AN12673

K32L3A6 器件的双核项目创建和转换

版本 0 — 2020 年 1 月

应用笔记

目录

1 介绍

K32 L3 器件的引入为 Kinetis 系列带来了双核设备的强大功能和多任务处理能力。双核设备虽然功能强大，但却会增加开发的复杂性。本应用笔记的目的是简化启动新的多核项目或将现有的单核项目转换为双核项目的过程。

注意

不同的 IDE 处理双核应用程序的方式有所不同。本应用笔记将介绍 IAR IDE 和 MCUXpresso IDE。

2 概述

在创建双核项目（或将单核项目转换为双核项目）之前，有必要讨论 K32L3A6 器件的结构。K32L3A6 是一个非对称双核设备（具有两个不同的内核，而不是对称的双核设备，对称的双核设备具有以锁步方式运行的相同内核），其中包含 Arm®Cortex®-M4 内核和 Arm Cortex-M0 +内核。像其他双核设备一样，一个核充当主要核或启动核，而另一个核充当从核。在 K32L3A6 中，默认情况下，Arm Cortex-M4 设备充当启动核（可以更改启动核）。 K32L3A6 器件的框图如 图 1 所示。

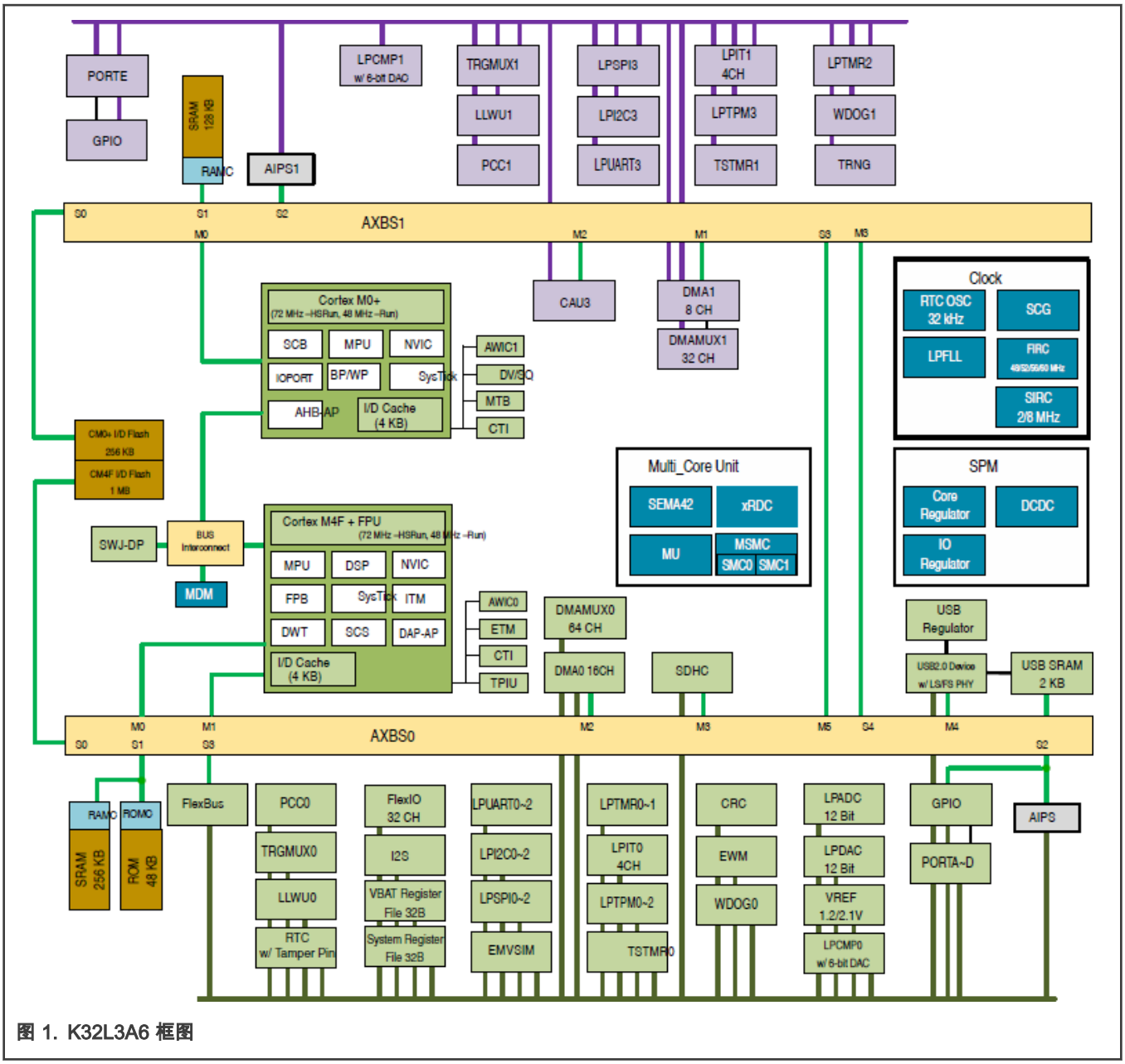

图 1. K32L3A6 框图

如 图 1 所示:

- 每个内核都有自己的缓存，RAM 存储器，闪存，中断控制器等。这两个内核可以完全独立地运行。
- 除了两个单独的核之外，还有两个单独的交叉开关 AXBS0 和 AXBS1。这有效地形成了一个逻辑上的划分，每个内核也都有自己的一组外围设备。 那些连接到 AXBS0 的外围设备是 Arm Cortex-M4 外围设备，而那些连接到 AXBS1 的外围设备是 Arm Cortex-M0 +外围设备。但是，交叉开关也通过 AXBS1 从端口 3（S3）连接到 AXBS0 主端口 5（M5）和 AXBS1 主端口 3（M3）连接到 AXBS0 从端口 4（S4）来相互连接。这使 Arm Cortex-M4 可以访问 Arm Cortex-M0 +外围设备和内存，反之亦然。
- 两个内核共有的公用部件包括调试单元，多内核单元（SEMA42，MU，资源域控制器和其他系统控制模块），时钟控制（SCG 和 RTC）和电源控制（SPM）。

现在，您已经对 K32L3A6 设备的架构有了更好的了解，让我们讨论一下 IDE 如何处理这种架构。首先，将讨论 IAR IDE，然后将讨论 MCUXpresso。

3 IAR 中的多核项目

IAR IDE 本质上将多核项目视为两个独立的项目。因此，在创建双核项目时需要两个独立的项目（每个核一个）。如果您是从单核项目开始的，则需要为第二个核创建（或合并）一个项目。您将需要修改代码以确保正确启动第二个核。您可能还需要考虑两个内核之间的交互作用（超出本应用笔记中的描述），并进行比本应用笔记中所描述的更多的修改。这里的可能性和选项是无限的，因此，本应用笔记无法讨论所有情况。

在 IAR 项目中，需要选择内部选项以将两个单独的项目链接在一起。在这一点上，重要的是要注意，多核项目的创建并不一定意味着调试器将与两个核建立调试连接。这是另一个独立的设置。因此，本节将分为多核项目创建和多核调试。

3.1 多核项目创建

如 IAR 中的多核项目中所述，IAR IDE 将多核项目的创建视为两个独立的项目，它们只是通过项目中的不同设置链接在一起。因此，创建多核项目的第一步是为不同的核创建两个独立的项目。K32L3A6 MCUXpresso SDK 软件包中 multicore_examples 文件夹中的 hello_world 示例将用作解释这些概念的示例。

3.1.1 主核项目

1. 检查启动核项目。这种情况下，这是 Arm Cortex-M4 项目，该项目仅配置一个终端并向该终端打印 hello_world 并（通过终端）通知用户从核正在启动。然后启动从核。在双核设备中，通常是引导核负责启动从核。图 2 显示了此源代码（有关源代码的更多详细信息，请参见本文档的以下章节）。

hello_world_core0.c

```
/*!
 * @brief Main function
 */
int main(void)
{
    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();

    /* Init board hardware.*/
    BOARD_InitPins_Core0();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();

    /* Print the initial banner from Primary core */
    PRINTF("\r\nHello World from the Primary Core!\r\n\n");

#ifdef CORE1_IMAGE_COPY_TO_RAM
    /* Calculate size of the image  - not required on MCUXpresso IDE. MCUXpresso copies the secondary core
       image to the target memory during startup automatically */
    uint32_t core1_image_size;
    core1_image_size = get_core1_image_size();
    PRINTF("Copy Secondary core image to address: 0x%x, size: %d\n", CORE1_BOOT_ADDRESS, core1_image_size);

    /* Copy Secondary core application from FLASH to the target memory. */
    memcpy(CORE1_BOOT_ADDRESS, (void *)CORE1_IMAGE_START, core1_image_size);
#endif

    /* Boot Secondary core application */
    PRINTF("Starting Secondary core.\r\n");
    MCMGR_StartCore(kMCMGR_Core1, CORE1_BOOT_ADDRESS, 5, kMCMGR_Start_Synchronous);
    PRINTF("The secondary core application has been started.\r\n");

    while (1)
    {
    }
}
```

图 2. 引导核源代码

引导核项目的另一个职责是对从核程序代码空间的内存进行编程。通过正确配置项目中的链接器设置以及正确配置链接器文件，可以实现此功能。必须将项目中的链接器设置配置为指向从核的正确二进制文件，并将其放置在内存中的正确位置。 图 3 显示了主核项目的 hello_world 链接器设置。

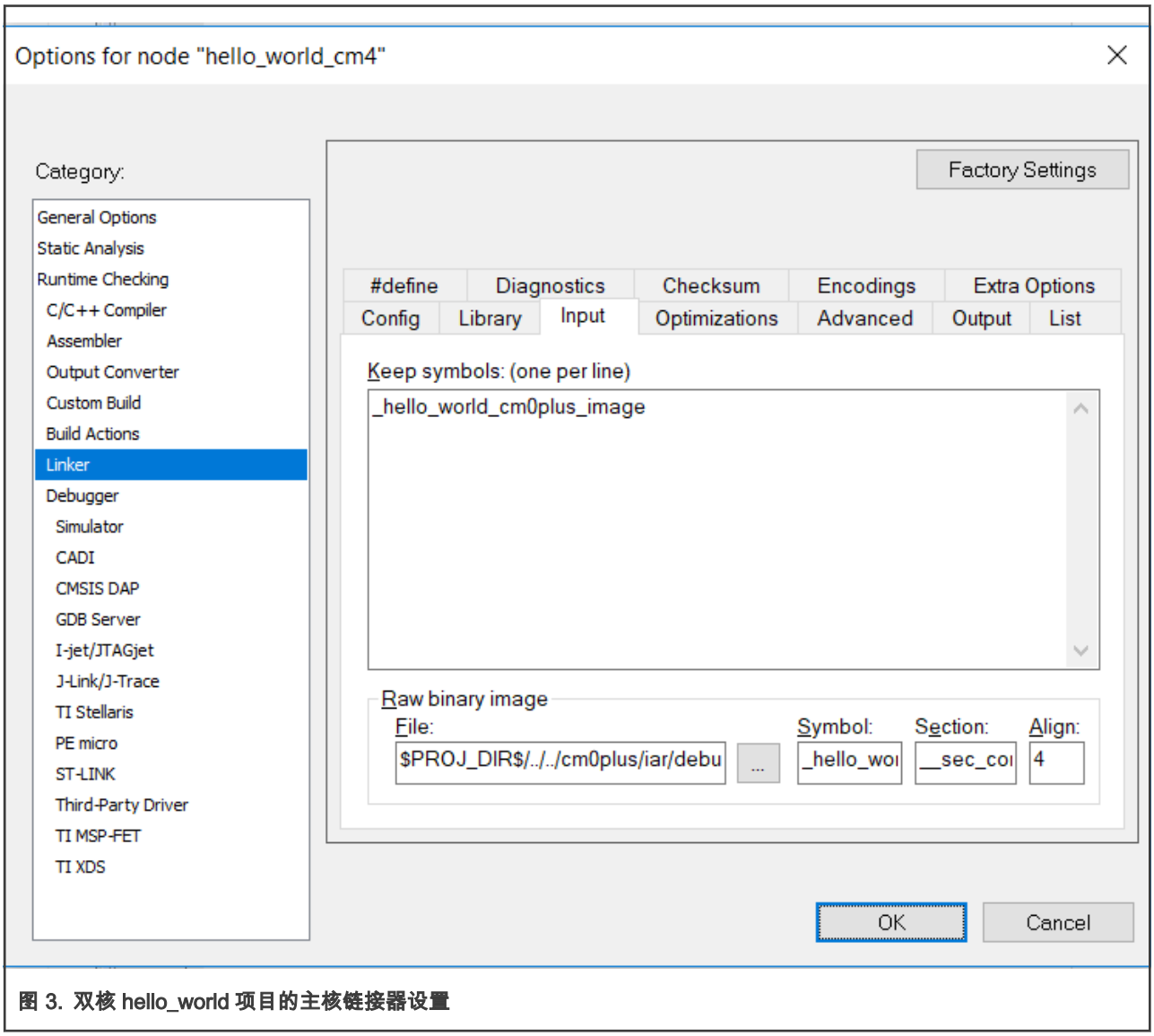

图 3. 双核 hello_world 项目的主核链接器设置

创建双核项目的所有必要设置都包含在项目设置的“链接器”类别的“输入”选项卡中。必须正确写出以下五点。

- 保留符号（Keep symbols）：在“保留符号”文本框中，列出了需要添加到输出文件中的其他二进制文件（单击“编译”按钮时生成）。在这种情况下，我们只有一个额外的映像要添加到最终输出中。 它被命名为 _hello_world_cm0plus_image。 该符号名称是任意的，可以是任何名字。
- 文件（File）：在文件文本框中，应包含 Arm Cortex-M0 +程序二进制映像的正确路径（* .bin）。
- 符号（Symbol）：“符号”字段必须与“保留符号”文本框中定义的符号匹配。
- 节（Section）：这定义了要包含的二进制文件的放置位置。必须在链接器文件中正确定义该二进制文件，以便对其进行编程和正确放置。
- 对齐（Align）：这定义二进制文件的对齐方式。 这确定数据是按字节，半字还是字对齐。这个示例是字对齐二进制文件，除了如图所示的字对齐，您什么都不需要。

2. 确保链接器文件已正确写入。链接器文件必须正确定义要放置从核二进制文件的部分。图 4 显示了 hello_world 项目中使用的链接器文件，重要部分用红色下划线标出。

K32L3A60xxx_cm4_flash.icf

```
define region TEXT_region = mem:[from m_interrupts_start to m_interrupts_end]
                          | mem:[from m_text_start to m_text_end];
define region DATA_region = mem:[from m_data_start to m_data_end-__size_cstack__];
define region CSTACK_region = mem:[from m_data_end-__size_cstack__+1 to m_data_end];
if (isdefinedsymbol(__use_shmem__)) {
  define region rpmsg_sh_mem_region     = mem:[from rpmsg_sh_mem_start to rpmsg_sh_mem_end];
}

define block CSTACK    with alignment = 8, size = __size_cstack__   { };
define block HEAP      with alignment = 8, size = __size_heap__     { };
define block RW        { readwrite };
define block ZI        { zi };

define region core1_region = mem:[from core1_image_start to core1_image_end];
define block SEC_CORE_IMAGE_BLOCK          { section   __sec_core };

/* regions for USB */
define region USB_BDT_region = mem:[from m_usb_sram_start to m_usb_sram_start + usb_bdt_size - 1];
define region USB_SRAM_region = mem:[from m_usb_sram_start + usb_bdt_size to m_usb_sram_end];
place in USB_BDT_region                    { section m_usb_bdt };
place in USB_SRAM_region                   { section m_usb_global };

initialize by copy { readwrite, section .textrw };
do not initialize  { section .noinit, section m_usb_bdt, section m_usb_global };
if (isdefinedsymbol(__use_shmem__)) {
  do not initialize  { section rpmsg_sh_mem_section };
}

place at address mem: m_interrupts_start    { readonly section .intvec };
place in TEXT_region                        { readonly };
place in DATA_region                        { block RW };
place in DATA_region                        { block ZI };
place in DATA_region                        { last block HEAP };
place in CSTACK_region                      { block CSTACK };
if (isdefinedsymbol(__use_shmem__)) {
  place in rpmsg_sh_mem_region                { section rpmsg_sh_mem_section };
}
place in core1_region                       { block SEC_CORE_IMAGE_BLOCK };
```

图 4. 链接器文件的底部

在此链接器文件的底部，core1_region {block SEC_CORE_IMAGE_BLOCK};中有 place 命令。它命令链接器将分配给 SEC_CORE_IMAGE_BLOCK 的所有内容放入 core1_region 定义的内存中。块 SEC_CORE_IMAGE_BLOCK 被定义为 sec_core 节，这意味着应用程序中带有该标记的所有内容都将放置在该块中。在这种情况下，唯一带有该标签的将是从核的映像（这是在项目设置的“链接器”->“输入”选项卡中完成的）。区域 core1_region 定义为从 core1_image_start 到 core1_image_end 的存储区域。这些界限在链接器文件的顶部定义（如 图 5 所示）。

K32L3A60xxx_cm4_flash.icf

```
18  **     http:                www.nxp.com
19  **     mail:                support@nxp.com
20  **
21  ** ###################################################################
22  */
23
24  define symbol m_interrupts_start       = 0x00000000;
25  define symbol m_interrupts_end         = 0x000003FF;
26
27  define symbol m_text_start             = 0x00000400;
28  define symbol m_text_end               = 0x000FFFFF;
29
30  define exported symbol core1_image_start     = 0x01000000;
31  define exported symbol core1_image_end       = 0x0103FFFF;
32
33  if (isdefinedsymbol(__use_shmem__)) {
34    define symbol m_data_start                   = 0x20000000;
35    define symbol m_data_end                     = 0x2002E7FF;
36    define exported symbol rpmsg_sh_mem_start    = 0x2002E800;
37    define exported symbol rpmsg_sh_mem_end      = 0x2002FFFF;
38  } else {
39    define symbol m_data_start           = 0x20000000;
40    define symbol m_data_end             = 0x2002FFFF;
41  }
42
```

图 5. 链接器文件的顶部

在链接器文件的顶部，变量 core1_image_start 到 core1_image_end 分别定义为 0x01000000 和 0x0103FFFF。因此，映像将放置在 Arm Cortex-M0 +闪存空间中。

注意

图像范围可以更改为 Arm Cortex-M0+ RAM 空间。然而，这将意味着 Arm Cortex-M4 也必须从 RAM 运行，因为图像加载工具无法在加载 Flash 或加载 RAM 之间切换。图像加载器实用程序要么加载 RAM，要么加载 Flash。

3.1.2 从核项目

从核项目应作为普通项目进行编译。为预期的配置使用正确的闪存加载器很重要。如果从核预期从闪存（FLASH）运行，则应使用闪存链接器将其链接。如果期望从 RAM 运行，则应使用 RAM 配置的链接器。

注意

将“保持符号（Keep symbols）”和“原始二进制映像（Raw binary images）”部分保留为空白（如 图 6 所示）。

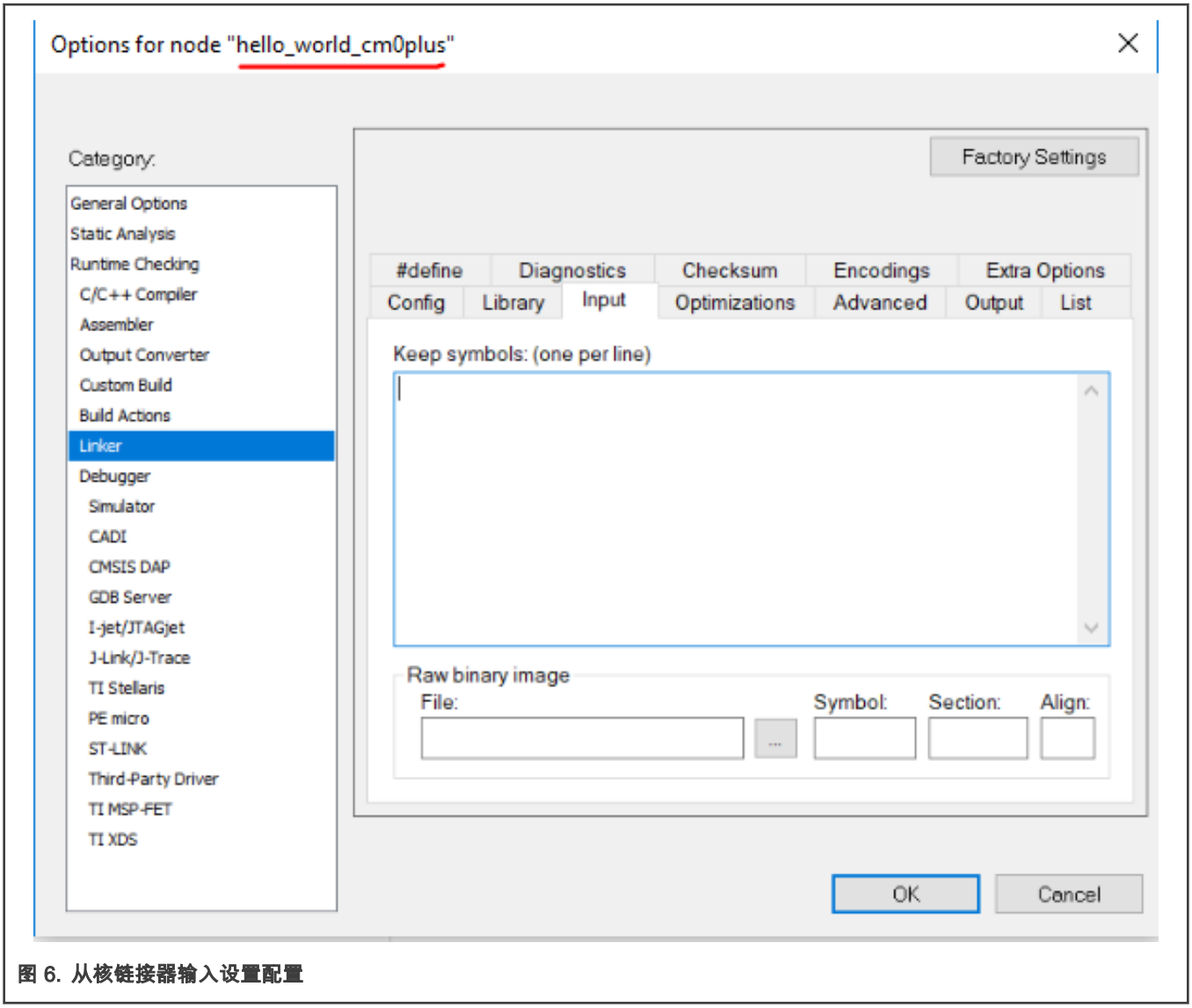

图 6. 从核链接器输入设置配置

4 在 IAR 中进行多核调试

多核调试通常是指同时调试两个内核的行为。

注意

可以仅调试主内核。 在这种情况下（假设项目是一个双核项目），从核仍将被编程并仍将运行，但是您将只能控制主核。这有时比真正的多核调试更容易和更简单。这样，您可能会发现这更方便（取决于您的目标）。本节将重点介绍多核调试。

与创建多核项目一样，同时调试两个核要求两个项目都配置有某些设置。错误的设置可能（通常）导致连接尝试失败。

将讨论以下信息：

1. 在 IAR 中编译多核项目
2. 主核项目调试设置
3. 从核项目调试设置
4. 调试

本示例检验了 FRDM-K32L3A6 SDK 包中的多核 hello_world 项目。在此示例中使用了 ijet 调试器，但该内容对于其他调试器仍然有效。

注意

在撰写本应用笔记时，IAR 不支持使用 JLink 调试器进行双核调试。确保检查您的 IAR 版本是否支持使用您喜欢的调试器进行双核调试。

4.1 在 IAR 中编译多核项目

主项目和从项目在 IAR 中分别进行编译，因此，应遵循正常程序来编译这些项目。但是，在这种情况下，顺序确实很重要。由于期望主项目也对从设备的内存空间进行编程，因此应该首先编译从项目，以便主项目能够链接该项目的二进制文件。

4.2 主核项目调试设置

1. 确保在“调试器（Debugger）”类别中选择了 ijet（或您选择的支持的调试器）。

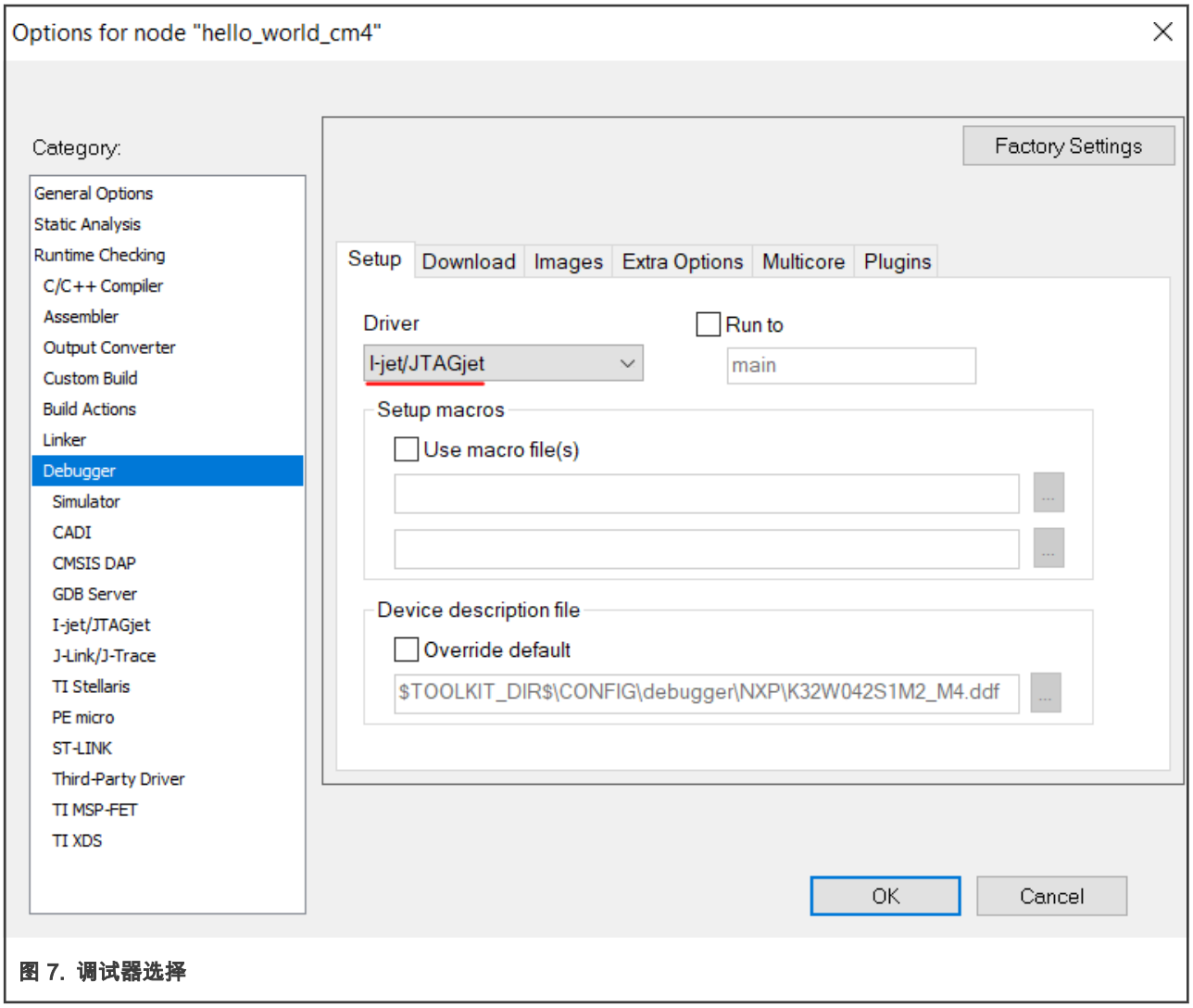

图 7. 调试器选择

让我们继续“设置（Setup）”标签。

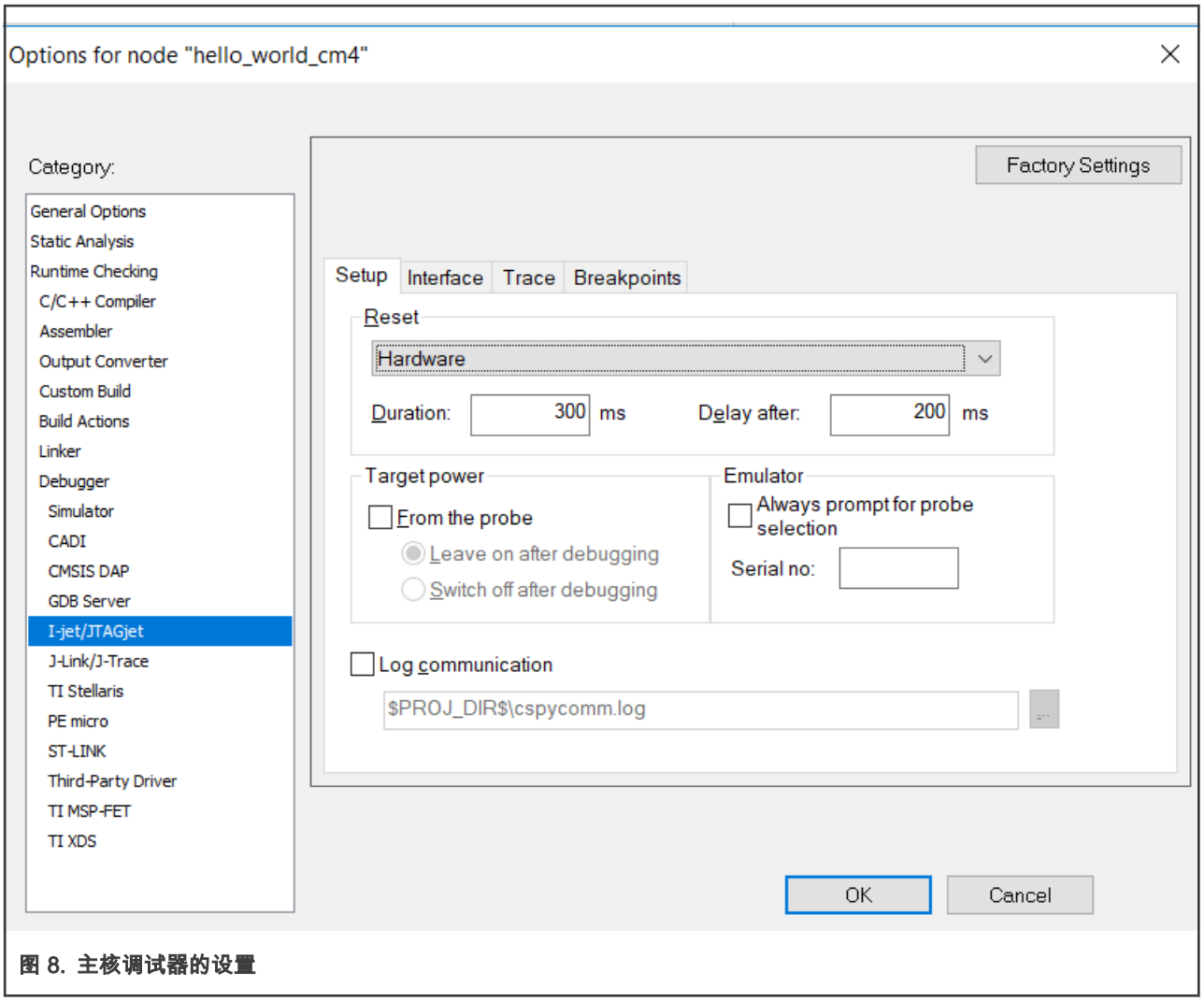

图 8. 主核调试器的设置

本例中的复位方法是使用硬件复位或复位引脚，但是由于它是主内核，因此可以是任何复位选项。

2. 让我们看一下“调试器（Debugger）”配置中的“下载（Download）”选项。hello_world 示例已配置为下载到闪存。图 9 显示了主核配置（如果下载到闪存，则必须使用闪存加载器）。

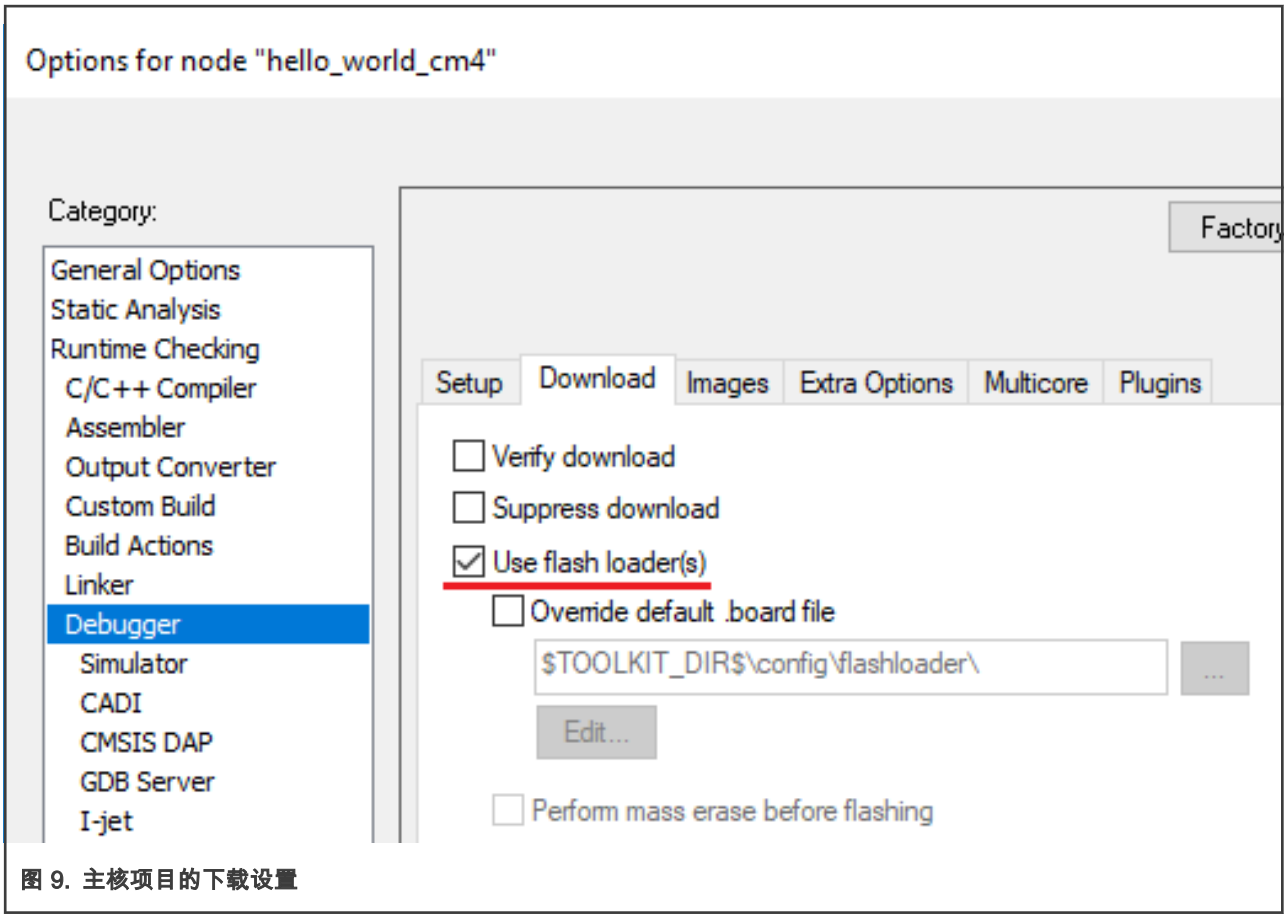

图 9. 主核项目的下载设置

注意

如果需要 RAM 项目，则应取消选中此复选框。

3. 检查“多核（Multicore）”选项卡。 可以说这是最重要的选项卡。 与链接器选项一样，此选项卡需要了解从属（或从核）工作空间。图 10 显示了“多核（Multicore）”选项卡。

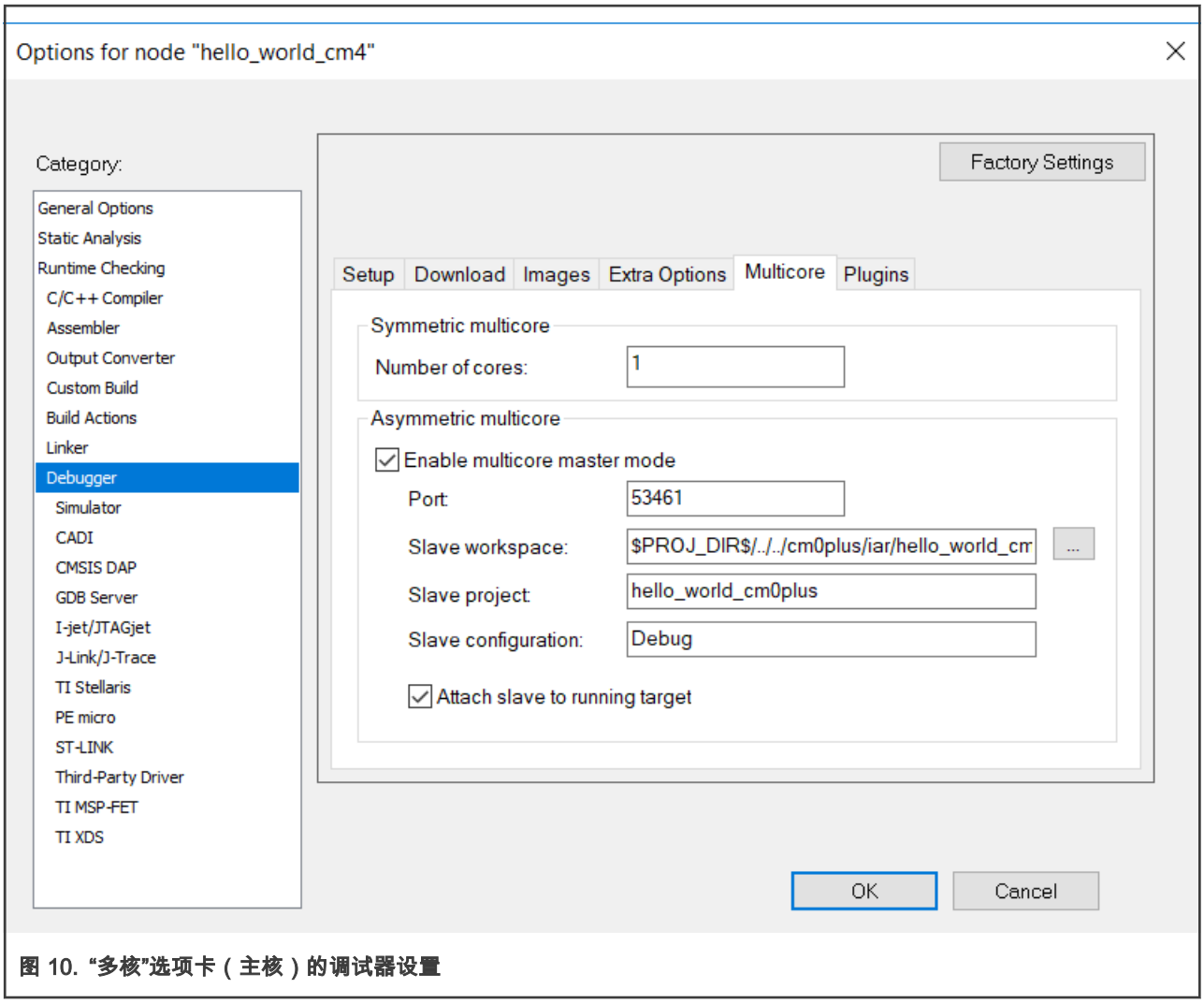

图 10. “多核”选项卡（主核）的调试器设置

由于 K32L3A6 是非对称设备，因此仅需要非对称多核设置。

注意

调试器选项卡中从属工作空间的路径与项目配置中的路径相同。

```
$PROJ_DIR$/../../cm0plus/iar/hello_world_cm0plus.eww
```

4. 确保正确设置了调试器接口。

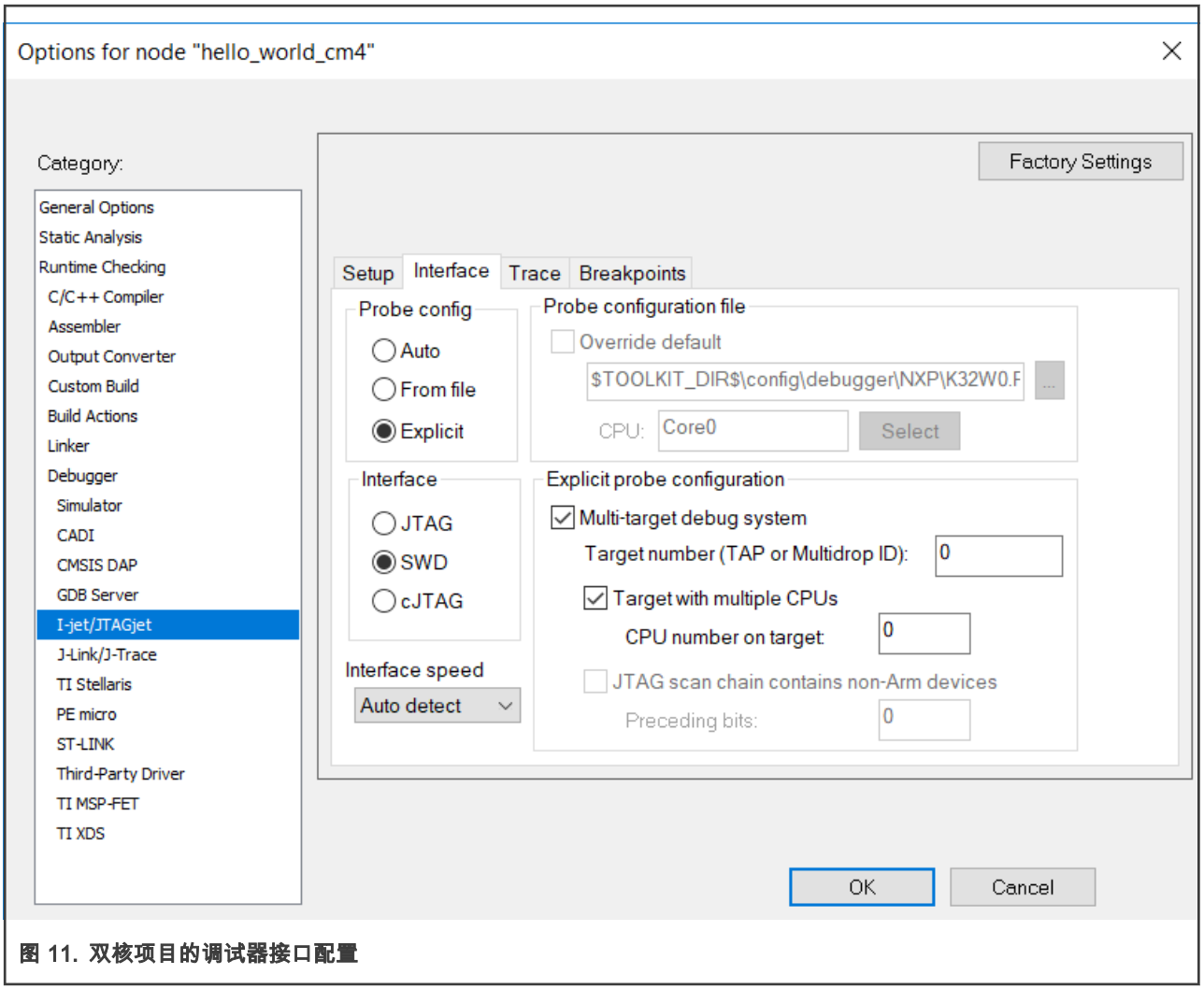

图 11. 双核项目的调试器接口配置

“界面（Interface）”选项卡中要重点关注的重要部分是“调试器配置（Probe config）”部分。可以从用户定义的文件自动确定调试器配置，也可以通过在此对话框中进行明确选择来确定调试器配置。本示例将重点介绍显式方法。重要的是，显式的调试器配置（如果选择）以正确的 CPU 编号为目标。由于此项目针对 Arm Cortex-M4 内核，因此 CPU 编号应为 0，因为它是调试链中的第一个 CPU。

如果使用了文件，则该文件应具有此信息，并且必须在“CPU”字段中选择正确的内核。但是，使用上面显示的设置，不会遇到任何问题。

注意

至关重要的一点是，所使用的接口（显示为 SWD）必须与从属项目相匹配。

4.3 从核项目调试设置

从核项目的调试设置主要由主设置决定。但是，两个项目之间没有自动链接。因此，必须在启动调试会话之前手动配置并更正这些设置。不这样做通常会导致连接尝试失败，但也可能导致 IAR IDE 的其他不稳定行为（有时导致程序崩溃）。

检查以下设置：

- 检查使用的驱动程序是否与主项目匹配。在“调试器（Debugger）”类别的“设置（Setup）”选项卡中可以找到它。

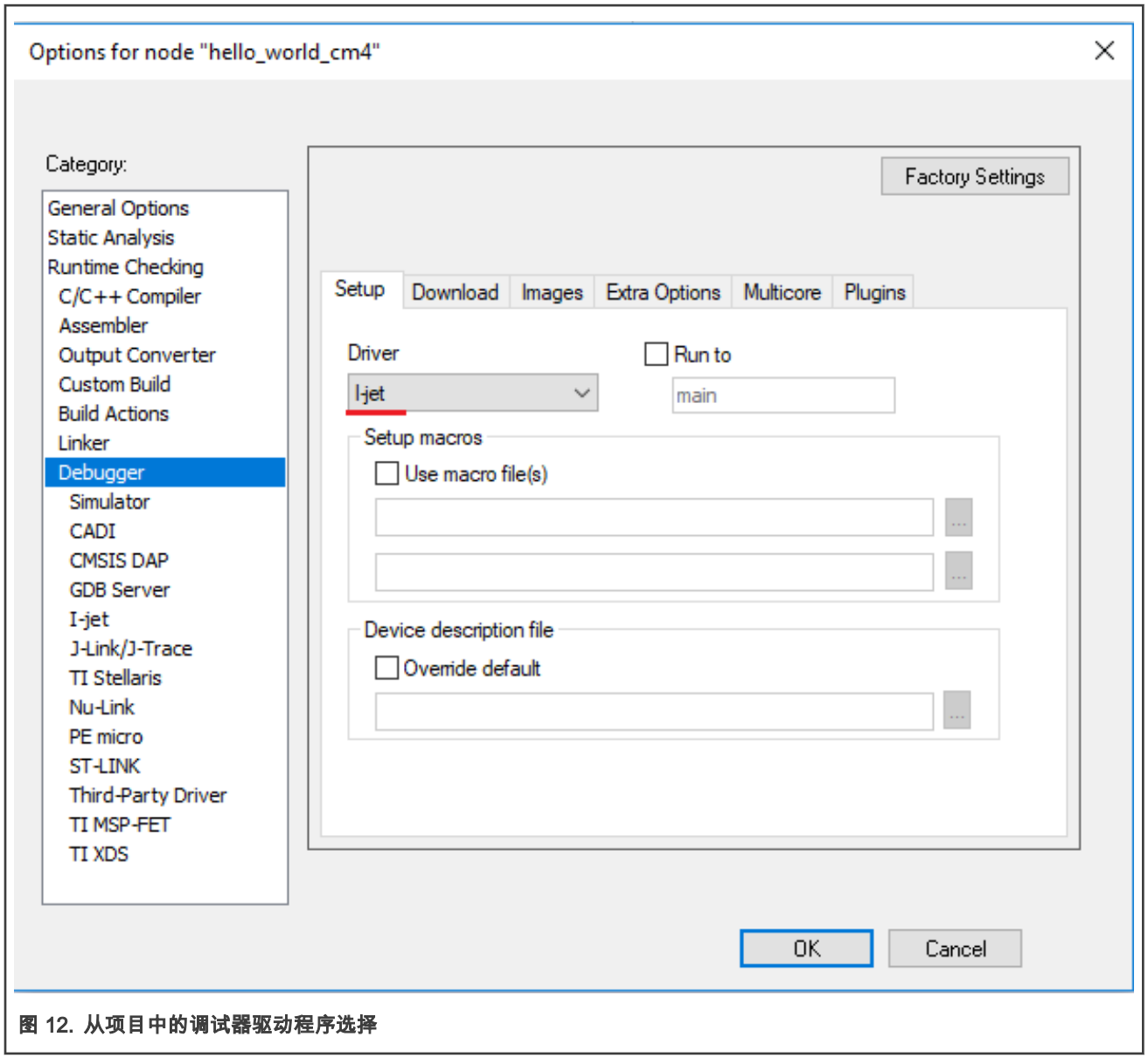

图 12. 从项目中的调试器驱动程序选择

- 检查下载设置。由于主项目应负责刷新要调试的设备，因此从项目不应尝试下载代码。此操作可能会导致错误。图 13 显示了从项目的下载设置。

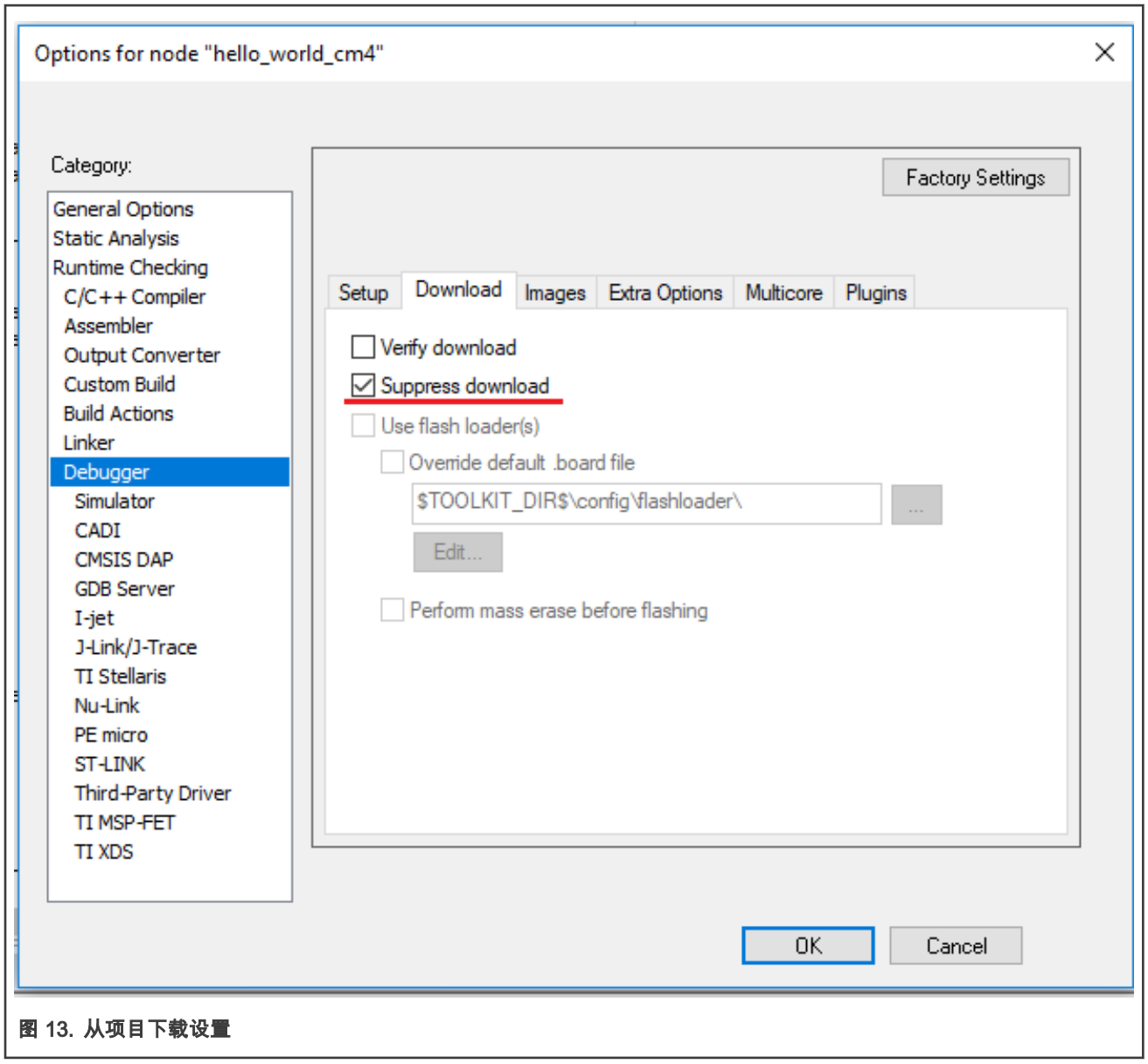

图 13. 从项目下载设置

同样，从项目在初始化时不应尝试复位设备。这也可能导致错误或不稳定的行为。因此，从项目需要禁止复位。这是通过在相应调试器的“设置（Setup）”选项卡中设置正确的复位选项来完成的。在这种情况下，这可以在 ijet / JTAGjet 类别中找到。

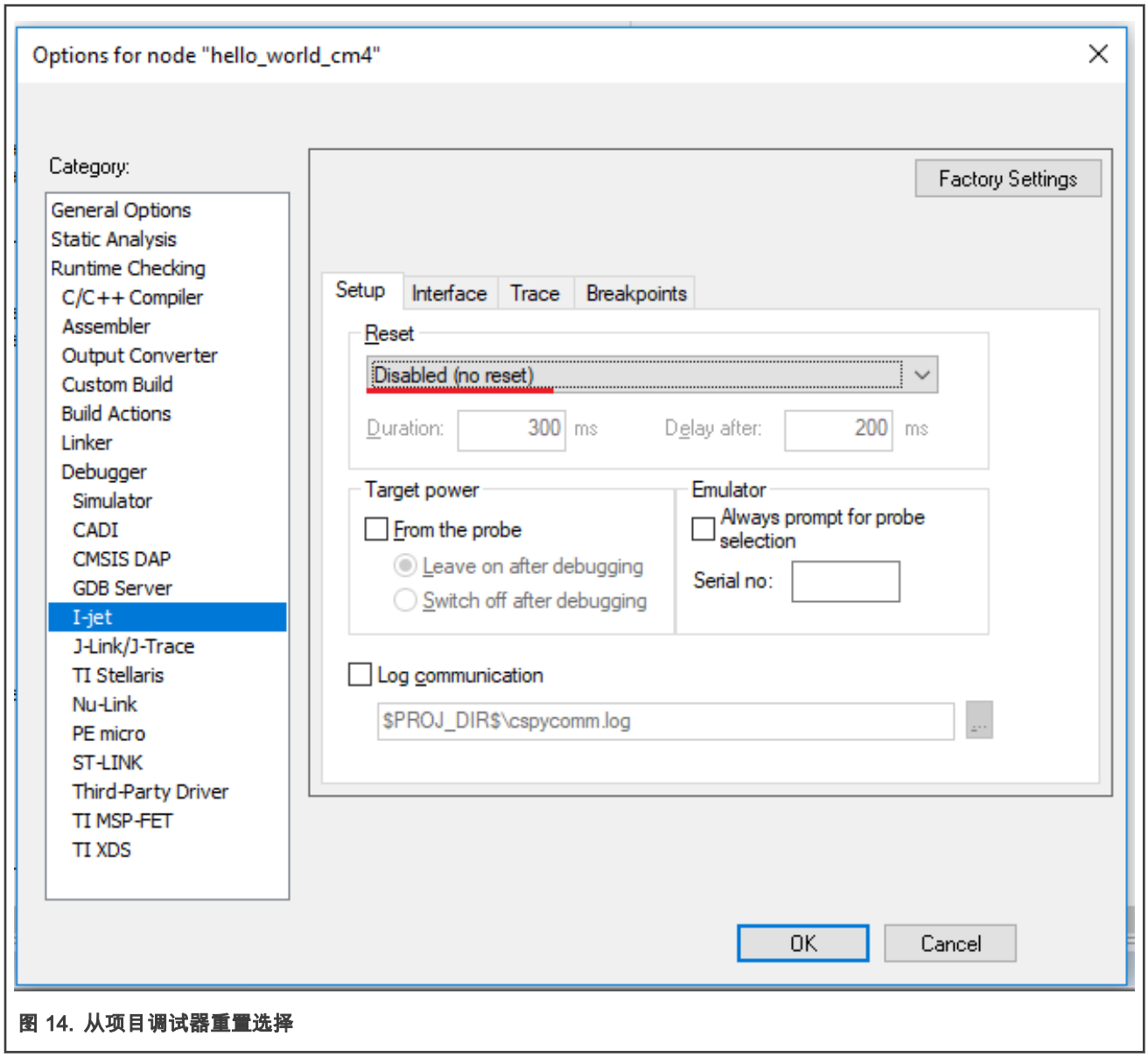

图 14. 从项目调试器重置选择

- 检查从项目的调试器的接口设置是否与主项目的接口设置匹配。

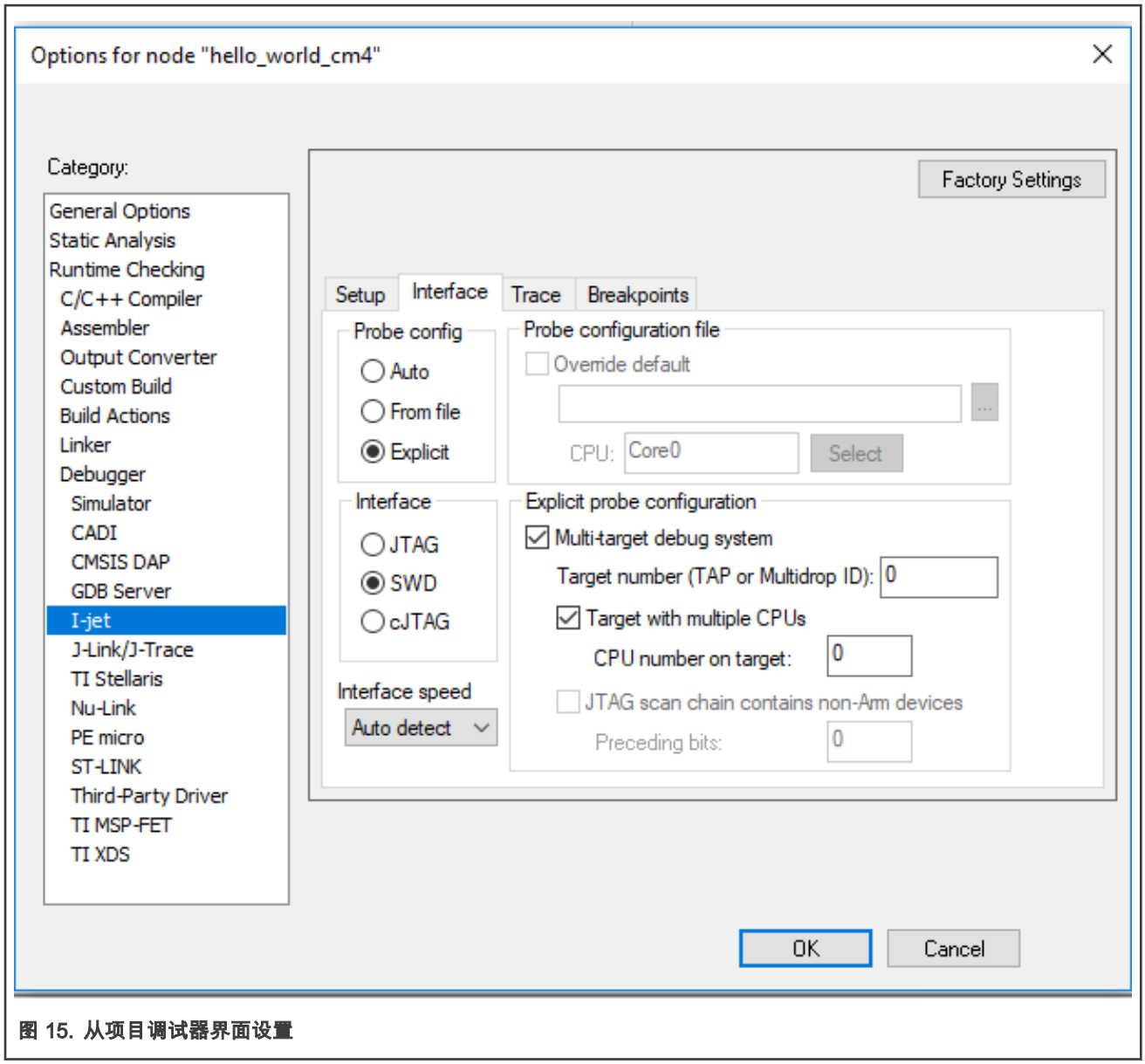

图 15. 从项目调试器界面设置

4.4 调试

1. 确保项目已编译。

   由于主项目包含来自从项目的二进制文件，因此必须首先成功编译从项目。只有这样才能编译主项目。正确编译两个项目后，关闭从属项目并使主项目保持打开状态。

2. 要启动调试会话，请单击主项目中的“下载并调试（Download and debug）”按钮。

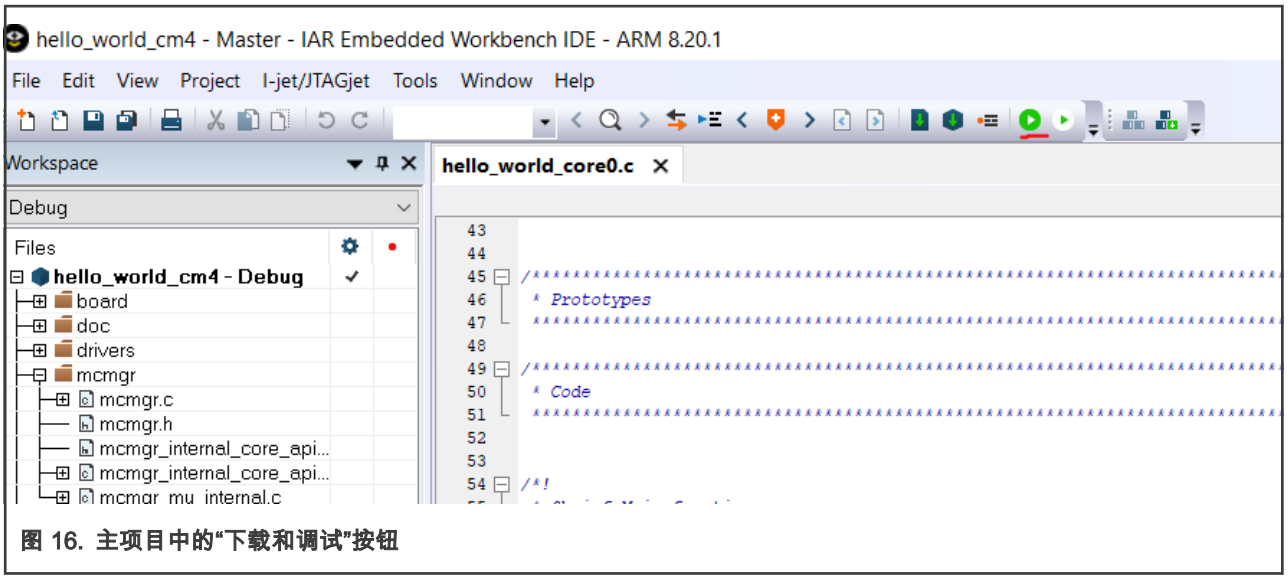

图 16. 主项目中的“下载和调试”按钮

3. 单击“下载并调试（Download and debug）”按钮后，将打开 IAR 的第二个实例，并在该实例中打开从工作区。

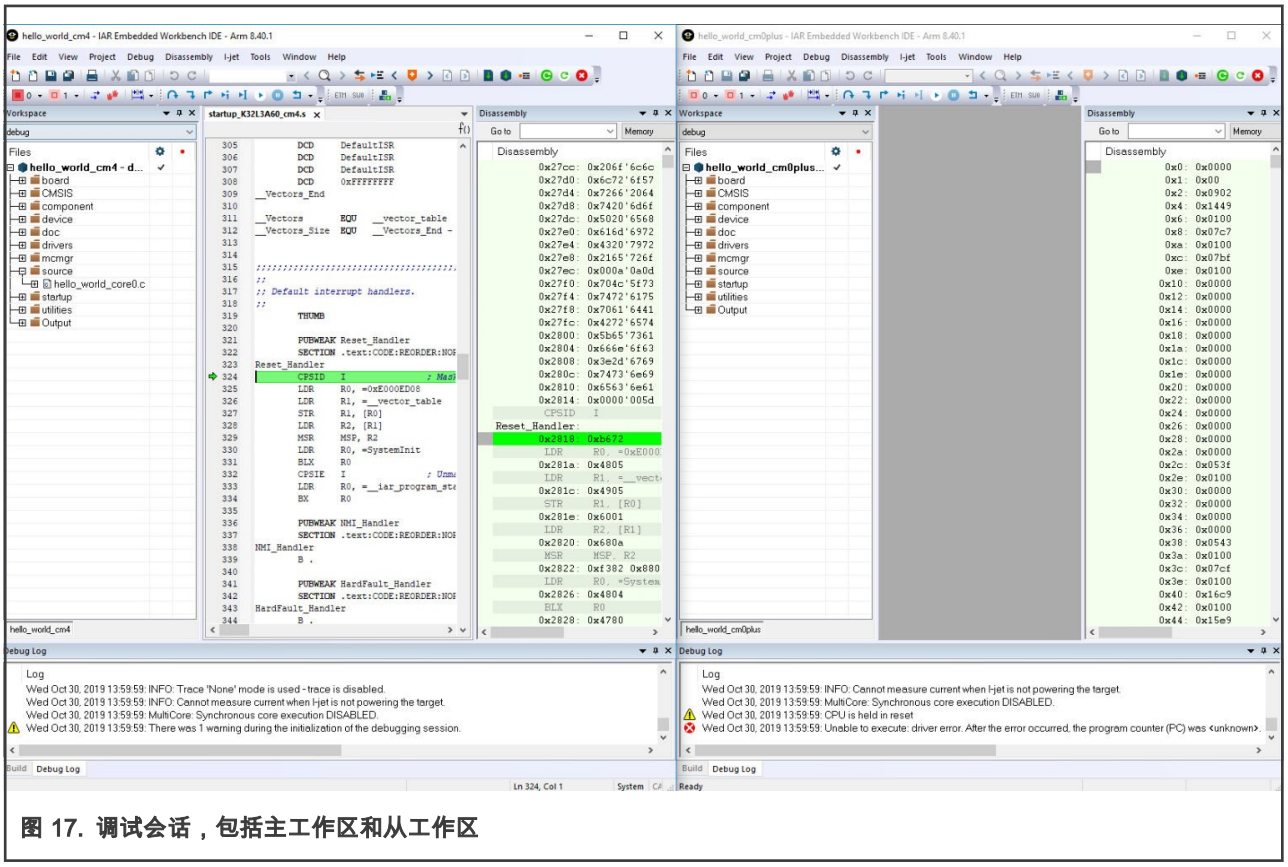

图 17. 调试会话，包括主工作区和从工作区

注意

- 整个 MCU 都可以从任一 IAR 实例进行控制。

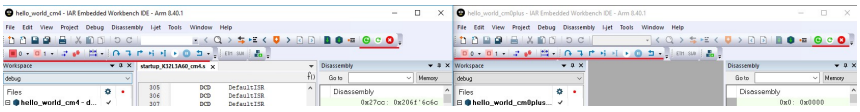

图 18. 两个 IAR 实例中的调试控件

- 有新的控件可以独立或同时控制内核。有控件可以启动两个内核，暂停两个内核的执行或获取两个内核的状态。图 19 显示并解释了 IAR 中的多核控件。

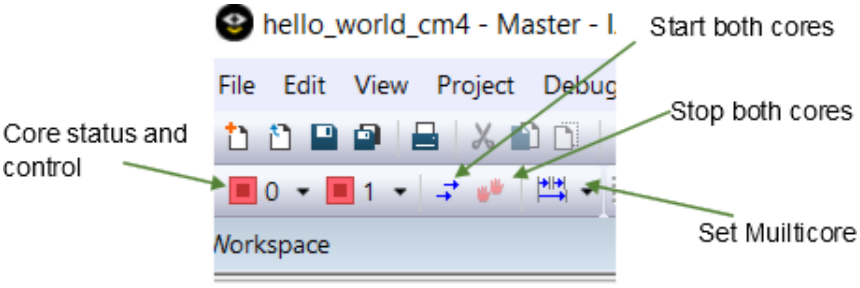

图 19. 多核控制

4. 如果将鼠标悬停在 Core 0 或 Core 1 状态和控件图标上，则会出现一个信息性文本框，其中提供了有关内核（Core）的详细信息。图 20 和 图 21 显示了一个示例。 。

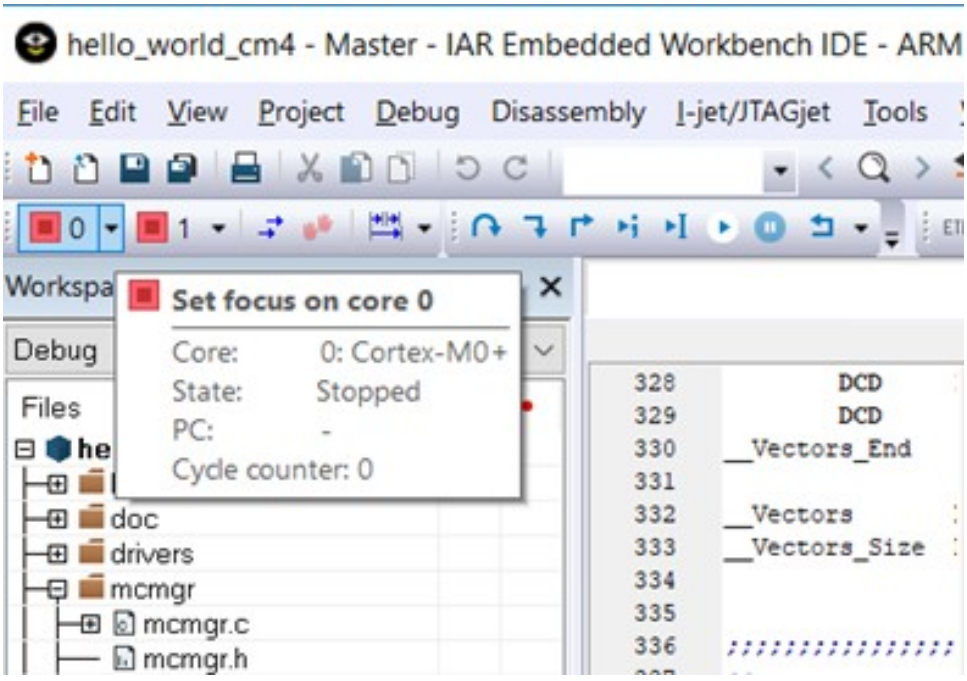

图 20. 核 0 状态

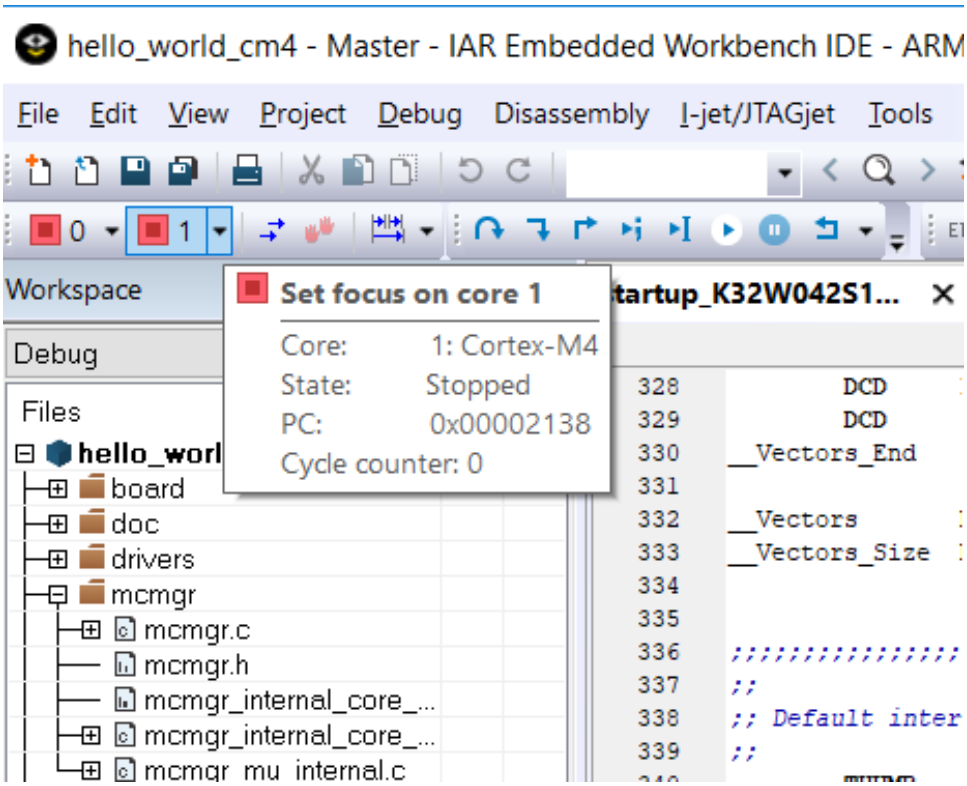

图 21. 核 1 状态

单击每个图标旁边的向下箭头将显示内核的控制选项。

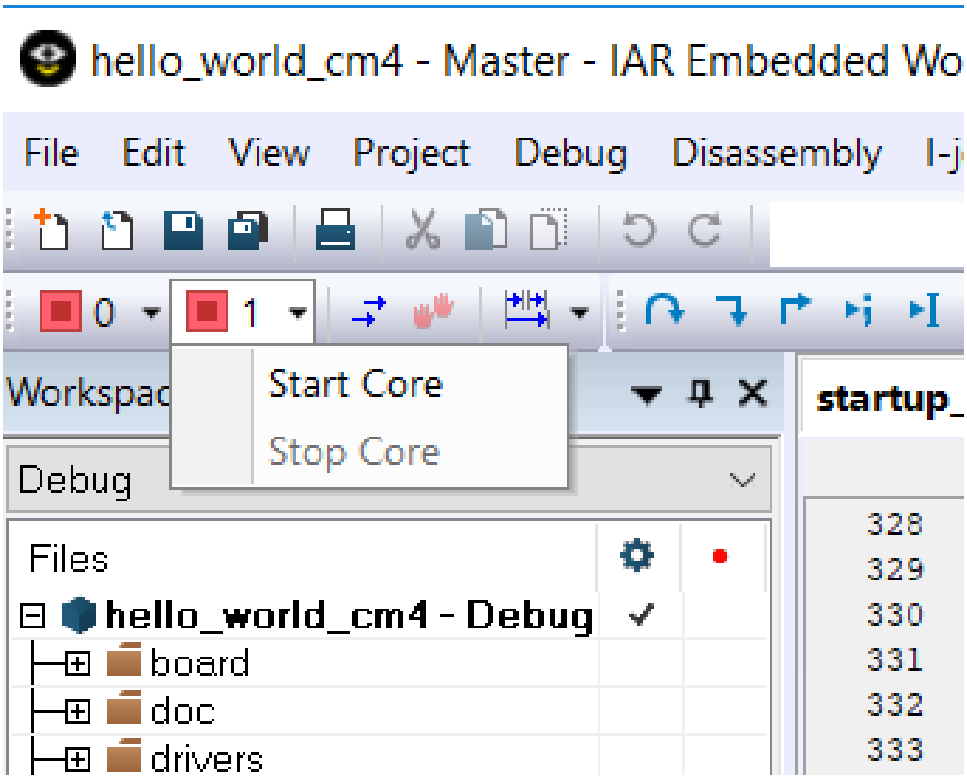

图 22. 核控制

5. 多核切换执行模式将设置其他控件的操作方式（单步执行，单步跨越执行等）。将鼠标悬停在此按钮上将显示当前的操作状态。

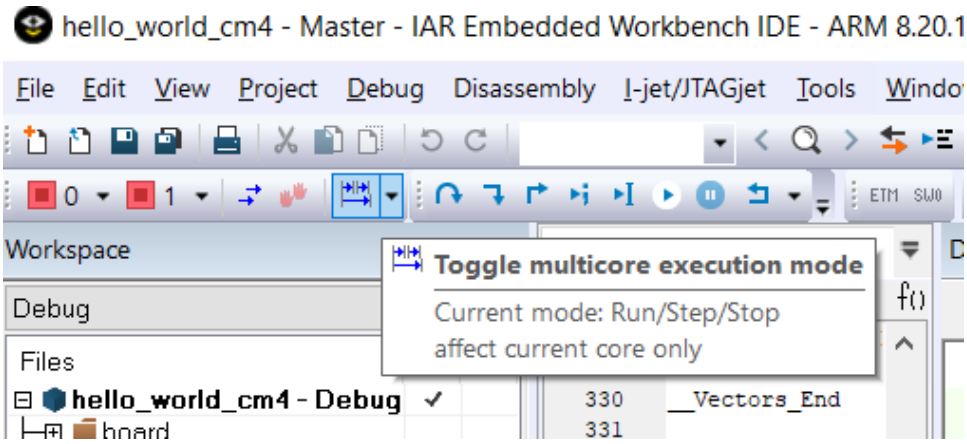

图 23. 切换多核执行模式按钮

6. 单击此按钮旁边的向下箭头将允许用户在单核操作和双核操作之间切换，如 图 24 所示。

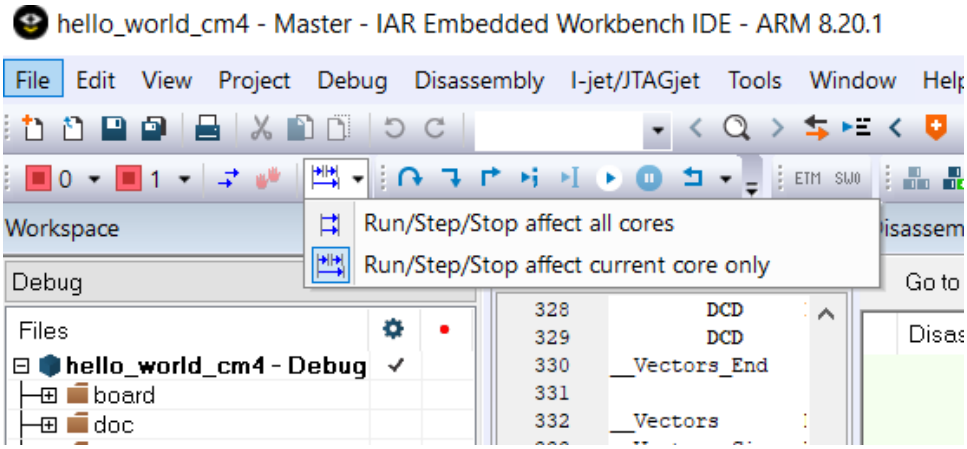

图 24. 多核模式选择选项

5 MCUXpresso 中的多核项目

多核 MCU 可以通过多种方式进行设计。但是，在 MCUXpresso IDE 中，一个基本的期望是一个内核（主核）将控制在其他（从属）内核上运行的代码的执行（或至少启动）。本节介绍：

- 如何制作全新的多核项目。
- 从一对现有项目（主从对）中创建一个多核项目。
- 如何在 MCUXpresso 中调试多核项目。

5.1 多核项目创建

如下所述的多核应用程序项目包含两个链接的项目：

- 一个包含主核代码的项目。
- 一个包含从核代码的项目。

主核（Master）项目包含一个到从核（Slave）项目的链接，当构建 Master 项目时，该链接将使得 Slave 的输出映像包含在 Master 映像中。

注意

构建 Master 项目将触发首先构建 Slave 项目。

5.1.1 创建主/从项目对（使用 SDK）

由于主项目的配置需要引用从项目，因此应首先创建从项目。 要创建从项目，请执行以下操作:

1. 将 SDK zip 文件拖放到“已安装的 SDK”视图中（如果尚未安装 SDK）以安装 SDK。 在出现的窗口中，单击“确定”，然后等待导入完成。

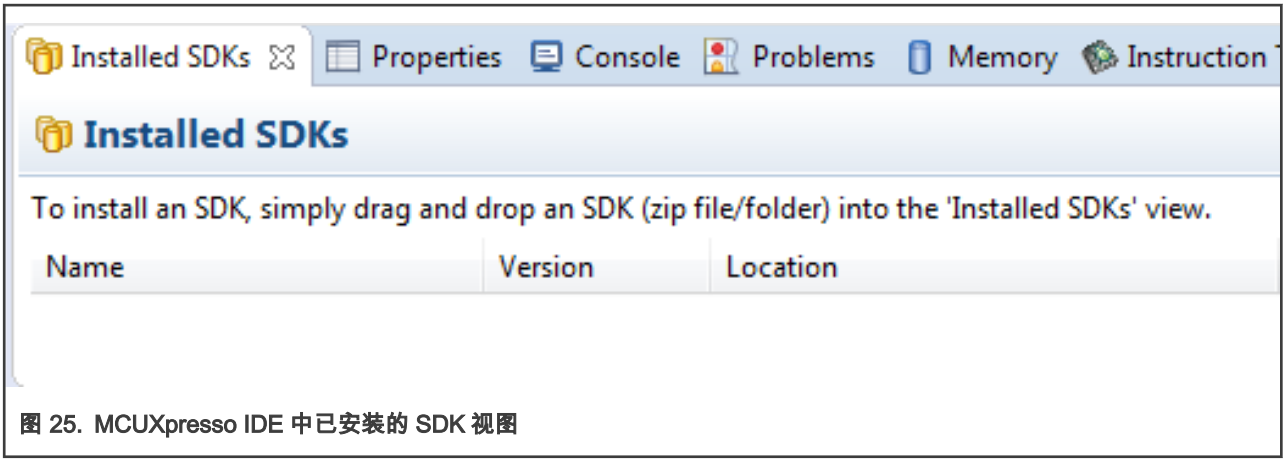

图 25. MCUXpresso IDE 中已安装的 SDK 视图

2. 启动“新建项目向导”，选择 frdmk32l3a6，然后单击“下一步”。

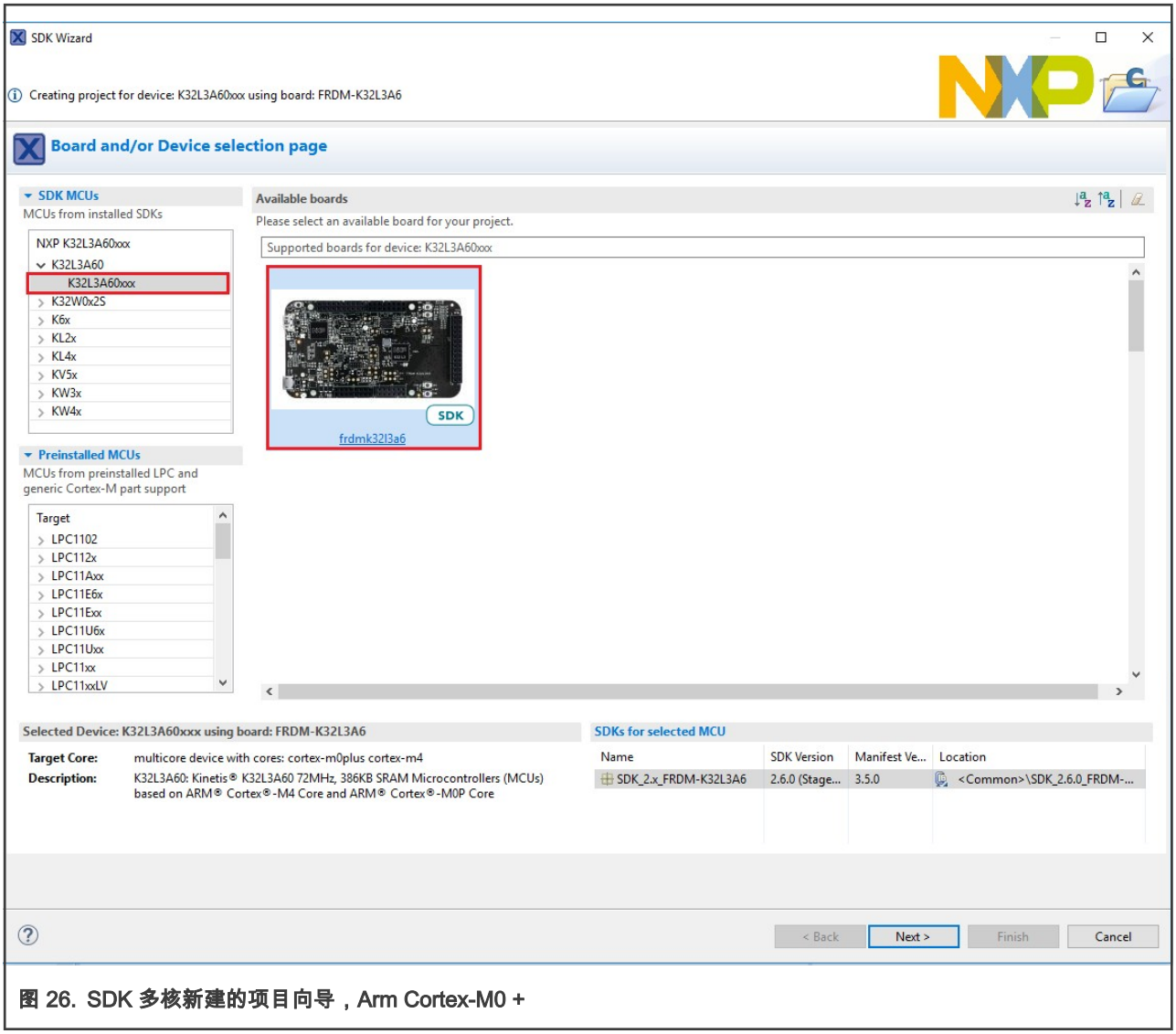

图 26. SDK 多核新建的项目向导，Arm Cortex-M0 +

3. 在“配置项目（Configure the project）”页面上，选择 cm0plus 核。确保在核选项中选择了 M0SLAVE（项目将自动获得 M0SLAVE 的后缀）。如果需要，可以在此阶段为 Slave 项目选择驱动程序，实用程序等。

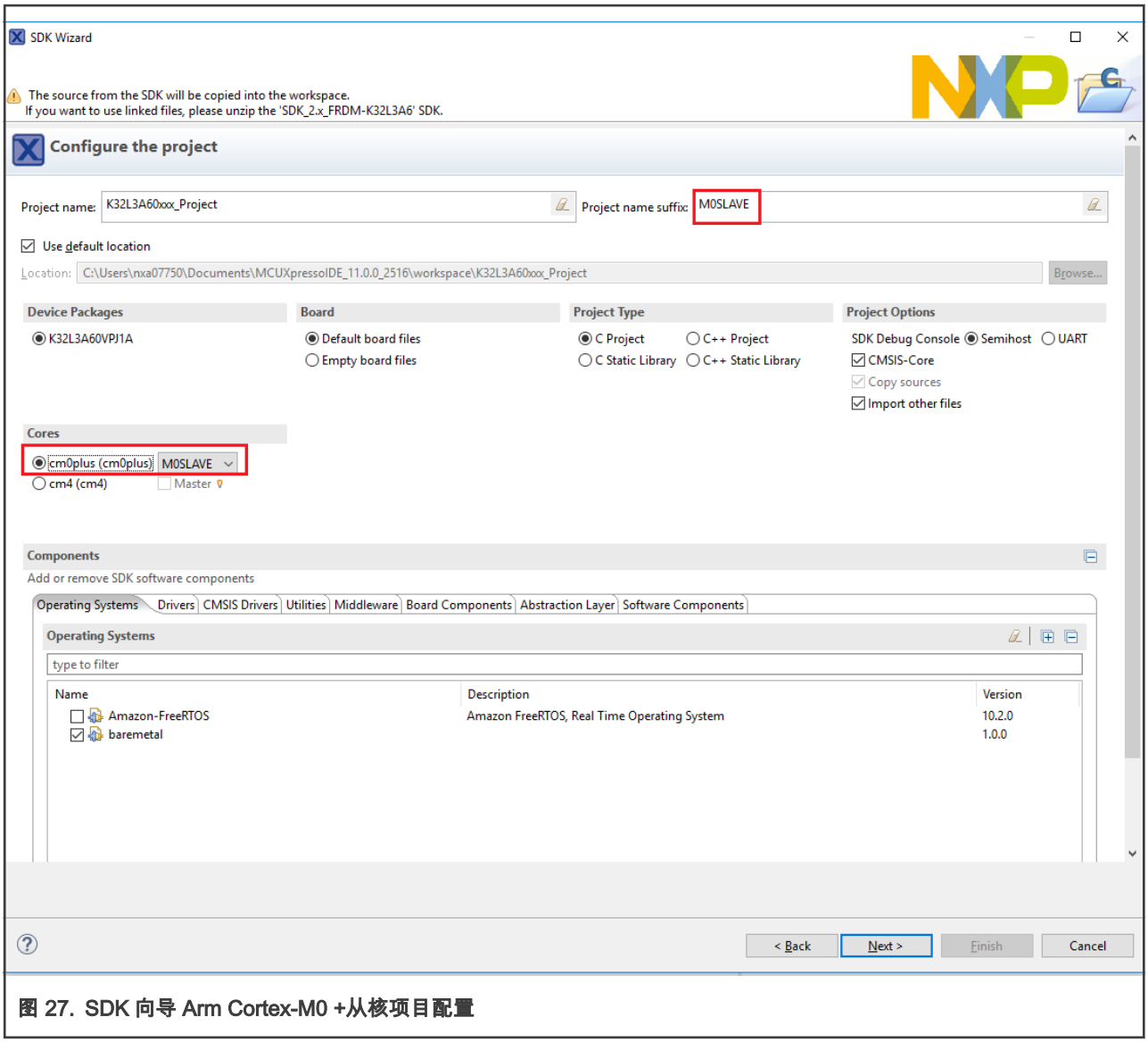

图 27. SDK 向导 Arm Cortex-M0 +从核项目配置

4. 设置 Arm Cortex-M0 +从内存配置。

> **注意**
>
> 默认情况下，MCUXpresso IDE 的托管链接器脚本机制会将代码链接到该视图中的第一个 Flash 区域（如果存在），并将第一个 RAM 区域用于数据，堆和堆栈。

要强制 Arm Cortex-M0 +代码链接到特定的内存区域，请确保所需的内存区域在内存配置列表的顶部。

> **注意**
>
> 要将项目放置在 RAM 中，请从前一个注释中，确保已删除 Flash 区域，并且所需的 RAM 组位于内存配置列表的顶部。

在此示例中，对于 Arm Cortex-M0 +代码，我们选择的存储区域从 0x1000000 开始，该区域对应于 Arm Cortex-M0 +Flash 空间；对于 Arm Cortex-M0 +数据，我们选择从 0x9000000 开始的 RAM。

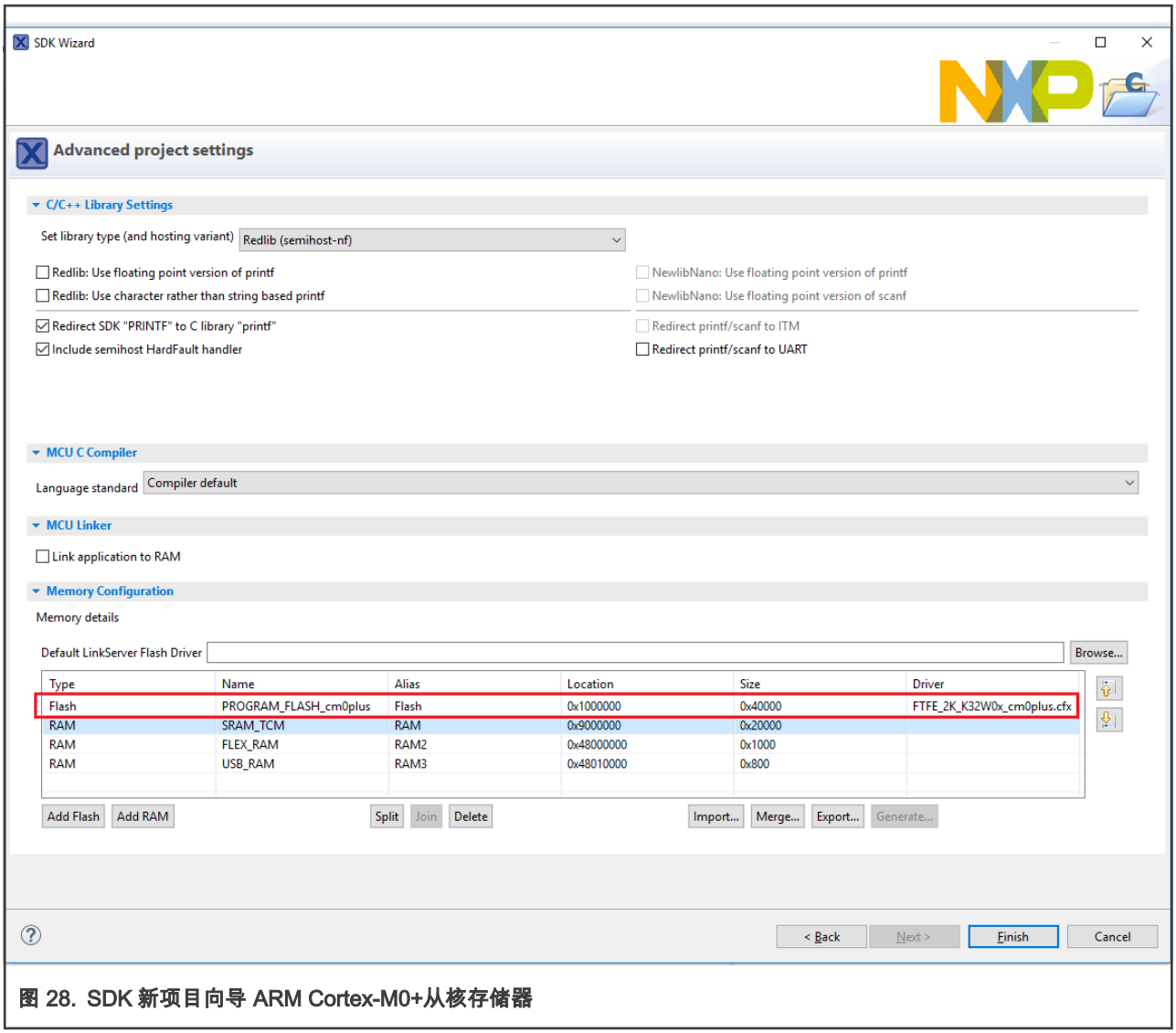

图 28. SDK 新项目向导 ARM Cortex-M0+从核存储器

5. 单击 Finish 完成从项目的创建。

若要创建 Master 项目，请执行以下操作：

1. 启动“New Project Wizard”，选择 FRDMK32L3A6 SDK，然后单击“Next”。 然后，选择 cm4Core 并单击 MASTER 复选框。 这将配置向导来创建多核项目。

注意

项目将自动被赋予后缀 MASTER。如果需要，可以在主项目的这个阶段选择驱动程序、实用程序等。

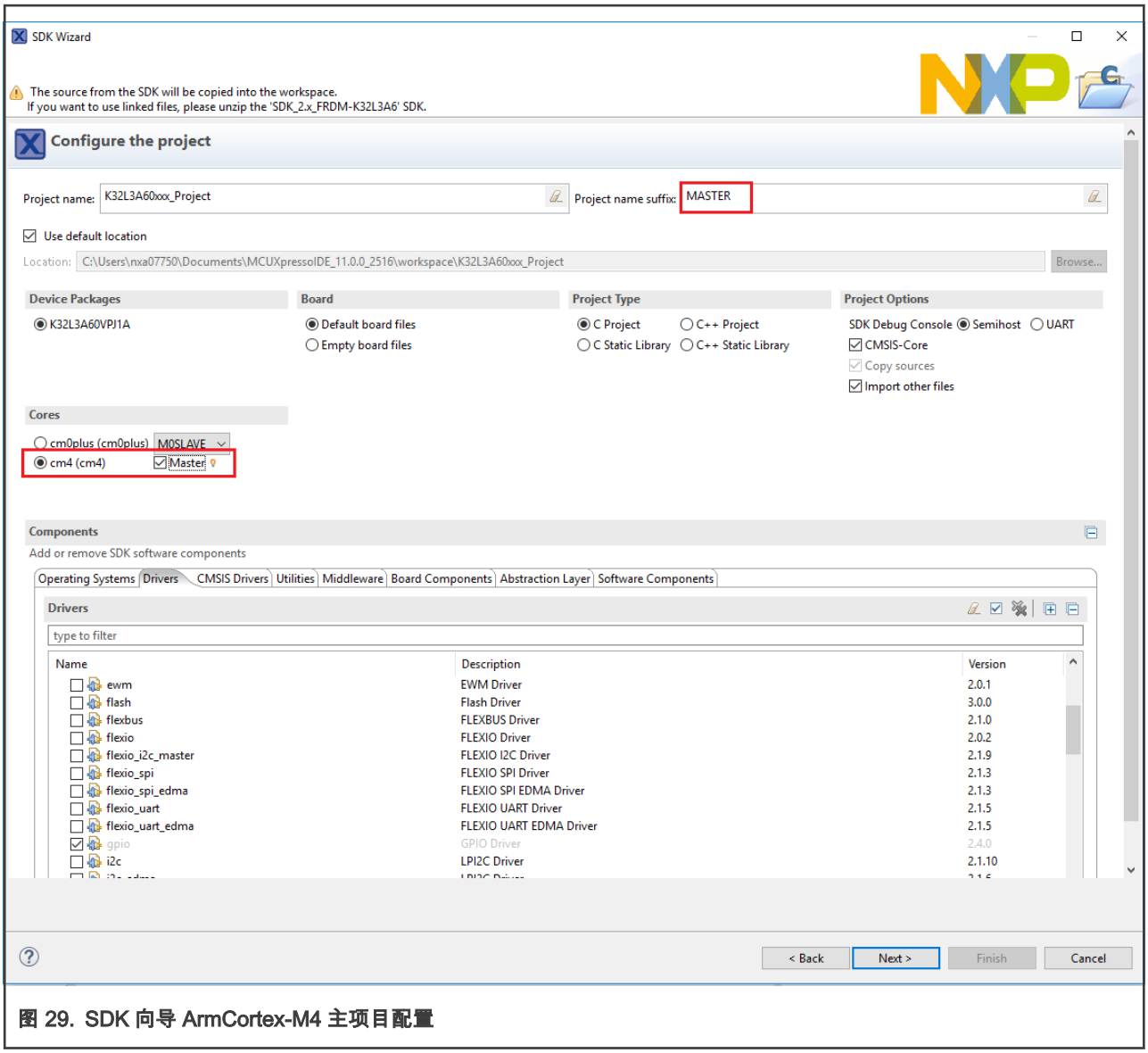

图 29. SDK 向导 ArmCortex-M4 主项目配置

2. 设置 ArmCortex-M4 主内存配置。

> **注意**
>
> MCUXpresso IDEs 托管链接器脚本机制将默认将代码链接到此视图中的第一个 Flash 区域（如果存在的话），并使用第一个 RAM 区域用于数据、堆和堆栈。

要将 ArmCortex-M4 项目代码放置在内存的特定部分，请确保所需的内存区域位于内存配置列表的顶部。在这个例子中，我们将 ArmCortex-M4 代码放置在 ArmCortex-M4 flash 中，从地址 0x0 开始，RAM 中的 ArmCortexM4 数据从 0x20000000 开始。

> **注意**
>
> 如果我们想把项目放在 RAM 中，从前面的注释中，确保 Flash 区域被移除，并且所需的 RAM 组位于内存配置列表的顶部。

3. 单击从项目旁边的 Browse 以打开 M0SLAVE 选择框（如 图 30 所示）来选择工作区中的从项目。

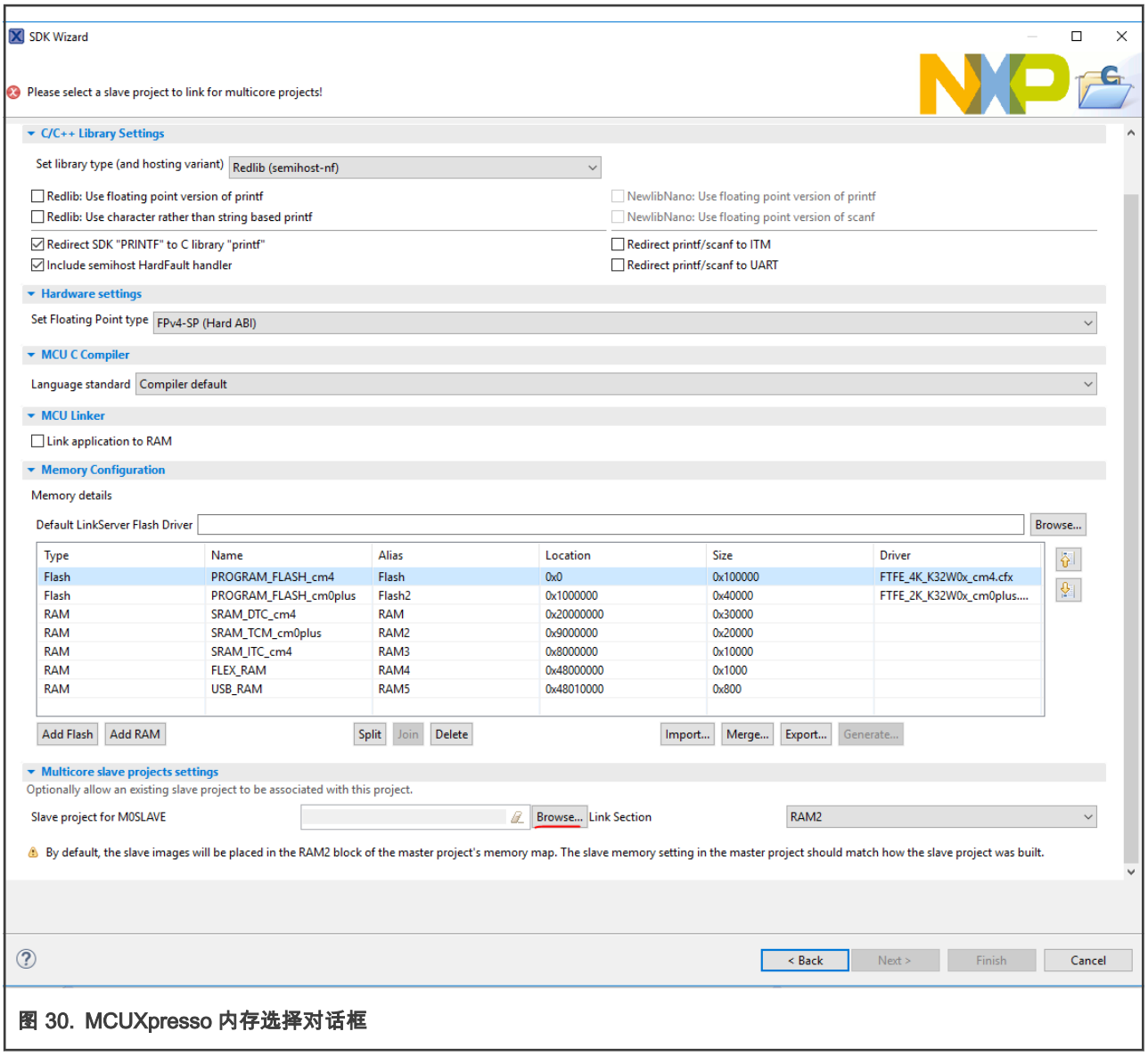

图 30. MCUXpresso 内存选择对话框

4. 选择先前创建的 Slave 项目并单击 OK。

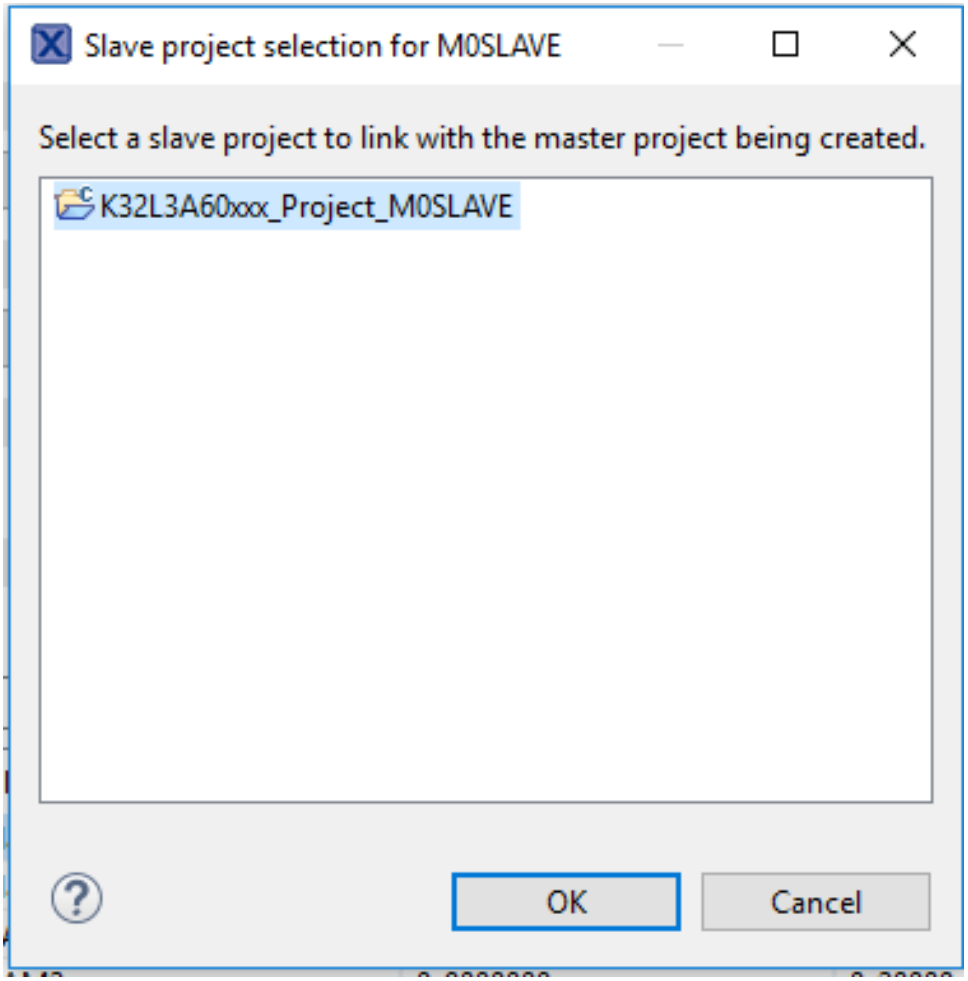

图 31. SDK 新项目向导 SDK Arm Cortex-M4 主从选择

5. 确保链接段名称（默认值为 RAM2）是与从项目的链接地址匹配的内存区域。在这种情况下，我们选择 PROGRAM_FLASH_cm0plus，因为它对应于地址 0x1000000。

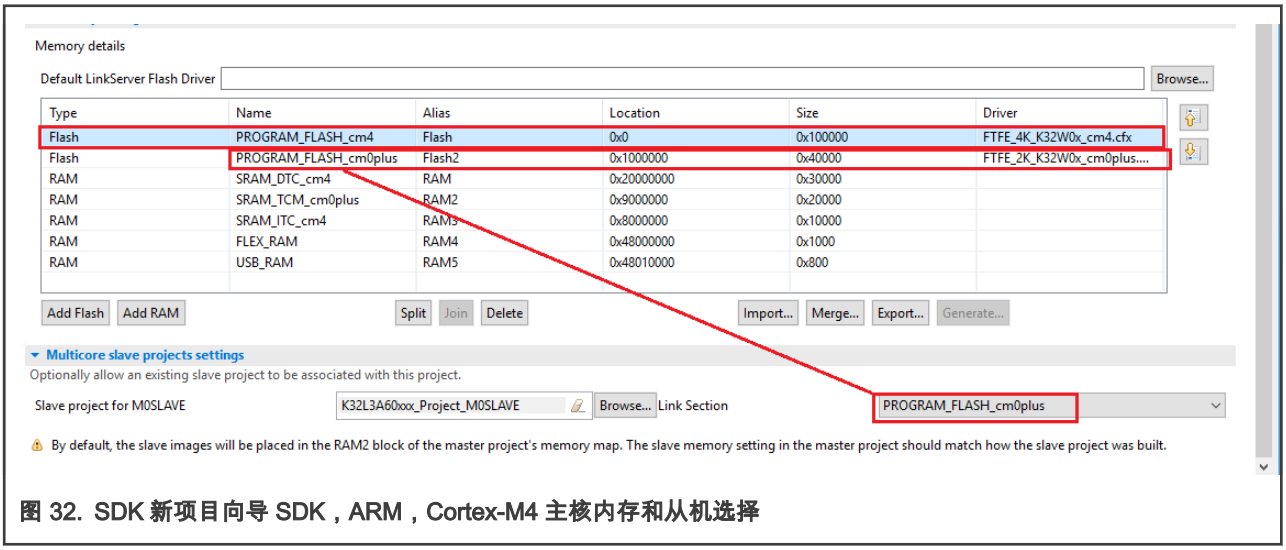

图 32. SDK 新项目向导 SDK，ARM，Cortex-M4 主核内存和从机选择

6. 单击 Finish 生成 Master 项目。

5.1.2 编辑多核的现有项目设置

如果您希望从两个现有项目（一个用于 Arm Cortex-M4，另一个用于 Arm Cortex-M0+）中创建一个多核项目，请对项目设置执行以下修改。位于 FRDMK32L3A6SDK 中的多核 hello_world 示例将作为指导示例。

1. 修改从项目设置。

   转到 Project Properties -> C/C++ Build -> MCU Settings。

   在 memory details（内存详细信息）中，编辑该列表，以便您希望 Arm Cortex-M0+代码所在的内存区域位于列表的顶部。对于本例，我们选择了从 0x1000000 开始的内存区域。

2. 转到项目属性 -> C/C++构建 -> 设置 -> 工具设置 -> MCU 链接 -> 多核，选择所需配置，并确保项目配置为 M0SLAVE。

3. 单击 OK 保存更改并关闭项目属性窗口。

注意

MCUXpresso IDE 的托管链接器脚本机制将默认将代码链接到此视图中的第一个 Flash 区域（如果存在的话），并将第一个 RAM 区域用于数据、堆和堆栈。

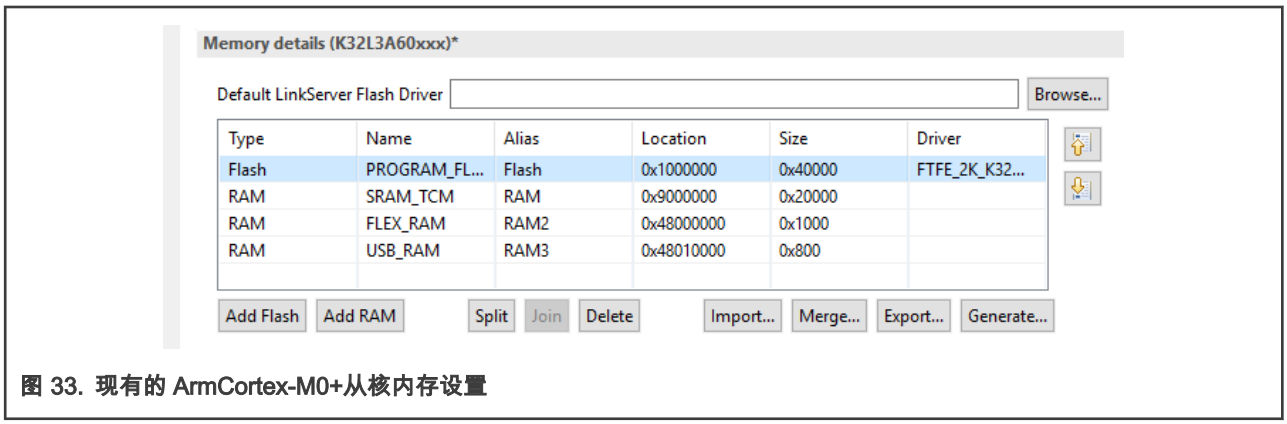

图 33. 现有的 ArmCortex-M0+从核内存设置

注意

要将项目放置在 RAM 中，从前面的注释中，确保删除 Flash 区域，并且所需的 RAM 库位于内存配置列表的顶部。

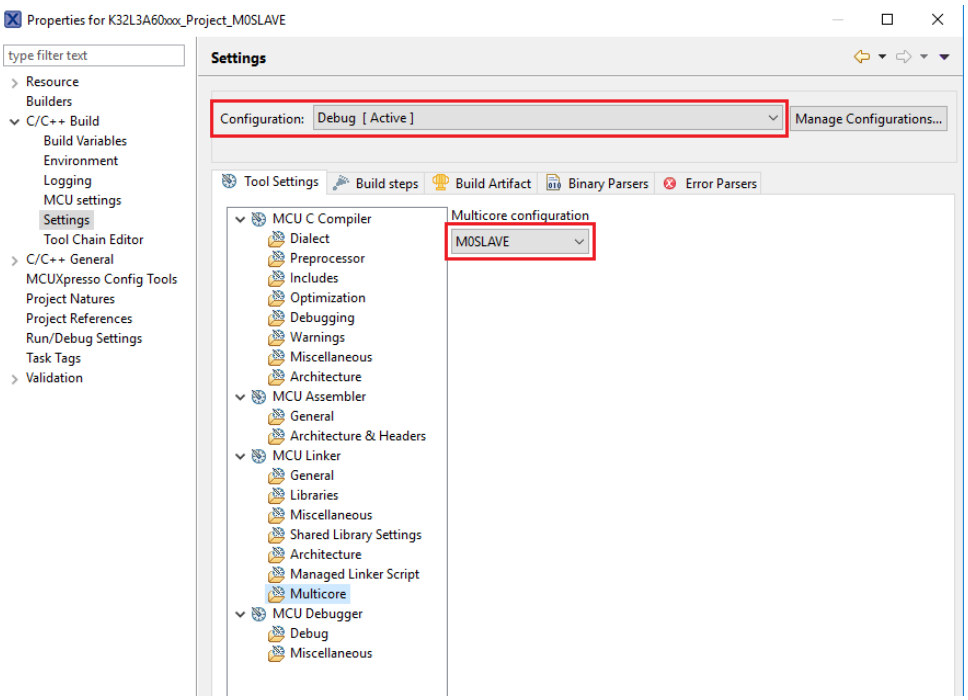

图 34. 现有的 ArmCortext-M0+从项目多核设置

注意

必须为 Arm Cortex-M4 和 Arm Cortex-M0+项目选择相同的构建配置。

接下来开始编辑主项目设置。

1. 转到项目属性 -> C/C++构建 -> MCU 设置。
2. 在内存详细信息中，编辑列表，以便要放置 Arm Cortex-M4 代码的内存区域位于列表的顶部。对于这个例子，我们选择了从 0x0 开始的内存区域。此外，确保您为 Arm Cortex-M0+代码选择的内存区域也在列表中。

注意

MCUXpresso IDE 的托管链接器脚本机制将默认将代码链接到此视图中的第一个 Flash 区域（如果存在的话），并使用第一个 RAM 区域用于数据、堆和堆栈。

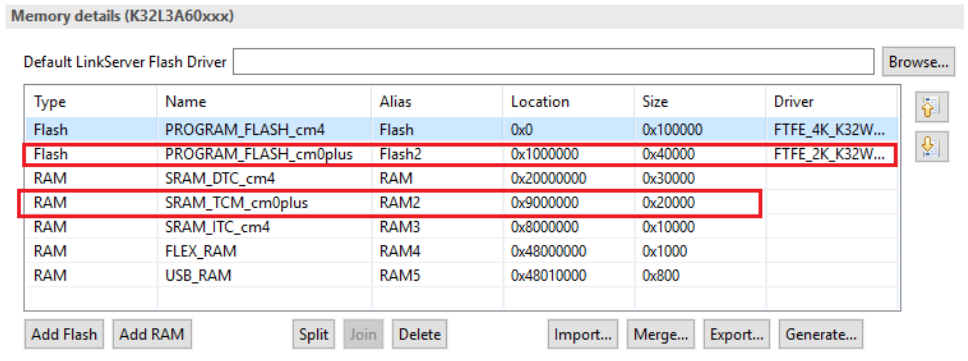

图 35. 现有的 ArmCortex-M4 主核内存设置

注意

要将项目放置在 RAM 中，从前面的注释中，确保删除 Flash 区域，并且所需的 RAM 库位于内存配置列表的顶部。

3. 转到项目属性 -> C/C++构建 -> 设置 -> 工具设置 -> MCU 链接 -> 多核，并选择所需的配置。所选配置应与 Slave 项目相同。 在这种情况下，我们为两者选择了 Debug 配置。
4. 选择 M0SLAVE 复选框以指示应该链接到 ArmCortex-M4 项目的从项目。然后，选择合适的主存储区，其中将放置 ArmCortex-M0+代码。这应该对应于 ArmCortex-M0+项目属性中选择的内存区域。在这个例子中，我们选择了 PROGRAM_FLASH_cm0plus，它对应于从 0x1000000 开始的内存。

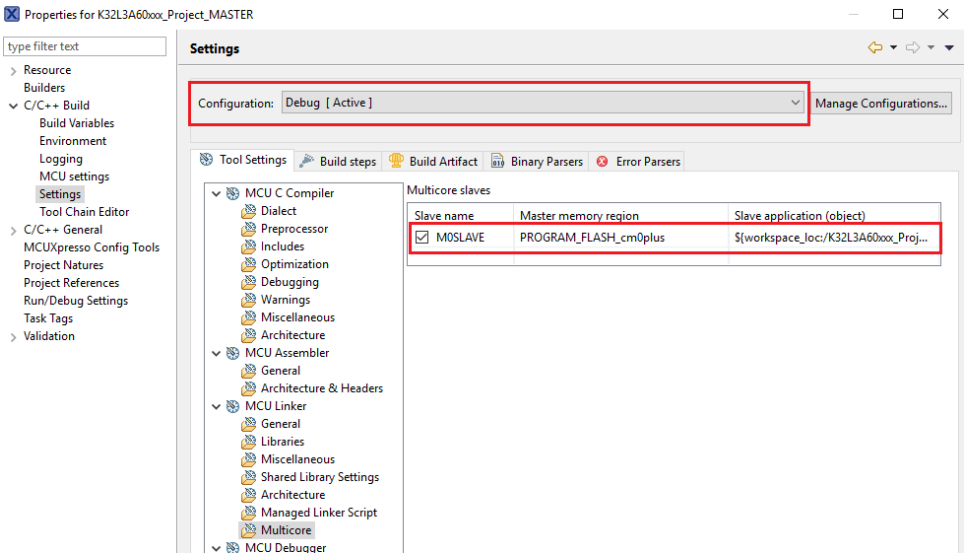

图 36. 现有的 ArmCortex-M4 主项目多核设置

5. 在 Slave 应用程序下，单击省略号打开一个窗口，以选择从项目的对象文件。

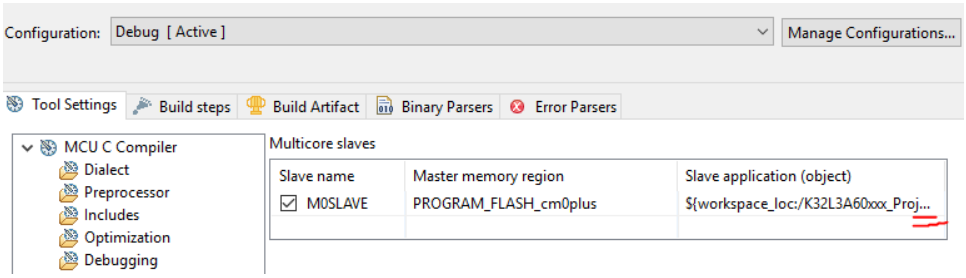

图 37. 点击省略号选择从项目

6. 此文件应在构建从项目后的选定配置文件夹中。

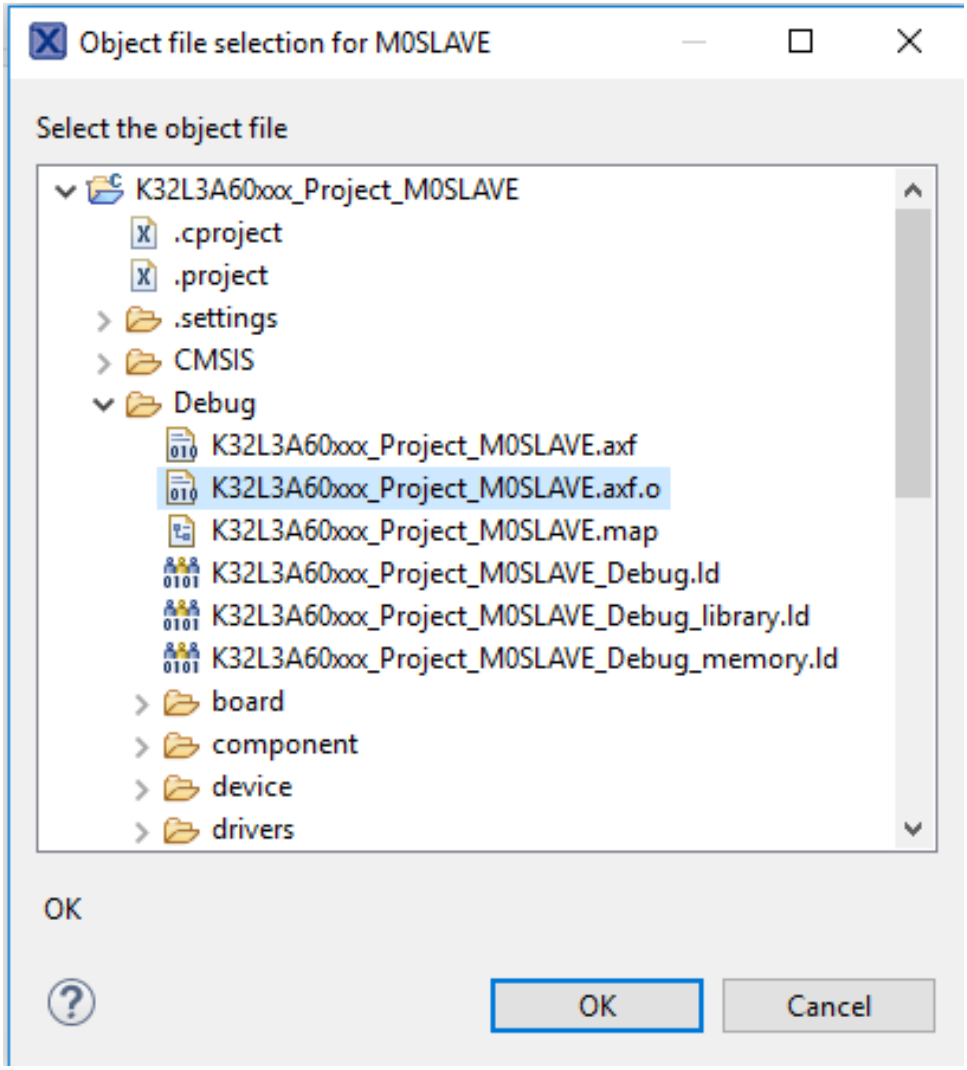

图 38. 现有的 ArmCortex-M4 主从选择

7. 转到 Project Properties -> Project References 并选择从项目的复选框。

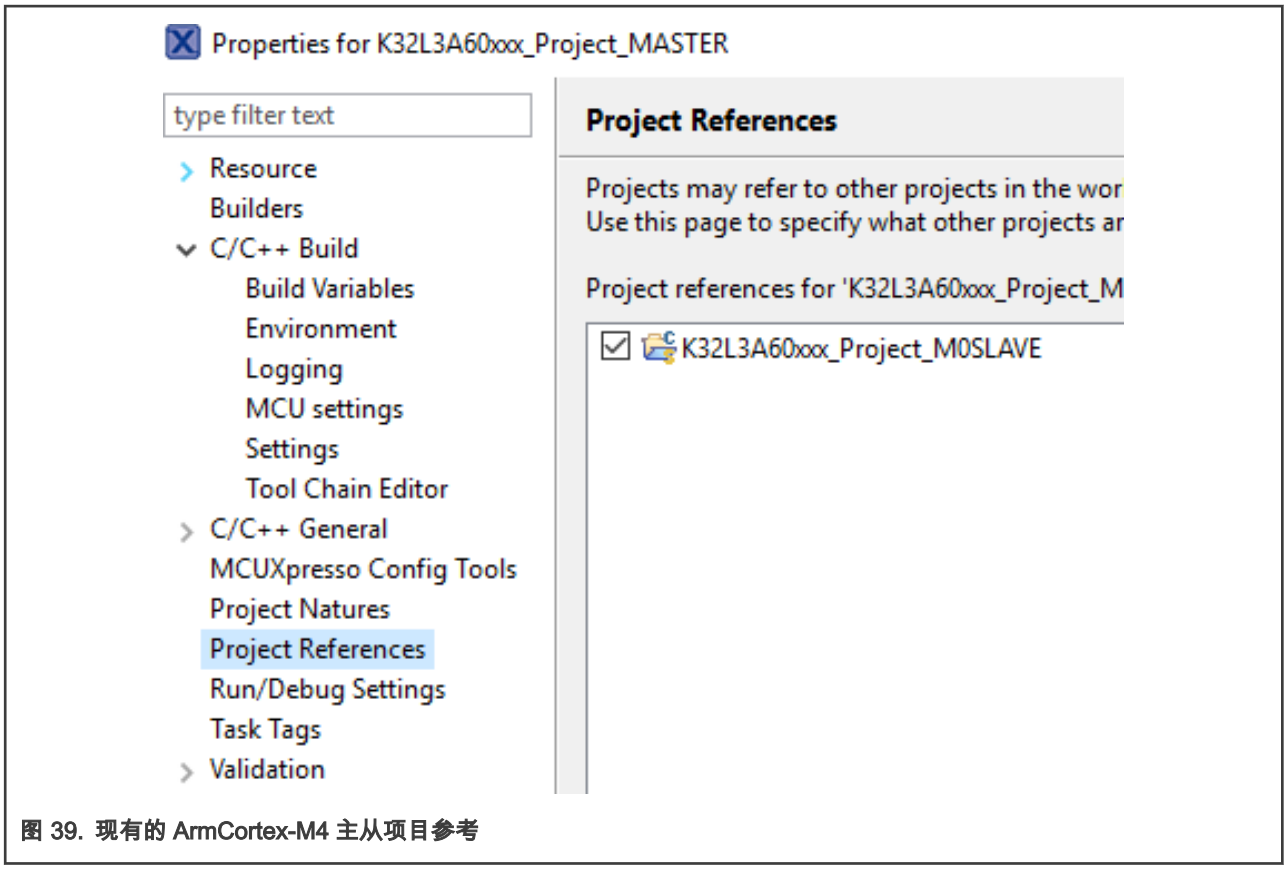

图 39. 现有的 ArmCortex-M4 主从项目参考

8. 单击 OK 保存更改并关闭项目属性窗口。

6 MCUXpresso 中的多核调试

主核调试器处理将主核和辅助核应用程序加载到 SoC 闪存中；因此，应该首先运行（调试）主项目。但是，在调试之前，必须确保已为多核调试正确配置从项目调试设置。

6.1 配置从核调试设置

选择从项目后，转到 Project Properties -> Run/Debug Settings。选择调试配置，点击编辑，如 图 40 所示。

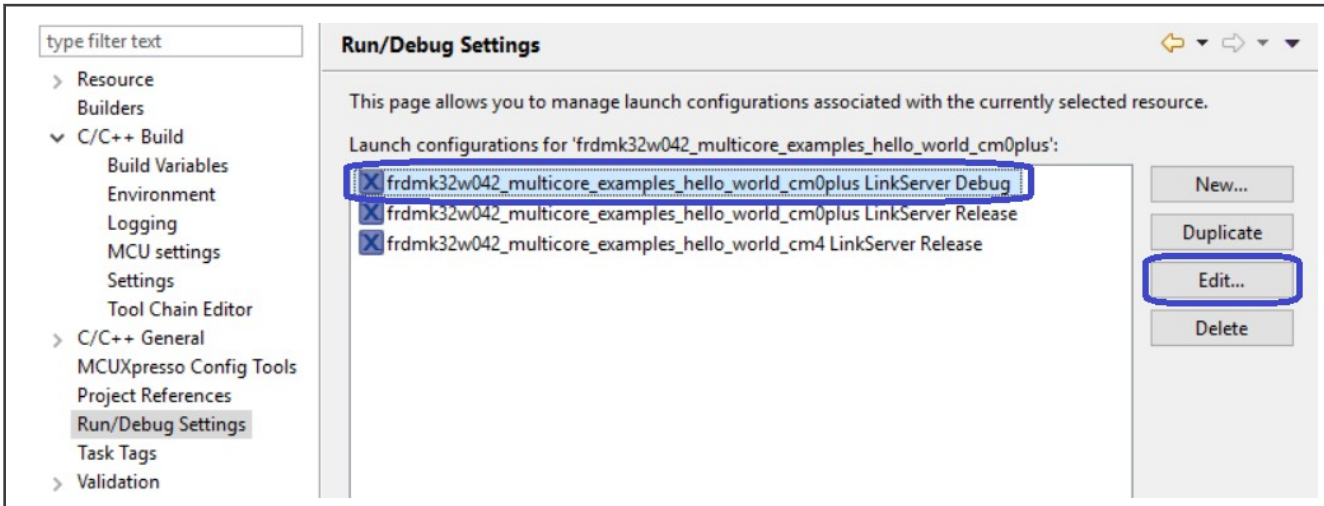

图 40. Arm Cortex-M0+从核编辑运行/调试设置

如果没有配置可用，请单击 New 创建新配置。请务必单击“Search”并从从项目中选择二进制文件。在执行此步骤之前，您可能还需要构建从项目。

在“Edit”启动配置属性窗口中，单击“LinkServer Debugge”选项卡，勾选“Attach only”，如 图 41 所示。

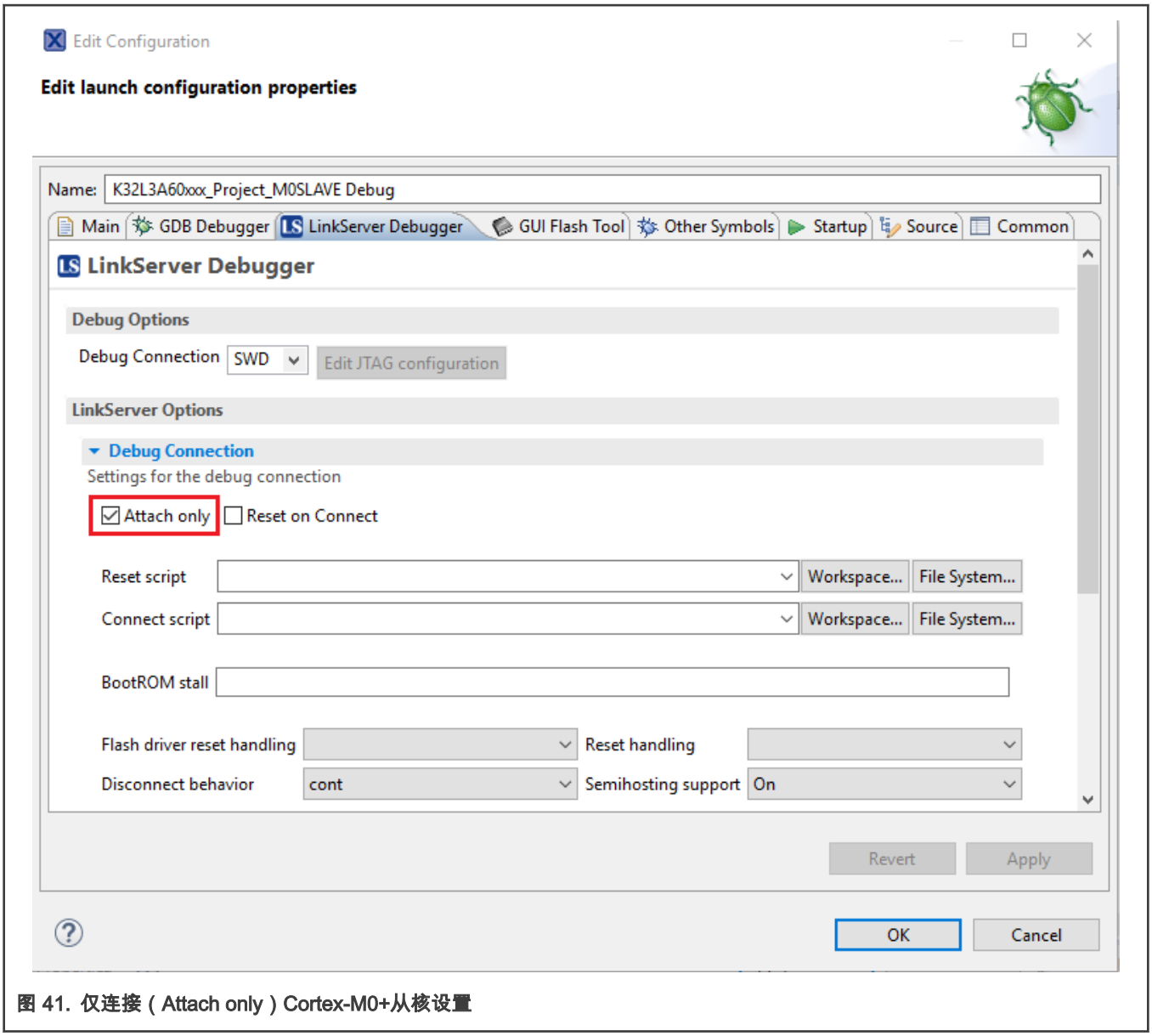

图 41. 仅连接（Attach only）Cortex-M0+从核设置

6.2 主核调试

要下载和运行多核应用程序，请切换到主应用程序项目并执行第 7.3 节中描述的所有步骤，在 MCUXpressoIDE 用户指南中运行示例应用程序（见<K32L SDK root>\docs）。这些步骤对于单核应用程序和双核应用程序的主端都是通用的，以确保多核应用程序的主端都正确加载和启动。

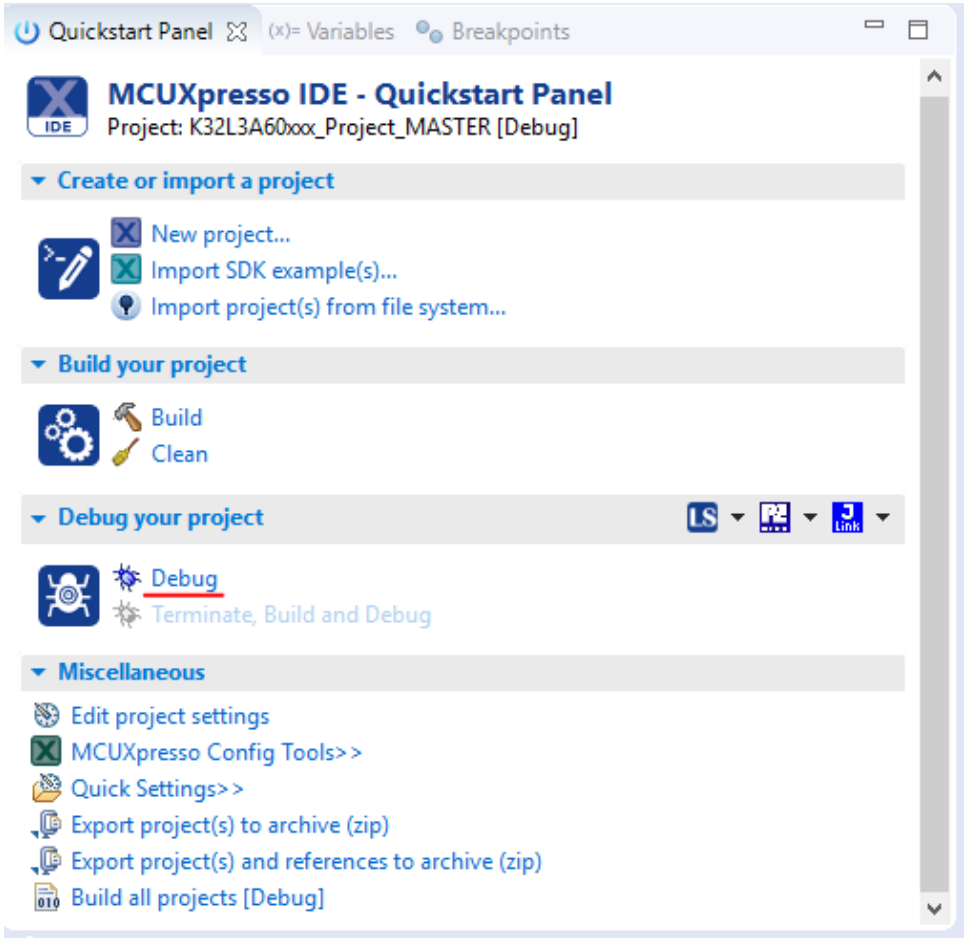

图 42. 调试 K32L3A60xxx_Project_MASTER

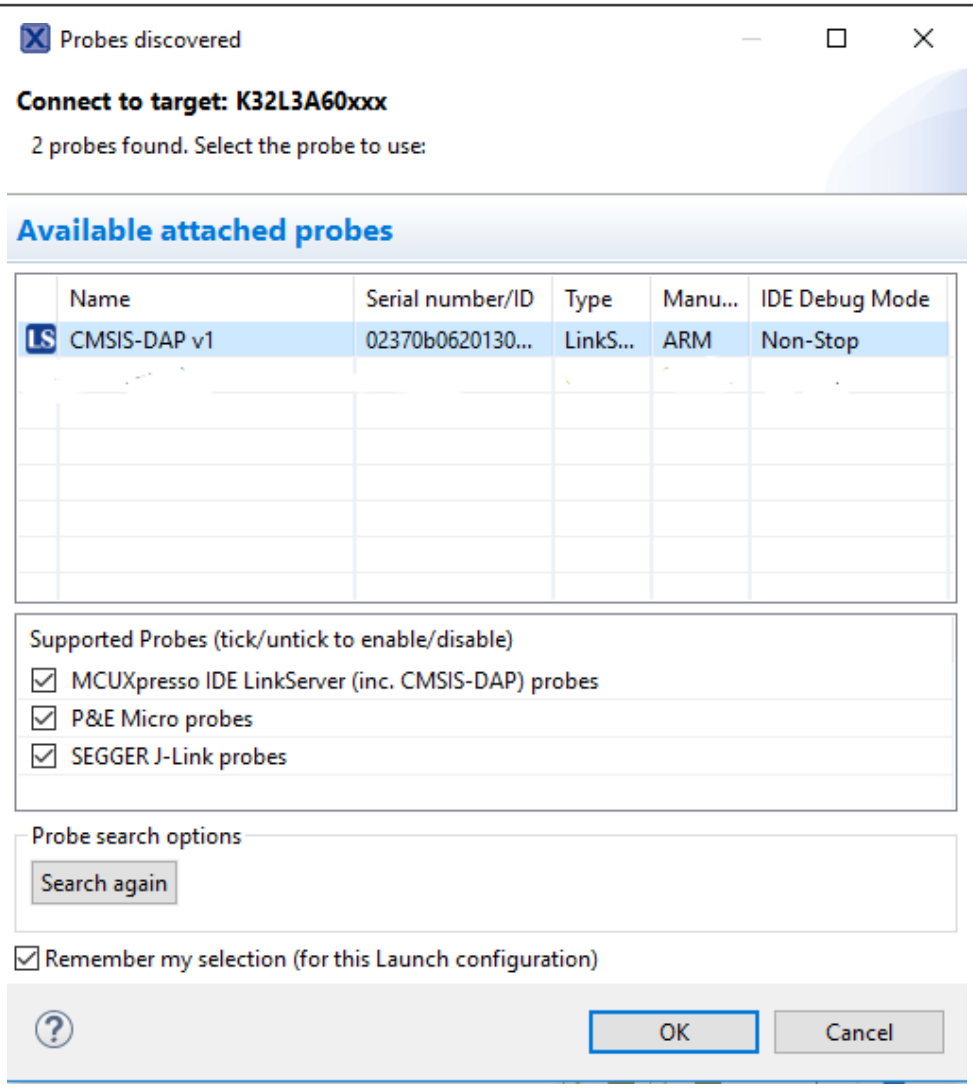

图 43. 连接调试器：调试仿真器选择

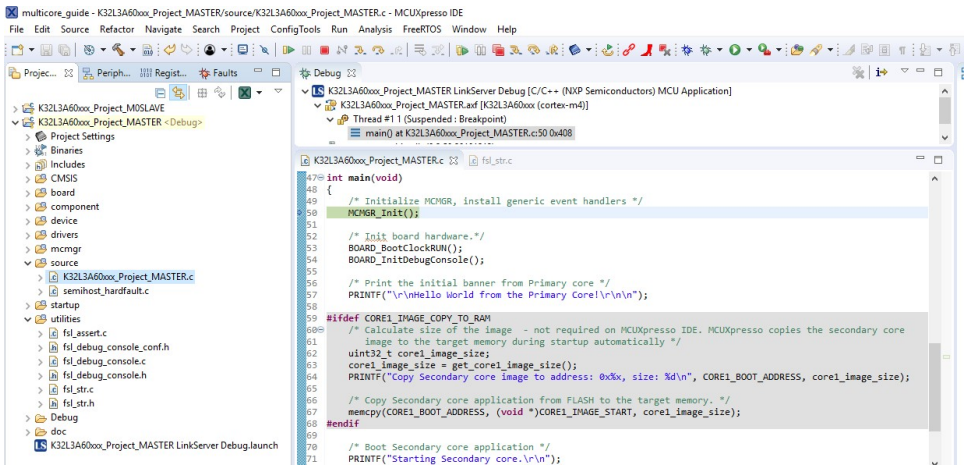

图 44. 运行调试时，在 main()停止主核应用程序

从这里开始，您可以继续调试应用程序，就像它是一个单核设备一样。或者您可以继续学习如何同时调试从核。

6.3 从核调试

通过连接到从核的运行应用程序，可以并行调试两边的多核应用程序。在为主核创建并运行调试会话之后，对从核应用程序执行相同的步骤。突显 ProjectExplorer 中的多核从项目，并在 Quickstart 面板中单击 Debug，如 图 45 所示。

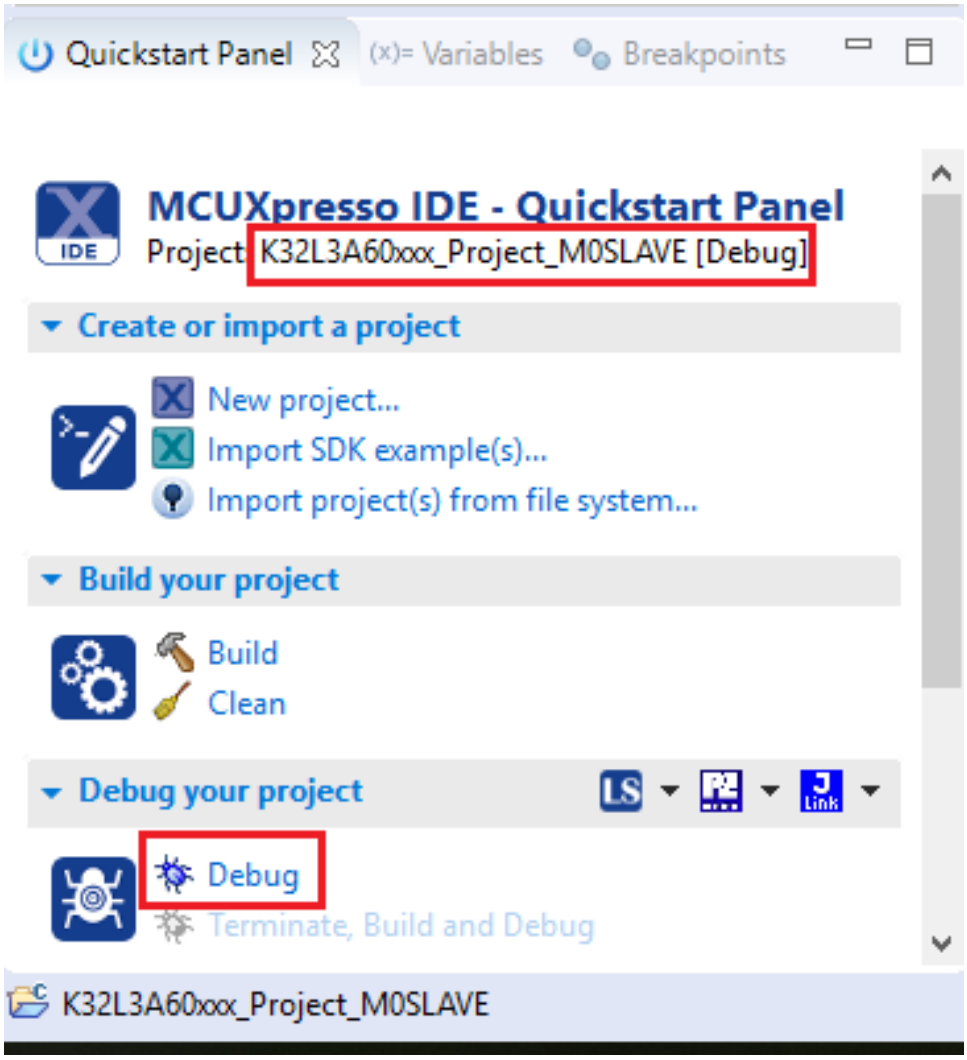

图 45. 调试 K32L3A60xxx_Project_M0SLAVE

在初始化从属调试会话之后，您可以看到两个独立的线程，您可以从单个 IDE 控制它们，如 图 46 所示。您可以使用 Suspend All Debug sessions 和 Resume All Debug sessions 控件同步两个内核的暂停/恢复。

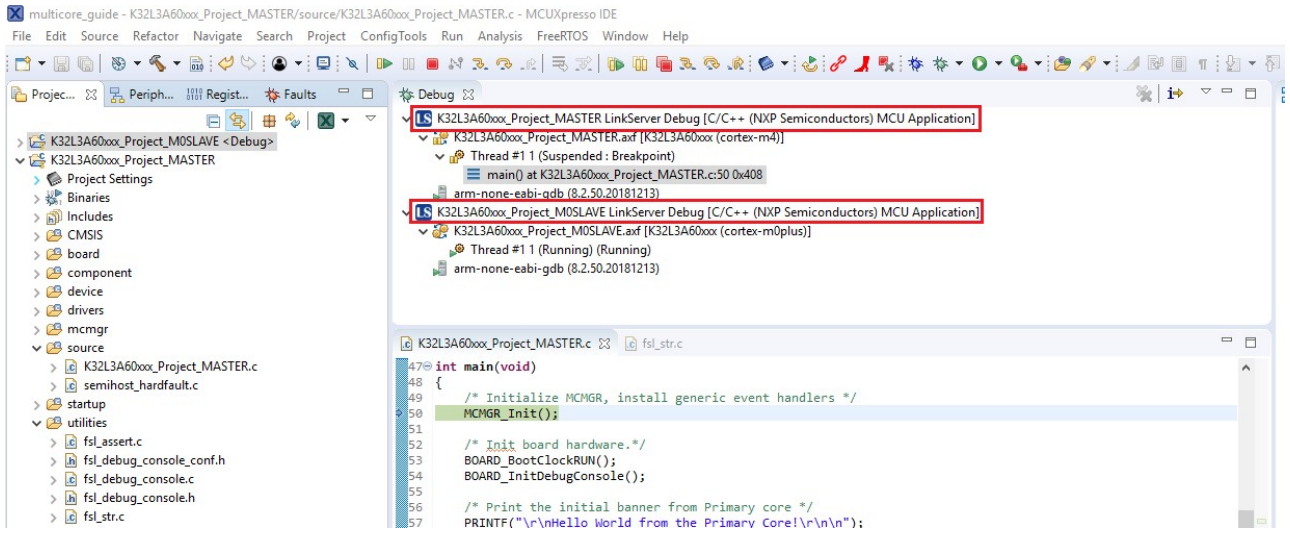

图 46. 两个打开的调试会话

现在，两个调试会话被打开，调试控件可以用于两个调试会话，这取决于调试会话的选择。此时，可以独立地暂停和恢复单个核。还可以通过以下两种方法同步暂停和恢复两个核：

- 选择两个打开的调试会话（多个选择），然后单击“暂停/恢复”（图 47 中突出显示）。

图 47. MCUXpressoIDE 单线程控制

- 使用暂停所有调试会话和恢复所有调试会话按钮（图 48 中突出显示）。

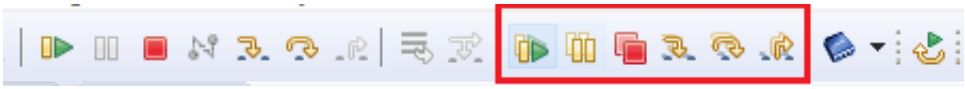

图 48. MCUXpressoIDE 多核同步控制

7 多核代码

主核和从核的项目设置只是双核项目方面的一部分。每个项目的源代码的编写必须考虑到其他核的操作。一般来说，主核需要配置公共时钟、外围设备和内存。在这样做的时候，一旦从复位中释放，从核将不得不做最小的工作，但是绝对没有什么可以阻止从核建立自己的区域。

由于没有从核自行释放的机制，主核必须使用以下任何方法从复位中释放从核：

1. 使用 MCMGR（多核管理器）高级驱动程序
2. 使用 MU（消息单元）单元低级驱动程序
3. 使用原始寄存器访问

MCMGR 高级驱动程序为复杂应用程序提供了灵活性和控制，还可以在内核之间注册和触发事件。MU 单元低级驱动程序只是以一种可移植的方式让您访问寄存器，并且只有在您有一个需要执行有限数量操作的简单程序时才可能使用它。使用此选项时，用户了解目标应用程序在做什么以及核的交互是如何发生的是很重要的。原始寄存器访问应该只用于简单的程序，或者当用户需要自定义函数、驱动程序，或者对 MU 和内核间交互有高水平的理解时。本应用说明仅侧重于推荐使用的 MCMGR 高级驱动程序。MCMGR 代码是 mcuxpressosdk 包的一部分（请参阅 middleware/multicore/MCMGR **文件夹**）。

项目（主从）需要以下文件：

- mcmgr.c
- mcmgr.h
- mcmgr_internal_core_api.h
- mcmgr_internal_core_api_k32l3a6.c
- mcmgr_mu_internal.c

注意

这些文件已经添加到 hello_world 多核项目中，但您可能需要手动将它们添加到您自己的项目中。

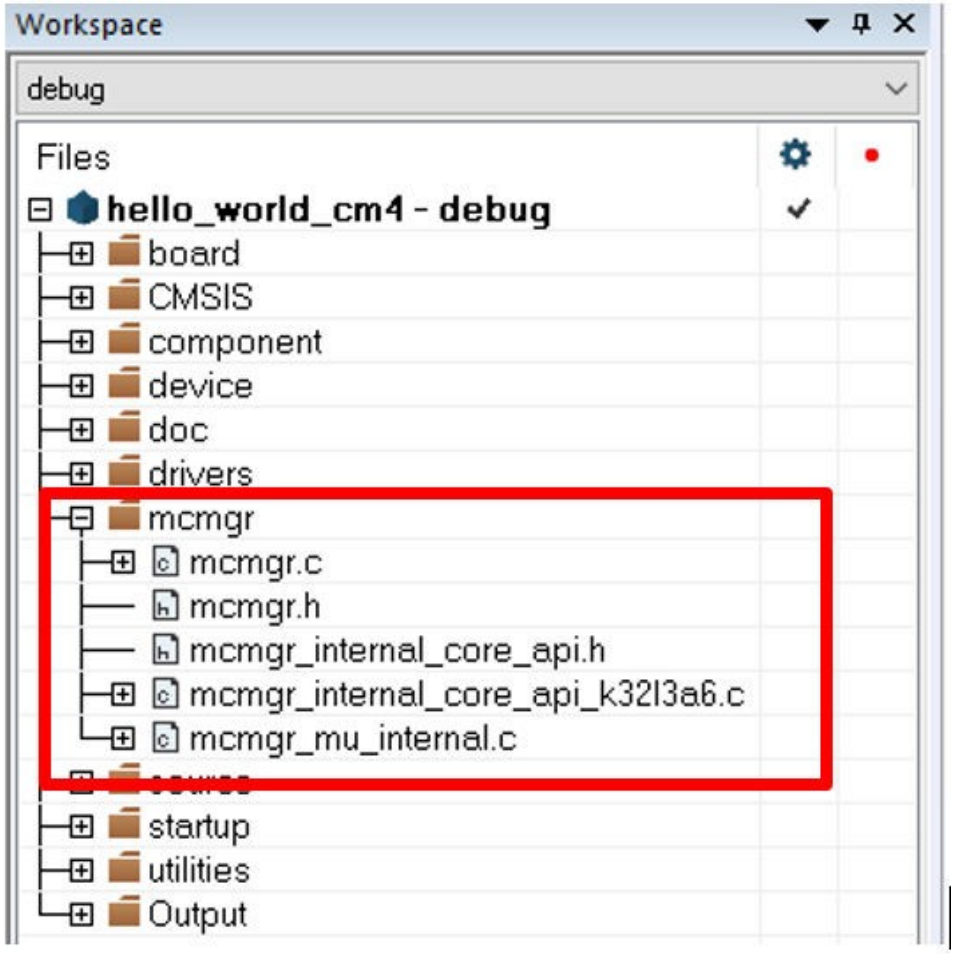

图 49. 多核项目中包含的 MCMGR 文件

此外，必须让编译器知道头文件的路径。图 50 所示的路径应该包含在预处理器搜索路径中。

注意

路径已经添加到 hello_world 多核项目中，但是您可能需要为正在转换的项目手动添加它。

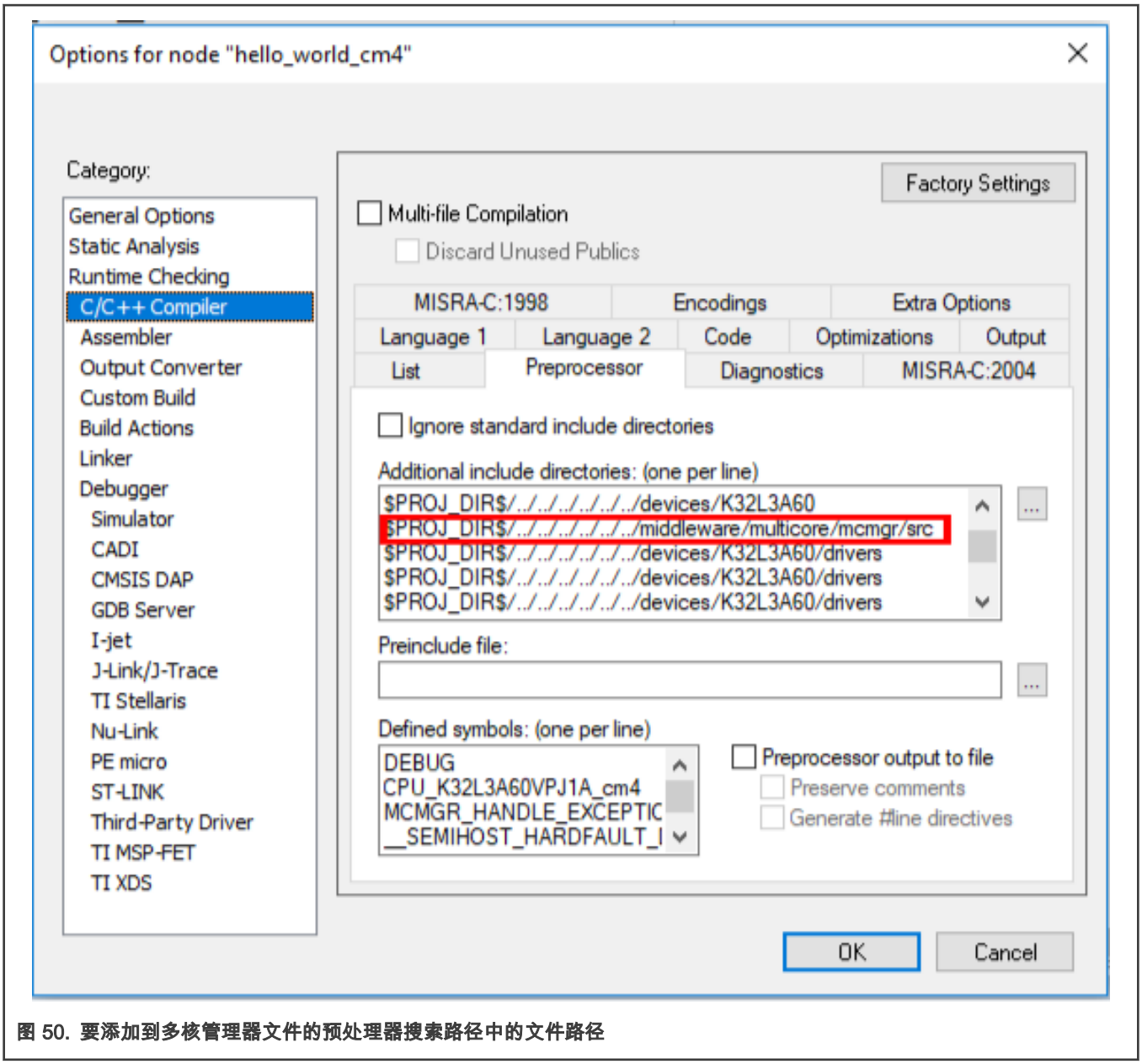

图 50. 要添加到多核管理器文件的预处理器搜索路径中的文件路径

此外，还需要 MU 驱动程序。它们是 `fsl_mu.c` 和 `fsl_mu.h`。

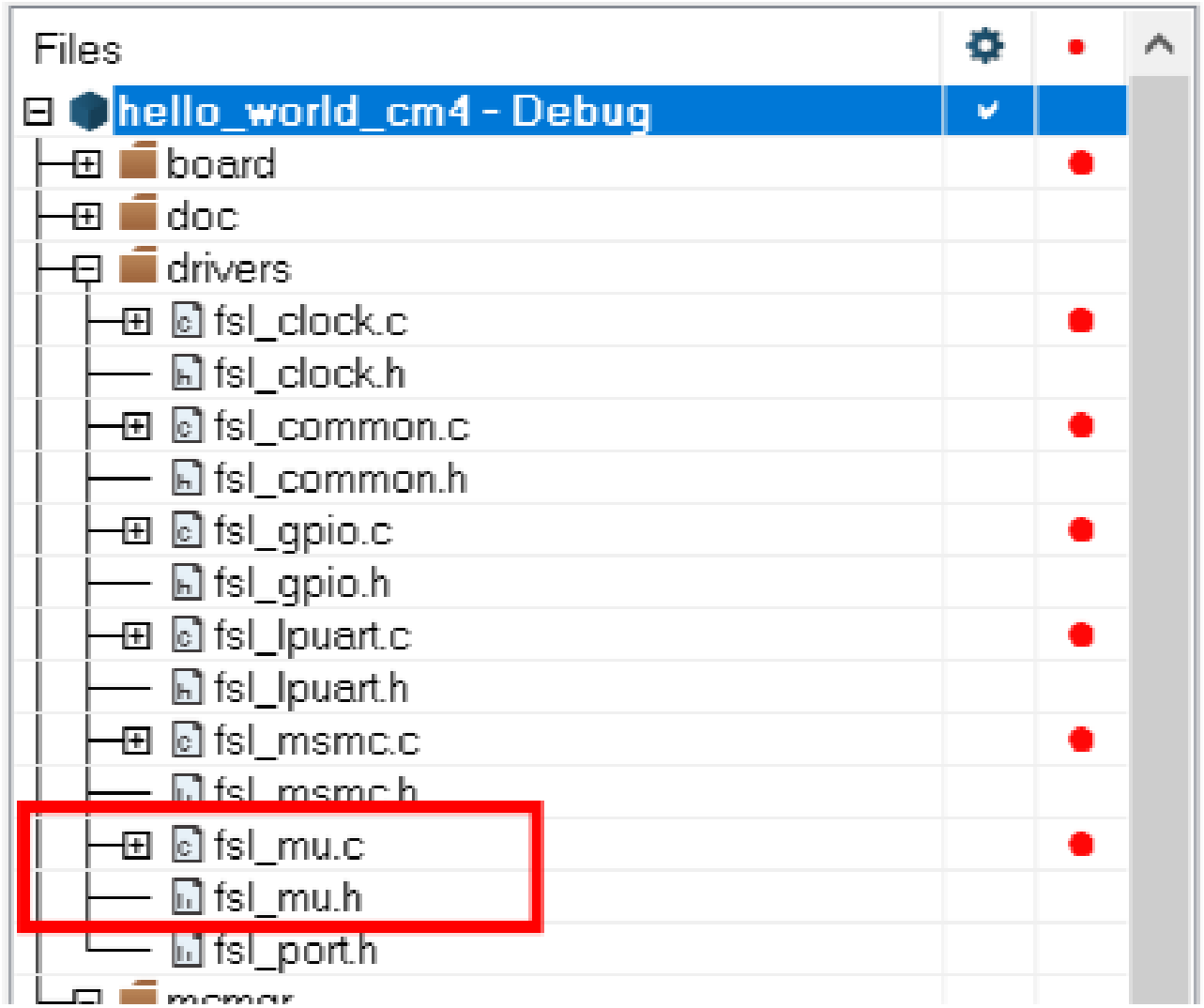

图 51. 将 MU 驱动程序添加到 hello_world 多核项目中

注意

预处理器搜索路径应该包含这些驱动程序文件的文件路径。

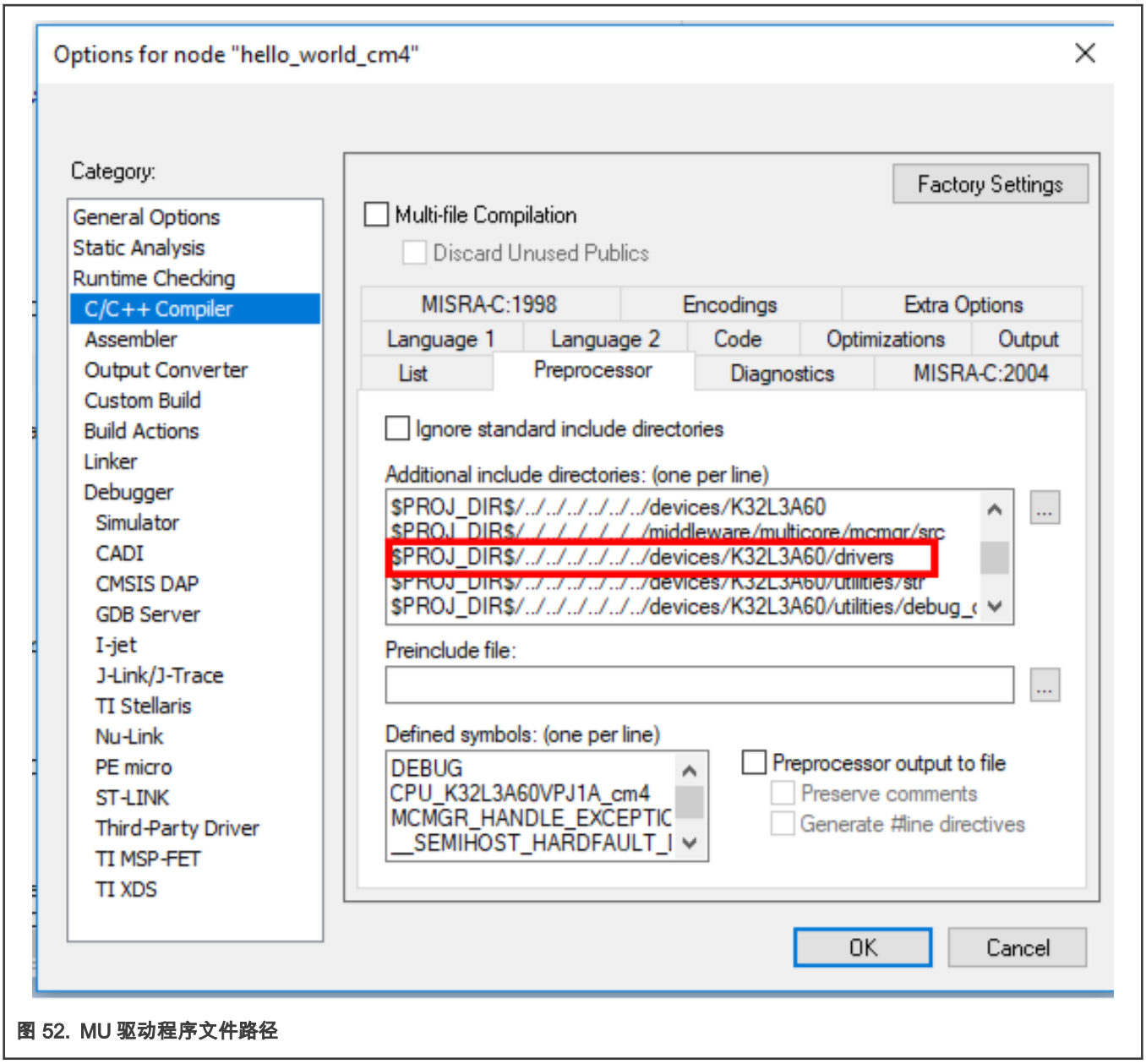

图 52. MU 驱动程序文件路径

要使用 MCMGR 高级驱动程序，需要调用两个初始化函数。 这些调用同时适用于主从项目，如 图 53 所示。

```
int main(void)
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();

    /* Init board hardware.*/
```

图 53. MCMGR 初始化函数调用

`MCMGR_EarlyInit` 函数的调用必须尽可能靠近 reset 条目。此函数启用到 MU 单元的时钟门，并触发传播到对应核的 core up 事件。必须从 main 调用 `MCMGR_Init` 函数，并启用所有其他要用到的 MCMGR API。

一旦调用了这些函数，主软件就应该调用 `MCMGR_StartCore` 函数来启动从核。此函数调用如 图 54 所示。

```
/* Boot Secondary core application */
PRINTF("Starting Secondary core.\r\n");
MCMGR_StartCore(kMCMGR_Core1, CORE1_BOOT_ADDRESS, 5, kMCMGR_Start_Synchronous);
PRINTF("The secondary core application has been started.\r\n");
```

图 54. MCMGR 函数启动从核

`MCMGR_StartCore` 函数需要以下参数：

- mcmgr_core_t coreNum：这应该是 kMCMGR_Core1 或者是 MCMGR_Core0。Core0 是 Arm Cortex-M4 核，Core1 是 Arm Cortex-M0+核。
- `void *bootAddress`：这是指向从核应用程序开始的指针。唯一有效的选项是从核的 flash 空间或 RAM 内存空间的开始。
- `uint32_t startupdata`：这是一些用户/应用程序数据（变量、数组或函数）在启动过程中从主核传递到从核。例如，在 rpmsg 内核间通信的情况下，可以利用它将共享内存基址从主端传递到远程端。
- `mcmgr_start_mode_t` 模式：这是您希望核启动的模式。 有效选项为 `kMCMGR_Start_Synchronous` 或 `kMCMGR_Start_Asynchronous`。如果使用 `kMCMGR_Start_Synchronous`，则主核将启动从核，然后等待从核的事件，然后继续操作。否则，主堆芯将启动从核并立即继续运行。

从核可能只是启动而不执行任何操作，或者检索它传递的任何启动数据。图 55 展示了一个例子。

```
/* Get the startup data */
do
{
    status = MCMGR_GetStartupData(&startupData);
} while (status != kStatus_MCMGR_Success);
```

图 55. 从核实现检索启动数据

从核如何处理这些数据取决于应用程序的具体内容。

8 结论

本应用笔记通过一个简单的示例展示了如何创建、编译和调试多核项目。K32L 系列设备的 SDK 包包含许多多核示例，可帮助您进行项目开发。这些示例包括嵌入式远程过程调用示例、远程处理器消息传递示例、SDK 多核管理器驱动程序的演示以及资源域管理器示例。有关所有这些资源的更多信息，请访问相应资源文件夹中的<KSDK_ROOT>/multicore

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 2020 年 1 月

Document identifier: AN12673

arm