Keil MDK ARM 项目迁移到 IAR Embedded Workbench for ARM

一．项目迁移概述

IAR Embedded Workbench 版本 7.60 以上的版本提供了一个名为 IAR Project Coverter 的工具，该工具可以将现有的 Keil MDK V5 版本的工程项目直接转换为 IAR Embedded Workbench for ARM 的工程项目源码文件及其目录结构维持不变。除此之外，项目迁移还需要对实际源代码做一些修改，工作量依据各个项目的情况而有所不同。另外，Convert To IAR 工具可以执行源代码替换，可以添加自己的替换规则，包括对正则表达式的支持。并且还允许导入一系列预定义的替换规则。

本文将详细介绍如何将 Keil MDK ARM 版本 5.23 的项目转换成 IAR Embedded Workbench 版本 8.30 的项目，使用的硬件平台是 ST 的 STM32F4-Discovery 官方开发板，迁移的项目是基于 FreeRTOS v9.0 的示例工程。

二．工程项目转换

2.1 使用 IAR Project Coverter 工具

从 IAR Embedded Workbenchv8.30 菜单栏中的“Tools”菜单选择“IAR Project Converter”打开工程转换工具，勾选 Enable Project file conversion，选择 Project type 为 Keil µVision5 for ARM，将 Root directory of source project 通过界面按钮选择 Keil µVision 工程文件所在目录。如有需要，可启用 Source code substitution 添加源码替换规则。

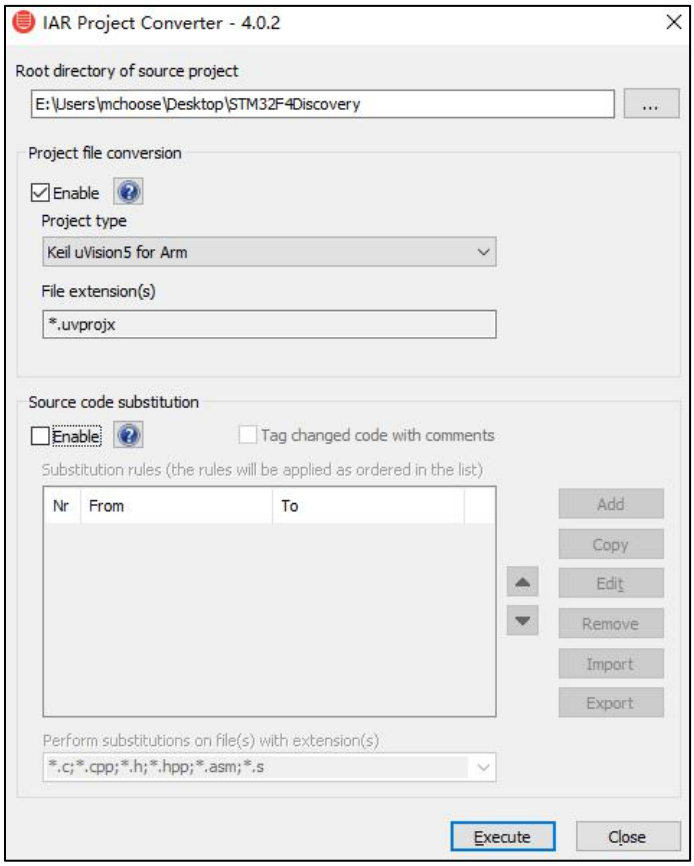

图 1. 选择源项目

然后单击“Execute”进入下一步，设置转换后 IAR Embedded Workbench for ARM 项目的存储路径，点击 OK 确认开始转换。

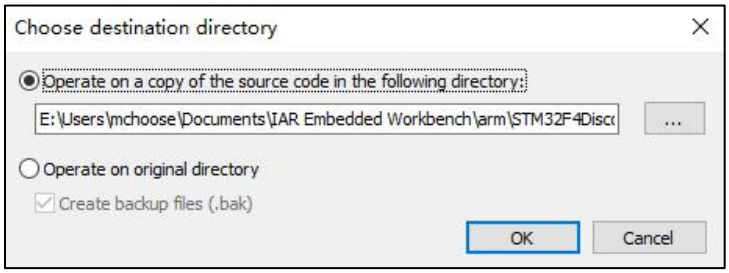

图 2. 设置转换后项目的目录

转换后项目的目录结构跟 Keil MDK 源项目结构一致，在目录下面会看到所生成的 IAR Embedded Workbench 的工作区文件(.eww)和工程文件(.ewp)。

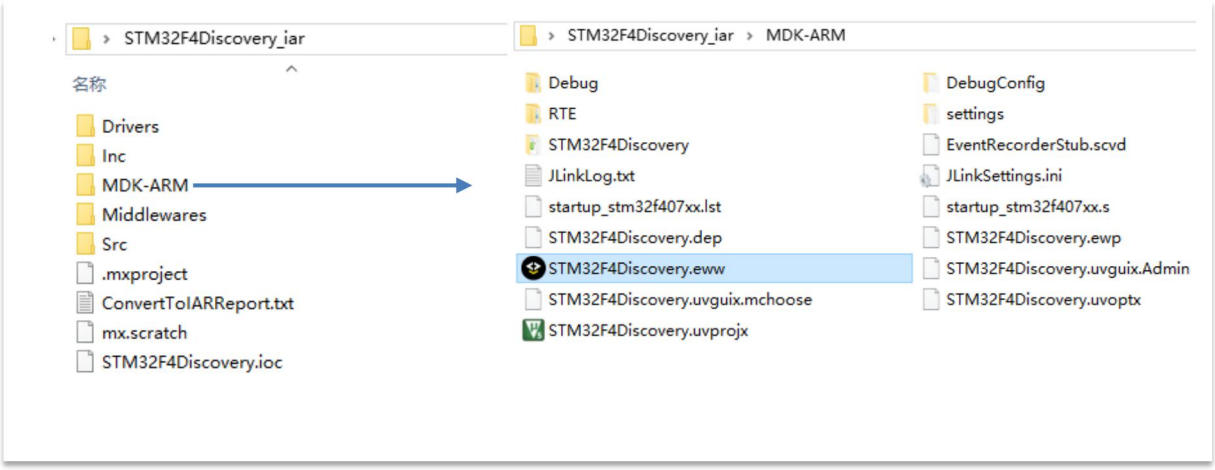

图 3. 转换后的项目文件夹

转换前后 IDE 工作区的虚拟目录命名保持不变，只是因为排序不同，先后顺序有所变化。

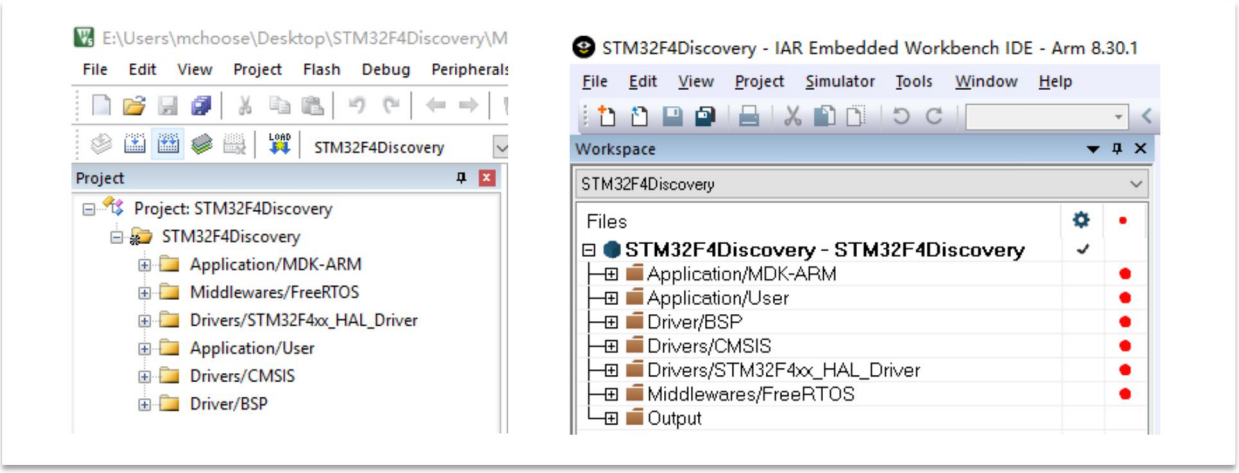

图 4. Keil 和 IAR 工作区的结构对比

2.2 IAR Embedded Workbench 项目选项设置

项目转换后，有些项目属性会丢失，比如芯片型号和调试器设置等。接下来就需要根据实际情况对基础的属性进行设置。

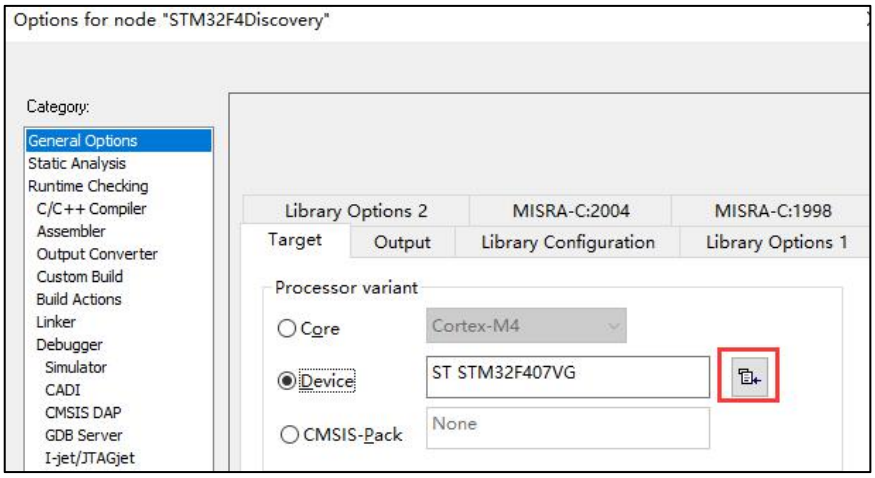

图 5. 选择芯片型号

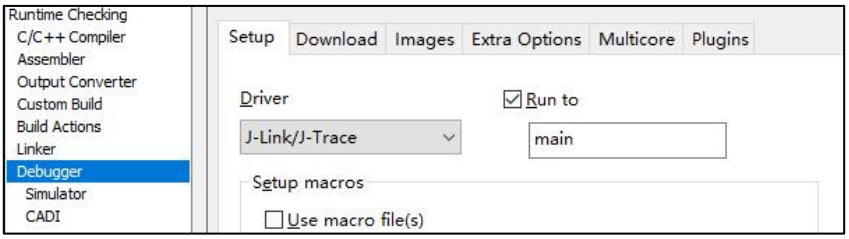

图 6. 选择调试器型号

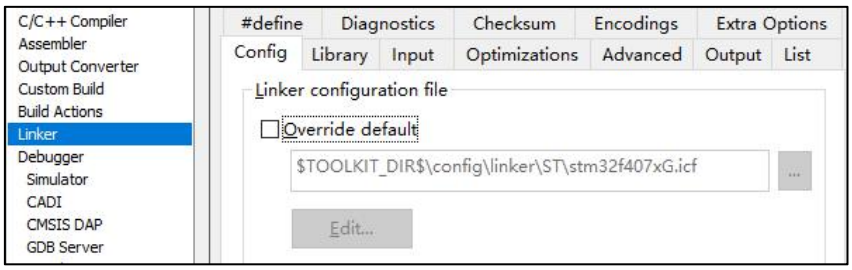

图 7. 设置链接器配置文件

此外，还需要检查开发语言设置项，例如选择 C 或者 C++语言，如果是 C 语言，是选择 C89 标准还是 C11 标准。

选择芯片型号后，会自动关联一个与该芯片匹配的默认链接器配置文件，如图 7 所示。如需自定义代码存储，则应在 Linker -> Config 勾选了“Override default”后，点击 Edit 按钮就会弹出链接器可视化配置页面。在页面可以修改向量表的地址，堆、栈空间的大小，以及存储器的地址区域。

IAR Embedded Workbench 8.30 版本增加了可配置多个外部存储空间的设置项，如图 8 所示。强烈建议先将默认的 Icf 文件复制并保存到当前工程目录下，并在这个副本的基础上进行修改(否则会将提供的 icf 默认值修改了)，而且建议将路径修改为相对路径（$PROJ_DIR$指代当前工程文件所在目录），例如：

$PROJ_DIR$\src\stmewf407xG.icf。

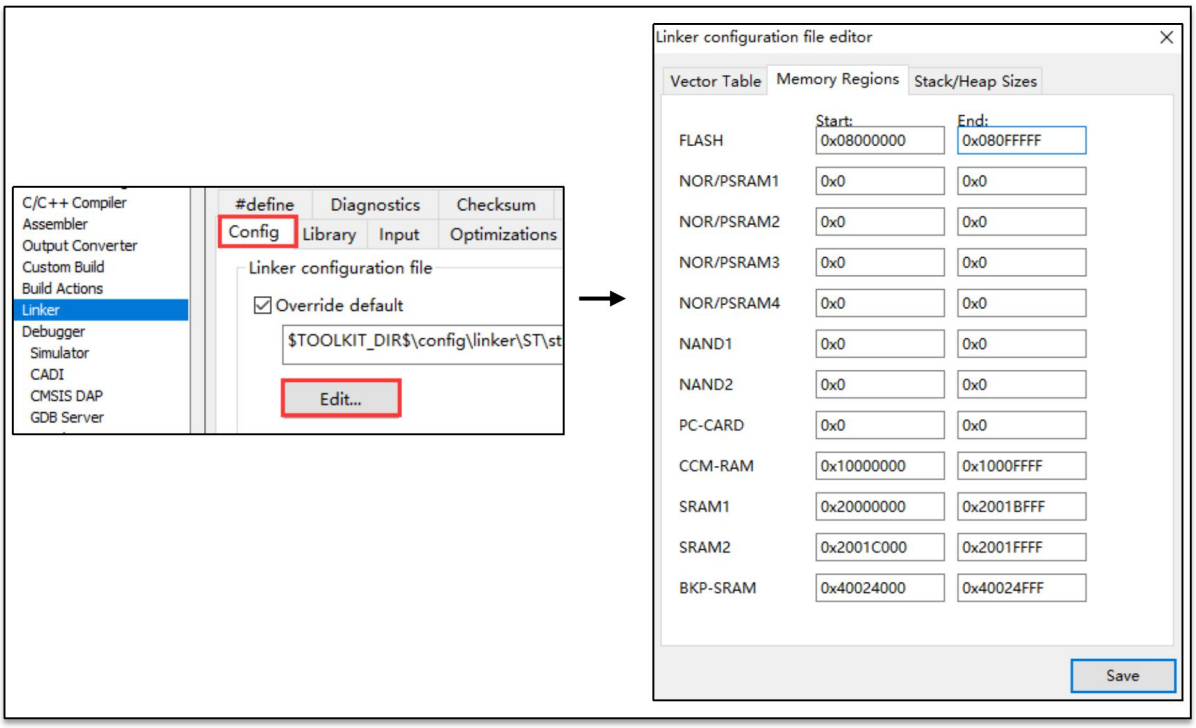

图 8. 链接器可视化配置

2.3 项目源代码修改

对基本的设置选项修改后，接下来要对源代码进行修改。最主要的工作是对汇编代码和编译关键字的修改。通常芯片的启动文件是由汇编写的，而且芯片厂商都会提供主流的编译环境的启动文件。一般不需要用户自己编写，用户只需替换对应的 IDE 的启动文件。以 ST 为例，其提供的 STM32F4 Cube 固件库在下图所示路径中可以看到 Keil、gcc、IAR 三个编译环境的启动文件。

图 9. ST 提供的启动文件

2.3.1 IAR ARM 汇编指令

IAR 的 ARM 汇编跟 Keil 存在差异，包括汇编指令，伪指令，内嵌汇编等。下面将会列举 IAR 部分 ARM 汇编常见指令，更多有关 IAR ARM 汇编的内容请参考[EWARM_AssemblerReference .ENU.pdf]产品手册。

MODULE：定义一个程序模块，也可以用 NAME 和 PROGRAM

SECTION：开始一个段(代码段或者数据段)

DATA：在代码部分中定义数据区域

EXTERN：导入外部符号

PUBLIC：将符号导出到其他模块

PUBWEAK：符号导出到其他模块，允许多定义

NOROOT：如果没有引用段当中的符号，该段也不会被 Linker 丢弃

LTORG：声明一个文字池

SFB：段起始

SFE：段结束 例如 SFE(CSTACK)，堆栈基地址

END：源文件结束

下面是 STM32F407VG 的 EWARM 启动文件中前面的一部分，一开始定义了一个名为“CSTACK”的数据段，用作系统的堆栈。NOROOT 后面的(3)指定了代码/数据的对齐字节数，为 2 的幂，所以堆栈是 8 字节对齐。

```
        MODULE    ?cstartup

        ;; Forward declaration of sections.
        SECTION CSTACK:DATA:NOROOT(3)

        SECTION .intvec:CODE:NOROOT(2)

        EXTERN    __iar_program_start
        EXTERN    SystemInit
        PUBLIC    __vector_table

        DATA
__vector_table
        DCD       sfe(CSTACK)
        DCD       Reset_Handler                 ; Reset Handler

        DCD       NMI_Handler                   ; NMI Handler
        DCD       HardFault_Handler             ; Hard Fault Handler
        DCD       MemManage_Handler             ; MPU Fault Handler
        DCD       BusFault_Handler              ; Bus Fault Handler
        DCD       UsageFault_Handler            ; Usage Fault Handler

                                  . . .
```

CortexM 向量表的第一个元素是堆栈的栈顶，且使用满减栈模型，所以 sfe(CSTACK)是获取 CSTACK 段的结束地址，该地址即为堆栈栈顶。

2.3.2 IAR 的静态库

另外，EWARM 和 MDK 的静态库格式也不一样，MDK 编译的静态库不能在 EWARM 中调用。除非 MDK 在编译的库符合 AEABI (Embedded Application Binary Interface for ARM)标准。例如，可以在 MDK 中使用“--library_interface=aeabi_clib”来生成 AEABI 兼容的库。芯片厂商一般都会同时提供 MDK 和 EWARM 的库。对于方案提供商等第三方，则用户在迁移工作开始之前应该确认是否有提供 IAR EWARM 的静态库。

2.3.3 内嵌汇编与 Intrinsic 函数

内嵌汇编可以在 C 或者 C++语言中插入汇编指令。通常是为了代码的效率，严格的时序要求或是访问 C 语言无法访问的硬件资源。“asm”和“__asm”关键字都可以用来插入汇编。但当使用了“--strict”时，“asm”不可用，而“__asm”关键字是一直可以用的。

IAR 内嵌汇编语法类似 gcc：

```
asm [volatile]( string [assembler-interface])
```

有多条汇编指令时，可以用“\n”分隔，如：

```
asm("label:nop\n"
    "b label");
```

内嵌汇编只是简单的插入到程序流中给定的位置，其有可能对周围的代码产生副作用，且不利于编译器对代码进行优化。如果是对某些硬件资源进行操作，例如获取堆栈指针，可以通过 Intrinsic 函数来实现。

使用 Intrinsic 函数要在文件中包含 intrinsics.h。以下是几个 Intrinsic 函数示例：

获取 MSP：__get_MSP
关闭中断：__disable_interrupt
设置 CortexM CONTROL 寄存器：__set_CONTROL
插入一条 WFI 指令：__WFI

更多内嵌汇编和 Intrinsic 函数的用法可以分别参考[EWARM_DevelopmentGuide.ENU.pdf]中的“Mixing C and assembler”和“Intrinsic functions”章节。

2.4 例程源码修改

本文的演示例程是基于 FreeRTOS 项目，所以除启动文件以外，与 RTOS 硬件移植相关的代码也需要替换成 IAR 环境所使用的文件，包含了一个 C 文件和一个汇编文件。在 FreeRTOS 源码目录下面的 portable->IAR->ARM_CM4 的目录里面提供了移植好适用于 IAR EWARM 的源码文件。

首先在 Workspace 当中移除工程当前所添加的两个源文件，然后再添加 ARM_CM4 目录下的 port.c 和 portasm.s。

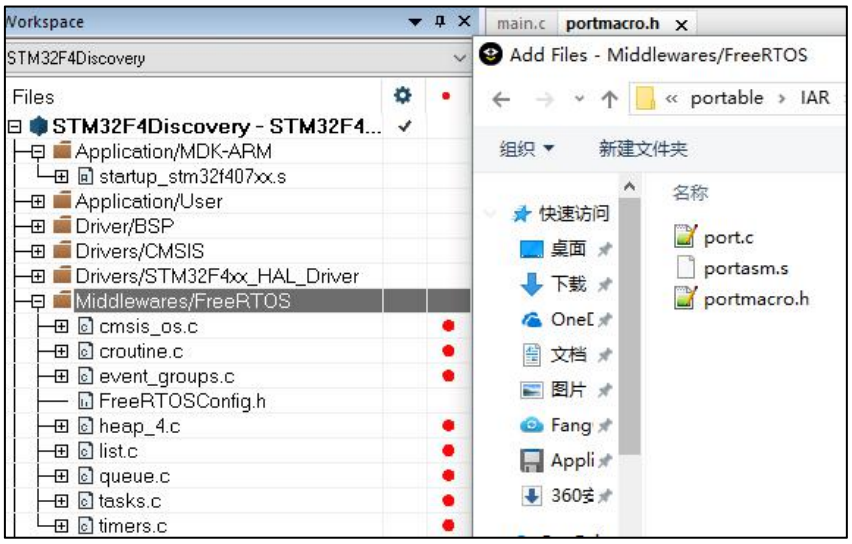

图 11. 替换 FreeRTOS 移植文件

此外，在工程的 Options -> C/C++ Compiler -> Preprocessor 设置选项添加 ARM_CM4 的路径，以便编译器可以找到 portmacro.h。

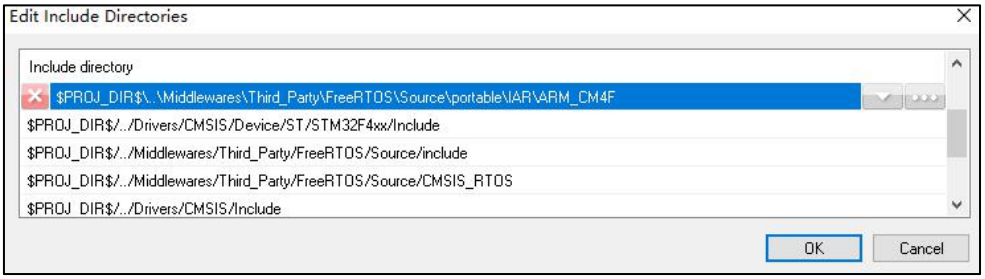

图 12. 设置头文件路径

完成以上工作之后就可以编译了。因为这个项目比较简单，编译没有遇到错误或者警告。如果是比较复杂的项目，可能会遇到错误，编译不能通过，还需要逐一的解决存在的问题。

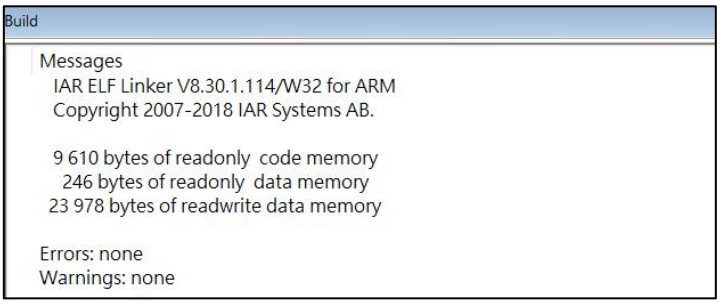

图 13. 编译结果

2.4.1 编译及问题处理

一般比较容易遇到的错误会有以下几种：

1.提示文件找不到或者无法打开

此问题一般是头文件的路径没有添加，或者源码文件的路径改变后没有在工程中作相应更改。工程目录存放的路径太深或者路径中包含中文或者特殊字符，也可能会导致编译器找不到文件。

2.编译时提示了大量的语法错误

编译出现大量的语法错误，甚至是厂商提供的固件库源码文件也有报错。这种问题多是由文件的编码格式导致的，可以尝试修改源码文件的编译格式，例如修改为 UTF-8 编码。

3.提示符号未定义

如果经过确认符号符号确实已经定义了，可以检查编译器关键字，不同 IDE 的关键字不尽相同。

如果遇到不能解决的错误或者警告，有以下途径可以尝试：

1. 可以从 EWARM 的 Help 菜单下打开帮助文档，搜索遇到的问题。直接从 Help 菜单点击“Search”，输入关键字进行搜索。如果想了解编译器相关功能的用法，打开[C/C++ Development Guide]开发手册参考。

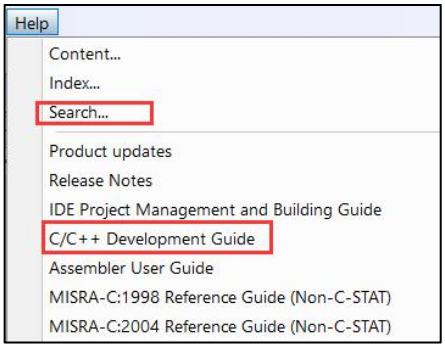

图 14. 获取文档帮助

2.还可以从 IAR Systems 官方网站的 Support 页面搜索问题的关键字，查找问题的解决方法。IAR 官网地址：https://www.iar.com。

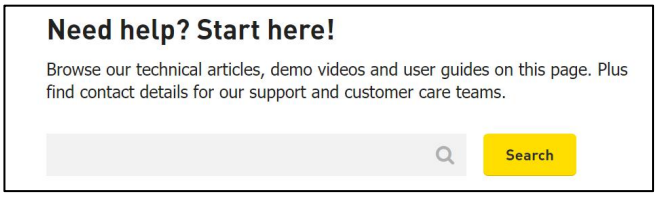

图 15. IAR 官网 Support 检索

3.如果以上两个途径无法自助解决问题，且所购买 EWARM 的 SUA 还在服务期，可以访问 https://www.iar.com/support/rfta/填写信息获取支持，此外还可以与购买的代理商联系，寻求技术支持。

三、IAR EWARM 链接器配置举例

在项目应用上，很多时候都有指定代码或者数据的存放位置的需求。比较常见的，如使用了外扩 Flash 来存放代码，在链接的时候就需要根据外扩 Flash 的映射的地址来对代码进行链接。IAR EWARM 是通过链接器配置文件(*.icf)的来定义代码和数据的存放地址，配合在源码中使用预处理指令可以实现用户的各种定位需求。

下面以 STM32F407VG 的默认链接器配置文件为例，来说明如何定位代码和数据的地址：

```
define symbol __ICFEDIT_intvec_start__   = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__        = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__          = 0x080FFFFF;
define symbol __ICFEDIT_region_RAM_start__        = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__          = 0x2001FFFF;
define symbol __ICFEDIT_region_CCMRAM_start__     = 0x10000000;
define symbol __ICFEDIT_region_CCMRAM_end__       = 0x1000FFFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__   = 0x400; ①
define symbol __ICFEDIT_size_heap__     = 0x200; ②
```

上面的 define symbol 的作用是定义一系列符号，替代地址和空间大小的数值，方便阅读和修改。①定义系统栈的大小；②的定义堆的大小。要改堆和栈的大小，直接修改后面的数值就可以了。

③ define memory mem with size = 4G;
④ define region ROM_region = mem:[from __ICFEDIT_region_ROM_start__
to__ICFEDIT_region_ROM_end__];
⑤ define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__
to __ICFEDIT_region_RAM_end__];
define region CCMRAM_region = mem:[from __ICFEDIT_region_CCMRAM_start__
to __ICFEDIT_region_CCMRAM_end__];
⑥ define block CSTACK with alignment = 8, size = __ICFEDIT_size_cstack__ { };
⑦ define block HEAP with alignment = 8, size = __ICFEDIT_size_heap__ { };

③使用 define memory 定义一个 32 位可寻址的存储空间；
④使用 define region 定义一个 ROM 区，大小由前面定义的符号决定；
⑤使用 define region 定义一个 RAM 区，大小由前面定义的符号决定；；
⑥使用 define block 定义一个 8 字节对齐的空 block 作为系统的堆栈，大小由前面定义的符号指定；
⑦使用 define block 定义系统的堆空间，大小由符号指定(如果没有使用 malloc 标准库函数实际不会分配堆空间)。

initialize by copy { readwrite };
do not initialize { section .noinit };
place at address mem: __ICFEDIT_intvec_start__ { readonly section .intvec };
⑧ place in ROM_region { readonly };
⑨ place in RAM_region { readwrite, block CSTACK, block HEAP };

⑧通过 place in 指令将只读的段定位到定义的 ROM 存储区；
⑨通过 place in 指令将可读/写的段、栈、堆定位到 RAM 存储区。

以上是 STM32F407VG 芯片内部 ROM 和 RAM 存储空间的定义。对于使用了外扩 Flash 或 RAM，也是使用同样的方法，定义一个或者多个 ROM 或 RAM 存储区，再使用 place in 指令将段定位到定义好的存储区。数据，如数组、变量还可以使用 place at 指令定位到某个具体的地址。下面是定位函数和变量到某个地址的实例。

3.1 定位函数到某个地址

因为函数占用空间大小事先无法确定，所以不能直接使用 place at 指令定位到具体地址。正确的做法是在源文件函数定义时，通过预处理指令指定函数编译到自定义的段，然后在链接器配置文件 icf 中定义段的存放位置，示例：

将函数定位到 0x08008000 为起始地址的 Flash 空间，将函数的代码放置在“.sec1”的段中：

```
unsigned int myfunction( ) @ “.sec1”
{
  ...
}
```

或者

```
#pragma location = “.sec1”
unsigned int myfunction( )
```

```
{
  ...
}
```

然后在 icf 文件中定义“.sec1”的位置：

```
place at address mem: 0x08008000 { section .sec1};
```

3.2 定位变量到某个地址

变量占用空间的大小是确定的，所以与函数不一样，定位变量可以不用修改 icf 文件，如下：

```
const int MyConst @ 0x08009000 = 0xFFFF0000;
```

当然，也可以使用定位函数的方法，先将变量放置到自定义的段，然后在 icf 文件中定义段的位置。

3.3 定位目标文件/静态库中的段

如果要将某个源文件编译后的代码/数据段定位到指定的空间，也可以使用 place at 或 place in 指令来进行操作。例如要将 MyFile.c 编译后的 MyFlie.o 当中的数据段.data 都放置到 RAM 区域：

```
place … { section .data object MyFile.o };        // place in RAM
```

定位 MyFile.a 文件中的代码段.text 到 Flash：

```
place … { section .text object MyLib.a };        // place in Flash
```

对于需要更复杂的定位要求，可以借助 block 来对段进行管理，block 可以按照固定的顺序排列放置的段。更多的有关于链接器配置的方法，可以参考[EWARM_DevelopmentGuide.ENU.pdf]的“Linking using ILINK”和“The linker configuration file”两个章节的内容。