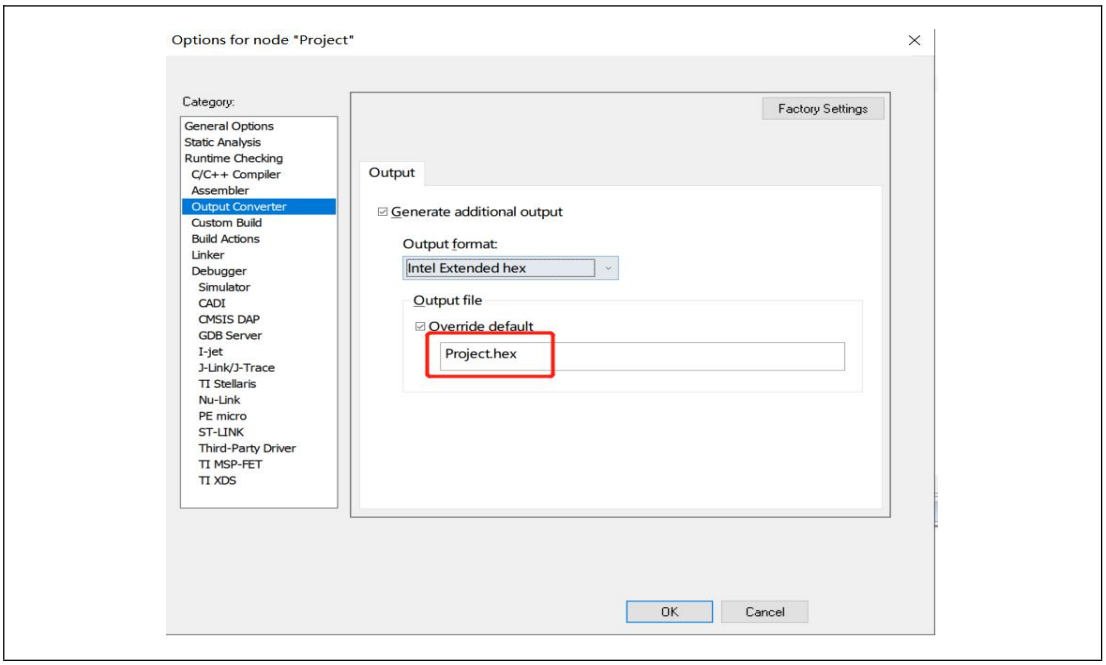
选择 **CC++ Compler　Options->Optimizations** 选项卡，会看到如图 3-6 所示的界面。项目前期可以选择 Low 等级，后面代码完善后选择 High 优化极。

图 3-6. 代码优化等级配置界面



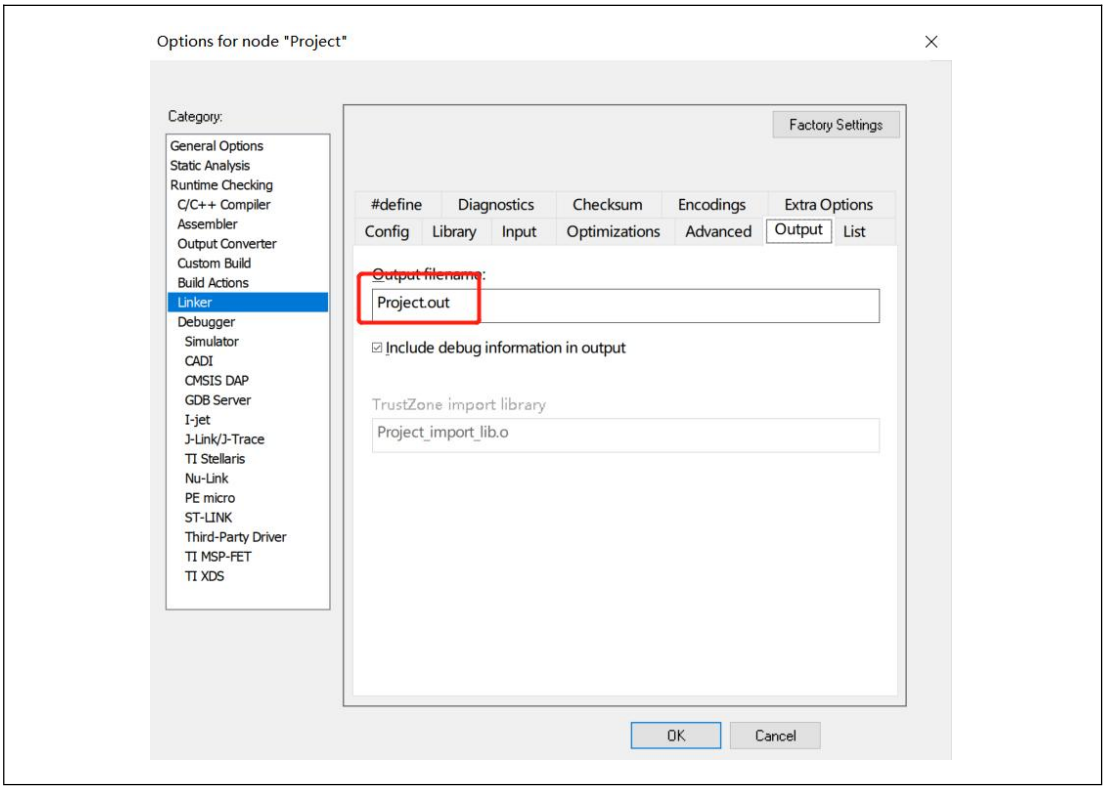
选择 **Output Converter** 选项卡，会看到如图 3-7 所示的界面。配置后编译代码则可以生成 hex 文件。

图 3-7. 输出 HEX 文件名称配置界面



选择 **Linker ->Output** 选项卡，会看到如图 3-8 所示的界面。可以修改输出 bin 文件的名称。

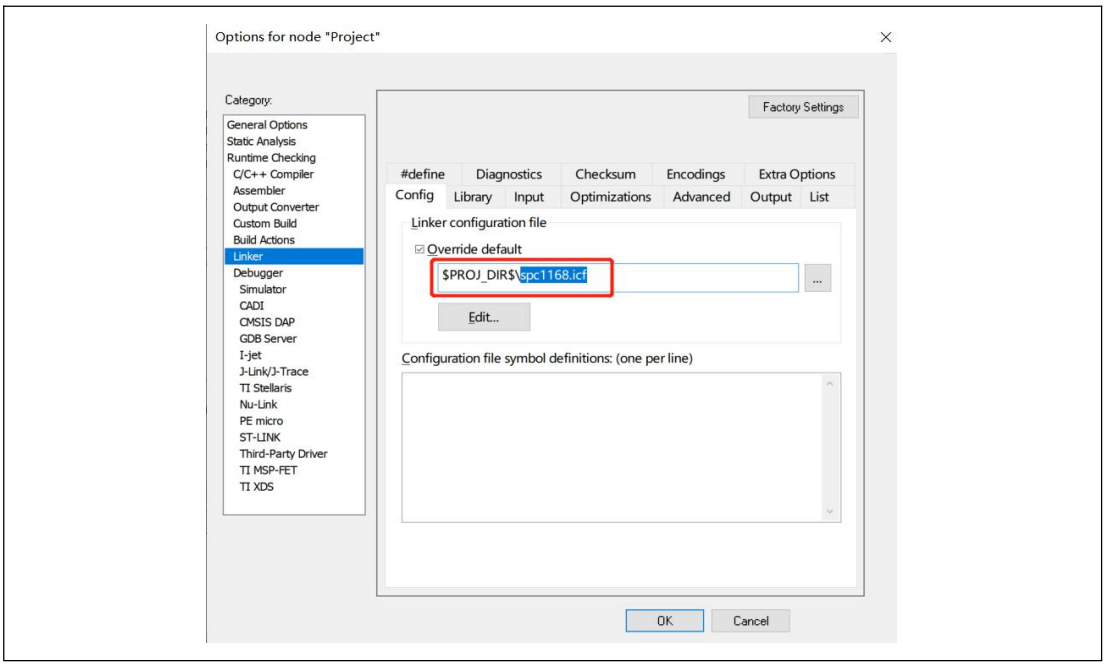
图 3-8. 输出 bin 文件名称配置界面



选择 **Linker ->Config** 选项卡，会看到如图 3-9 所示的界面。

spc11X8.icf/spd11X8.icf 文件定义了 rom 与 ram 的起始地址，以及堆栈的大小。位于 Project.eww 当前目前下。

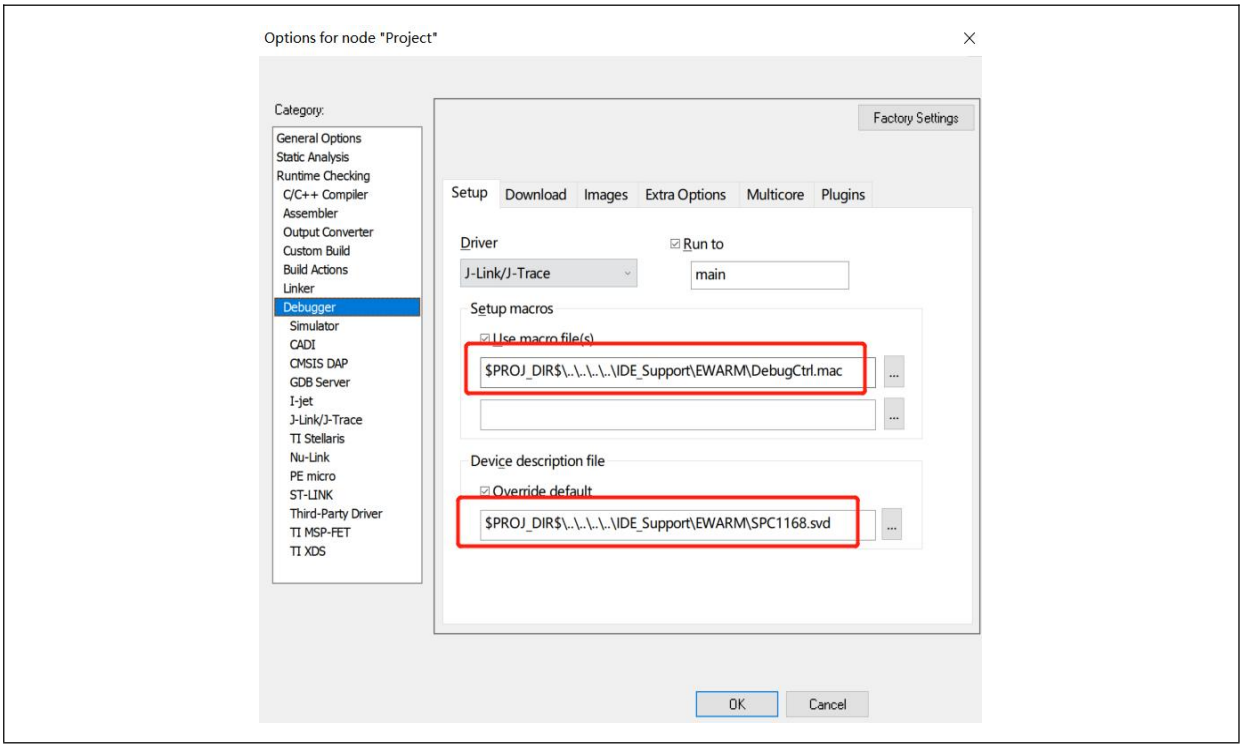
图 3-9. CONFIG 文件配置界面



选择 **Debugger ->Setup** 选项卡，会看到如 图 3-10 所示的界面。

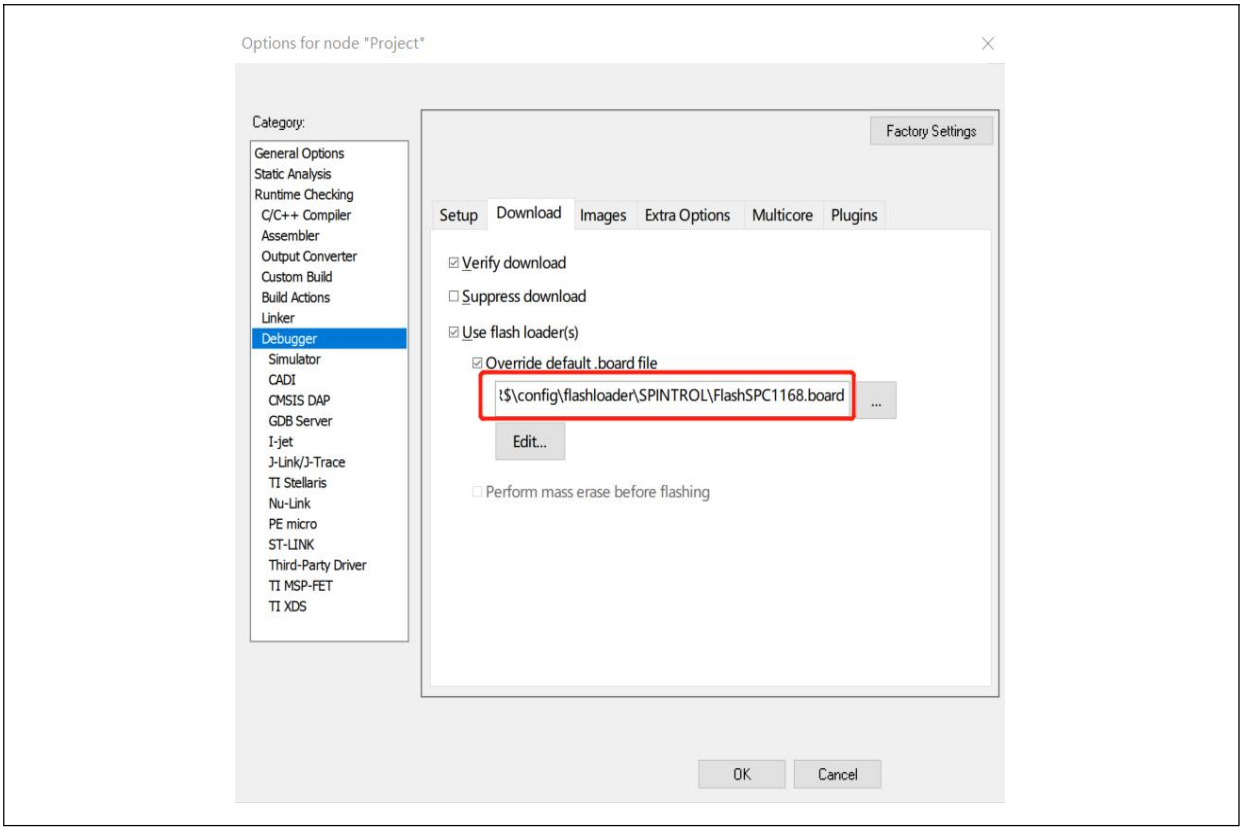
Use macro file 选择 DebugCtrl.mac 文件，位于目录 V1_X\IDE_Support\EWARM 中。
Override default 选择 SPC1168.svd 文件，位于目录 V1_X\IDE_Support\EWARM 中。

图 3- 10. Debuger Setup 配置界面



选择 **Debugger ->Download** 选项卡，会看到如图 3- 11 所示的界面。选择 FlashSPC1168.board 文件。 完整路径为：$TOOLKIT_DIR$\config\flashloader\SPINTROL\FlashSPC1168.board。
另外需要把 V1_X\IDE_Support\EWARM\flashloader 该目录下的文件复制到 IAR 安装目录下的 arm\config\flashloader\SPINTROL 文件夹下，如果没有 SPINTROL 文件夹，用户需要手工新建一个。

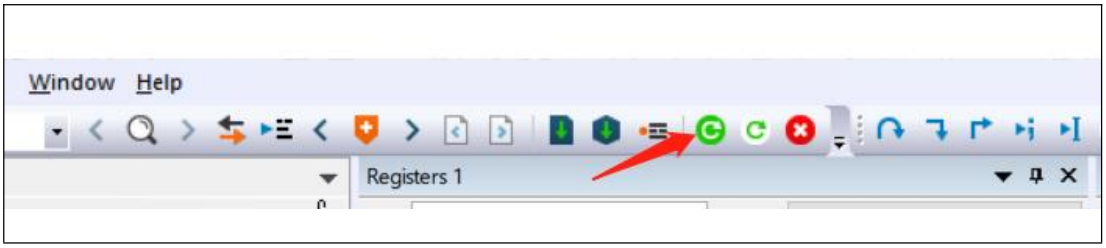
图 3- 11. Debuger Download 配置界面



3.2 IAR 环境下使用 J-LINK 调试

根据前面的介绍，将 J-LINK 设备与 SPC11X8/SPD11X8 正确连接后，按照图 3-10、图 3-11、图 3-11 设置配置以及 Debug 的相关选项，就可以使用 J-LINK 设备调试程序了。用户只需勾选 Make & RestStart Debugger 选项，那么在每次启动 Debug 会话时，IAR 软件会自动通过 J-LINK 设备将程序下载到 Flash 中，从而保证了 Flash 中的程序与当前调试的程序一致，并开始调试。用户如果勾选 RestStart Debugger 选项，则不会下载当前程序，直接进入到 Debug 调试。

图 3-12. Make&Reststart Debugger 调试器



单击工具栏上的按钮进入 Debug 状态，程序界面如图 3-14 所示。如果先出现 图 3-13 则需要先选择 Contex-M4 的 Device。 执行到 main 函数入口处后停止，等待用户的进一步操作。此时，KEIL 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，黄色底纹的汇编代码对应于用户代码窗口中光标所在位置的 C 代码；此外，菜单栏上也出现了一些与 Debug 相关的菜单选项，如表 2-1 所示。

注意：在程序进入 Debug 状态后，代码是不可以修改的。如果想修改代码，需要单击按钮退

出 Debug 模式，然后才能修改代码。修改后的代码编译通过后，将代码重新下载到 Flash 中，用户可以继续单击按钮进行 Debug。

图 3- 13. 启动 debug 后选择 device 界面

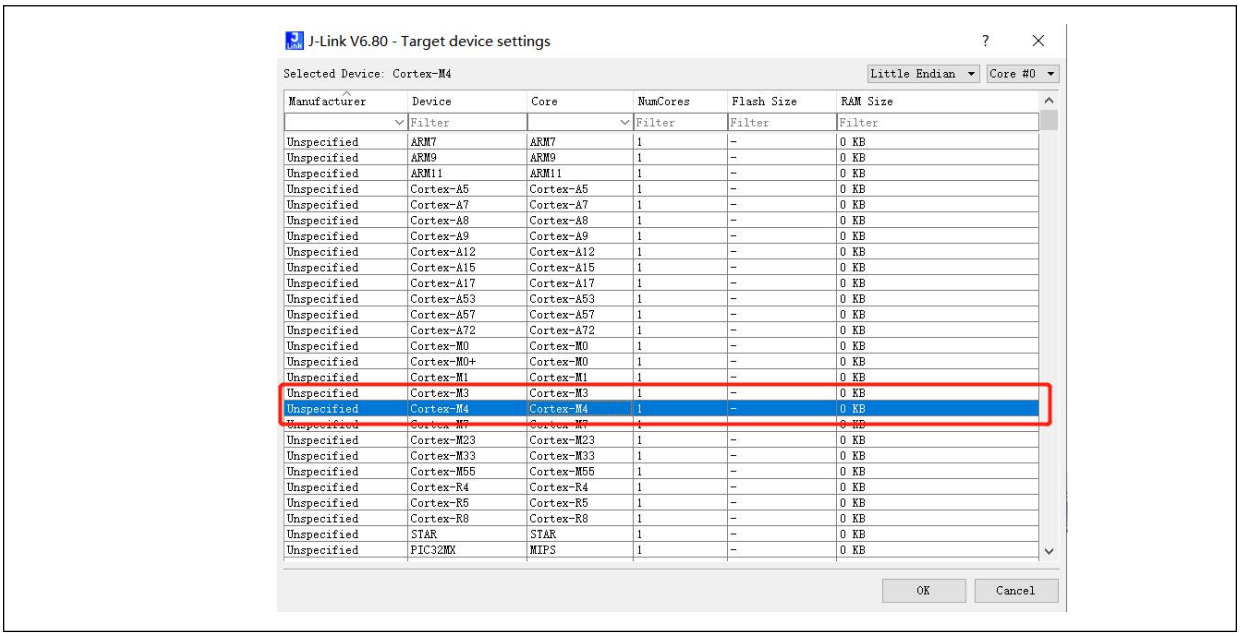


图 3- 14. 启动 Debug 后的界面

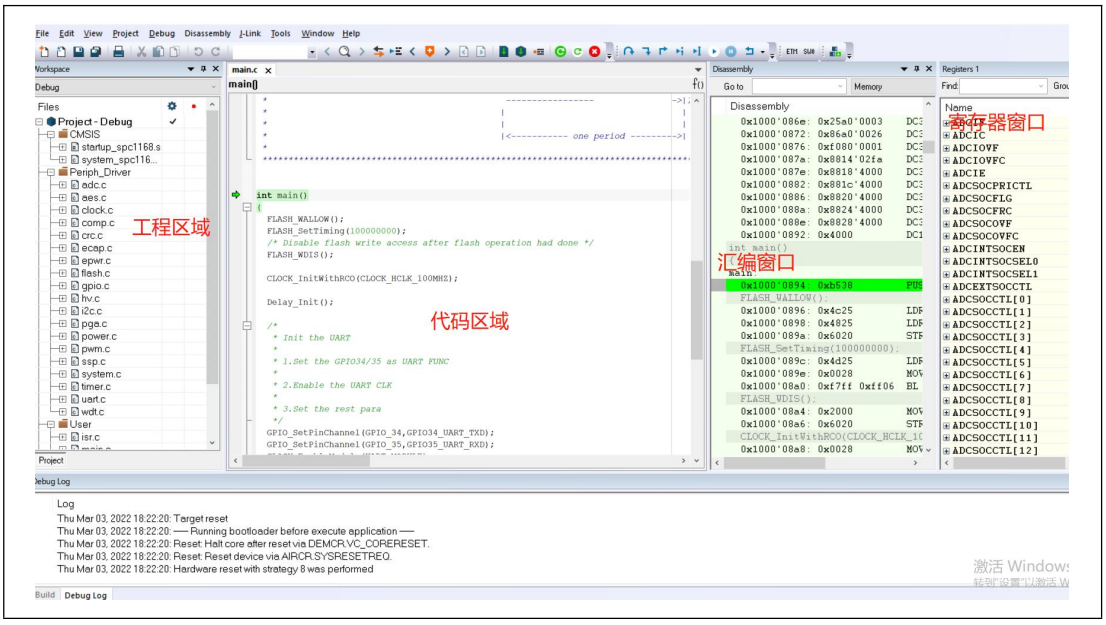


表 3-1. Debug Menu and Commands

Debug Menu	Toolbar	Shortcut	Description
Start/Stop Debug Session		**Ctrl+R**	Starts a debugging session.

Stop Debuging		**Ctrl+Shift+D**	stops a debugging session.
Reset CPU			Sets the CPU to RESET state.
Run		**F5**	Continues executing the program until the next active breakpoint is reached.
Stop			Stops the program execution immediately.
Step Into		**F11**	Executes a single-step into a function; Executes the current instruction line.
Step Over		**F10**	Executes a single-step over a function.
Step Out		**Shift+F11**	Finishes executing the current function and stops afterwards.
Run to Cursor Line			Executes the program until the current cursor line is reached.
Show Next Statement			Shows the next executable statement/instruction.
Breakpoints		**F9**	Opens the dialog Breakpoints.
Insert/Remove Breakpoint		**F9**	Toggles the breakpoint on the current line.

3.2.1 单步调试

单击工具栏上的  按钮后，程序进入 Debug 会话状态。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有个三角箭头：  表示当前光标所在的位置。因此，可以通过这个三角箭头快速判断程序执行到哪条语句以及光标的位置。

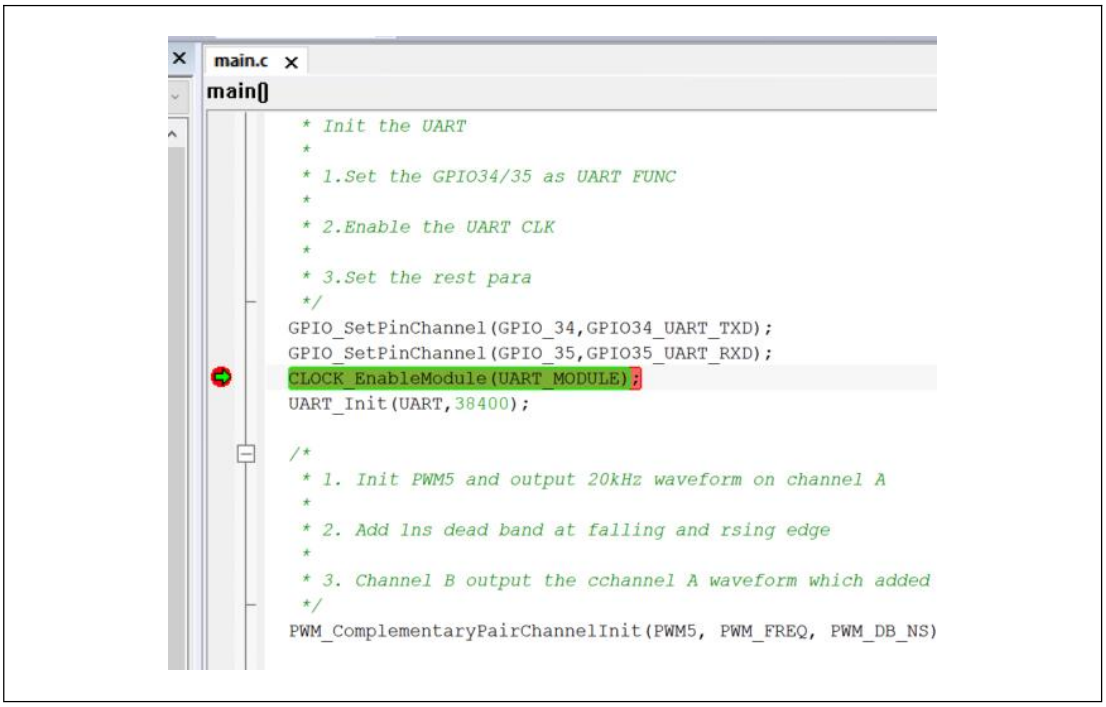
有时候，我们希望程序能够快速地执行到某个位置，再进行单步调试。这时我们可以将光标定位到该位置，然后单击工具栏上的  按钮，程序就会立即执行到当前光标处。此外，我们也可以通过设置断点的方式来实现上述功能。

3.2.2 断点设置

当启动 Debug 会话后，在用户代码所在行左侧边框处单击鼠标左键，即可快速地设置断点，此时左侧边框会出现一个红色的圆形标记，如图 3-15 所示。当然，也可以通过工具栏上的  按钮设置断点，具体做法是：将光标定位到欲设置断点的代码行，然后单击  按钮，即可设置断点。此时，单击工具栏上的按钮  或者按下快捷键 F5，程序就会执行起来，一直执行到断点

处停下来，如 图 3-15 所示。

图 3-15. 设置断点



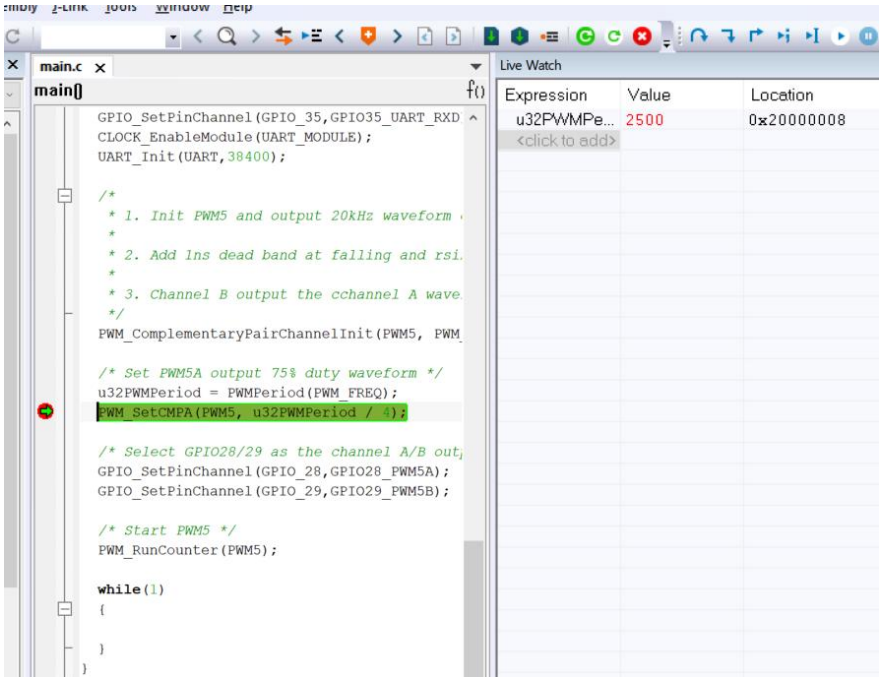
设置断点除了可以让程序快速的执行到某个位置外，在 Debug 程序时也非常有用。例如，可以在中断服务函数中设置断点，用来判断相应的中断是否发生。

3.2.3 观察变量值

在调试程序的时候，常常需要观察变量的值。这个可以通过 IAR 软件提供的 Window 来实现。具体实现过程如下：将光标定位到要观察的变量（光标移到变量名左侧），单击鼠标右键，在弹出的快捷菜单中选择 Add to live watch: xxxx 到观察窗口中，如图 3-16 所示。

从图 3-16 可以看出，我们将变量 u32PWMPeriod 添加到观察窗口 Watch1 里，结果如图所示。

图 3-16. 添加变量到观察窗口的结果



3.2.4 观察外设寄存器

在调试程序的时候，我们不仅需要观察变量的值，也需要查看芯片外设 Register 的值。本节以芯片 SPC11X8/SPD11X8 外设模块 PWM5 为例介绍实现过程。

单击  按钮，进入 Debug 模式，将芯片外设 PWM5 添加到窗口，如图 3-17 所示。不仅可以看到 PWM5 模块各个 Register 的值，而且还可以看到 Register 各个位段的值。

按照上面的步骤将 PWM5 添加到 System Viewer 窗口后，就可以在 Debug 程序的过程中观察到 PWM5 各个寄存器的值。我们也可以即可修改外设寄存器的值让其生效，如图 3-18。

图 3-17. 添加外设 PWM 到窗口

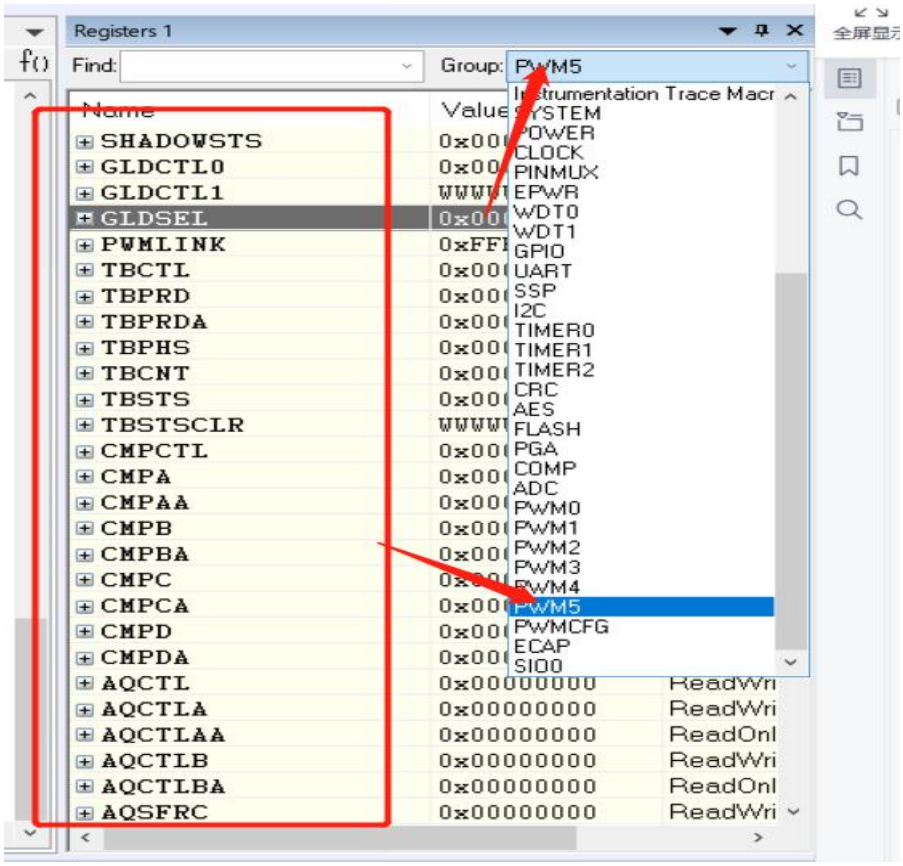
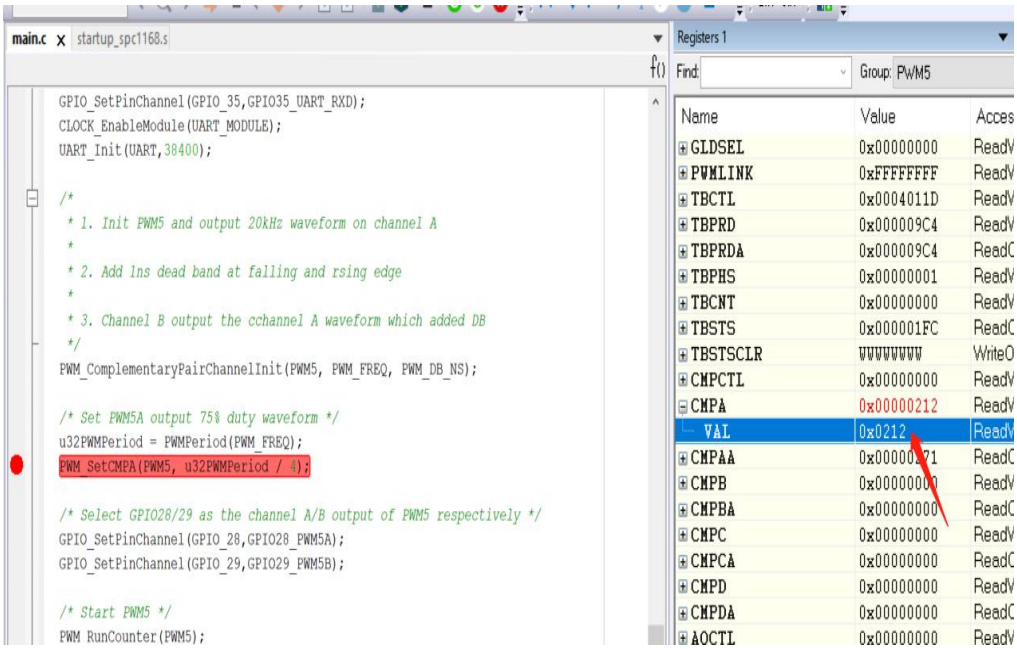


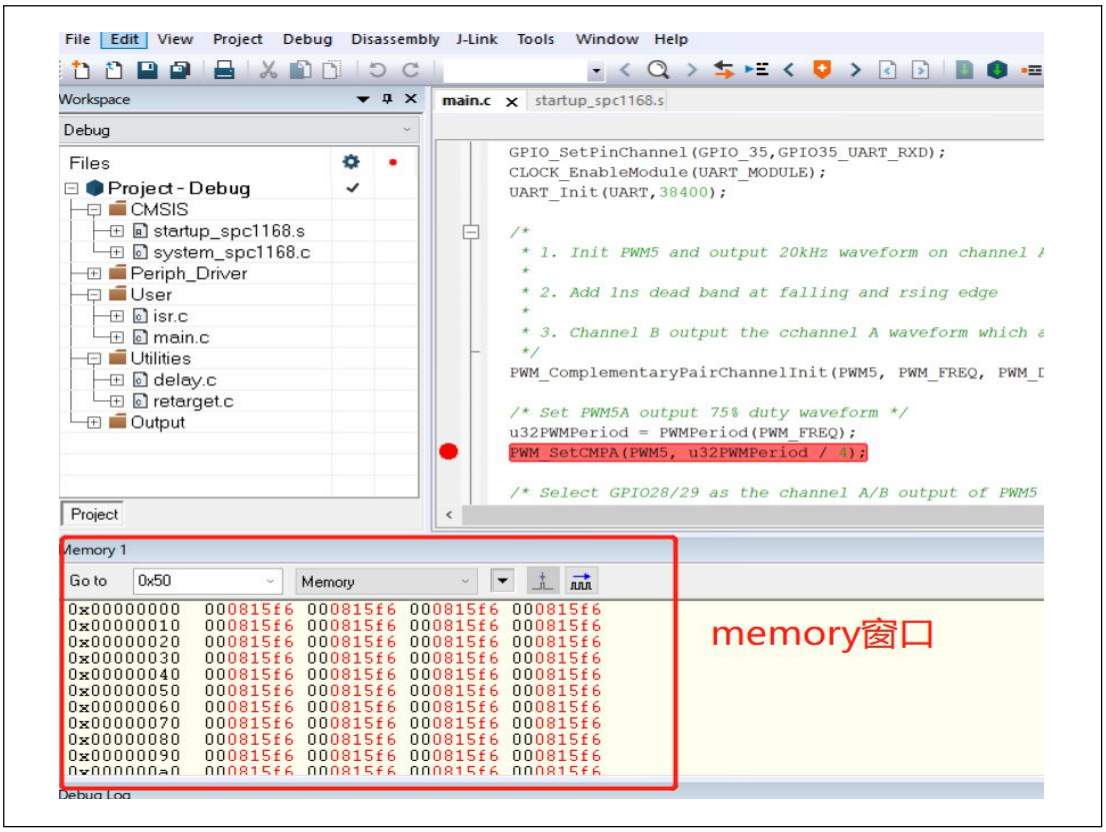
图 3- 18. 修改 PWM5 的寄存器值



3.2.5 Memory 窗口

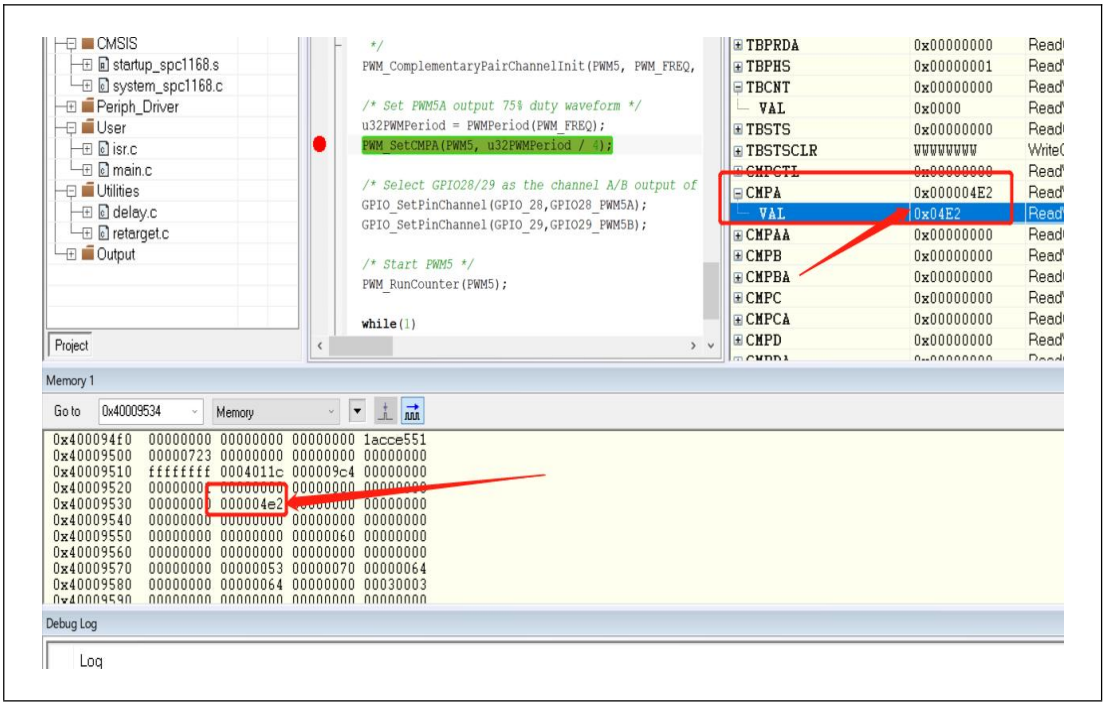
在 Debug 程序的过程中，我们还可以通过 Memory 窗口观察芯片内任一存储单元的地址。其中 SPC11X8/SPD11X8 芯片 PWM0 模块 TBCTL 寄存器的地址为 0x40009014。首先，打开一个 Memory 观察窗口（Memory1），如图 3-19 所示。

图 3-19. 打开 Memory 观察窗口



在 Memory1 窗口中输入地址 0x40009534 后回车，结果如下：

图 3-20. Memory 窗口中观察到的 CMPA 初始值



3.3 IAR 环境下分散加载文件

分散加载文件(即 scatter file，后缀为.scf)是一个文本文件，通过此文件指定 ARM 连接器再生成映像文件时如何分配 RO,RW,ZI 等数据的存放地址。 编译器在链接的时候，是根据分散加载(.scf 后缀的文件)来确定程序的加载域和运行域的。加载域就是程序运行前在 flash 中具体分区情况，执行域就是程序运行后，程序在 flash 和 ram 中的分区情况。

如图 2-24 所示左边是加载视图，RW 段和 RO 段对应的存储空间我们称为加载域。当程序运行后，RW 段中的数据会被复制到 RAM 中，同时还会初始化一个 ZI 段用来存放没有初始化和被初始化为零的相关变量。因此右边的相关储存空间我们称为执行域。

3.3.1 Section 与 linker

代码按功能可以分为很多种类，比如常量、变量、函数、堆栈等，而相同类型的代码的集合便是一个 section，链接器在链接时组织数据的基本单元便是 section。linker 文件是按 IDE 规定的语法写成的用于指示链接器分配各 section 在嵌入式系统存储器中存放位置的文件。section 根据存放的存储器位置不同分为两类属性：readonly, readwrite。实际上 linker 文件的工作就是将 readonly section 放进 ROM，readwrite section 放进 RAM。linker 文件语法规则如

图 3-21. Linker 语法规则

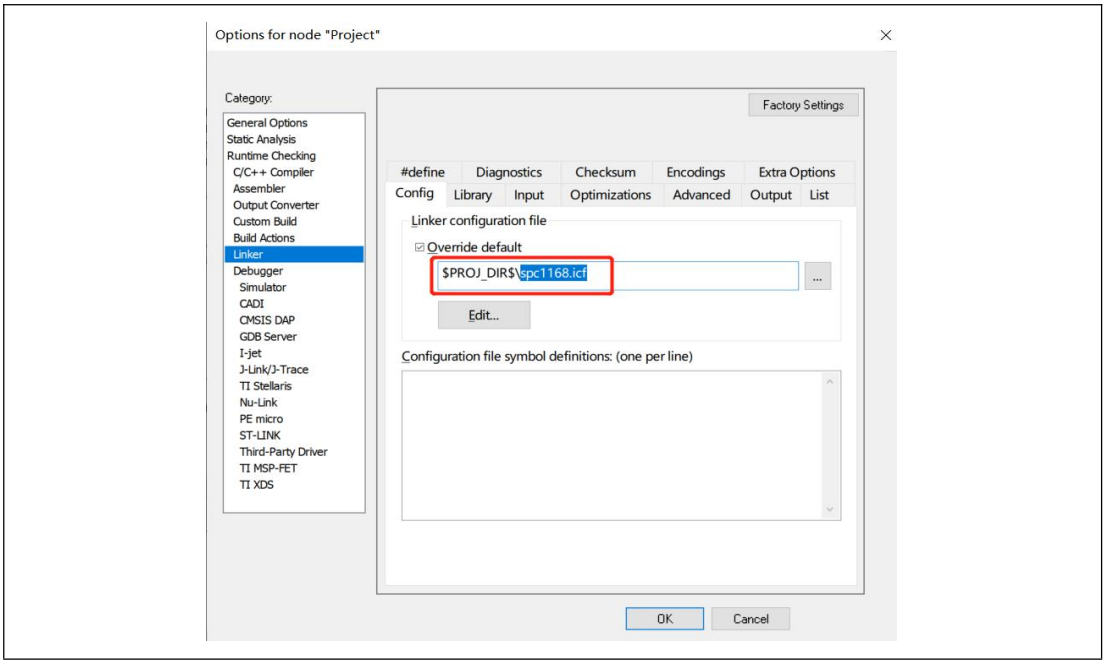
```
// 动词类关键字
define // 定义各种空间范围、长度
initialize // 设置 section 初始化方法
place in // 放置 section 于某 region 中（具体地址由链接器分配）
place at // 放置 section 于某绝对地址处 //
名词类关键字
symbol // 各种空间范围、长度的标识
memory // 整个 ARM 内存空间的标识
region // 在整个 ARM 内存空间中划分某 region 空间的标识
block // 多个 section 的集合块的标识
```

3.3.1 IAR 配置添加 scf 文件

选择 **Linker ->Config** 选项卡，会看到如图 3-9 所示的界面。

spc11X8.icf/spd11X8.icf 文件定义了 rom 与 ram 的起始地址，以及堆栈的大小。位于 Project.eww 当前目前下。如果没有需要自己创建，内容参考图 3-23。

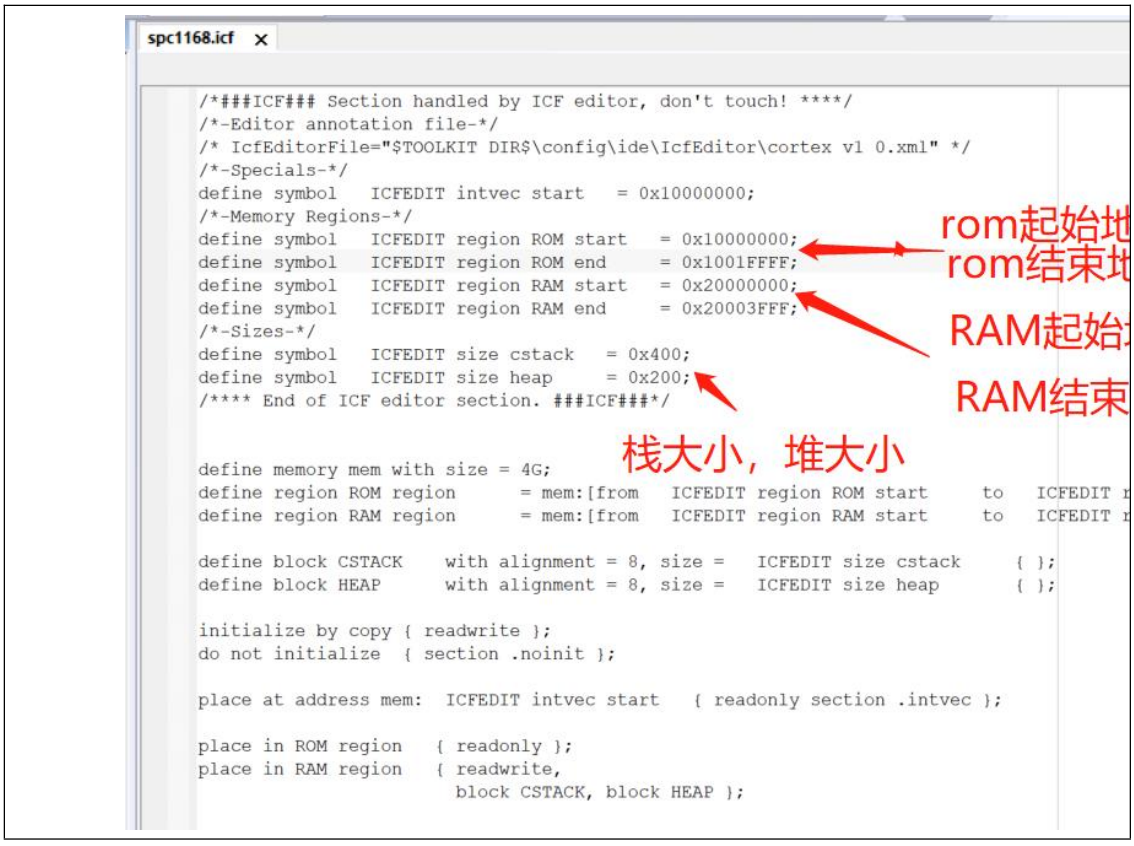
图 3-22. CONFIG 文件配置界面



3.3.2 scf 文件配置

Scatter 文件内容如下所示：具体的 rom 地址与 ram 地址请查看芯片手册的内存映射章节。

图 3- 23. scf 文件内容



flash 也可以配置为两个 RAM 假设:

RW_IRAM(0x20000000, 16k)　RW_DRAM(0x1FFFC000, 16k)。 IRAM 用来存放所有 readwrite 段，

DRAM 用来存放自定义的 myFuntionSection 的 section。则配置 scf 文件如图 3-24 所示。

注意：两个 RAM 地址不要冲突越界。

图 3-24.SCF 两个 RAM 配置

```
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT DIR$\config\ide\IcfEditor\cortex v1 0.xml" */
/*-Specials-*/
define symbol   ICFEDIT intvec start   = 0x10000000;

define memory mem with size = 4G;
define region TEXT region = mem:[from 0x10000000 to 0x1001FFFF];
define region IRAM region = mem:[from 0x1FFFC000 to 0x1FFFFFFF];
define region DRAM region = mem:[from 0x20000000 to 0x2001FBFF];
define region CSTACK region = mem:[from 0x2001FA00 to 0x2001FFFF];
/*-Sizes-*/
define symbol   ICFEDIT size cstack   = 0x400;
define symbol   ICFEDIT size heap     = 0x200;
/**** End of ICF editor section. ###ICF###*/


define block CSTACK    with alignment = 8, size =   ICFEDIT size cstack
define block HEAP      with alignment = 8, size =   ICFEDIT size heap

initialize by copy { readwrite };
do not initialize  { section .noinit };


place at address mem:  ICFEDIT intvec start   { readonly section .intvec
place in TEXT region               { readonly };
place in IRAM region               { readwrite, zi };
place in CSTACK region             { block CSTACK,block HEAP};
/*将.myFuntionSection section的数据链接到地址0x20200100 */
place in DRAM region               { section .myFuntionSection };
```

3.3.3 Scf 文件指定相关函数文件在 ram 区域运行

如果需要某个文件全部函数在 ram 上 run，如 flash.c 文件，则可以直接把 flash.o 写到 RAM 里面。如图 3-25 所示。

图 3-25. 指定文件到 ram 区域运行

spc1168.icf * | main.c | main.c | APP.map

```
/*-Specials-*/
define symbol   ICFEDIT intvec start   = 0x10000000;

define memory mem with size = 4G;
define region TEXT region = mem:[from 0x10000000 to 0x1001FFFF];
define region IRAM region = mem:[from 0x1FFFC000 to 0x1FFFFFFF];
define region DRAM region = mem:[from 0x20000000 to 0x2001FBFF];
define region CSTACK region = mem:[from 0x2001FA00 to 0x2001FFFF];
/*-Sizes-*/
define symbol   ICFEDIT size cstack   = 0x400;
define symbol   ICFEDIT size heap     = 0x200;
/**** End of ICF editor section. ###ICF###*/


define block CSTACK    with alignment = 8, size =   ICFEDIT size cstack     { };
define block HEAP      with alignment = 8, size =   ICFEDIT size heap       { };

initialize by copy { readwrite };
do not initialize  { section .noinit };

initialize by copy { readwrite, section .textrw,
                     object flash.o, };  // 添加 flash.o 全部目标


place at address mem:  ICFEDIT intvec start   { readonly section .intvec };
place in TEXT region                { readonly };
place in IRAM region                { readwrite, zi };
place in CSTACK region              { block CSTACK,block HEAP};
/*将.myFuntionSection section的数据链接到地址0x20200100 */
place in DRAM region                { section .myFuntionSection };
```

此时可以通过映射文件（.map）看到 flash.o 的内部函数已经链接到 0x1FFFC009 地址了。

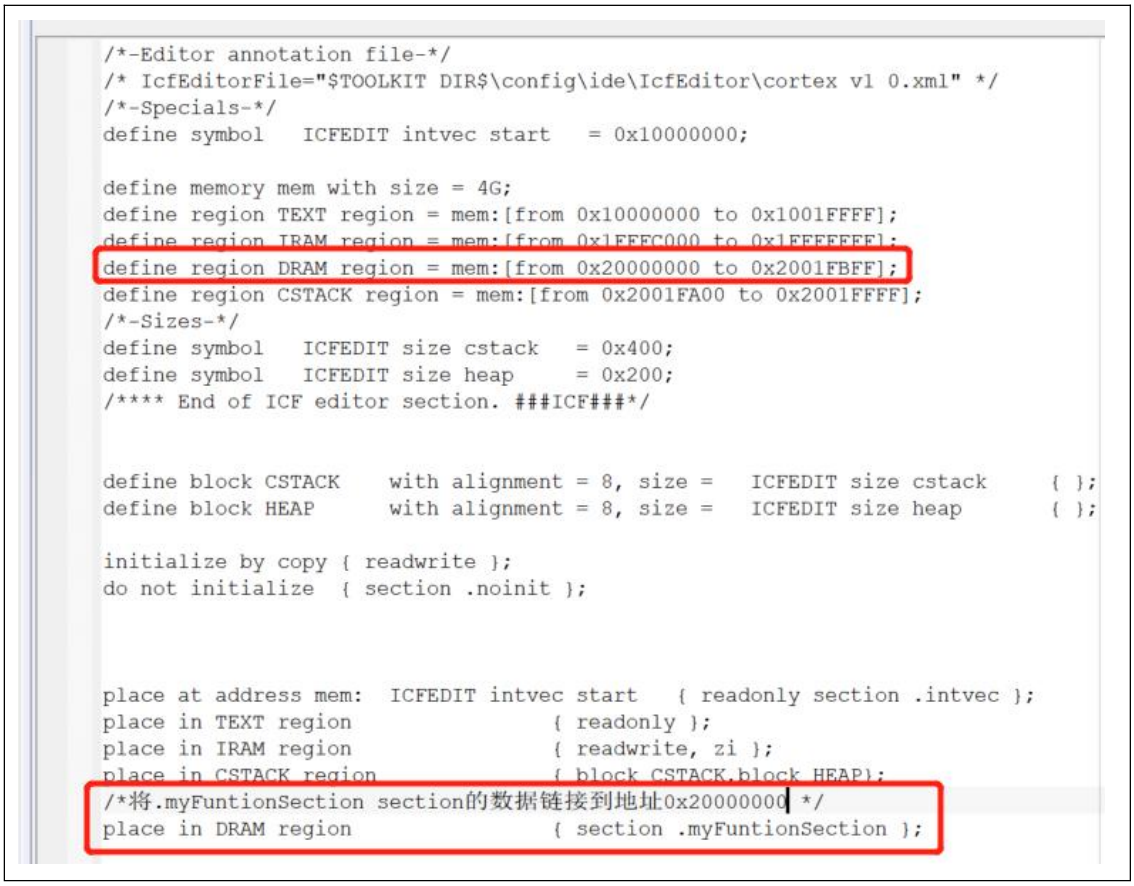
图 3-26. 查看其映射文件（iled_blinky.map）文件相关函数位置

spc1168.icf * | main.c | main.c | APP.map

```
ADCPPU5 IRQHandler      0x10001a4f    0x2  Code  Gb  isr.o [1]
AES IRQHandler          0x10001a73    0x2  Code  Gb  isr.o [1]
BusFault Handler        0x10001a03    0x2  Code  Gb  isr.o [1]
CLOCK ConfigurePLLWithRCO
                        0x10001205   0xdc  Code  Gb  clock.o [1]
CLOCK EnableModule      0x100012e1   0x24  Code  Gb  clock.o [1]
CLOCK GetModuleClock    0x10001325   0x2a  Code  Gb  clock.o [1]
CLOCK GetModuleDiv      0x10001305   0x20  Code  Gb  clock.o [1]
CLOCK IRQHandler        0x10001a13    0x2  Code  Gb  isr.o [1]
CLOCK InitWithRCO       0x100010fd  0x108  Code  Gb  clock.o [1]
CRC IRQHandler          0x10001a71    0x2  Code  Gb  isr.o [1]
CSTACK$$Base            0x2001fa00          --   Gb  - Linker created -
CSTACK$$Limit           0x2001fe00          --   Gb  - Linker created -
DebugMon Handler        0x10001a09    0x2  Code  Gb  isr.o [1]
Delay Init              0x1000180d    0x8  Code  Gb  delay.o [1]
Delay Us                0x10001815    0xe  Code  Gb  delay.o [1]
EGAP IRQHandler         0x10001a69    0x2  Code  Gb  isr.o [1]
FLASH SetTiming         0x1fffc009  0x130  Code  Gb  flash.o [1]
GPIO EDGE IRQHandler    0x10001a19    0x2  Code  Gb  isr.o [1]
GPIO LEVEL IRQHandler   0x10001a17    0x2  Code  Gb  isr.o [1]
HardFault Handler       0x100019ff    0x2  Code  Gb  isr.o [1]
I2C IRQHandler          0x10001a23    0x2  Code  Gb  isr.o [1]
MEM IRQHandler          0x10001a0f    0x2  Code  Gb  isr.o [1]
```

也可以设置相关的 section, section 会在这个 RAM 区域，让特定的函数在这个 sction 区域里面运行，也就是在 RAM 运行。如图 3-27 示。代码再按照如图 3-28 所示。则 myfuntion 函数就会运行再相关的区域。最终查看映射文件如图 3-29 所示。

图 3-27. 指定函数在 RAM 空间运行

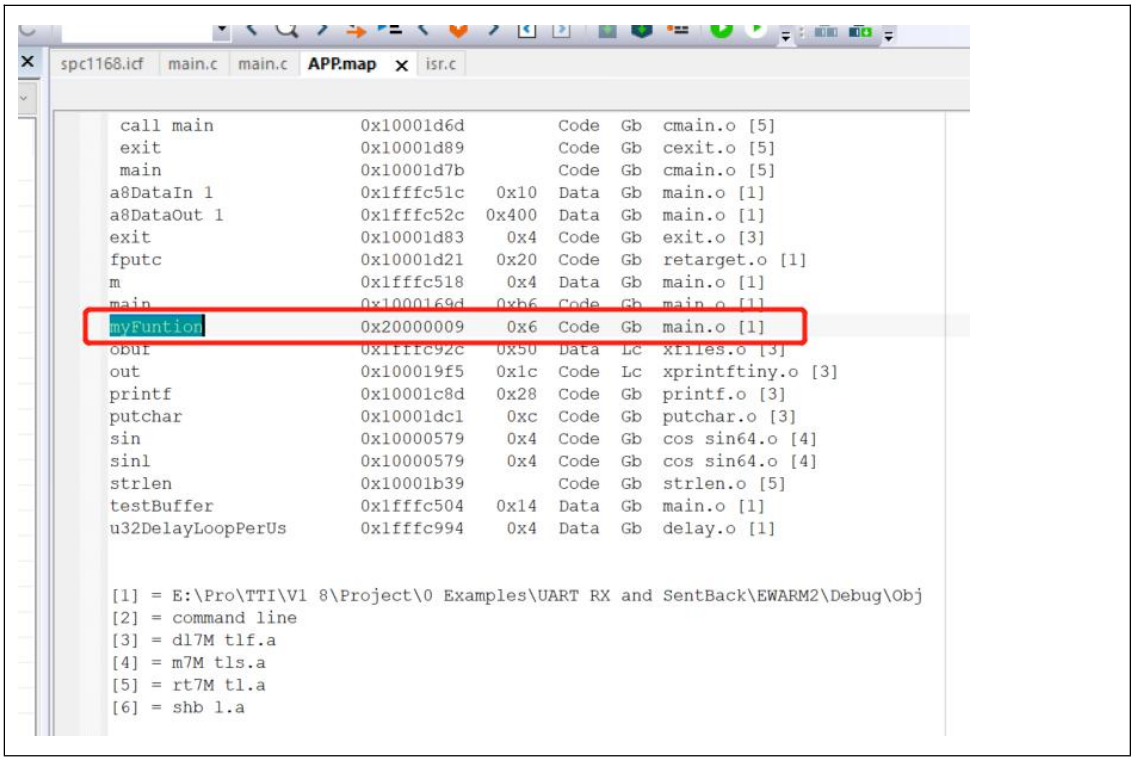


```
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT DIR$\config\ide\IcfEditor\cortex v1 0.xml" */
/*-Specials-*/
define symbol   ICFEDIT intvec start   = 0x10000000;

define memory mem with size = 4G;
define region TEXT region = mem:[from 0x10000000 to 0x1001FFFF];
define region IRAM region = mem:[from 0x1FFFC000 to 0x1FFFFFFF];
define region DRAM region = mem:[from 0x20000000 to 0x2001FBFF];
define region CSTACK region = mem:[from 0x2001FA00 to 0x2001FFFF];
/*-Sizes-*/
define symbol   ICFEDIT size cstack   = 0x400;
define symbol   ICFEDIT size heap     = 0x200;
/**** End of ICF editor section. ###ICF###*/


define block CSTACK    with alignment = 8, size =   ICFEDIT size cstack     { };
define block HEAP      with alignment = 8, size =   ICFEDIT size heap       { };

initialize by copy { readwrite };
do not initialize  { section .noinit };



place at address mem:  ICFEDIT intvec start   { readonly section .intvec };
place in TEXT region                { readonly };
place in IRAM region                { readwrite, zi };
place in CSTACK region              { block CSTACK,block HEAP};
/*将.myFuntionSection section的数据链接到地址0x20000000 */
place in DRAM region                { section .myFuntionSection };
```

图 3-28. 指定函数加载到 RAMCODE 里面

```
#pragma location = ".myFuntionSection"
void myFuntion()
{
   printf("hello world.\r\n");
}
// 第二种写法
/*
void myFuntion( ) @".myFuntionSection"
{
    printf("hello world.\r\n");
}
*/
```

图 3-29. 指定函数加载到 DRAM 里面



利用 **__ramfunc** 修饰符来链接函数到 RAM 地址。这个修饰符是 IAR 链接器能特殊识别的，主要适用重定向单个关键函数。比如我们用它来修饰 myfuntion () 函数。

图 3-30. 指定函数链接到 RAM 里面

```
//Note: __ramfunc 仅重定向被修饰的函数体本身代码，而该函数中调用的其他函数体本身并不受
__ramfunc void myFuntion()
{
   printf("hello world.\r\n");
}
```

4 JFLASH

4.1 JFLASH 配置

4.1.1 JLINK 配置添加烧录文件

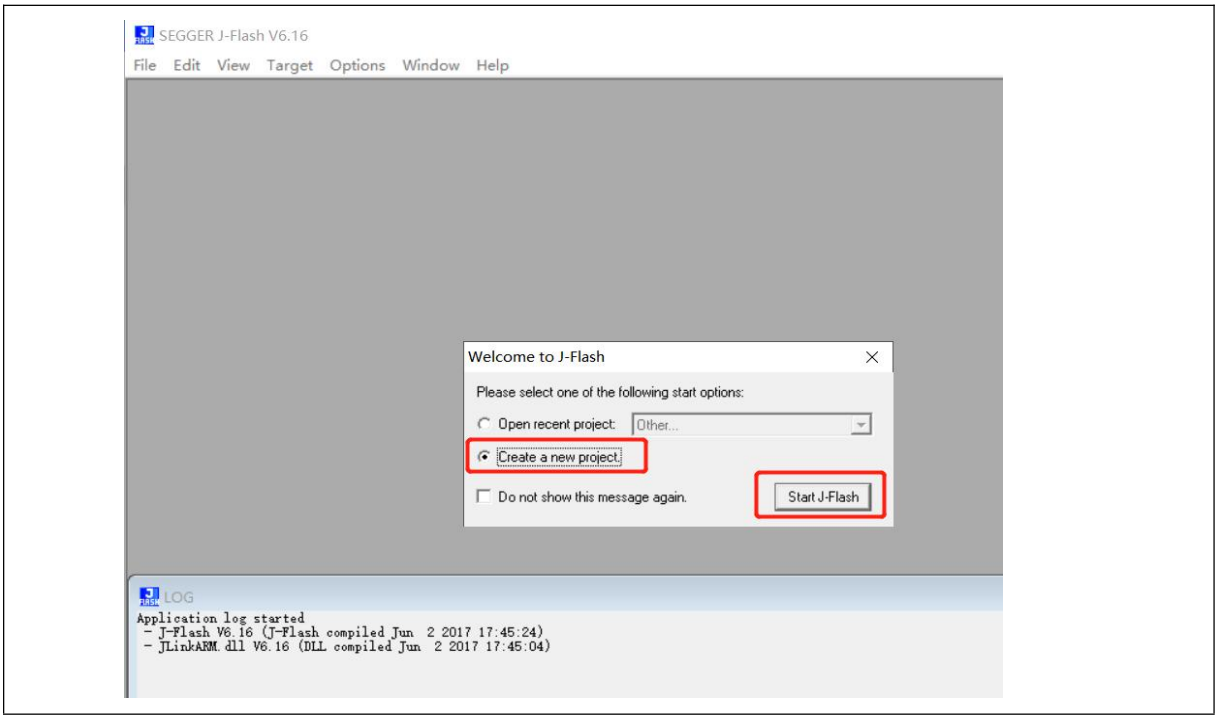
参考《 JLINK 配置 Device 》章节。

4.2 用 J-Flash 软件烧录 Hex 文件了

4.2.1 运行 JFlash.exe，新建一个工程，选择要烧录的芯片。

打开 Jflash 软件后，显示如图 4-1 所示，如果第一配置，则选择 Create a new project。如果不是选择前一次的配置文件。

图 4- 1. Create a new project 配置



点击如图 4- 2 的箭头，再弹出的对话框选择对应的 MCU，如图 4- 3 所示。

图 4- 2. Target Device 配置

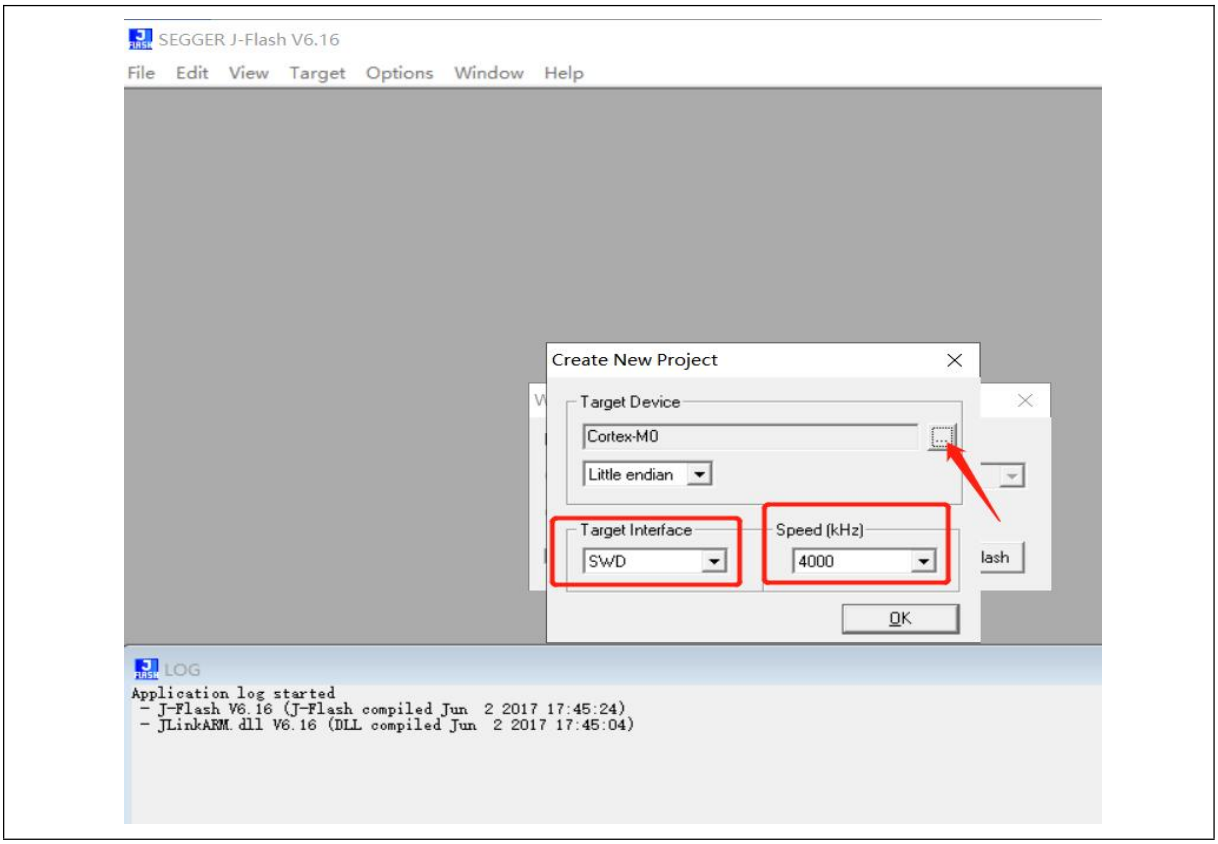
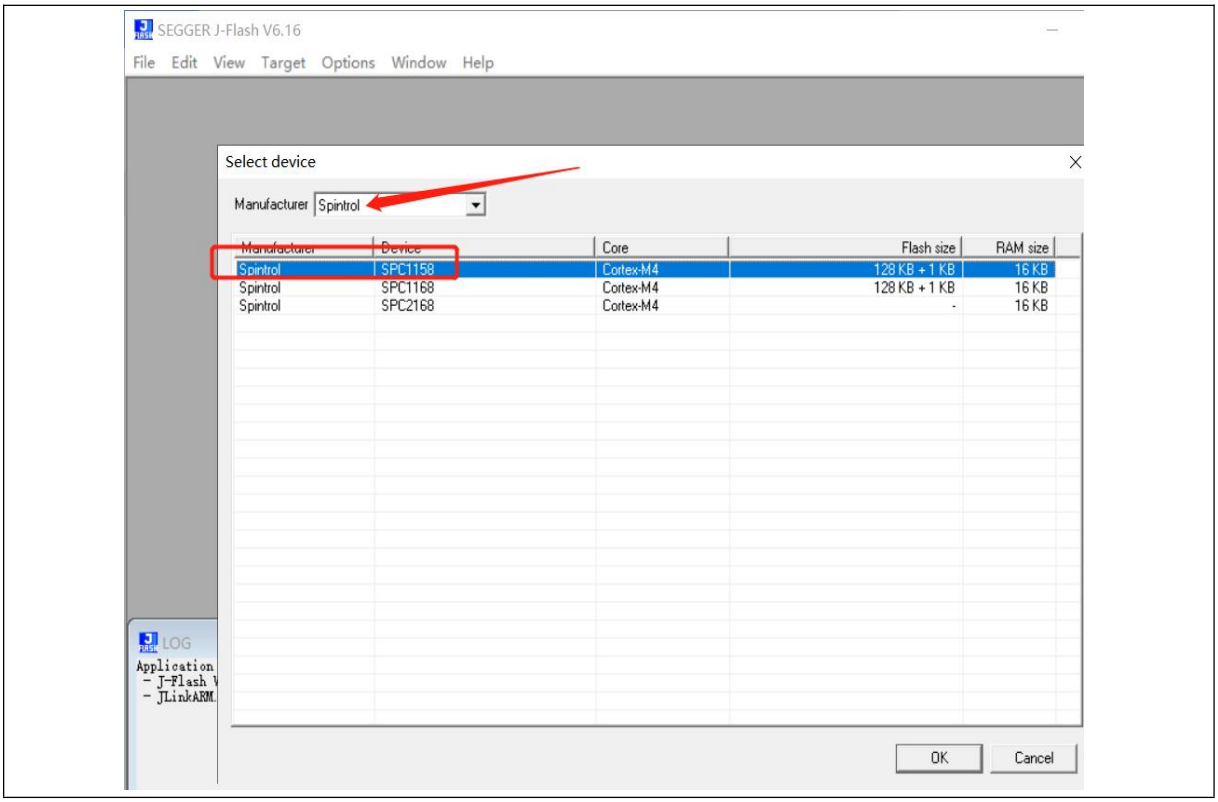


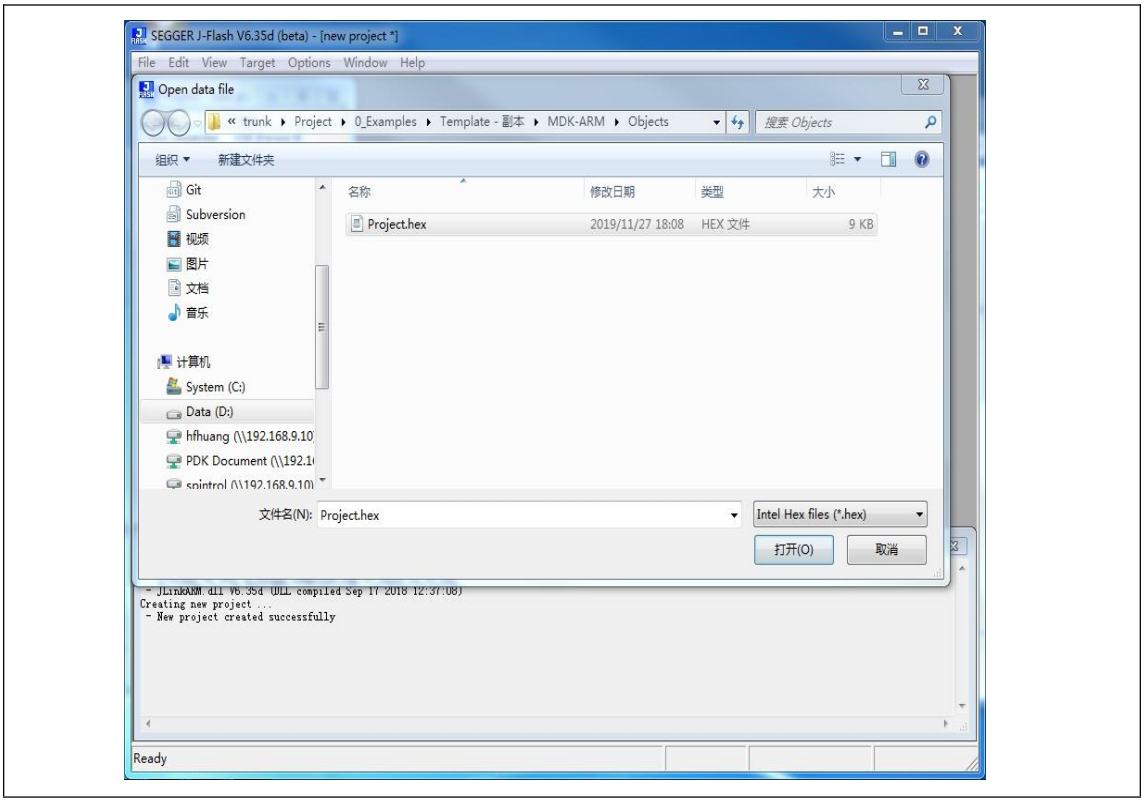
图 4- 3. Target Device 选择



4.2.2 打开需要下载的文件

点击 File ->Open data file，选择要下载的 Hex 文件，如图 4- 4 所示。

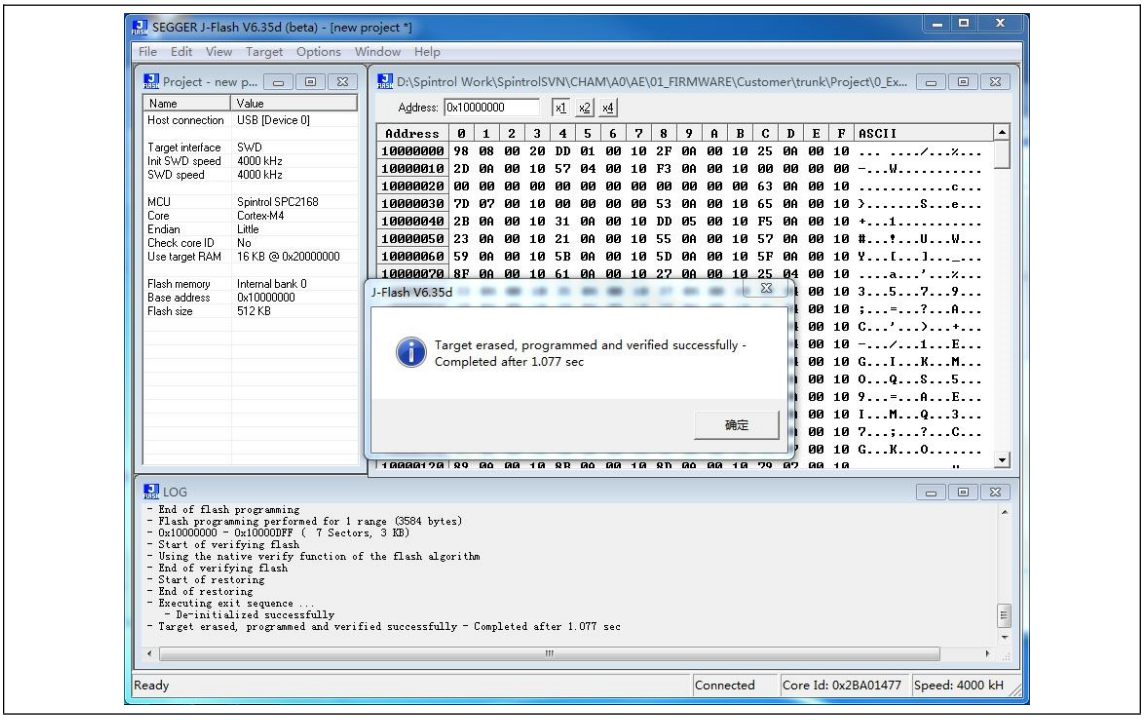
图 4- 4. 烧录文件 Hex 选择



4.2.3 烧录固件

点击 Target -> Production Programming，开始烧录芯片。也可以直接使用 F3 擦除/F7 烧录快捷键进行下载，烧录完成后如图 4-5 所示。

图 4-5. 烧录固件



4.2.4 读取固件

点击 Target -> Manual Programming->Read back->Entire chip，开始读取芯片 flash 数据。如图 4-6 所示。最终读取成功如图 4-7 所示。

图 4-6. 读取固件

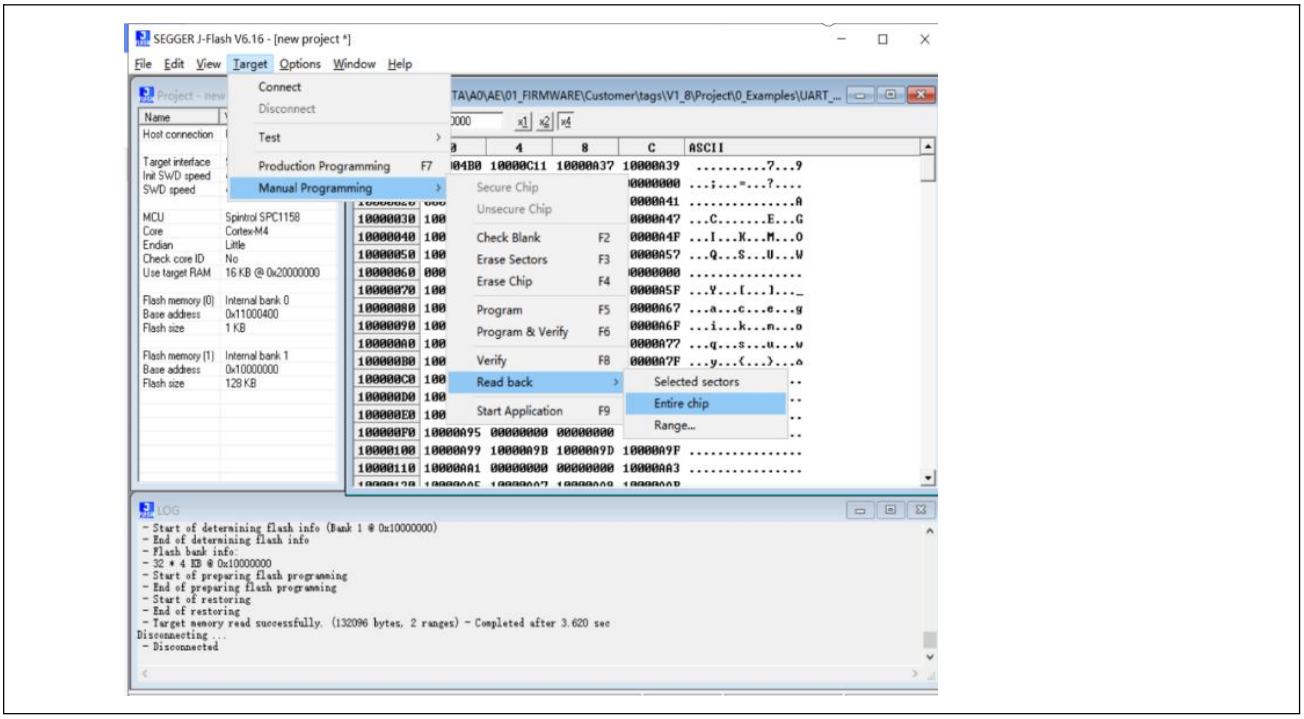
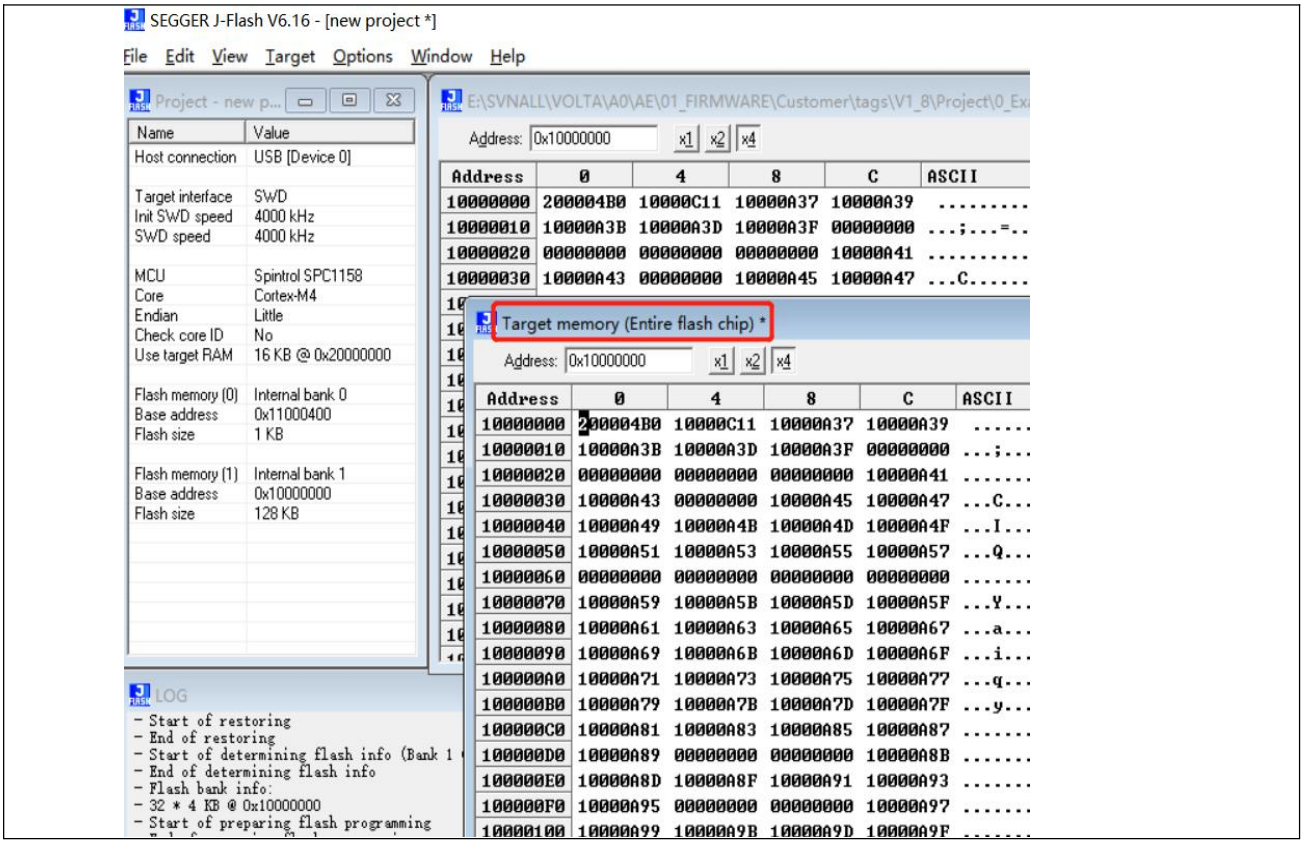


图 4-7.读取固件成功



4.2.5 注意事项

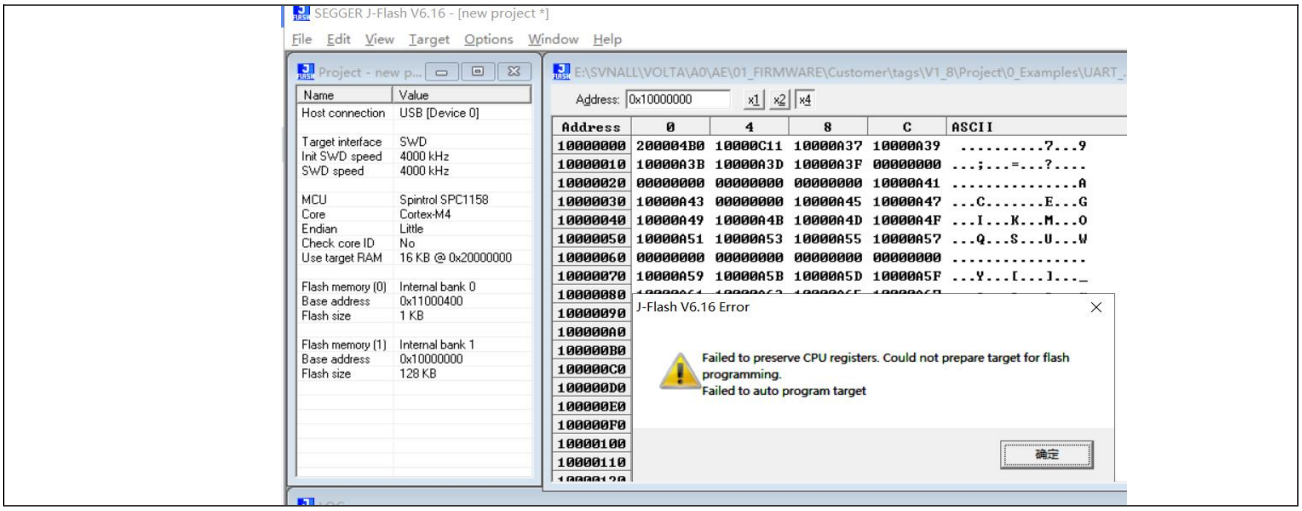
使用快捷键操作，擦除烧写读取都能正常。

但是如果用工具栏的选项，**先 connnect 再开始擦除，或者烧写**，就会遇到如图 4-8 等问题。

这类有些是软件版本导致的，有些需要重复操作两次才成功。

建议 **target->connect 选项不要先点击，直接就进行擦除烧写读取操作。**

图 4-8.烧录问题



4.2.6 烧录脚本

a) WINDOWS 的 bat 脚本运行需要用到 JLFASH 的指令，JFLASH 需要把执行路径添加到 windows 的环境变量里面。如图 4-9 所示。

b) 当配置好后，按照《4.2.1》章节新建 JFLASH 工程，然后 FILE->Project 保存.JFLASH 文件如图 4-11 所示。

c) 可以新建 .bat 后缀的文本，然后按照图 4-11 所示添加内容。

-openprjD:\SPC1158.jflash 表示刚才步骤 b 保存的.jflash 文件。

-openE:\Project.hex -connect -auto -exit 表示需要烧录的 hex 文件。

ECHO J-Flash Production: Error!!! 表示烧录失败打印错误.

d) 烧录失败后如图 4-12 所示。

图 4-9. 添加 JFLASH 路径到环境变量

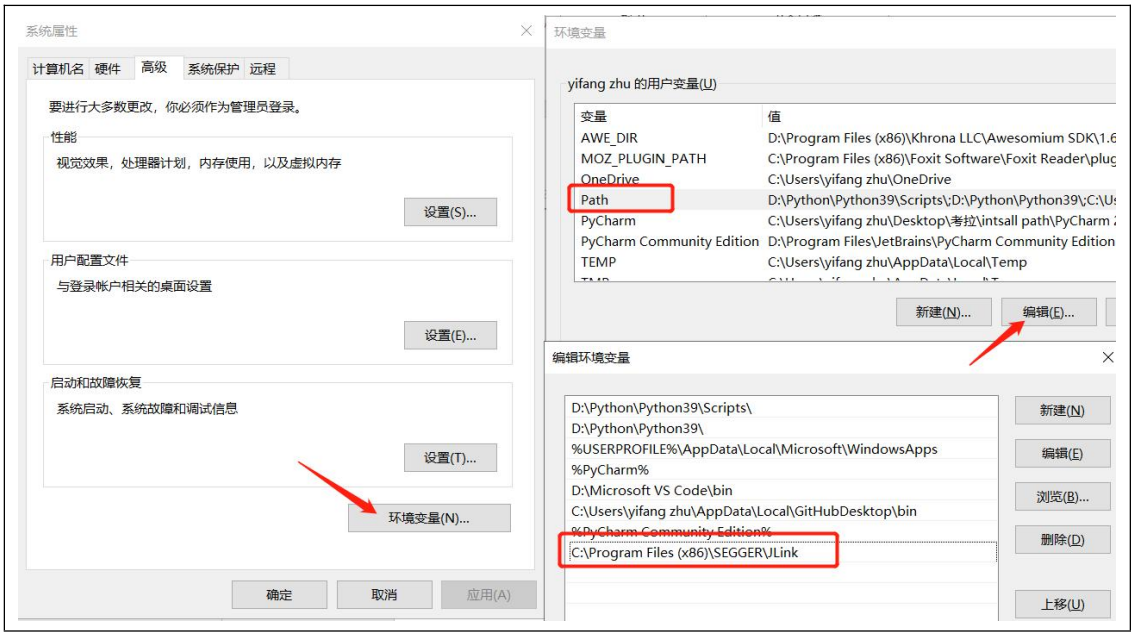


图 4-10. 保存.flash 文件

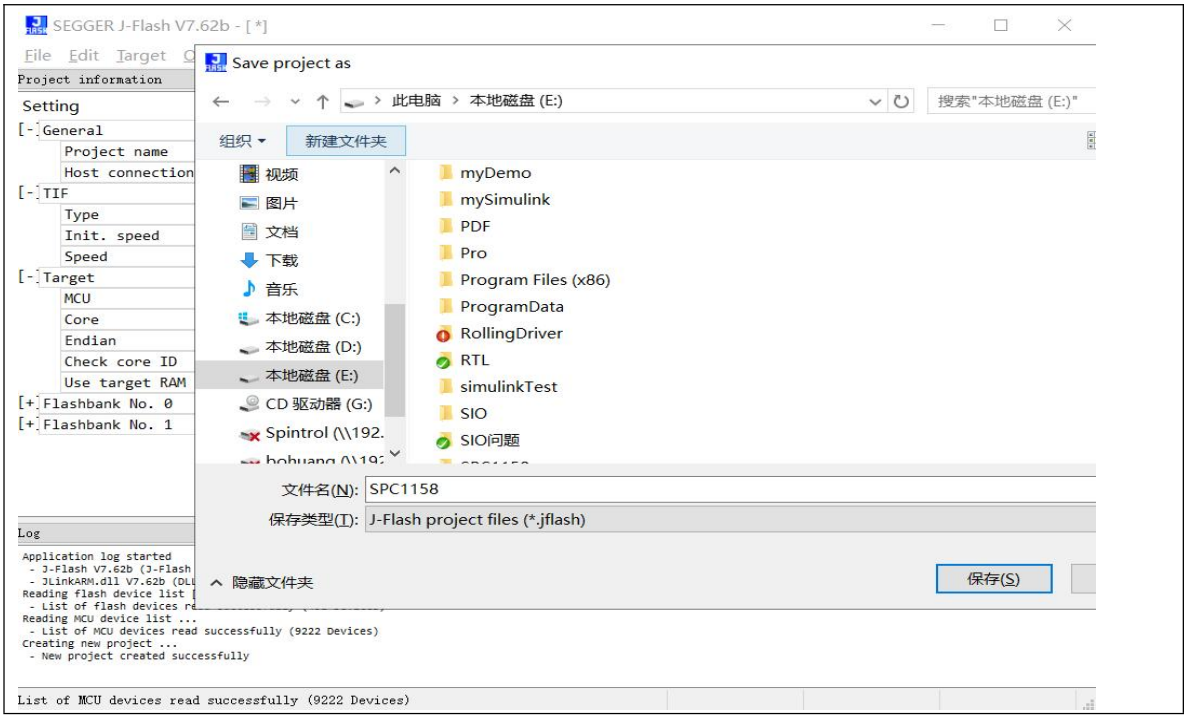


图 4- 11. JFLASH 下载脚本

```
@ECHO off

ECHO Start auto-production
JFlash.exe -openprjD:\SPC1158.jflash          -openE:\Project.hex -connect -auto -exit

IF ERRORLEVEL 1 goto ERROR
goto END
:ERROR
ECHO ----------------------------
ECHO J-Flash Production: Error!!!
ECHO ----------------------------

pause
:END
```

图 4- 12. 烧录失败

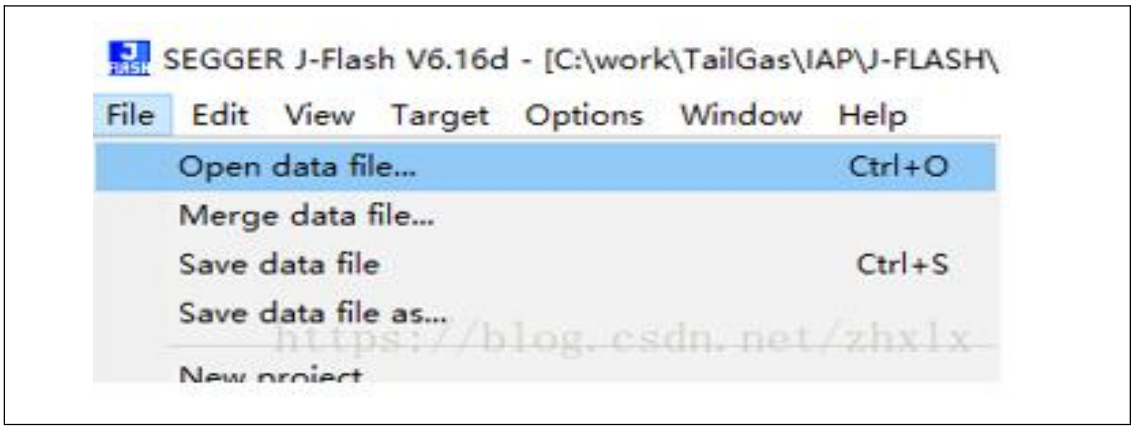
4.3 用 J-Flash 软件合并 Hex 文件

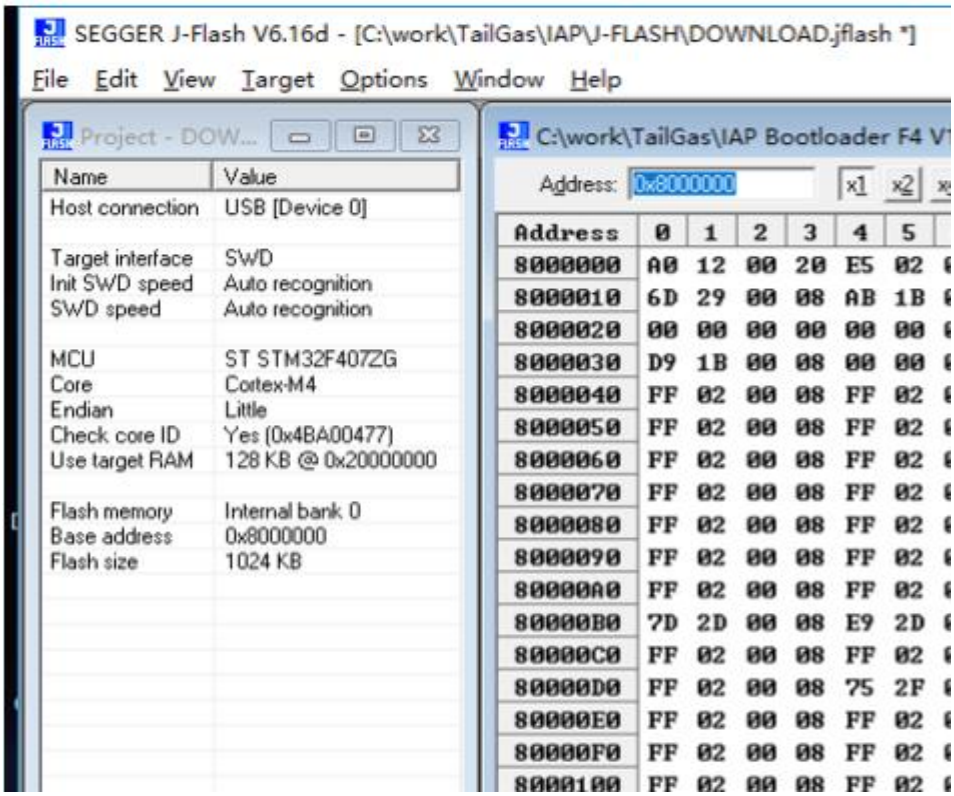
在开发过程中，一份 code 实现 IAP 升级功能作为，一份 code 实现 APP 应用功能。再生产中需要根据烧写两份程序，为了简化方便，需要将两个 hex 文件合并一起为一个 hex 文件。

4.3.1 打开 Jflash 软件并打开任意一个 hex 文件。

FILE-》open data file,然后选择需要合并的任意一个 hex 文件。

图 4- 13.打开任意一个 hex 文件

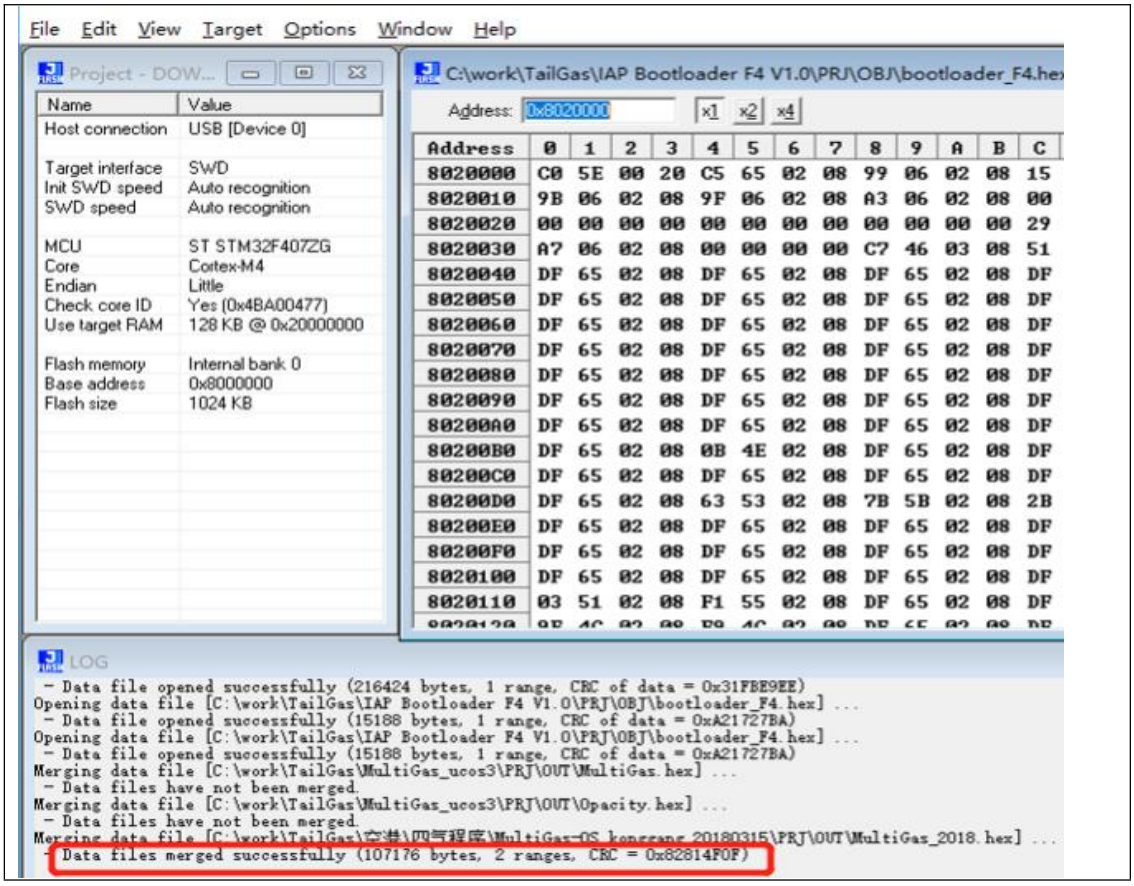




4.3.1 合并另外一个 hex 文件。

file->merge data file. 需要合并的另外一个 hex 文件。

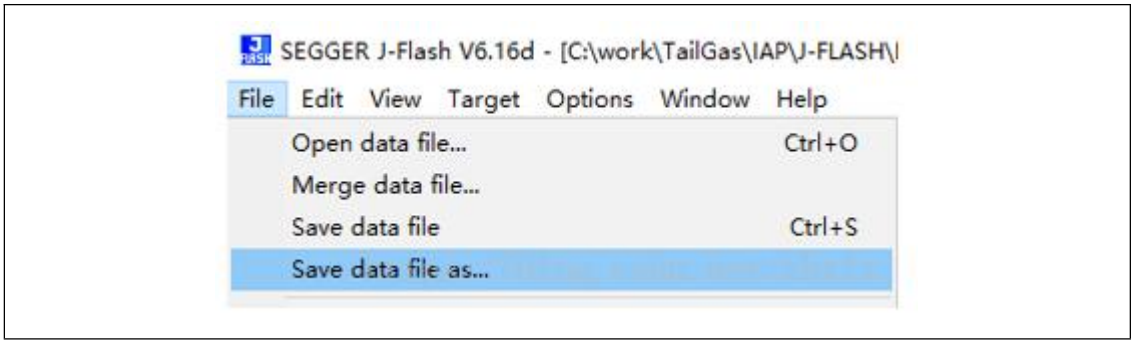
图 4- 14. 合并另外一个 hex 文件



4.3.2 保存合并后的 hex 文件。

file->save data file as...，保存合并好的 hex 文件。

图 4- 15.保存合并文件



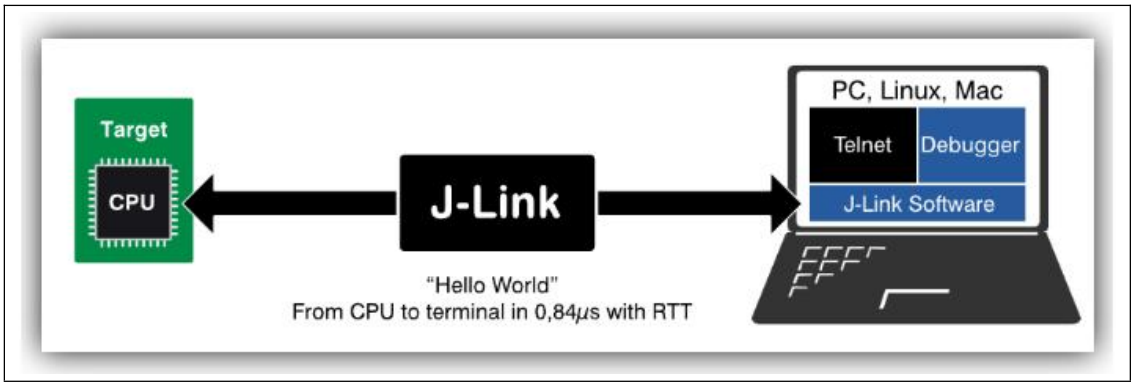
5 Segger Rtt

5.1 RTT 框图

使用 RTT(Real Time Transfer)，可以从目标微控制器输出信息，以及以很高的速度将输入发送到应用程序，而不会影响目标的实时行为。 SEGGER RTT 可与任何 J-Link 模型和支持后台内存访问的任何受支持的目标处理器一起使用，这些处理器是 Cortex-M 和 RX 目标。

RTT 在双向上都支持多个通道，直至主机和目标，都可以用于不同目的，并为用户提供最大的自由度。 RTT 默认实现是每个方向使用一个通道，该通道用于可打印的终端输入和输出。 使用 J-Link RTT Viewer，该通道可用于多个“虚拟”终端，从而仅使用一个目标缓冲区即可打印到多个窗口（例如，一个用于标准输出，一个用于错误输出，一个用于调试输出）。 例如，可以使用一个附加的（主机）通道来发送配置文件或事件跟踪数据。使用 RTT 可以把 MCU 的串口资源释放出来了。与 J-Link RTT Viewer 、J-Link RTT Logger 、J-Link RTT Client 等工具进行数据采集查看调试开发 。

图 5-1：RTT 框图结构示意图

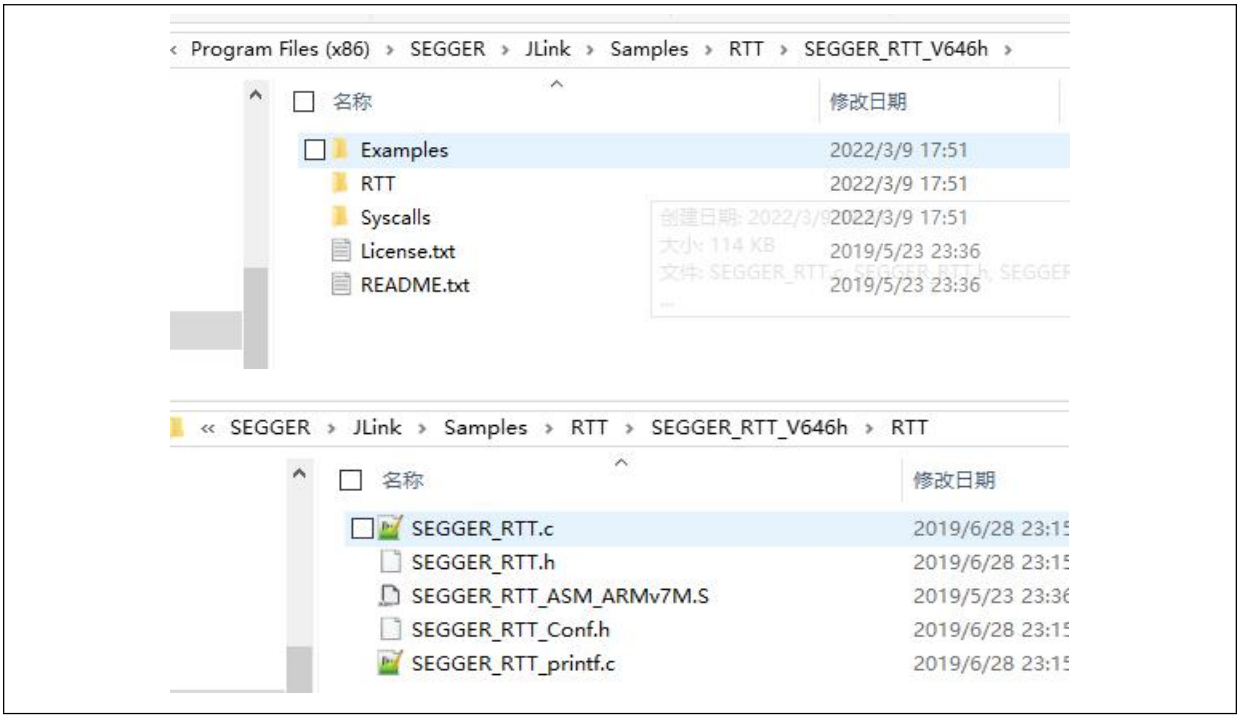


5.2 项目工程上添加库

5.2.1 获取 RTT 库

从官网或者从 C:\Program Files (x86)\SEGGER\JLink_VXXX\Samples\RTT\SEGGER_RTT_VXXX.zip 获取 RTT 库。本文测试使用 SEGGER_RTT_V616.zip 进行测试验证。如图 5-2 所示。

图 5-2. RTT_V616 库



5.2.2 工程代码添加库

需要将 SEGGER_RTT_V616\RTT 目录下的文件添加到工程。如图 5-3 所示。并且将头文件目录加入工程，如图 5-4 所示。

图 5-3. 工程添加 RTT 文件

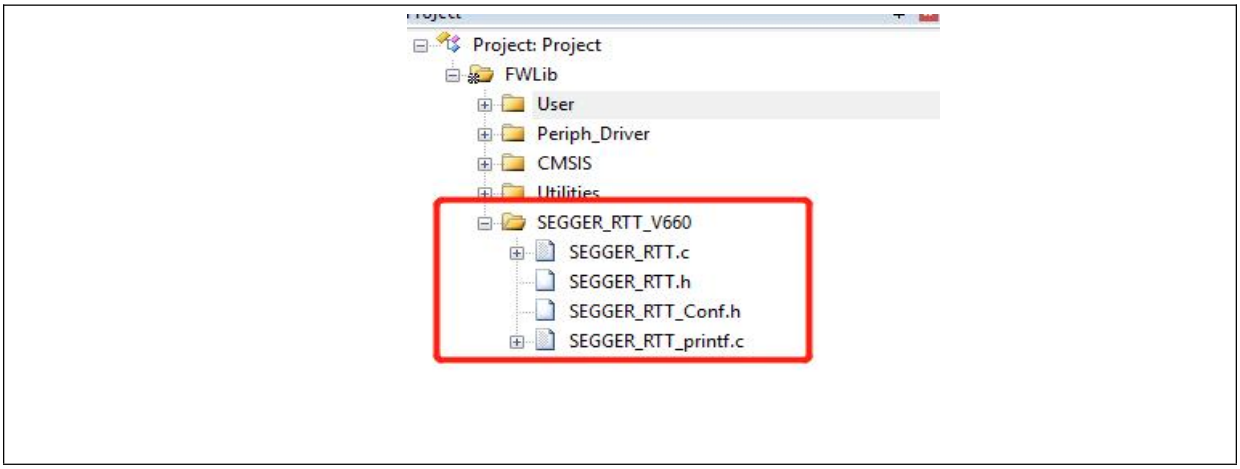
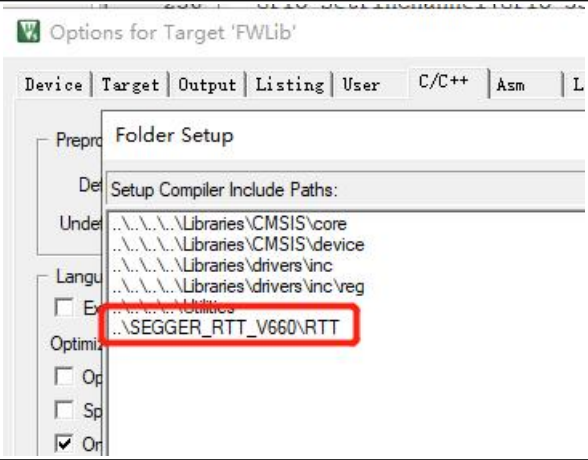


图 5-4. 工程添加 RTT 头文件



5.2.3 RTT 初始化发送数据

RTT 初始化并向 RTT 通道 0 发送数据。

注意需要加\r\n 换行符。RTT Viewer 等工具会识别不到 BUFFER 结束。

图 5- 5.RTT 初始化以及通道 0 发送数据

```
#include "math.h"
#include<stdio.h>
#include<stdlib.h>
#include "SEGGER_RTT.h"
#include "SEGGER_RTT_Conf.h"
int main(void)
{
        SEGGER_RTT_Init();
        while(1)
        {
          SEGGER_RTT_printf(0,"channel 0 test\r\n");
          Delay_Ms(1);
        }
}
```

图 5- 6.RTT 初始化以及通道 1 发送数据

```
#include "math.h"
#include<stdio.h>
#include<stdlib.h>
#include "SEGGER_RTT.h"
#include "SEGGER_RTT_Conf.h"
 char a8DataIn_1[BUFFER_SIZE_DOWN];
 char a8DataOut_1[BUFFER_SIZE_UP];
int main(void)
{
        SEGGER_RTT_Init();
        SEGGER_RTT_ConfigDownBuffer(1, "DataIn", &a8DataIn_1[0],
                                    sizeof(a8DataIn_1),
```

```
                                        SEGGER_RTT_MODE_NO_BLOCK_SKIP);

        SEGGER_RTT_ConfigUpBuffer(1, "DataOut", &a8DataOut_1[0],
                                        sizeof(a8DataOut_1),
                                        SEGGER_RTT_MODE_BLOCK_IF_FIFO_FULL);

        while(1)
        {
            SEGGER_RTT_printf(0,"channel 0 test\r\n");
            SEGGER_RTT_TerminalOut(1,"channel 1 test\r\n");
            Delay_Ms(1);
        }
}
```

5.2.4 RTT 接收数据

图 5- 7. MCU 使用 RTT 通道 0 接收数据

```
#include "math.h"
#include<stdio.h>
#include<stdlib.h>
#include "SEGGER_RTT.h"
#include "SEGGER_RTT_Conf.h"
  char a8DataIn_1[BUFFER_SIZE_DOWN];
  char a8DataOut_1[BUFFER_SIZE_UP];
int main(void)
{
        SEGGER_RTT_Init();
        SEGGER_RTT_ConfigDownBuffer(1, "DataIn", &a8DataIn_1[0],
                                        sizeof(a8DataIn_1),
                                        SEGGER_RTT_MODE_NO_BLOCK_SKIP);

        SEGGER_RTT_ConfigUpBuffer(1, "DataOut", &a8DataOut_1[0],
                                        sizeof(a8DataOut_1),
                                        SEGGER_RTT_MODE_BLOCK_IF_FIFO_FULL);

        while(1)
        {
                char acIn[4];
                char cIn='a';
                uint8_t NumBytes=0;
                NumBytes = SEGGER_RTT_Read(0, &acIn[0], sizeof(acIn));
                //cIn = SEGGER_RTT_GetKey();
                if(NumBytes>0)
```

```
                {
                    SEGGER_RTT_printf(0,"0R:%c \r\n",acIn[0]);
                }
                Delay_Ms(1);
            }
        }
```

5.3 J-Link RTT Viewer

安装 JLINK 驱动后，一般都安装了 RTT Viewer，根据 C:\Program Files (x86)\SEGGER\JLink 类似目录查找。

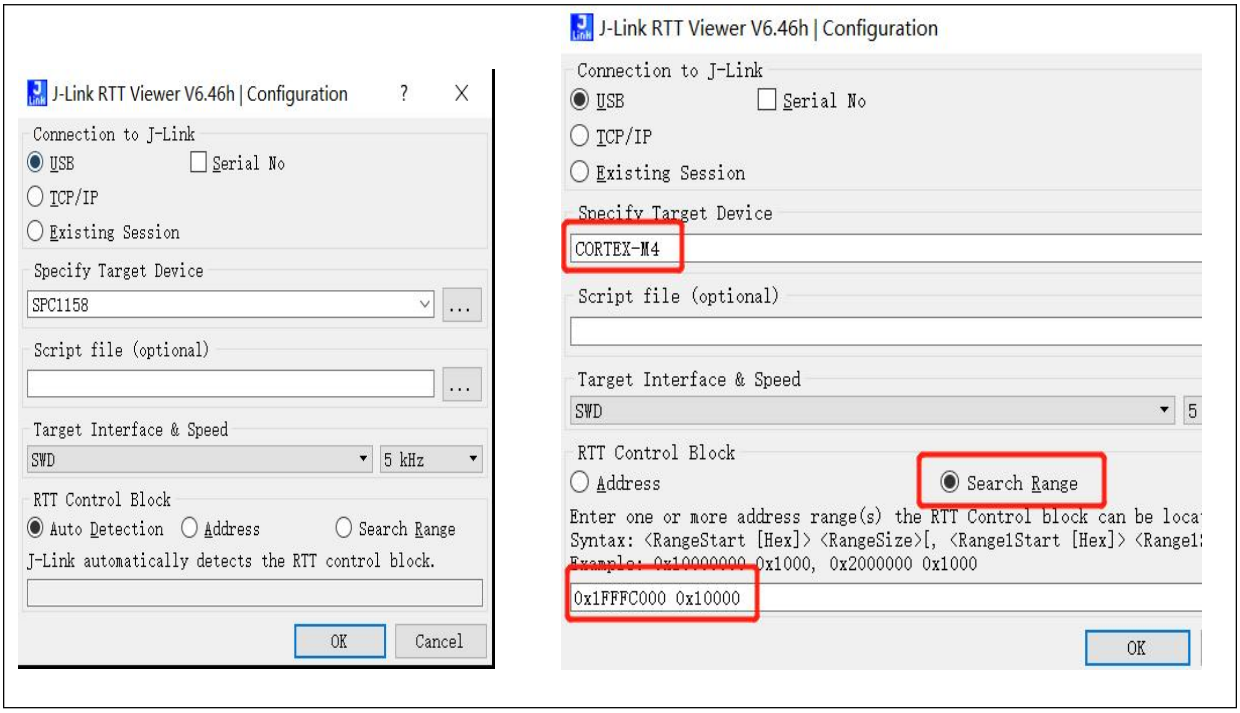
点击 RTT Viewer 时候或者使用菜单 File->connect 可弹出 RTT Viewer 配置选项， 按照如图 5-8 进行配置。

NOTE：

使用 RTT Viewer 需要先按照《 JLINK 配置 Device》章节进行添加设备。

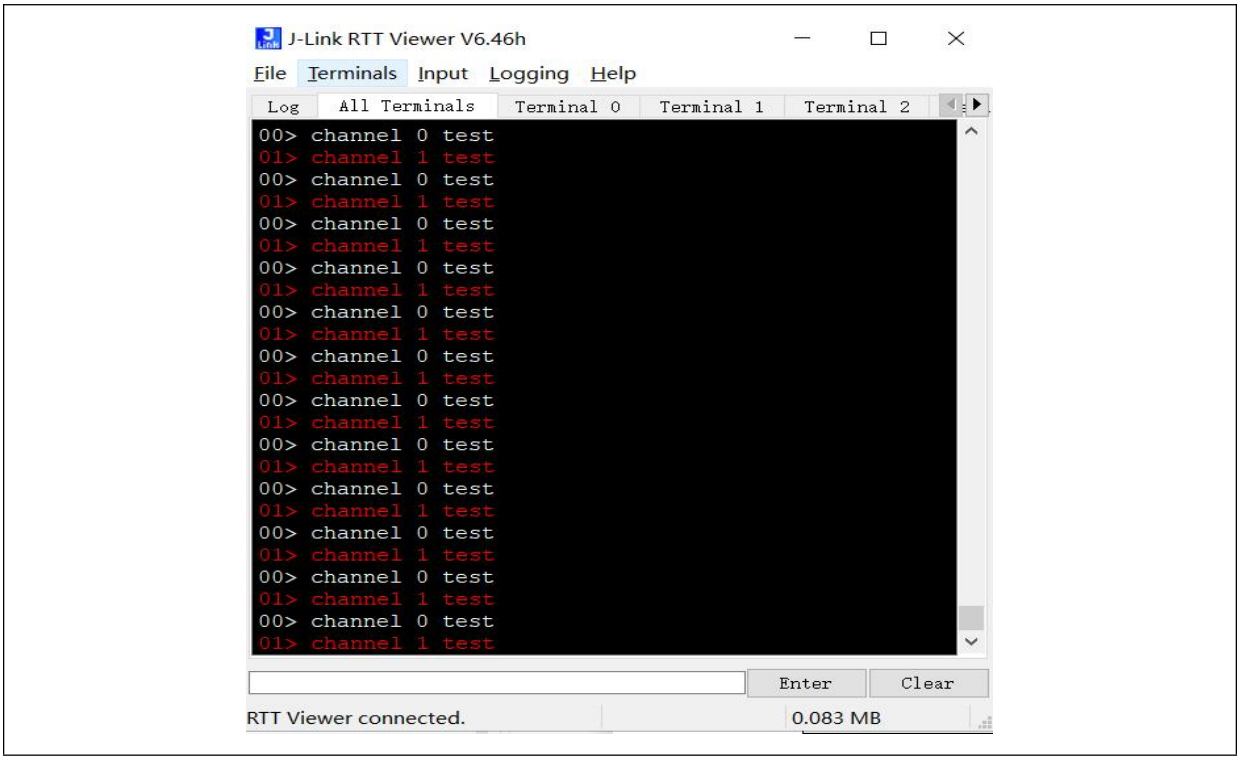
如果 Specify target Device 选项选择 CORTEX-M4（MCU 内核类型），RTT Control Block 选项需要选择 Search Range，并且配置 RAM 起始地址与大小。

图 5-8. RTT Viewer 配置



最终效果如下：

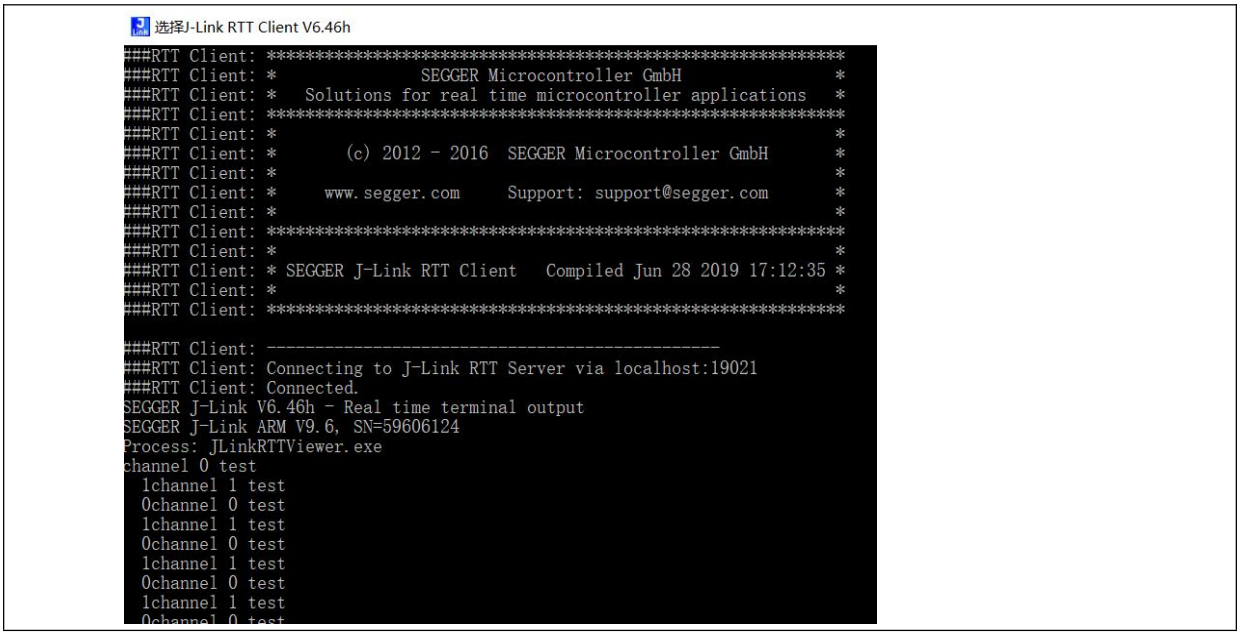
图 5-9. RTT Viewer 效果图



5.4 J-Link RTT Client

JLINK RTT Client 需要配合 JLINK RTT Viewer 或者 JLINK RTT Logger 或者 keil debug 调试模式进行使用。代码按照图 5-6 配置后再配合 JLINK RTT Viewer，采集数据如图 5-10 所示。

图 5-10. RTT Client 效果图



5.5 J-Link RTT Logger

JLinkRTTLogger 按照图 5-6 所配置工程代码后，需要按照如图 5-11 指定 SRAM 地址给

_SEGGER_RTT，按照如图 5-12 执行执行启动 RTTlogger。最终效果如图 5-13 所示。详细的 RTT 指令如图 5-14 所示。

图 5-11. RTT 指定固定 SRAM 地址_SEGGER_RTT

```
__attribute__((at(0x20001000)))      SEGGER_RTT_PUT_CB_SECTION(SEGGER_RTT_CB_ALIGN(SEGGER_RTT_CB _SEGGER_RTT))  ;
```

图 5-12. RTTLogger 启动

```
"C:\Program Files (x86)\SEGGER\JLink\JLinkRTTLogger.exe"  -Device SPC1158  -if swd -Speed 4000
-RTTAddress 0x20001000  -RTTChannel  0  %~dp0\RTT.log
```

图 5-13. RTT Logget 采集数据

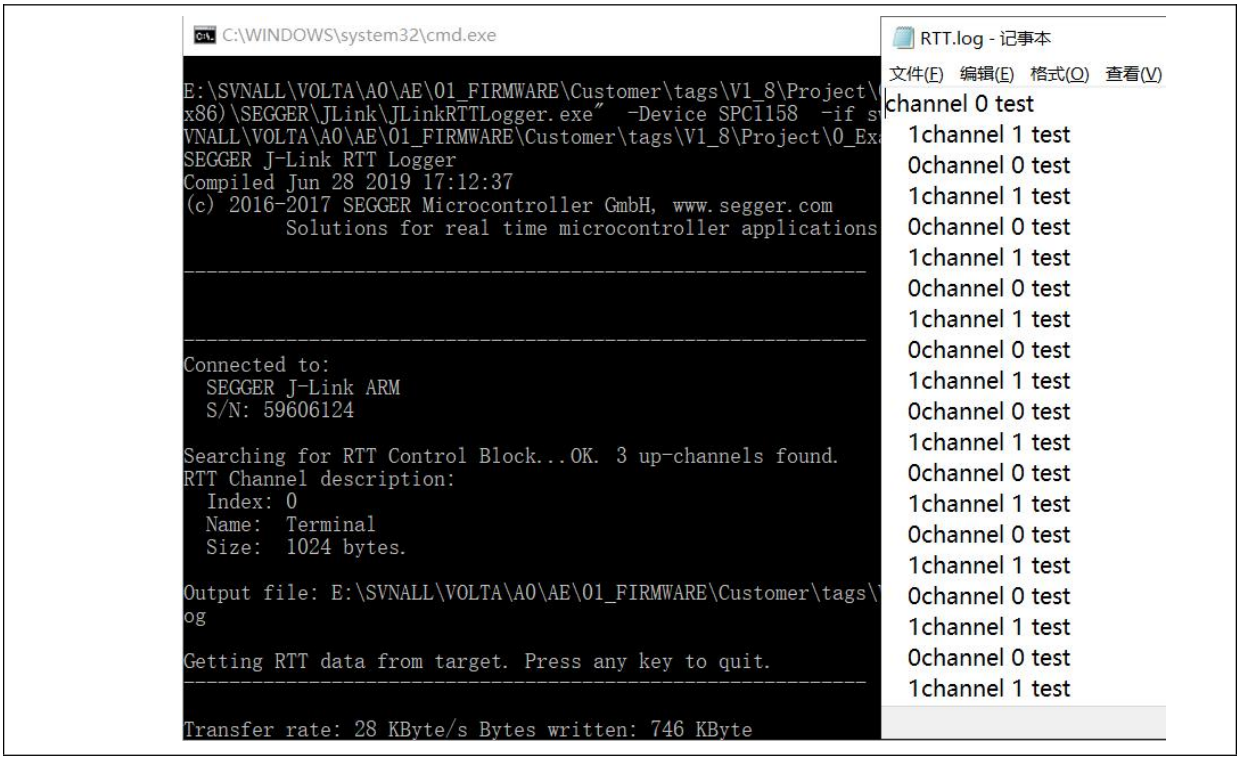


图 5-14. RTTLogger 指令

Command line options:

Command	Explanation
-?	Shows commandline options
-Device <DeviceName>	Sets target device to <DeviceName>
-if <Interface>	Sets target interface to <Interface>
-Speed <SpeedInKHZ>	Sets speed to <SpeedInKHZ>
-SelectEmuBySN <SN>	Connects to J-Link with serial numbet <SN> via USB
-RTTAddress <RTTAddress>	Sets RTT address to <RTTAddress>
-RTTChannel <RTTChannel>	Sets RTT channel to <RTTChannel>

6 Jscope 使用

JScope 软件下载地址:https://www.segger.com/j-link-j-scope.html 。

安装完成之后，找到安装目录，如图所示，会看到 J-Scope 的启动文件和他的帮助文件。或者点击软件上面的 Help->J-scope Manual 就会弹出帮助文档。

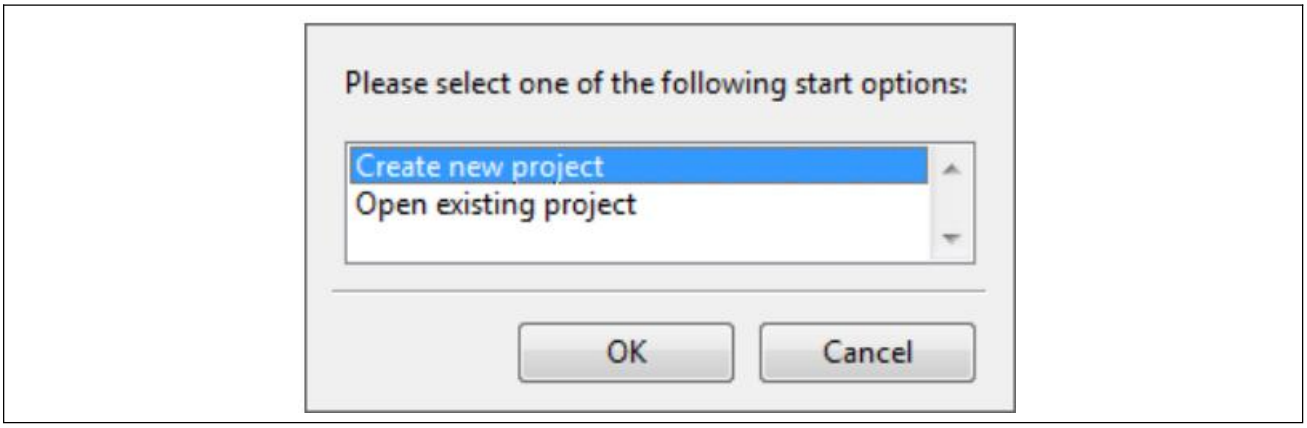
UM08028_JScope.pdf 是官方的帮助文档。

本文使用 Setup_JScope_V611m.exe

6.1 Jscope 配置

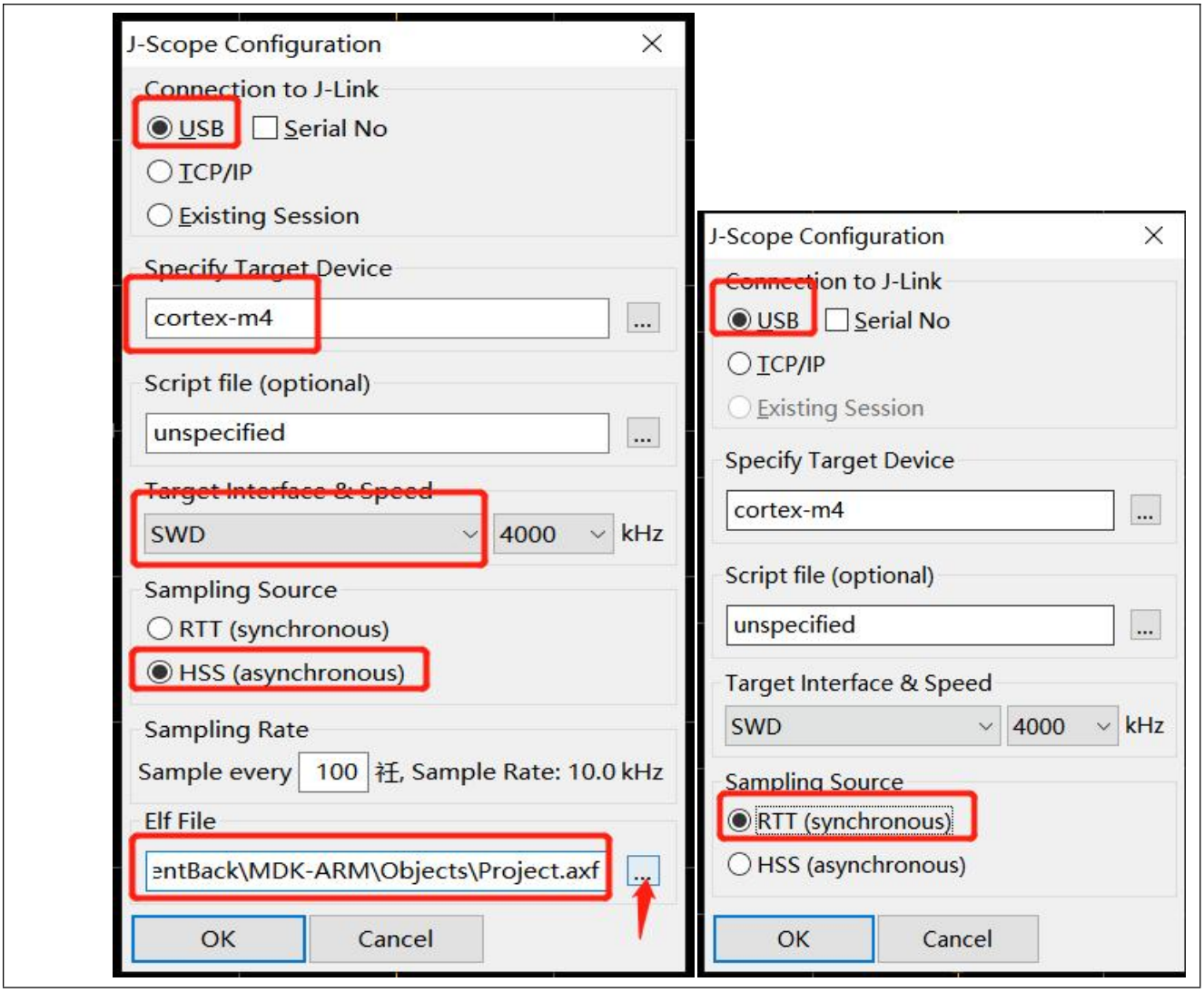
在使用 jscope 之前请确认是否目标板子 MCU 已经烧写了固件， 可以通过 J-Flash, the J-Link Commander 或者 Keil IDE 来烧录固件。开始 Jscope 时候，出现如图 6-1 点击使用“Create new project”和点击 OK。

图 6-1. 创建新工程



然后 J-Scope 配置对话框将会打开。

图 6-2. J-Scope 配置



Asynchronous Mode (HSS)

HSS 模式的工作间隔是定期对内存位置进行采样。因此，采样率和一个 elf 文件是强制性的。ELF 文件用于确定被采样符号的内存地址。代码中声明变量为“volatile”是为了更好的更快将它们写入 memory 中。HSS 模式可以随时随地都可以连接目标板，不影响目标板的正常功能，不需要额外的资源。

Synchronous Mode (RTT)

为了使用 RTT 模式，必须在目标应用程序中实现 RTT。有关 RTT 使用的更多信息，请参阅第《Jscope 高速 RTT 模式采集》章节，应用程序将其与 J-Scope 一起使用。类似串口输出数据到 PC 的串口软件。可以有 2MB/s 的数据吞吐量，即使有 512 字节的缓冲区，也可以达到 1MB/s 的速度。

J-Link 的链接选择

Serial No，TCP/IP，Exitsting Session，USB。常用的基本是 USB 接口的。

Target Device 配置

SPC11XX/SPD11XX 系列 都填上 contex-M4.

注意:有些 win10 选择 device 时候会出现闪退现象，直接填写 cortex-m4 就可以了。

Script file(optional)

脚本文件，可不配置。

Sampling Source

一般选择 HSS 模式， 如果选择 RTT 模式，则需要参考《Jscope 高速 RTT 模式采集》。

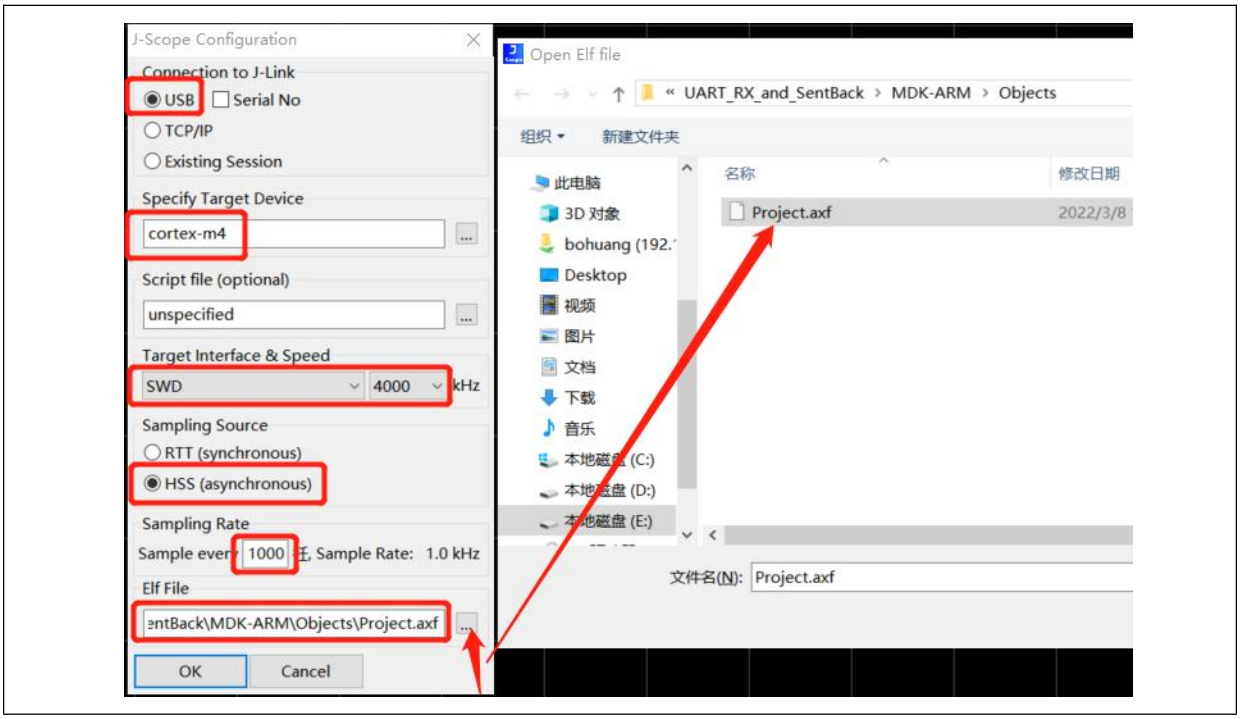
Sampling Rate

1000 us. 即 1.0khz。 HSS 模式速度较慢，采样速度基本固定在 1khz 左右。

ELF file

需要调试的项目工程目录编译出来的 xx.axf 文件。如图 6-3 所示。

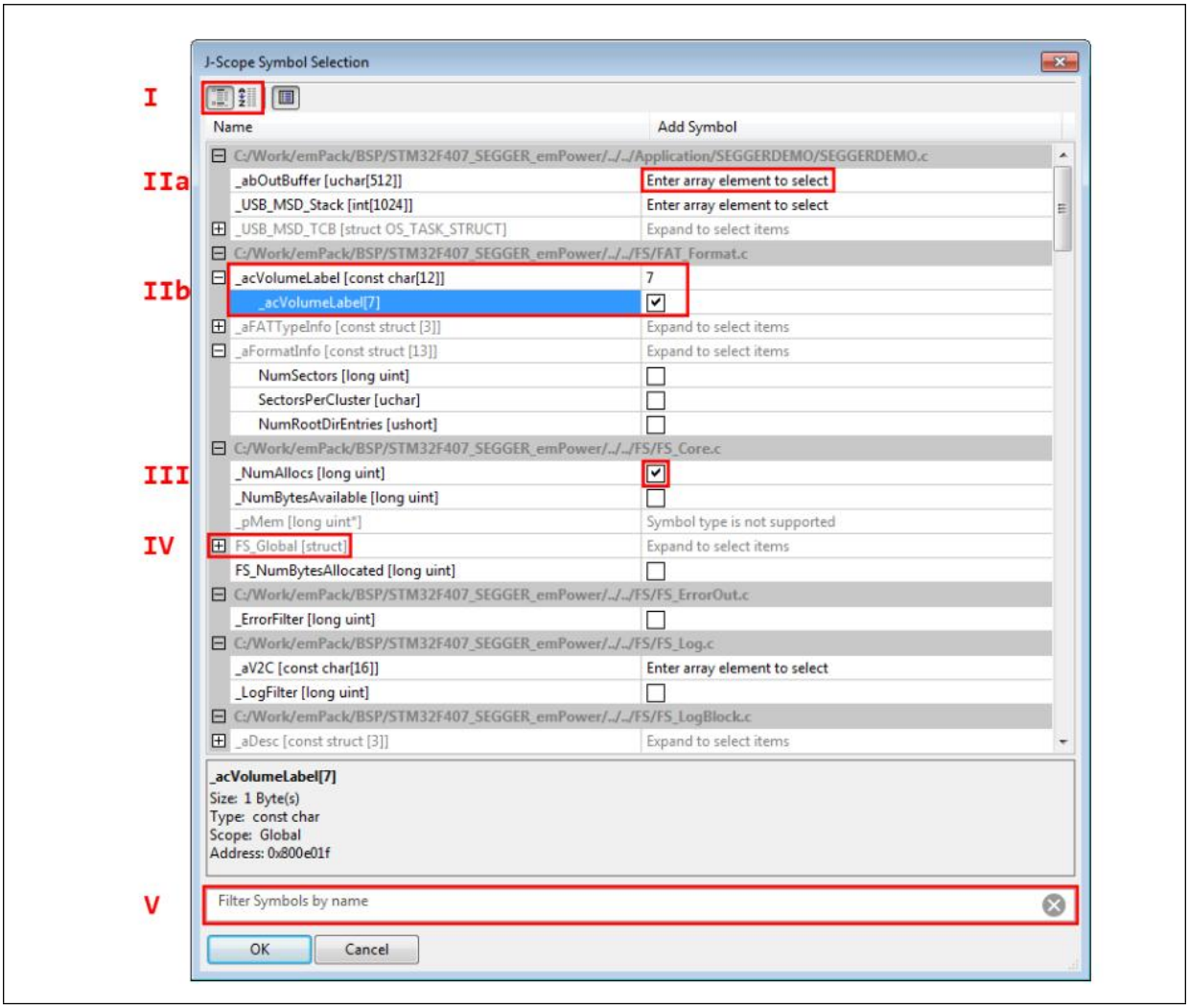
图 6-3. 选择 ELF FILE



6.2 Jscope 添加观察变量

当配置完成点击 OK 时候，就会进入到如
图 6-4 的界面。

图 6-4. 添加观察符号变量



I.分类模式

Categorized mode（默认）

将树视图中的符号显示为其编译单元文件(例如_NumAllocs 在 FS_Core.c 中定义)。

Alphabetical mode 按字母模式按字母顺序

II.数组元素的选择

虽然完整的数组不可用于采样，但可以对支持的基本类型的数组元素进行采样。为了对特定的数组元素进行采样，可以在相应的字段(IIa)中填充数字。用户按 enter 确认输入后，就将加入对话框(IIb)

III。基本类型的选择支持

通过启用该复选框，可以将符号添加到已采样符号的列表中。数据结构成员的选择可以将特定结构的成员通过单击结构名称旁边的“+”显示在树视图中。

V.搜索

提供了一个不区分大小写的搜索。搜索结果中不显示无法选择的符号。结构显示为长字符串，它们的名称或至少有一个成员的名称包含搜索字符串。用户可以右键单击一个符号，以显示单击符号的所有子项，即使它们与搜索不匹配。

6.3 Jscope 数据显示

点击开始采集按钮 ，则启动采集，此时如果弹出如图 6-6，则无需理会，直接点击 OK 即可。

然后就可以看到符号变量的图形。如图 6-7。

图 6-5. 开始采集

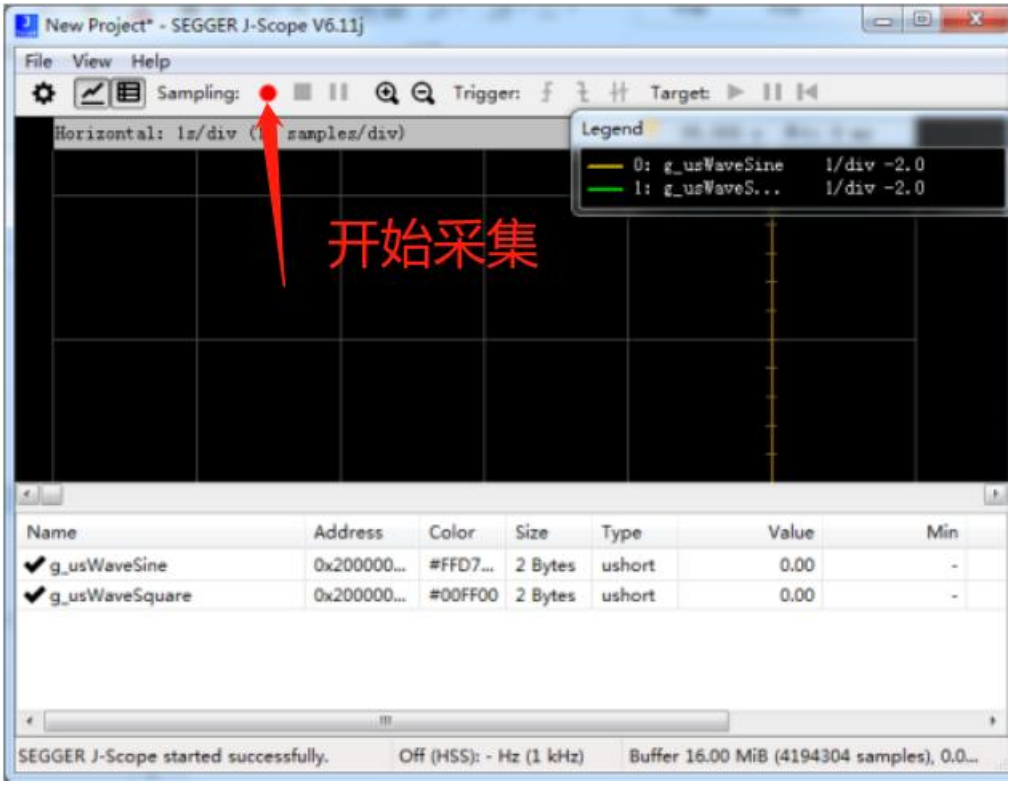


图 6-6. Jscope 启动报 Warning

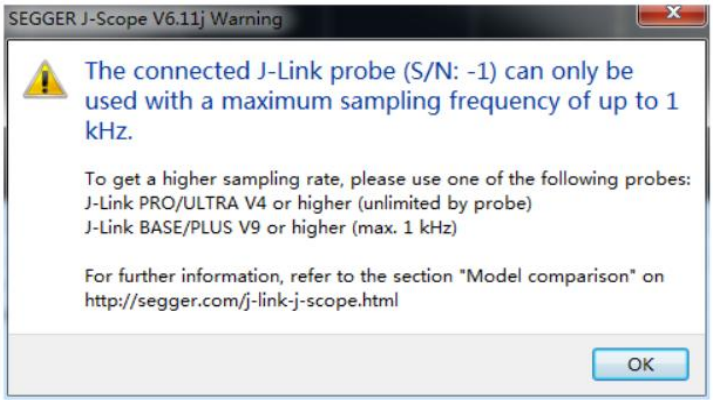
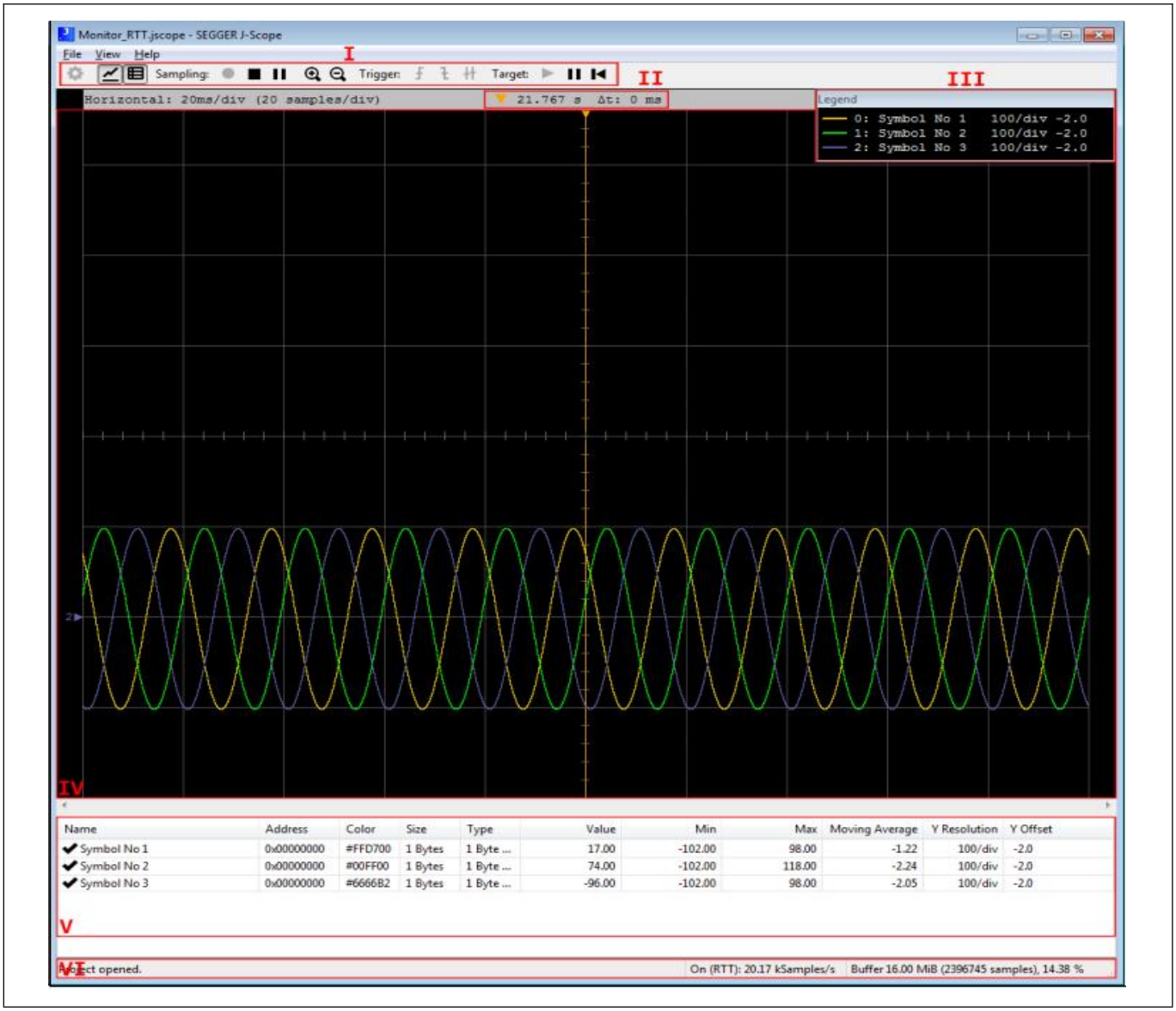


图 6-7. Jscope 主界面



6.4 Jscope control

当鼠标对着变量或者某些局部位置右键时候，可以看到弹出的的一些操作选项。如

图 6-8 所示。

这里不做更详细的描述。

图 6-8. 操作选项

Function	GUI Input	Shortcut
Graph Area Settings		
Zoom in (X-Axis)	Tool-bar	Left, Ctrl + Wheel up
Zoom out (X-Axis)	Tool-bar	Right, Ctrl + Wheel down
Set trigger level	Drag Trigger Indicator	N/A
Set Zoom target	Drag ZTI	N/A
Change X-Axis scope	Move Scrollbar / Drag Graph Area	Wheel up / down
Symbol Settings *(Apply to the symbol currently selected)*		
Change color	Symbol Context menu	N/A
Show / Hide toggle	Symbol Context menu	Enter, Space
AC coupling toggle	Symbol Context menu	N/A
Move up	Symbol Context menu	PGUP
Move down	Symbol Context menu	PGDOWN
Zoom in (Y-Axis)	Symbol Context menu	+
Zoom out (Y-Axis)	Symbol Context menu	-
Change draw style	Symbol Context menu	N/A
Y-Offset up	Drag Base Line Indicator	Ctrl + +
Y-Offset down	Drag Base Line Indicator	Ctrl + -
Y-Resolution up	Symbol Context menu	+
Y-Resolution down	Symbol Context menu	-
Remove Symbol	Symbol Context menu	Delete
Add Symbol	Symbol Context menu	N/A
Trigger Controls		
Set trigger on rising edge	Tool-bar	N/A
Set trigger on falling edge	Tool-bar	N/A

Function	GUI Input	Shortcut
Set trigger on both edges	Tool-bar	N/A
Sampling Controls		
Start Sampling	Tool-bar	F5
Stop Sampling	Tool-bar	F9
Pause Sampling	Tool-bar	N/A
Start Target	Tool-bar	N/A
Halt Target	Tool-bar	N/A
Reset Target	Tool-bar	N/A
General		
Exit J-Scope	File Menu	Alt + X, Alt + F4
J-Scope Manual	Help Menu	F11
About Dialog	Help Menu	F12
Project Management		
Open Project	File Menu	Ctrl + O
New Project	File Menu	Ctrl + N
Save Project	File Menu	Ctrl + S
Save Project as	File Menu	Ctrl + Shift + S
Recent Projects	File Menu	N/A
Export CSV	File Menu	Ctrl + D
Export RAW	File Menu	Ctrl + R
Import RAW	File Menu	Ctrl + T

6.5 Jscope 高速 RTT 模式采集

6.5.1 RTT 配置

如章节《项目工程上添加库》所示，在工程上添加 RTT 库。

6.5.2 RTT 代码配置

代码先初始化 SEGGER_RTT_Init，然后进行 buffer 配置 SEGGER_RTT_ConfigUpBuffer 配置，最后通过 SEGGER_RTT_Write 发送数据到 Jscope。最终代码如图 6-9 所示。

图 6-9. 工程代码 RTT 发送正旋波

```
#include "math.h"
#include<stdio.h>
#include<stdlib.h>
#include "SEGGER_RTT.h"
#include "SEGGER_RTT_Conf.h"

char a8DataIn_1[BUFFER_SIZE_DOWN];
char a8DataOut_1[BUFFER_SIZE_UP];

void SendToJscope(void) {
  //
  // RTT block structure
```

```
    //
    #pragma pack(push, 1)
    struct {
      signed int     Sine1;
      signed int     Sine2;
      signed int     Sine3;
      float           count;
    } acValBuffer;
    #pragma pack(pop)

    static float XCnt=0;
    float _SinA,_SinB,_SinC;
    int     Sine1,Sine2,Sine3;

    static unsigned    int i = 0;
    #define SINUS_SAMPLE_M    100
    #define PI 3.14159265358979323846f
    #define PI_RAD_Coef (3.14159265358979323846f*0.00555555555f)

    i++;
    i = i%SINUS_SAMPLE_M;

    XCnt = (float)i/SINUS_SAMPLE_M*(360)*(PI_RAD_Coef); // calcs sin data for 0-90?(0-PI/2)
    _SinA = XCnt - 30*(PI_RAD_Coef);
    _SinB = XCnt - 20*(PI_RAD_Coef);
    _SinC = XCnt + 0*(PI_RAD_Coef);

    Sine1 = (signed int)(sin((float)(_SinA))*100000);
    Sine2 = (signed int)(sin((float)(_SinB))*100000);
    Sine3 = (signed int)(sin((float)(_SinC))*100000);

    //
    // send over RTT channel
    //
    acValBuffer.Sine1 = Sine1;
    acValBuffer.Sine2 = Sine2;
    acValBuffer.Sine3 = Sine3;
    acValBuffer.count = XCnt;
    SEGGER_RTT_Write(1, &acValBuffer, sizeof(acValBuffer));

}
int main(void)
{
        SEGGER_RTT_Init();
```

```
        SEGGER_RTT_ConfigDownBuffer(1, "DataIn", &a8DataIn_1[0],
                                    sizeof(a8DataIn_1),
                                    SEGGER_RTT_MODE_NO_BLOCK_SKIP);

        SEGGER_RTT_ConfigUpBuffer(1, "JScope_i4i4i4f4", &a8DataOut_1[0],
                                    sizeof(a8DataOut_1),
                                    SEGGER_RTT_MODE_BLOCK_IF_FIFO_FULL);
        while(1)
        {
          SendToJscope();
          Delay_Us(100);
        }
    }
```

图 6- 10. RTT 配置上传通道

```
SEGGER_RTT_ConfigUpBuffer(1, "JScope_i4i4i4f4", &a8DataOut_1[0],
                          sizeof(a8DataOut_1),
                          SEGGER_RTT_MODE_BLOCK_IF_FIFO_FULL);
```

注意:

- **JScope_FORMAT**

  如图 6- 10 配置通道 buffer 的时候，RTT 通道命名格式需要与发送数据格式相匹配。"JScope_FORMAT" 的定义目标应用程序写入缓冲区的数据格式，所有的变量都是由类型标志符和大小指示组成。数据包中的数据必须与 FORMAT 中声明的顺序相同。如果提供了时间戳，则需要将其声明为第一个变量。规则如图 6- 11 所示。

  JScope_t4u1i4

  表示一个 32bit 的时间戳，紧跟着一个 unsigned char，和一个 signed int。

  JScope_u4i1

  表示一个 unsigned int 与一个 signed char。

图 6- 11. 格式列表

Type	Supported Sizes	Note
t	4	Indicates that every packet is preceded by a 32bit value containing a timestamp in µs
i	1, 2, 4	Specifies a signed 8-, 16- or 32bit value
u	1, 2, 4	Specifies an unsigned 8-, 16- or 32bit value

Naming Examples

JScope_t4u1i4	Specifies that the data packets are headed by a 32bit time value, followed by an unsigned char and a signed int
JScope_u4i1	Specifies that the data packets contain an unsigned int, followed by a signed char

- **结构体**

  定义自己需要发送的结构体，如 JScope_i4i4i4f4 对应如下结构体。

图 6- 12. RTT 发送结构体数据

```
#pragma pack(push, 1)
struct {
signed int     Sine1;
signed int     Sine2;
signed int     Sine3;
float          count;
} acValBuffer;
#pragma pack(pop)
```

- **发送数据**

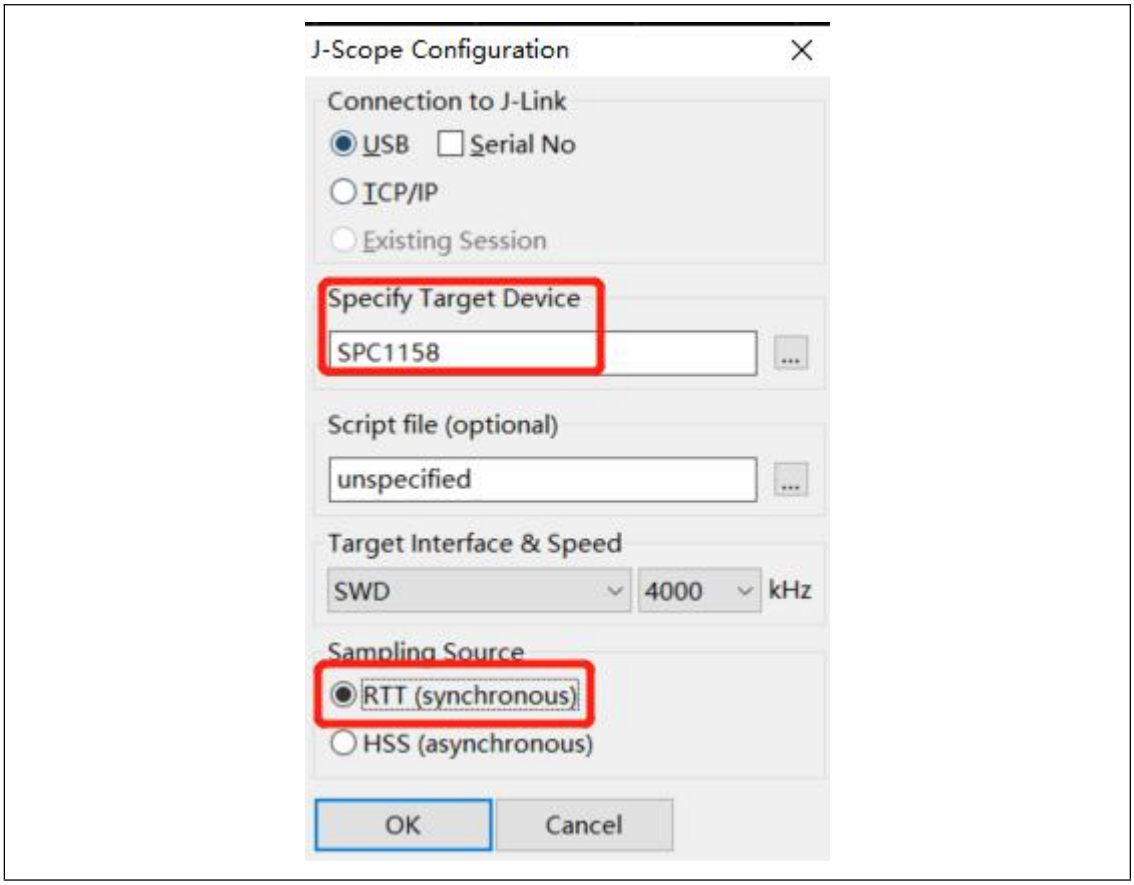
图 6- 13. RTT 发送结构体数据

```
acValBuffer.Sine1 = Sine1;
acValBuffer.Sine2 = Sine2;
acValBuffer.Sine3 = Sine3;
acValBuffer.count = XCnt;
SEGGER_RTT_Write(1, &acValBuffer, sizeof(acValBuffer));
```

6.5.3 Jscope 配置 RTT 模式

如图配置相应的 RTT 模式。选择相应的 MCU， 以及 SWD 模式。

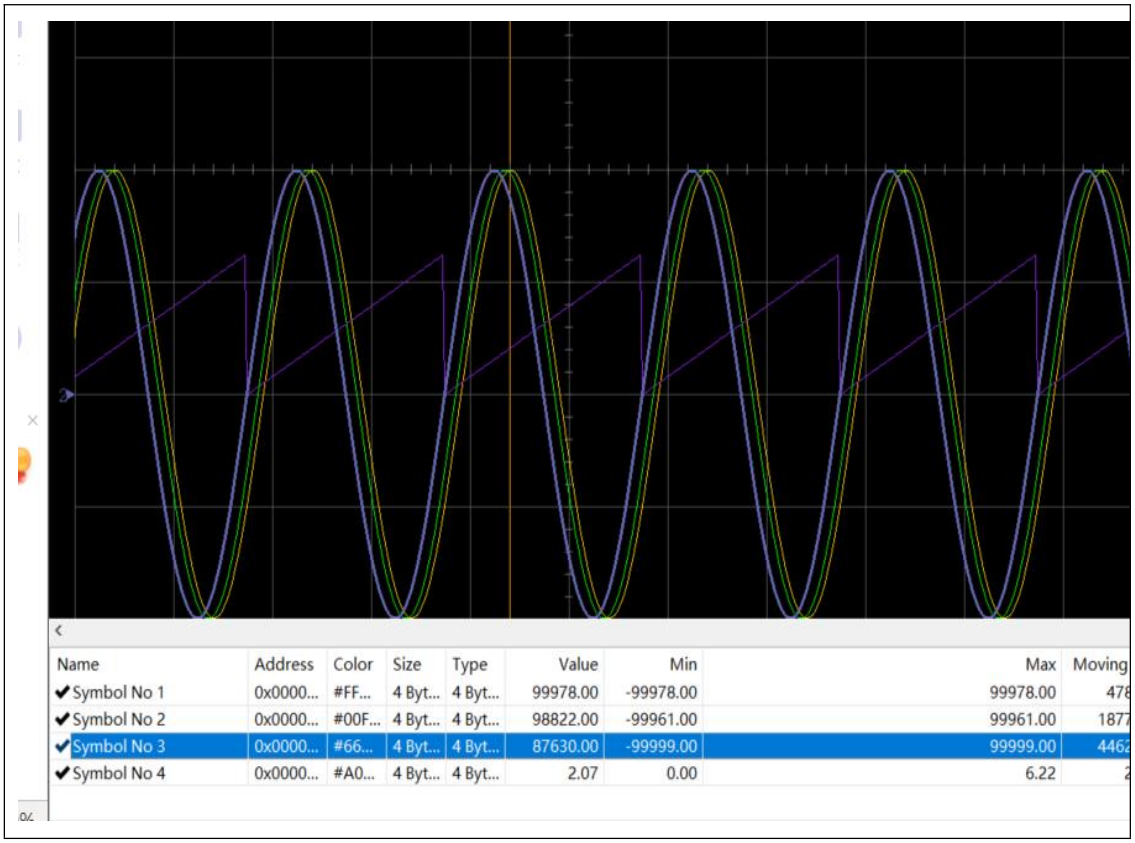
图 6- 14. JScope 配置



6.5.4 Jscope 显示 采集到的数据

最终效果如图 6- 15:

图 6- 15. JScope 采集数据



Name	Address	Color	Size	Type	Value	Min	Max	Moving
✔Symbol No 1	0x0000...	#FF...	4 Byt...	4 Byt...	99978.00	-99978.00	99978.00	478
✔Symbol No 2	0x0000...	#00F...	4 Byt...	4 Byt...	98822.00	-99961.00	99961.00	1877
✔Symbol No 3	0x0000...	#66...	4 Byt...	4 Byt...	87630.00	-99999.00	99999.00	4462
✔Symbol No 4	0x0000...	#A0...	4 Byt...	4 Byt...	2.07	0.00	6.22	2