AN12673
Dual-core Project Creation and Conversion for K32L3A6 Devices

Rev. 0 — 01/2020

Application Note

Contents

1 Introduction

The introduction of the K32 L3 device brings to the Kinetis family the power and multitasking capabilities of a dual-core device. Dual-core devices, while powerful, can add complexity to your development. The goal of this application note is to simplify the process of starting a new multi-core project or convert an existing single core project to a dual-core project.

NOTE

Different IDEs handle dual-core applications differently. This application note will cover IAR IDE and MCUXpresso IDEs.

2 Overview

Before creating a dual-core project (or converting a single-core project to a dual-core project), it is necessary to discuss the structure of the K32L3A6 device. The K32L3A6 is an asymmetric dual-core device (with two different cores as opposed to symmetric dual-core devices which have identical cores that operate in lockstep) containing an Arm® Cortex®-M4 core and an Arm Cortex-M0+ core. Like other dual-core devices, one core acts as the primary core or boot core, while the other core acts as the secondary core. In the K32L3A6, the Arm Cortex-M4 device acts as the boot core by default (the boot core can be changed). The block diagram of the K32L3A6 device is as shown in Figure 1.

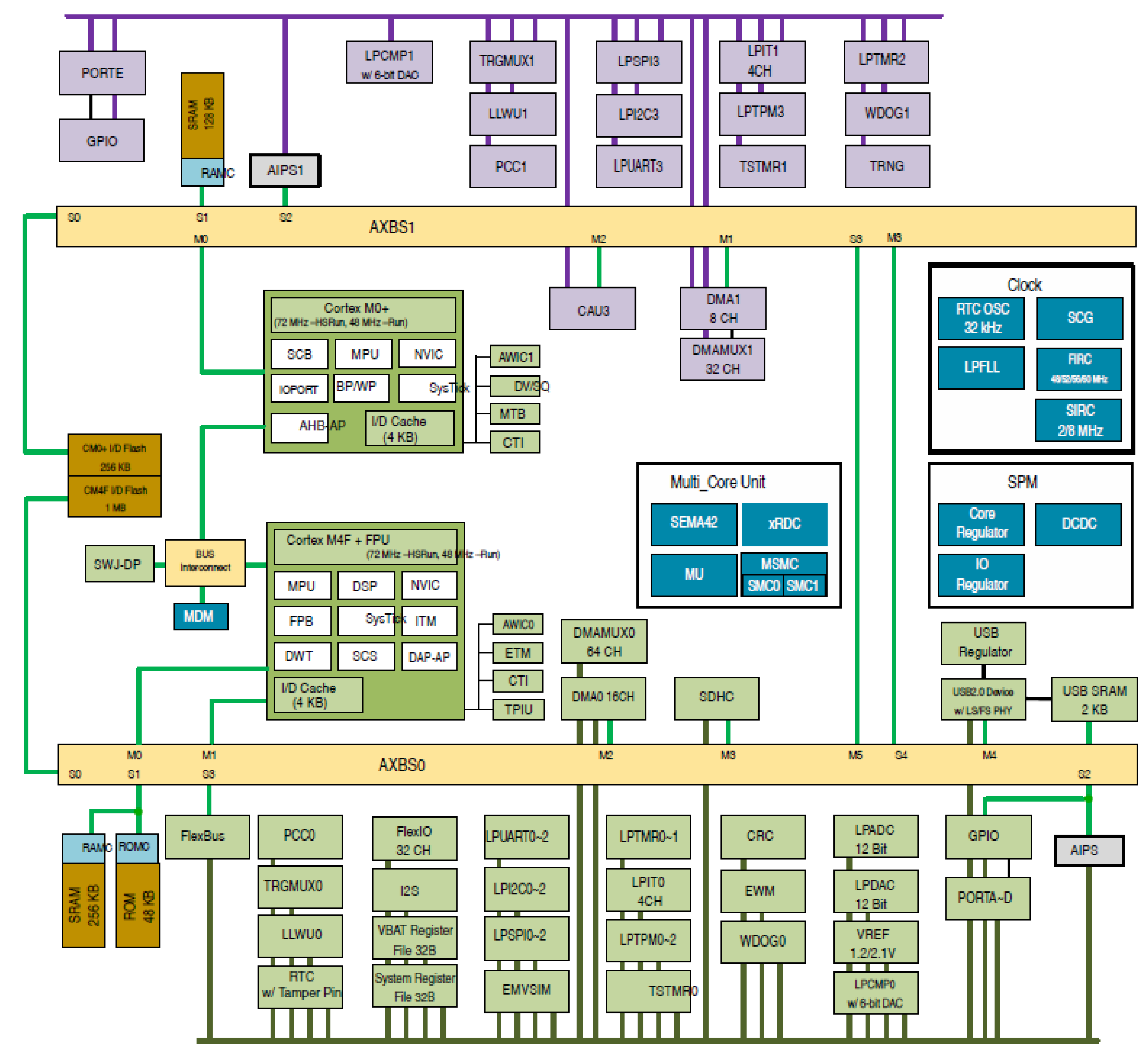


Figure 1. K32L3A6 block diagram

As shown in Figure 1:

- Each core has its own cache, RAM memory, Flash memory, interrupt controllers, and so on. The two cores can operate completely independently of each other.
- Besides the two separate cores, there are two separate crossbars, AXBS0 and AXBS1. This effectively creates a logical divide such that each core also has their own set of peripherals. Those peripherals attached to AXBS0 are the Arm Cortex-M4 peripherals, and those attached to AXBS1 are Arm Cortex-M0+ peripherals. However, the crossbars are also connected to each other through AXBS0 master port 5 (M5) connected to AXBS1 slave port 3 (S3) and AXBS0 slave port 4 (S4) connected to AXBS1 master port 3 (M3). This allows the Arm Cortex-M4 to access Arm Cortex-M0+ peripherals and memory and vice versa.
- Utilities common to both cores include the debug unit, Multi-core units (SEMA42, MU, resource domain controller, and miscellaneous system control module), clock controls (SCG and RTC), and power controls (SPM).

Now that you have a better understanding of how the K32L3A6 device is architected, let's discuss how the IDEs handle this architecture. First, the IAR IDE will be discussed, and then MCUXpresso will be discussed.

3 Multi-core projects in IAR

IAR IDE essentially treats multi-core projects as two independent projects. Thus you need two independent projects (one for each core) when creating a dual-core project. If you are starting from a single-core project, you will need to create (or incorporate) a project for the second core. You will need to modify your code to ensure that the second core is started correctly. You may also need to consider the interaction between the two cores (beyond what is described in this application note) and make more modifications than are described in this application note. The possibilities and options here are limitless and thus, all situations cannot be discussed in this application note.

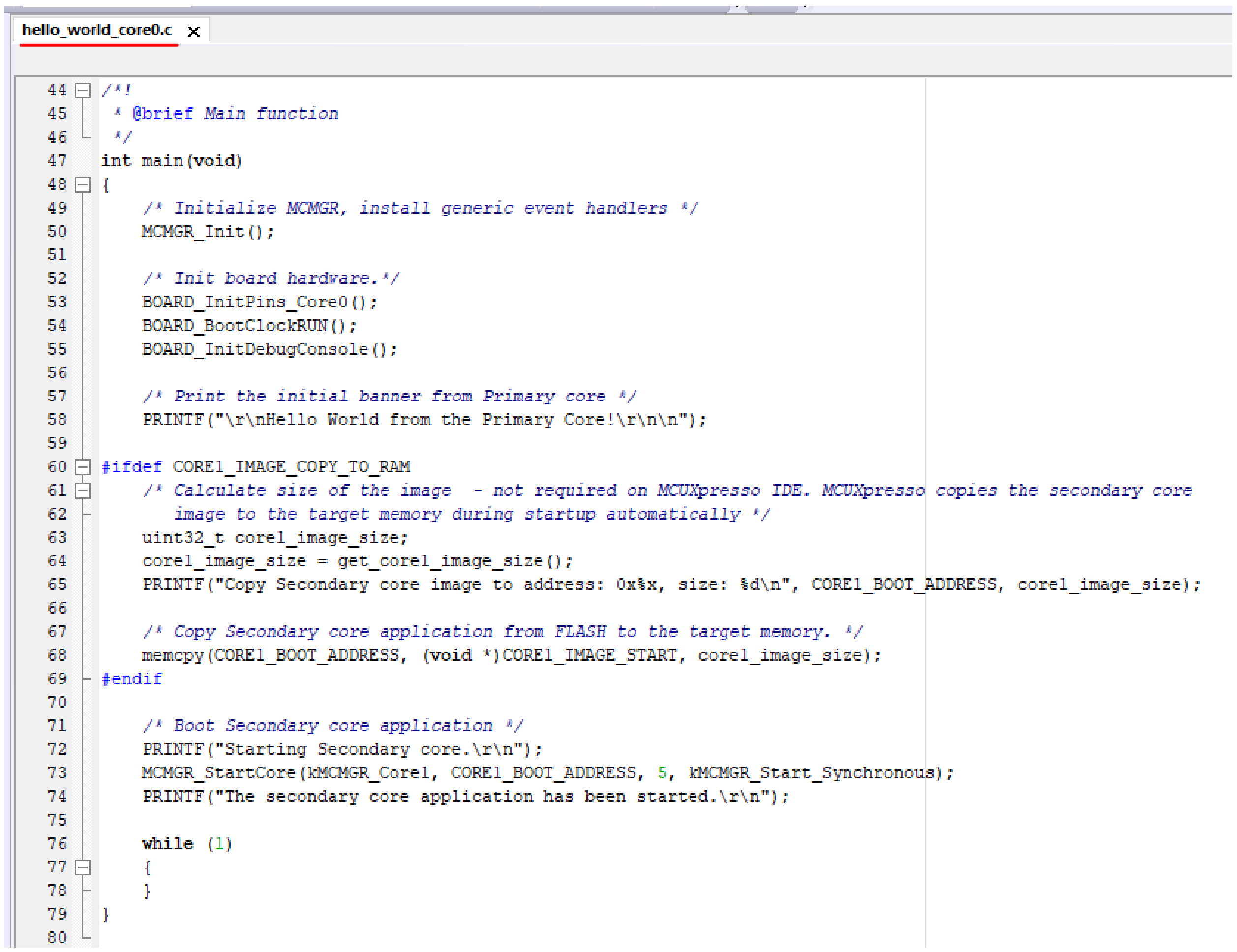
In the IAR projects, there are internal options that will need to be selected to link the two separate projects together. It is important at this point to note that a multi-core project creation does not necessarily mean that the debugger will establish a debug connection with both cores. This is another independent setting. So this section will be broken up into multi-core project creation and multi-core debug.

3.1 Multi-core project creation

As mentioned in Multi-core projects in IAR, IAR IDE treats multi-core project creation as two independent projects just linked by different settings in the projects. Therefore, the first step to multi-core project creation is to create two independent projects for the different cores. The `hello_world` example from the `multicore_examples` folder from the K32L3A6 MCUXpresso SDK package will be used as an example to explain these concepts.

3.1.1 Primary core project

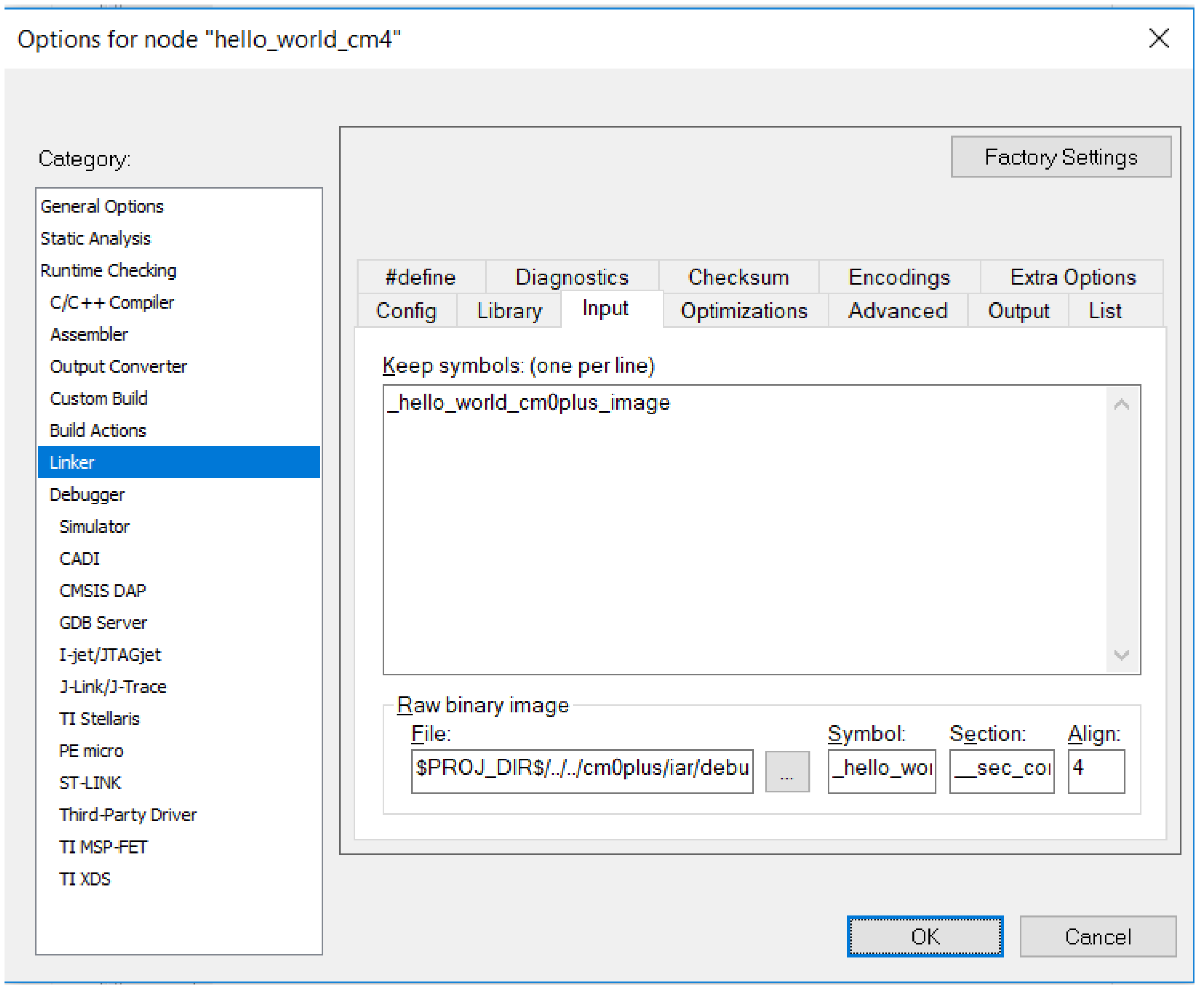
1. Examine the boot core project. In this case, this is the Arm Cortex-M4 project and this project simply configures a terminal and prints `hello_world` to the terminal and notifies the user (via the terminal) that the secondary core is starting. Then the secondary core is started. In dual-core devices, it is generally the responsibility of the boot core to start the secondary core. Figure 2 shows the source code for this (for more details about the source code, see the following chapters in this document).



```
/*!
 * @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)
    {
    }
}
```

Figure 2. Boot core source code

Another responsibility of the boot core project is to program the memory that the secondary core will use for its program code space. This responsibility is achieved by configuring the linker settings in the project correctly as well as configuring the linker file correctly. The linker settings in the project must be configured to point to the correct binary for the secondary core and placed in the correct location in memory. Figure 3 shows the `hello_world` linker settings for the primary core project.



Keep symbols: (one per line)

_hello_world_cm0plus_image

Raw binary image

File: $PROJ_DIR$/../../cm0plus/iar/debu

Symbol: _hello_wo

Section: __sec_co

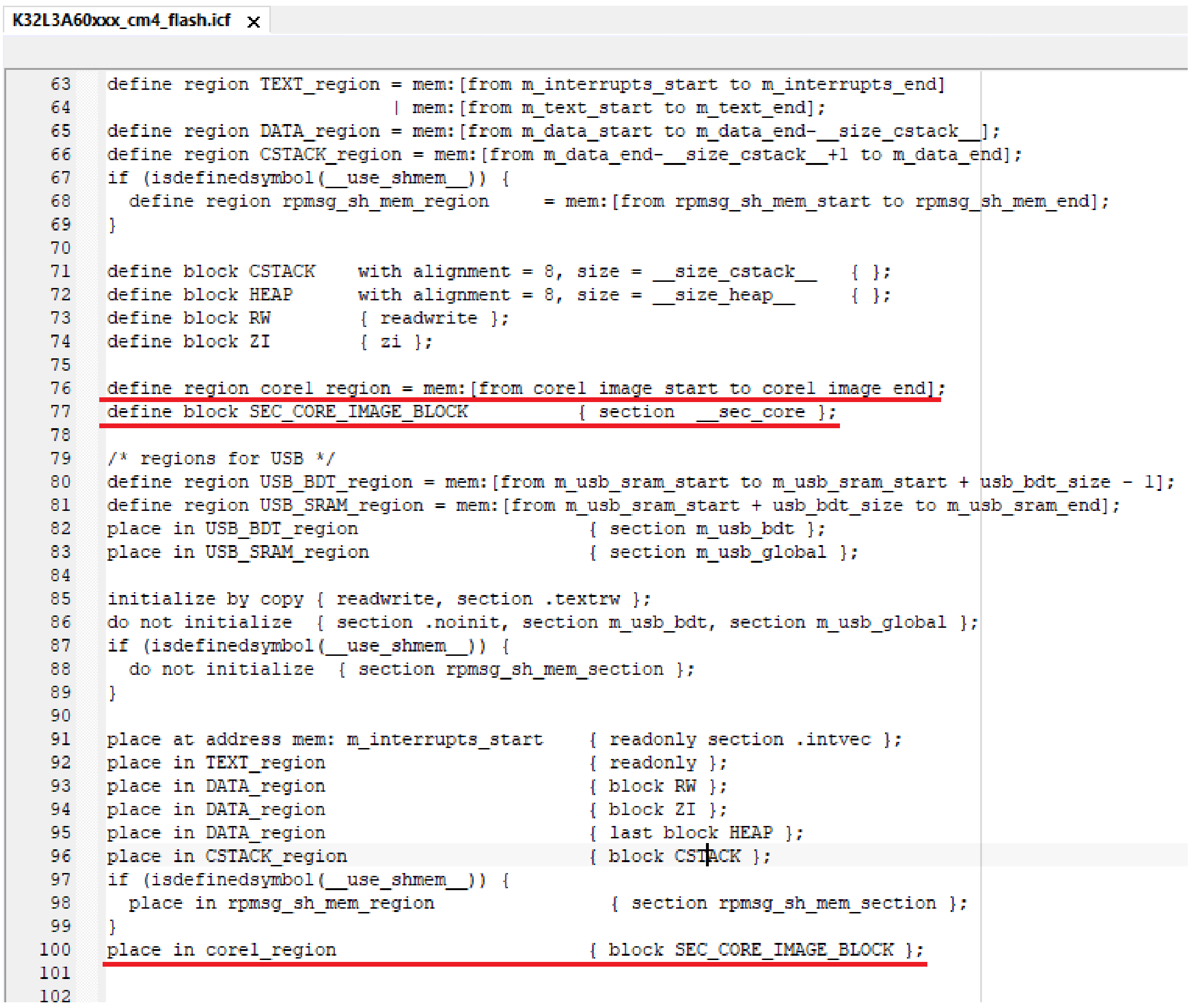
Align: 4

Figure 3. Linker settings for primary core of the dual-core `hello_world` project

All necessary settings to create a dual-core project are contained within the Input tab of the Linker category of the project settings. The following five boxes must be filled in correctly.

- **Keep symbols**: In the Keep symbols text box, the additional binaries that need to be added to the output file (that is generated when the compile button is clicked) are listed. In this case, we have only one additional image to be added to the final output. It is named as `_hello_world_cm0plus_image`. This symbol name is arbitrary and can be anything.
- **File**: In the File text box, the correct path to the binary image for the Arm Cortex-M0+ program should be included (`*.bin`).
- **Symbol**: The Symbol field must match the symbol defined in the Keep Symbols text box.
- **Section**: This defines where to place the binary file to be included. This MUST be correctly defined in the linker file for the binary to be programmed and placed correctly.
- **Align**: This defines the alignment of the binary. This determines whether the data is aligned by byte, half-word, or word. This example word aligns the binary and you shouldn't ever need anything other than word alignment as shown.

2. Make sure that the linker file is properly written. The linker file must properly define the section for the secondary core binary to be placed. Figure 4 shows the linker file used in the `hello_world` project and the important parts have been underlined in red.

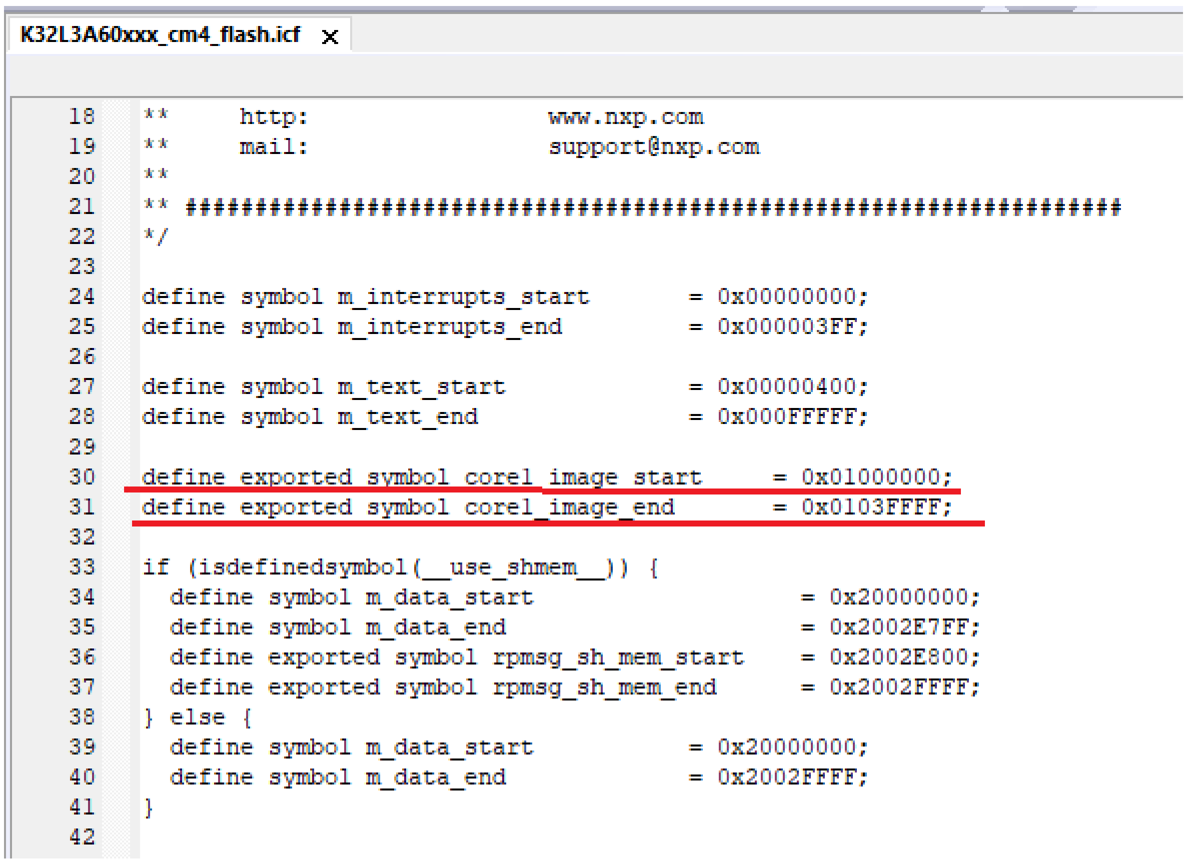


K32L3A60xxx_cm4_flash.icf

```
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 };
```

Figure 4. Bottom of linker file

At the bottom of this linker file, there is the command of `place in core1_region {block SEC_CORE_IMAGE_BLOCK };`. It commands the linker to place whatever is assigned to the `SEC_CORE_IMAGE_BLOCK` to the memory defined by `core1_region`. The block, `SEC_CORE_IMAGE_BLOCK`, is defined as `section __sec_core`, which means anything in the application with that tag will be placed in that block. In this case, the only thing with that tag will be the image for the secondary core (and that was done in the **Linker->Input** tab in the project settings). The region `core1_region` is defined as a memory region from `core1_image_start` to core1_image_end. These bounds are defined at the top of the linker file (as shown in Figure 5).


K32L3A60xxx_cm4_flash.icf

```
**     http:                     www.nxp.com
**     mail:                     support@nxp.com
**
** ################################################################
*/

define symbol m_interrupts_start       = 0x00000000;
define symbol m_interrupts_end         = 0x000003FF;

define symbol m_text_start             = 0x00000400;
define symbol m_text_end               = 0x000FFFFF;

define exported symbol core1_image_start     = 0x01000000;
define exported symbol core1_image_end       = 0x0103FFFF;

if (isdefinedsymbol(__use_shmem__)) {
  define symbol m_data_start                    = 0x20000000;
  define symbol m_data_end                      = 0x2002E7FF;
  define exported symbol rpmsg_sh_mem_start     = 0x2002E800;
  define exported symbol rpmsg_sh_mem_end       = 0x2002FFFF;
} else {
  define symbol m_data_start           = 0x20000000;
  define symbol m_data_end             = 0x2002FFFF;
}
```

Figure 5. Top of Linker file

At the top of the linker file, variables `core1_image_start` to `core1_image_end` are defined as `0x01000000` and `0x0103FFFF` respectively. Therefore, the image will be placed in the Arm Cortex-M0+ flash space.

NOTE

The image range could be changed to be the Arm Cortex-M0+ RAM space. However, this would mean that the Arm Cortex-M4 would also have to run from RAM, as the image loading tools cannot switch between loading Flash or loading RAM. The image loader utility either loads RAM, or loads Flash.

3.1.2 Secondary core project

The secondary core project should be compiled as a normal project. It is important to use the correct flash loader for the expected configuration. If the secondary core is expected to run from flash, it should be linked using a flash linker. If it is expected to run from RAM, a RAM-configured linker should be used.

NOTE

Keep the **Keep symbols** and **Raw binary image** sections blank (as shown in Figure 6).

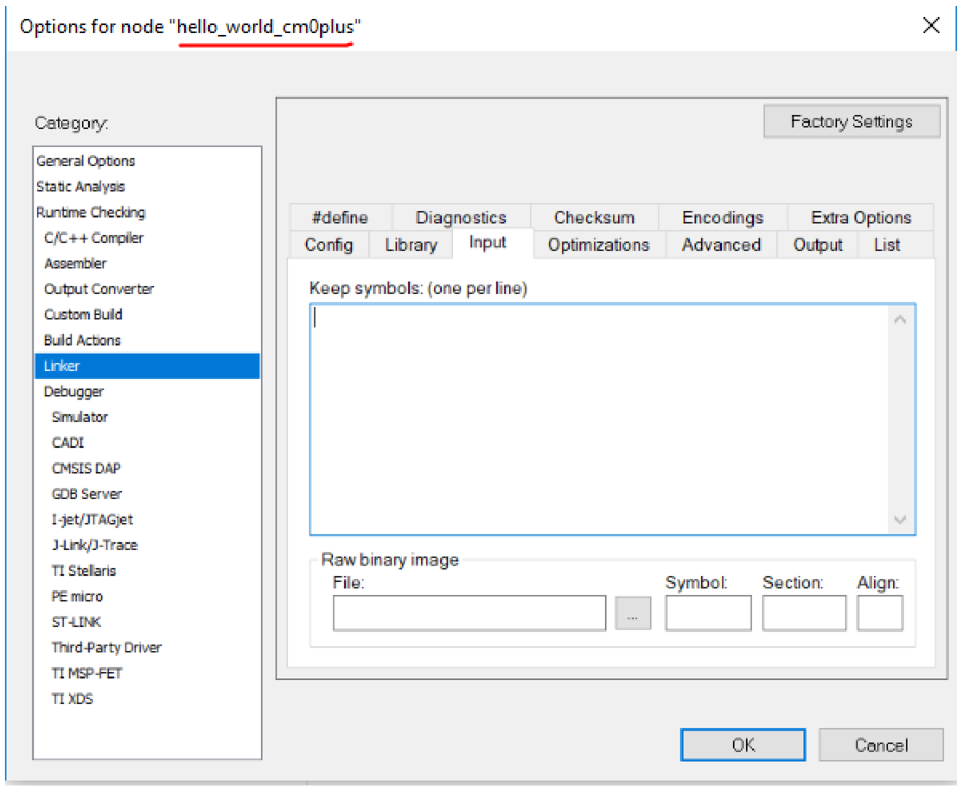


Figure 6. Secondary Core Linker Input settings configuration

4 Multicore debug in IAR

Multi-core debugging generally refers to the act of debugging two cores simultaneously.

NOTE

It is possible to debug just the primary core. In this type of situation (assuming the project is a dual-core project), the secondary core will still be programmed and will still run, but you will only have control over the primary core. This is sometimes easier and simpler than true multi-core debugging. As such, you may find this more convenient (depending on your goals). This section will focus on multi-core debugging.

As with multi-core project creation, debugging two cores simultaneously requires that both projects have certain settings configured. Incorrect settings can (and usually) result in failed connection attempts.

The following information will be discussed:

1. Compiling a Multi-core project in IAR
2. Primary core project debug settings
3. Secondary project debug settings
4. Debugging

This example examines the multicore `hello_world` project from the FRDM-K32L3A6 SDK package. The `ijet` debug probe is used in this example but the content is still valid for other debug probes.

NOTE

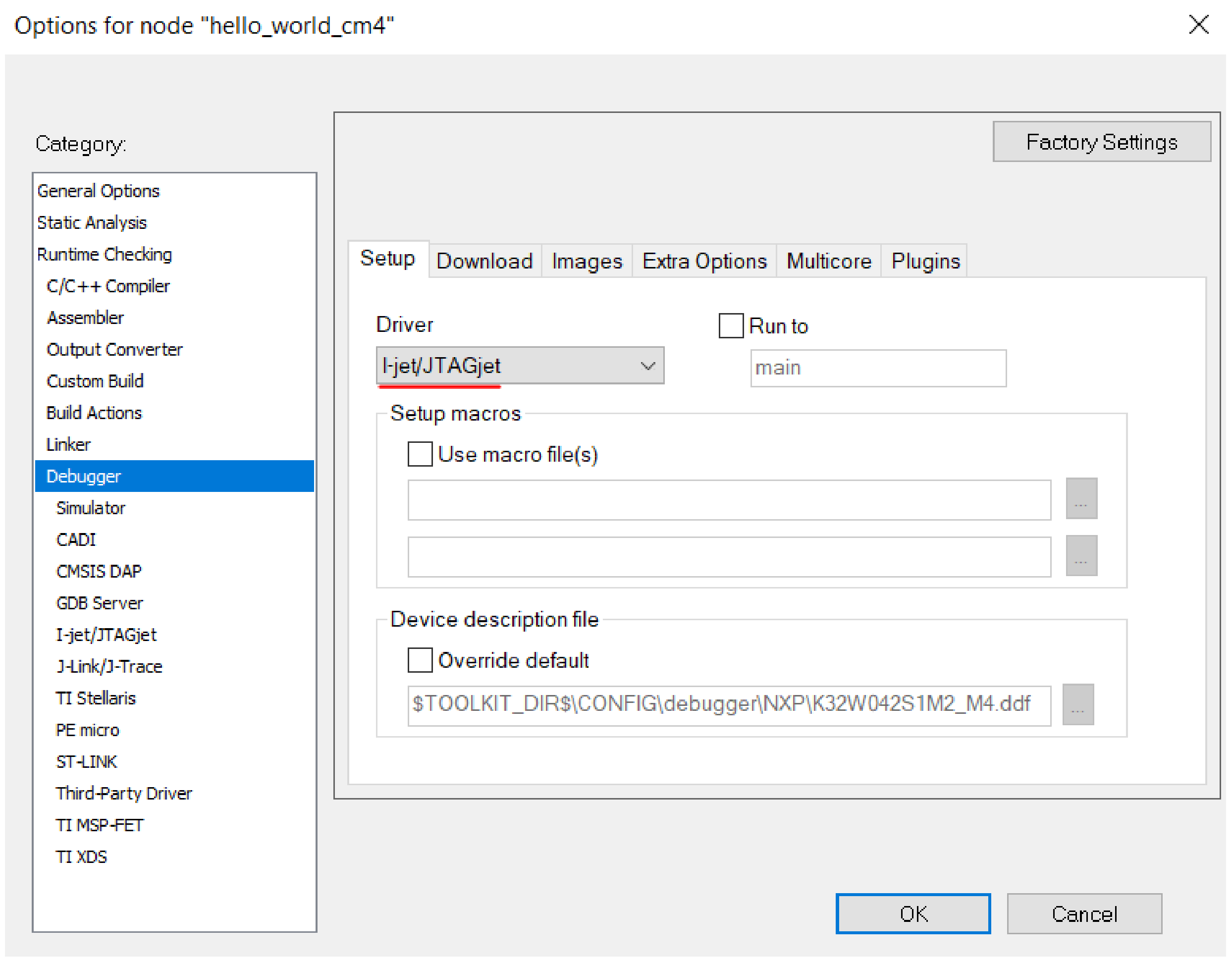
At the time of writing this application note, IAR does not support dual-core debug with the JLink debug probe. Be sure to check if your version of IAR supports dual-core debug with your preferred debug probe.

4.1 Compiling a Multi-core project in IAR

Master and slave projects are compiled separately in IAR, and as such, normal procedures should be followed to compile these projects. However, the order does matter in this case. Because the master project is expected to program the slave's memory space as well, it follows that the slave project should be compiled first, so that the master project is able to link in the binary for that project.

4.2 Primary core project debug settings

1. Make sure that the `ijet` (or supported debug probe of your choice) is selected in the Debugger category.



Options for node "hello_world_cm4"

Category: General Options, Static Analysis, Runtime Checking, C/C++ Compiler, Assembler, Output Converter, Custom Build, Build Actions, Linker, Debugger (selected), Simulator, CADI, CMSIS DAP, GDB Server, I-jet/JTAGjet, J-Link/J-Trace, TI Stellaris, PE micro, ST-LINK, Third-Party Driver, TI MSP-FET, TI XDS

Tabs: Setup, Download, Images, Extra Options, Multicore, Plugins

Driver: I-jet/JTAGjet

Run to: [ ] main

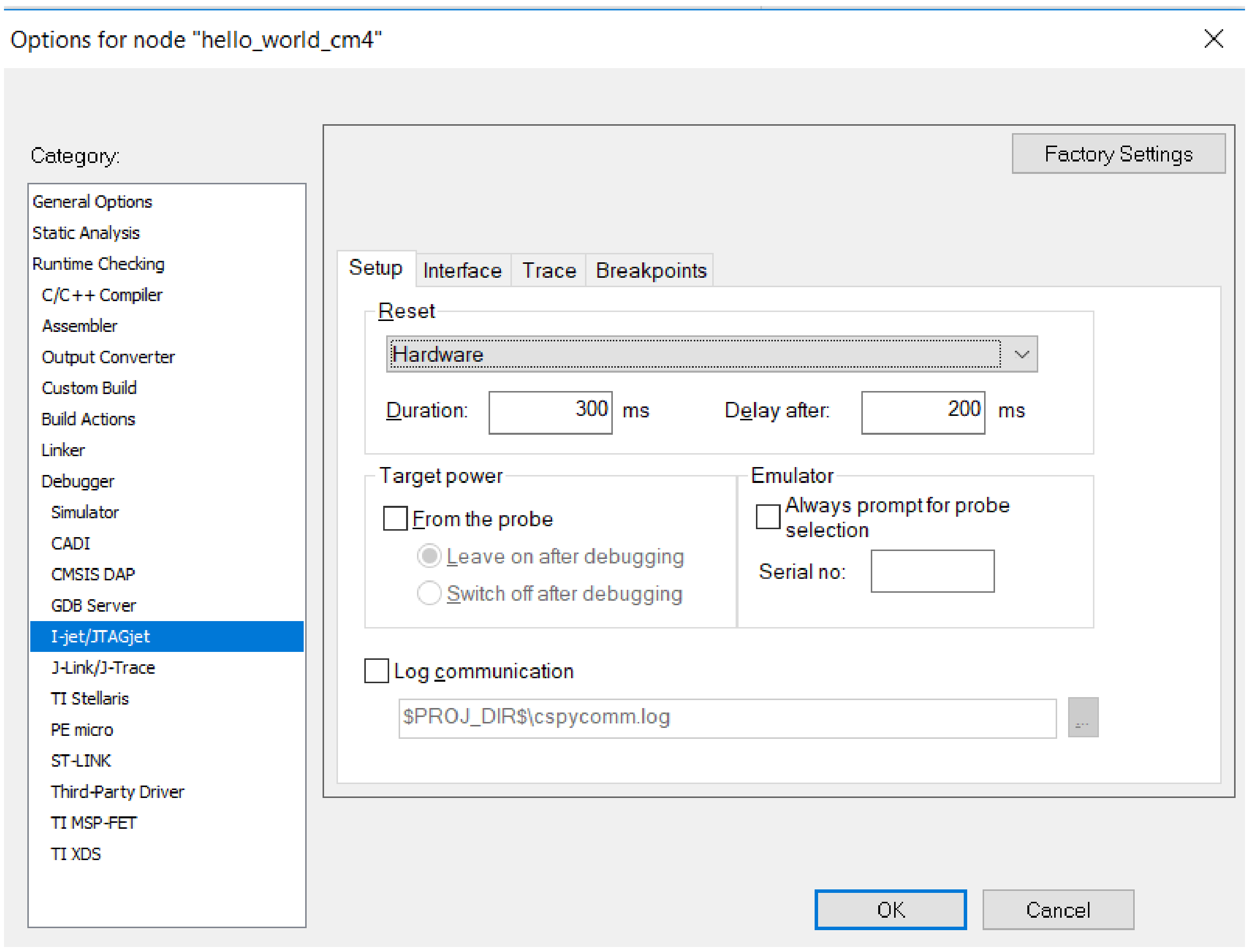
Setup macros: [ ] Use macro file(s)

Device description file: [ ] Override default

$TOOLKIT_DIR$\CONFIG\debugger\NXP\K32W042S1M2_M4.ddf

Figure 7. Debug probe selection

Let's continue with the **Setup** tab.



Options for node "hello_world_cm4"

Reset: Hardware

Duration: 300 ms

Delay after: 200 ms

Target power

☐ From the probe

Leave on after debugging

Switch off after debugging

Emulator

☐ Always prompt for probe selection

Serial no:

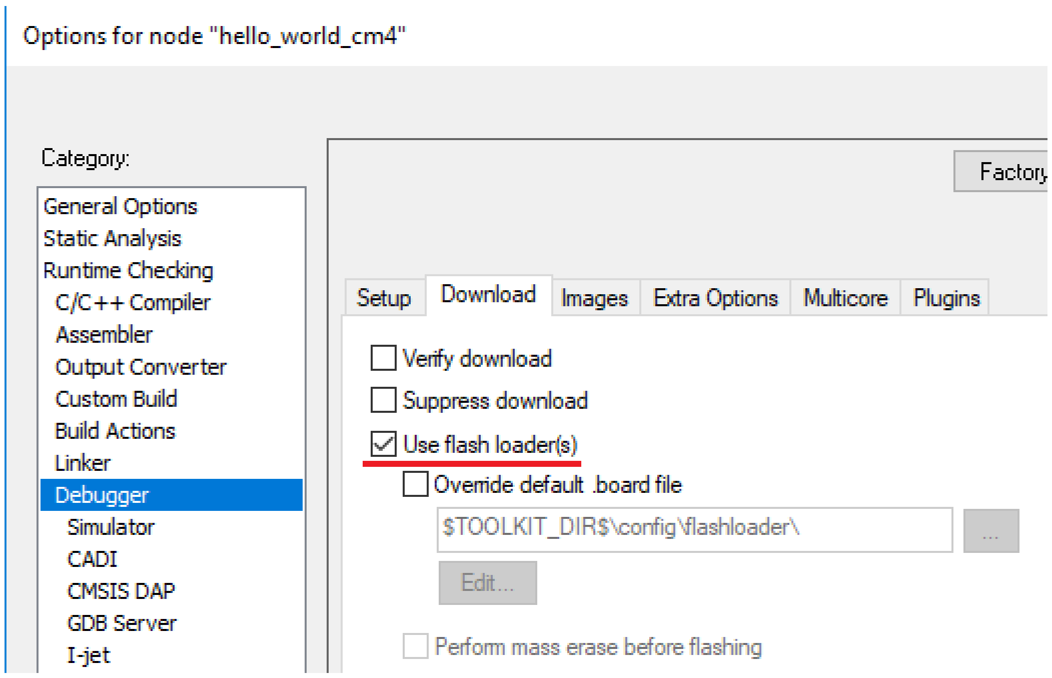
☐ Log communication

$PROJ_DIR$\cspycomm.log

Figure 8. Setup of the primary core debugger

The reset method in this example is to use the Hardware reset, or reset pin, but this can be any of the reset options since it is the primary, or master, core.

2. Let's look at the **Download** options of the Debugger configuration. The `hello_world` example is configured to download to flash. Figure 9 shows the primary core configuration (must use a flash loader if downloading to flash).


Options for node "hello_world_cm4"

Category:

- General Options
- Static Analysis
- Runtime Checking
- C/C++ Compiler
- Assembler
- Output Converter
- Custom Build
- Build Actions
- Linker
- Debugger
- Simulator
- CADI
- CMSIS DAP
- GDB Server
- I-jet

- [ ] Verify download
- [ ] Suppress download
- [x] Use flash loader(s)
- [ ] Override default .board file

$TOOLKIT_DIR$\config\flashloader\

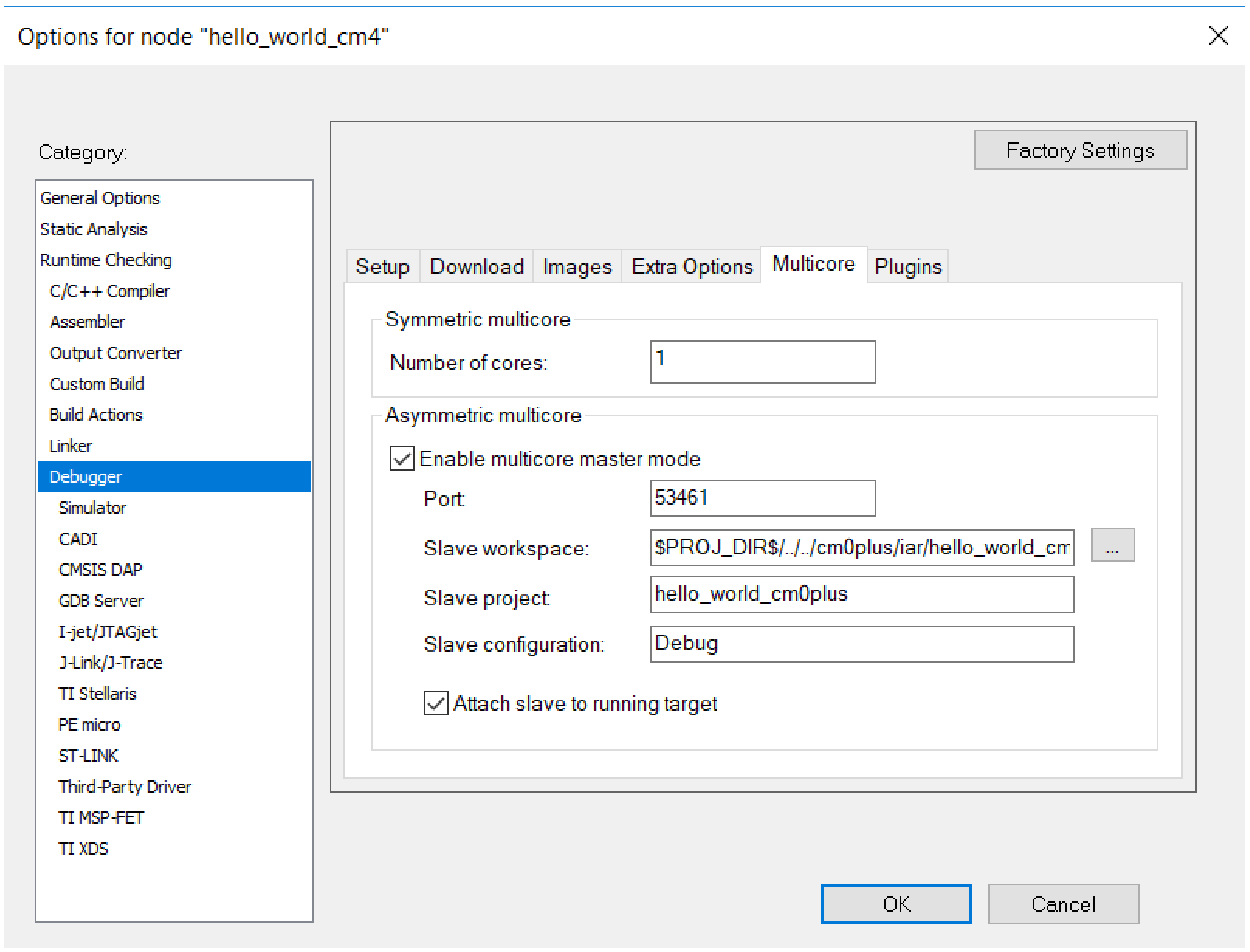
- [ ] Perform mass erase before flashing

Figure 9. Download settings for primary core project

NOTE

If a RAM project is desired, this check-box should be left unchecked.

3. Examine the **Multicore** tab. This is arguably the most important tab. As with the linker options, this tab requires knowledge of the slave (or secondary core) workspace. Figure 10 displays the **Multicore** tab.



Symmetric multicore

Number of cores: 1

Asymmetric multicore

☑ Enable multicore master mode

Port: 53461

Slave workspace: $PROJ_DIR$/../../cm0plus/iar/hello_world_cm

Slave project: hello_world_cm0plus

Slave configuration: Debug

☑ Attach slave to running target

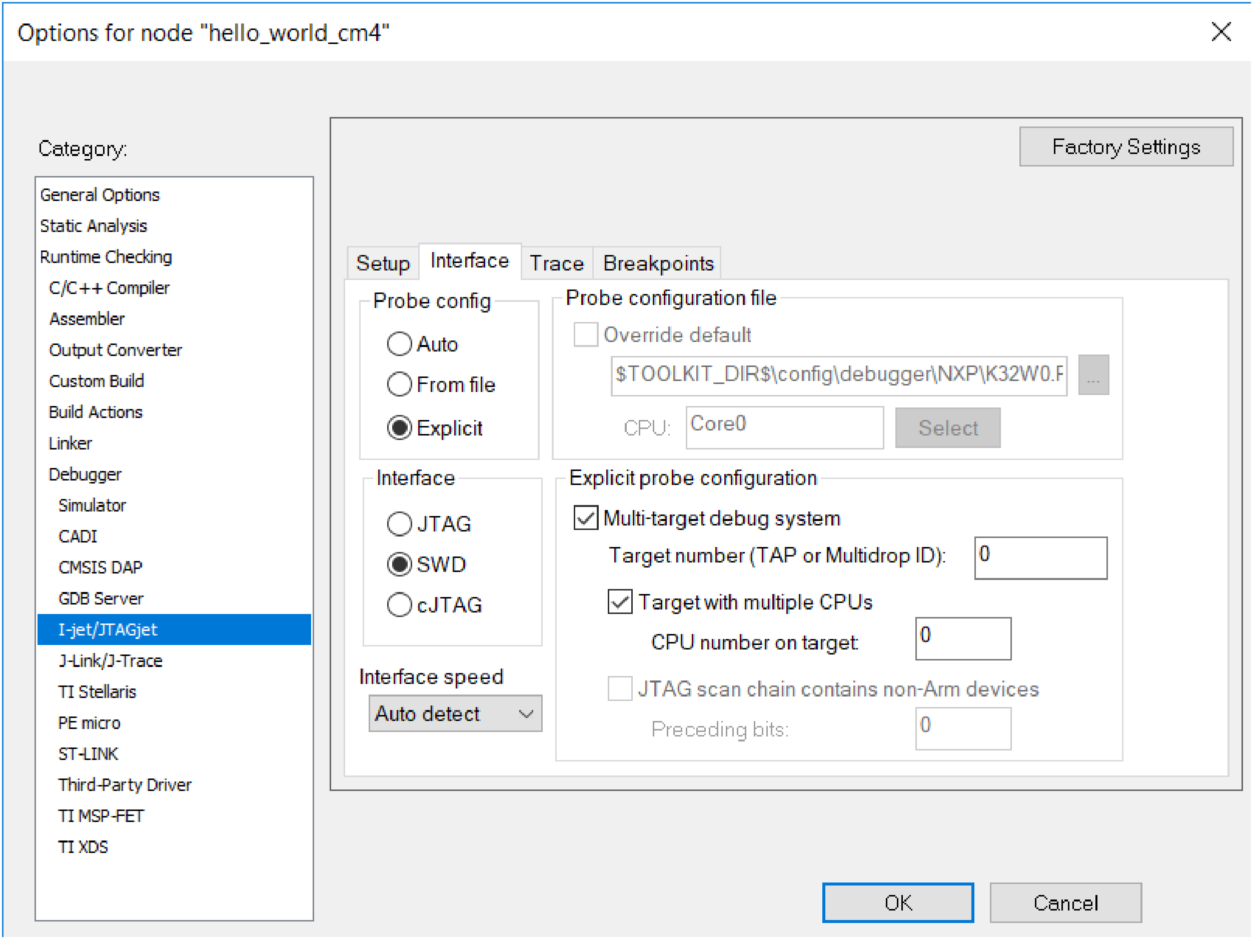
Figure 10. Debugger settings for the Multicore tab (primary core)

Only the Asymmetric multicore settings are necessary as K32L3A6 is an asymmetric device.

NOTE

The path to the slave workspace in the debugger tab is the same as in the project configuration (`$PROJ_DIR$/../../cm0plus/iar/hello_world_cm0plus.eww`).

4. Make sure that the debugger interface is setup correctly.


Options for node "hello_world_cm4"

Category: I-jet/JTAGjet

Interface tab:

Probe config: ( ) Auto ( ) From file (● Explicit)

Probe configuration file: [ ] Override default; $TOOLKIT_DIR$\config\debugger\NXP\K32W0.F; CPU: Core0

Interface: ( ) JTAG (● SWD) ( ) cJTAG

Interface speed: Auto detect

Explicit probe configuration:
[✓] Multi-target debug system
Target number (TAP or Multidrop ID): 0
[✓] Target with multiple CPUs
CPU number on target: 0
[ ] JTAG scan chain contains non-Arm devices
Preceding bits: 0

Figure 11. Debugger interface configuration for a dual-core project

The important sections to focus on in the interface tab are the Probe config section. The probe configuration can be determined automatically, from a user defined file, or by explicit selections in this dialog box. This example will focus on the explicit method. It is important that the Explicit probe configuration (if selected) targets the correct CPU number. Since this project targets the Arm Cortex-M4 core, the CPU number should be **0** as it is the first CPU in the debug chain.

If a file is used, the file should have this information and the correct core must be selected in the CPU field. However, using the settings shown above, no problems should be encountered.

NOTE

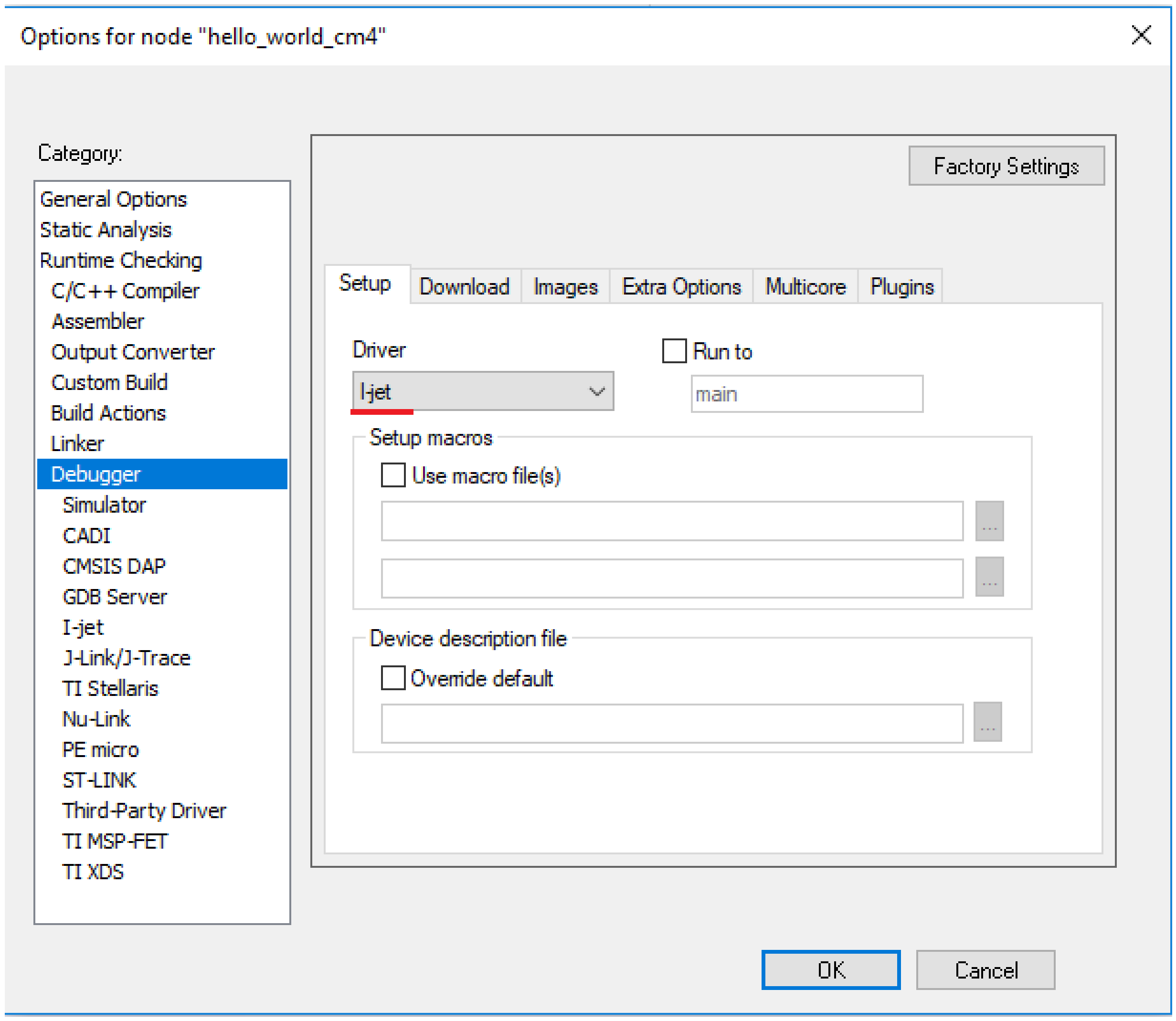
Another important note here is that it is crucial that the Interface used (SWD is shown) matches the slave project.

4.3 Secondary project debug settings

The secondary project's debug settings are largely determined by the master's settings. However there is no automatic link between the two projects. So it is imperative that these settings are manually configured and correct before initiating a debug session. Not doing so usually results in failed connection attempts, but can also lead to other erratic behavior of the IAR IDE (sometimes causing the program to crash).

Check the following settings:

- Check whether the driver used matches the master project. This is found in the **Setup** tab of the Debugger category.



Options for node "hello_world_cm4"

Category:

- General Options
- Static Analysis
- Runtime Checking
- C/C++ Compiler
- Assembler
- Output Converter
- Custom Build
- Build Actions
- Linker
- Debugger
- Simulator
- CADI
- CMSIS DAP
- GDB Server
- I-jet
- J-Link/J-Trace
- TI Stellaris
- Nu-Link
- PE micro
- ST-LINK
- Third-Party Driver
- TI MSP-FET
- TI XDS

Driver: I-jet

☐ Run to: main

Setup macros

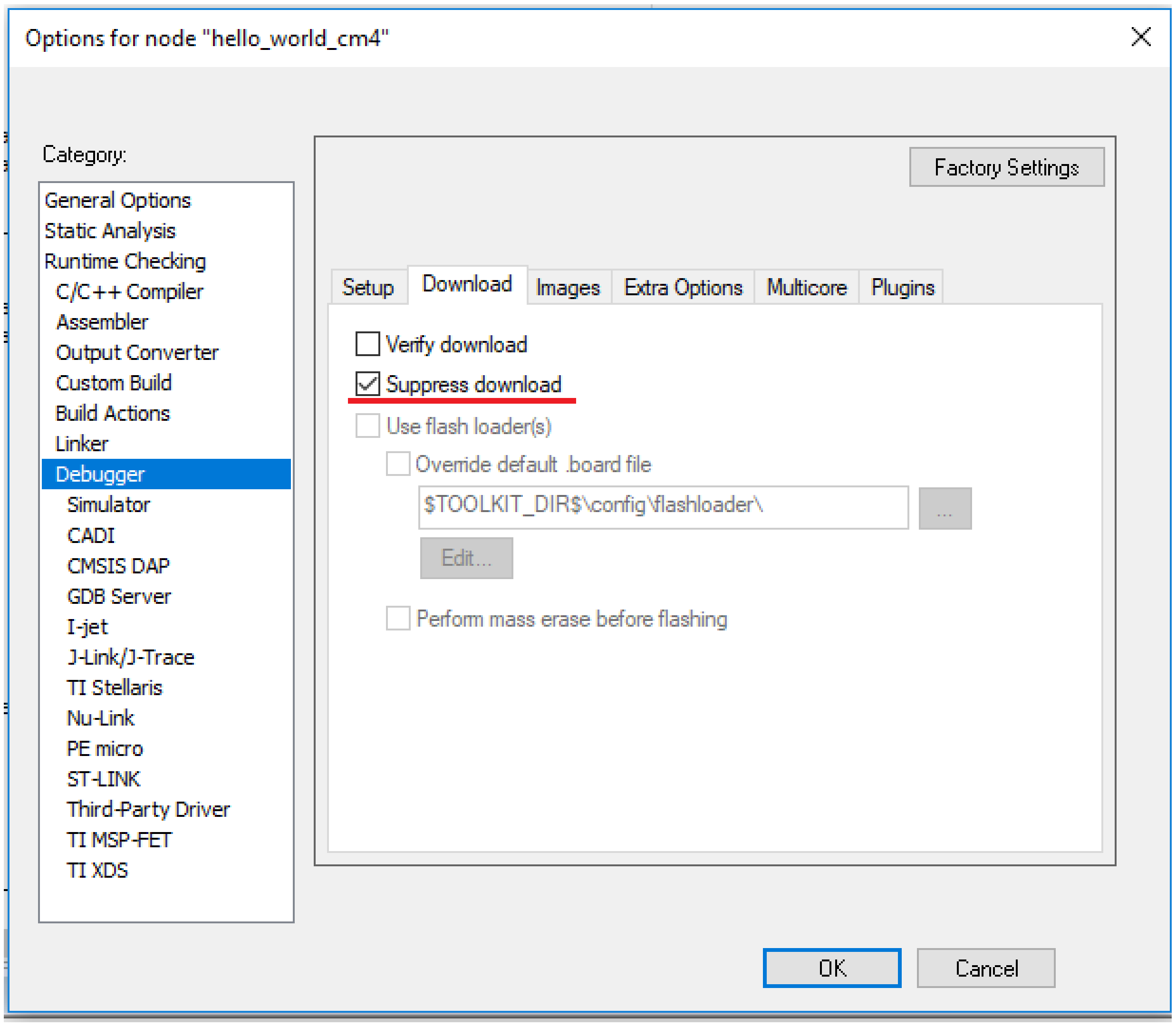
☐ Use macro file(s)

Device description file

☐ Override default

Figure 12. Debugger driver selection in slave project

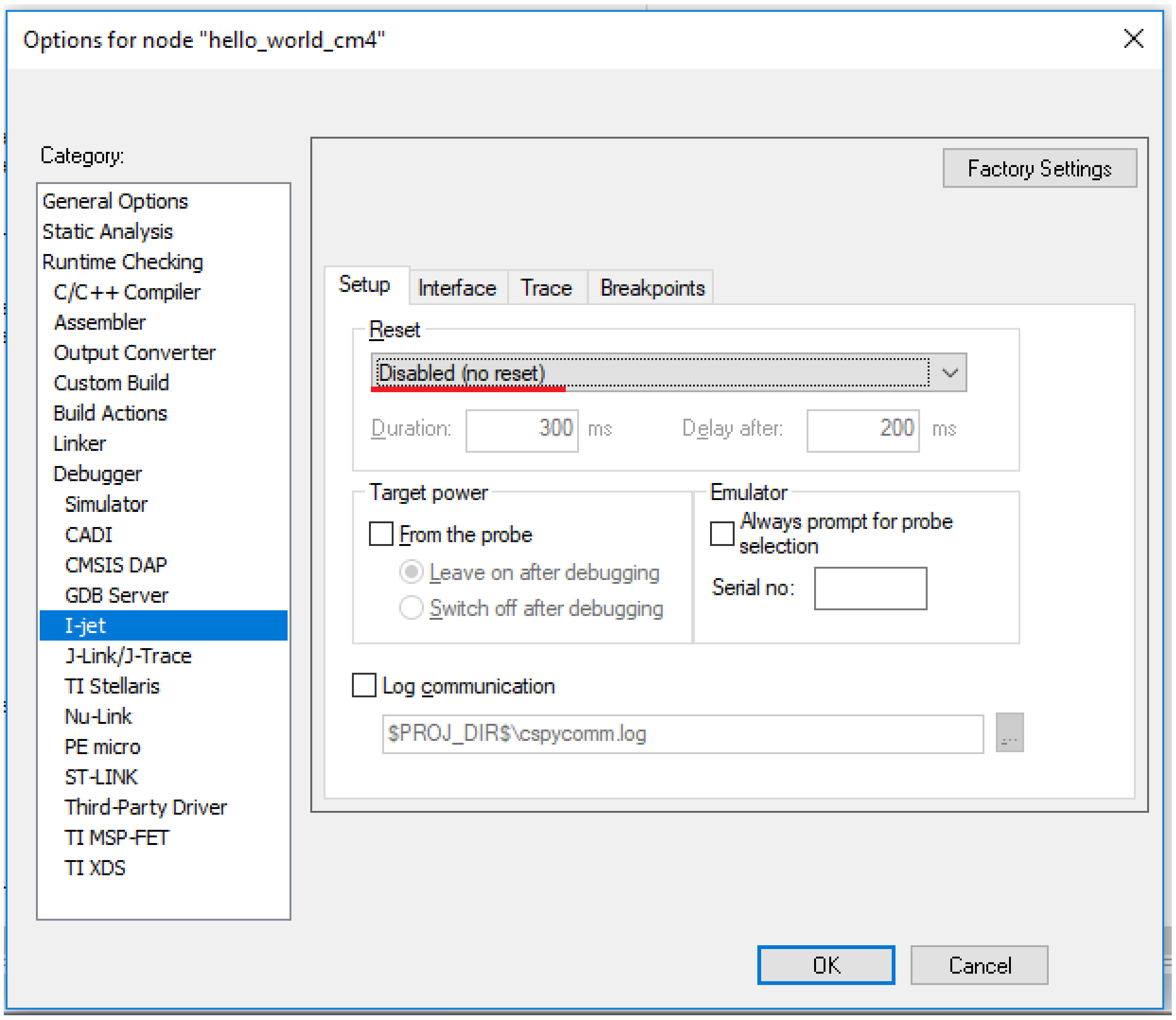
- Check the **Download** setting. Since the master project should take care of flashing the device to be debugged, the slave project should not try to download the code. This operation could result in errors. Figure 13 displays the slave project's download settings.



☐ Verify download
☑ Suppress download
☐ Use flash loader(s)
☐ Override default .board file
$TOOLKIT_DIR$\config\flashloader\
☐ Perform mass erase before flashing

Figure 13. Slave project download settings

Similarly, the slave project should not try to reset the device when initializing. This could also result in errors or erratic behavior. So the slave project needs to suppress resets. This is accomplished by setting the correct reset option in the Setup tab of the appropriate debugger. In this case, this is found in the ijet/JTAGjet category.


Options for node "hello_world_cm4"

Category:

- General Options
- Static Analysis
- Runtime Checking
- C/C++ Compiler
- Assembler
- Output Converter
- Custom Build
- Build Actions
- Linker
- Debugger
- Simulator
- CADI
- CMSIS DAP
- GDB Server
- I-jet
- J-Link/J-Trace
- TI Stellaris
- Nu-Link
- PE micro
- ST-LINK
- Third-Party Driver
- TI MSP-FET
- TI XDS

Factory Settings

Setup | Interface | Trace | Breakpoints

Reset: Disabled (no reset)

Duration: 300 ms   Delay after: 200 ms

Target power
- [ ] From the probe
- Leave on after debugging
- Switch off after debugging

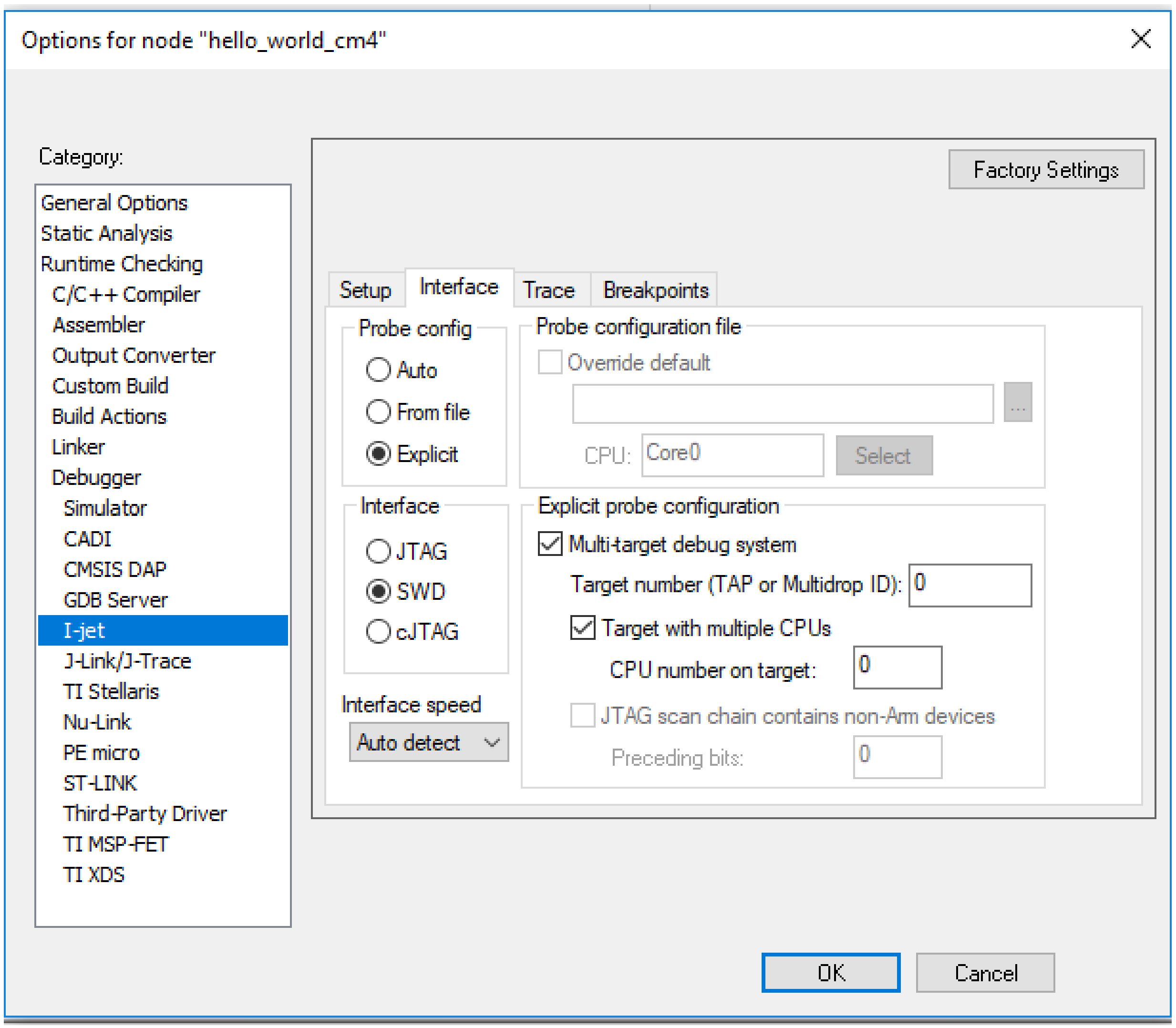
Emulator
- [ ] Always prompt for probe selection
- Serial no:

- [ ] Log communication

$PROJ_DIR$\cspycomm.log

Figure 14. Slave project debugger reset selection

- Check whether the **Interface** settings of the debugger of the slave project match those of the master project.


Options for node "hello_world_cm4"

Category:

General Options
Static Analysis
Runtime Checking
C/C++ Compiler
Assembler
Output Converter
Custom Build
Build Actions
Linker
Debugger
Simulator
CADI
CMSIS DAP
GDB Server
I-jet
J-Link/J-Trace
TI Stellaris
Nu-Link
PE micro
ST-LINK
Third-Party Driver
TI MSP-FET
TI XDS

Setup | Interface | Trace | Breakpoints

Probe config: Auto [✓ Explicit] From file

Probe configuration file: Override default (unchecked); CPU: Core0

Interface: JTAG [✓ SWD] cJTAG

Interface speed: Auto detect

Explicit probe configuration:
- ✓ Multi-target debug system
- Target number (TAP or Multidrop ID): 0
- ✓ Target with multiple CPUs
- CPU number on target: 0
- JTAG scan chain contains non-Arm devices (unchecked)
- Preceding bits: 0

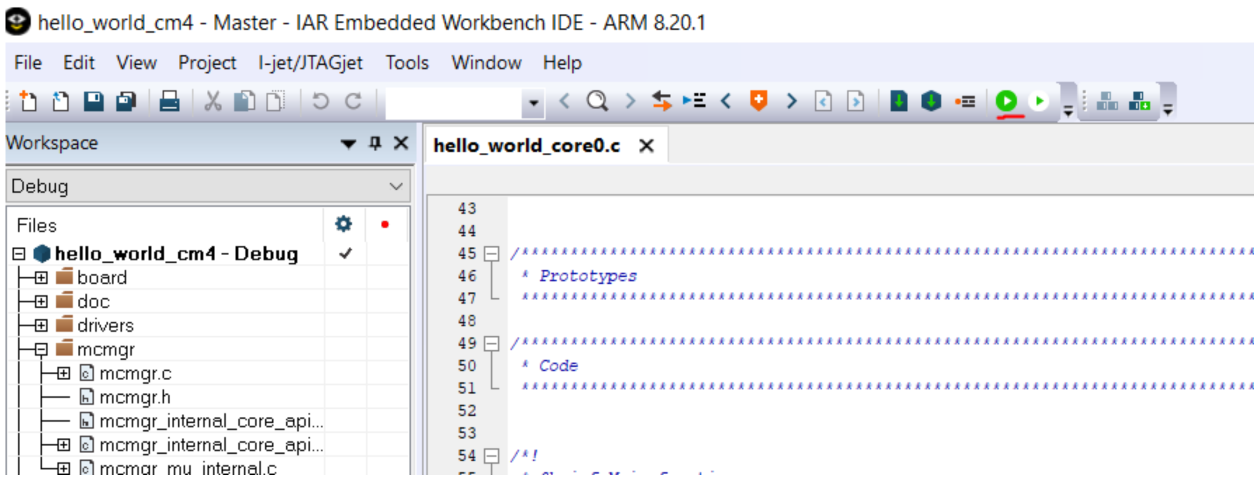
Figure 15. Slave project debugger interface settings

4.4 Debugging

1. Make sure that the projects are compiled.

   Since the master project includes a binary from the slave project, the slave project must be successfully compiled first. Only then can the master project be compiled. Once both projects are compiled properly, close the slave project and leave the master project open.

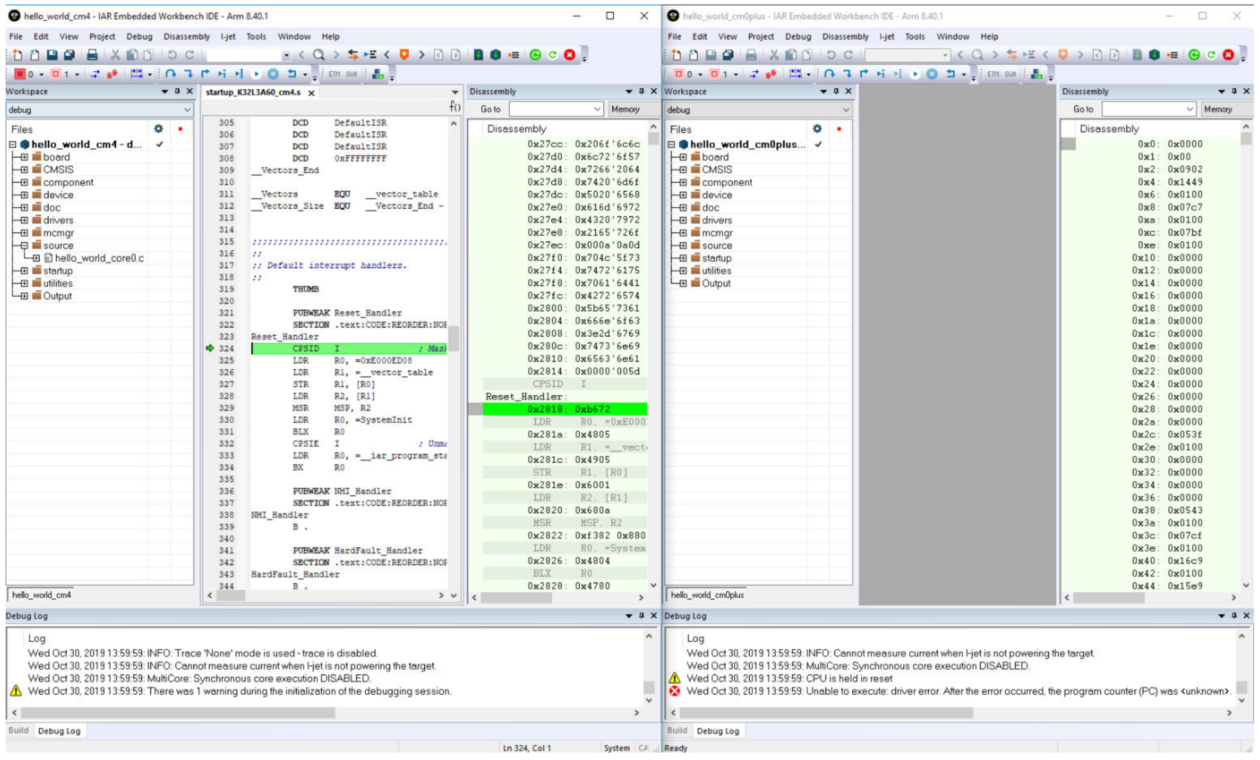
2. To initiate a debug session, click the **Download and debug** button in the master project.


hello_world_cm4 - Master - IAR Embedded Workbench IDE - ARM 8.20.1
File Edit View Project I-jet/JTAGjet Tools Window Help
Workspace
Debug
Files
hello_world_cm4 - Debug
board
doc
drivers
mcmgr
mcmgr.c
mcmgr.h
mcmgr_internal_core_api...
mcmgr_internal_core_api...
mcmgr_mu_internal.c
hello_world_core0.c

```
/****************************************************************************
 * Prototypes
 ****************************************************************************

/****************************************************************************
 * Code
 ****************************************************************************

/*!
```

Figure 16. Download and Debug button in the master project

3. Upon clicking the **Download and debug** button, a second instance of IAR will open with the slave workspace open in that instance.



```
        DCD     DefaultISR
        DCD     DefaultISR
        DCD     DefaultISR
        DCD     0xFFFFFFFF
__Vectors_End

__Vectors       EQU     __vector_table
__Vectors_Size  EQU     __Vectors_End -


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Default interrupt handlers.
;;
        THUMB

        PUBWEAK Reset_Handler
        SECTION .text:CODE:REORDER:NOR
Reset_Handler
        CPSID   I                       ; Mask
        LDR     R0, =0xE000ED08
        LDR     R1, =__vector_table
        STR     R1, [R0]
        LDR     R2, [R1]
        MSR     MSP, R2
        LDR     R0, =SystemInit
        BLX     R0
        CPSIE   I                       ; Unma
        LDR     R0, =__iar_program_sta
        BX      R0

        PUBWEAK NMI_Handler
        SECTION .text:CODE:REORDER:NOR
NMI_Handler
        B .

        PUBWEAK HardFault_Handler
        SECTION .text:CODE:REORDER:NOR
HardFault_Handler
        B .
```

```
Disassembly
        0x27cc:  0x206f'6c6c
        0x27d0:  0x6c72'6f57
        0x27d4:  0x7266'2064
        0x27d8:  0x7420'6d6f
        0x27dc:  0x5020'6568
        0x27e0:  0x616d'6972
        0x27e4:  0x4320'7972
        0x27e8:  0x2165'726f
        0x27ec:  0x000a'0a0d
        0x27f0:  0x704c'5f73
        0x27f4:  0x7472'6175
        0x27f8:  0x7061'6441
        0x27fc:  0x4272'6574
        0x2800:  0x5b65'7361
        0x2804:  0x666e'6f63
        0x2808:  0x3e2d'6769
        0x280c:  0x7473'6e69
        0x2810:  0x6563'6e61
        0x2814:  0x0000'005d
          CPSID    I
Reset_Handler:
        0x2818:  0xb672
          LDR      R0, =0xE000
        0x281a:  0x4805
          LDR      R1, =__vect
        0x281c:  0x4905
          STR      R1, [R0]
        0x281e:  0x6001
          LDR      R2, [R1]
        0x2820:  0x680a
          MSR      MSP, R2
        0x2822:  0xf382 0x880
          LDR      R0, =System
        0x2826:  0x4804
          BLX      R0
        0x2828:  0x4780
```

```
Disassembly
        0x0:  0x0000
        0x1:  0x00
        0x2:  0x0902
        0x4:  0x1449
        0x6:  0x0100
        0x8:  0x07c7
        0xa:  0x0100
        0xc:  0x07bf
        0xe:  0x0100
       0x10:  0x0000
       0x12:  0x0000
       0x14:  0x0000
       0x16:  0x0000
       0x18:  0x0000
       0x1a:  0x0000
       0x1c:  0x0000
       0x1e:  0x0000
       0x20:  0x0000
       0x22:  0x0000
       0x24:  0x0000
       0x26:  0x0000
       0x28:  0x0000
       0x2a:  0x0000
       0x2c:  0x053f
       0x2e:  0x0100
       0x30:  0x0000
       0x32:  0x0000
       0x34:  0x0000
       0x36:  0x0000
       0x38:  0x0543
       0x3a:  0x0100
       0x3c:  0x07cf
       0x3e:  0x0100
       0x40:  0x16c9
       0x42:  0x0100
       0x44:  0x15e9
```

```
Log
Wed Oct 30, 2019 13:59:59: INFO: Trace 'None' mode is used - trace is disabled.
Wed Oct 30, 2019 13:59:59: INFO: Cannot measure current when I-jet is not powering the target.
Wed Oct 30, 2019 13:59:59: MultiCore: Synchronous core execution DISABLED.
Wed Oct 30, 2019 13:59:59: There was 1 warning during the initialization of the debugging session.
```

```
Log
Wed Oct 30, 2019 13:59:59: INFO: Cannot measure current when I-jet is not powering the target.
Wed Oct 30, 2019 13:59:59: MultiCore: Synchronous core execution DISABLED.
Wed Oct 30, 2019 13:59:59: CPU is held in reset
Wed Oct 30, 2019 13:59:59: Unable to execute: driver error. After the error occurred, the program counter (PC) was <unknown>.
```

Figure 17. Debug session including Master and Slave workspaces

NOTE

- The entire MCU can be controlled from either IAR instance.

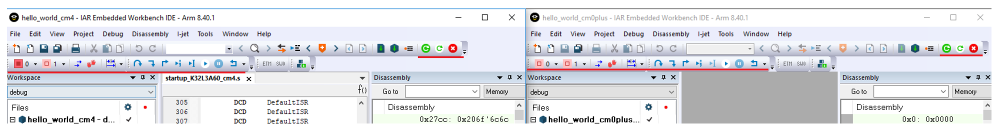

hello_world_cm4 - IAR Embedded Workbench IDE - Arm 8.40.1
File Edit View Project Debug Disassembly I-jet Tools Window Help
Workspace
debug
Files
hello_world_cm4 - d...
startup_K32L3A60_cm4.s
Disassembly
Go to
Memory
hello_world_cm0plus - IAR Embedded Workbench IDE - Arm 8.40.1
hello_world_cm0plus...

Figure 18. Debug controls in both IAR instances

- There are new controls to control the cores independently or simultaneously. There are controls start both cores, pause execution of both cores, or get the status of both cores. Figure 19 displays and explains the multicore controls in IAR.

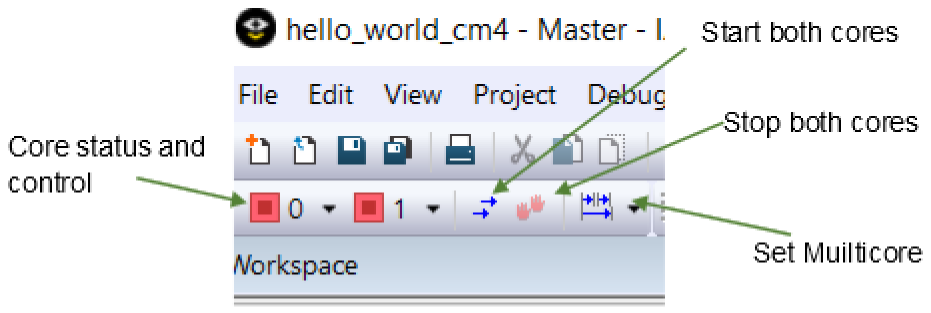
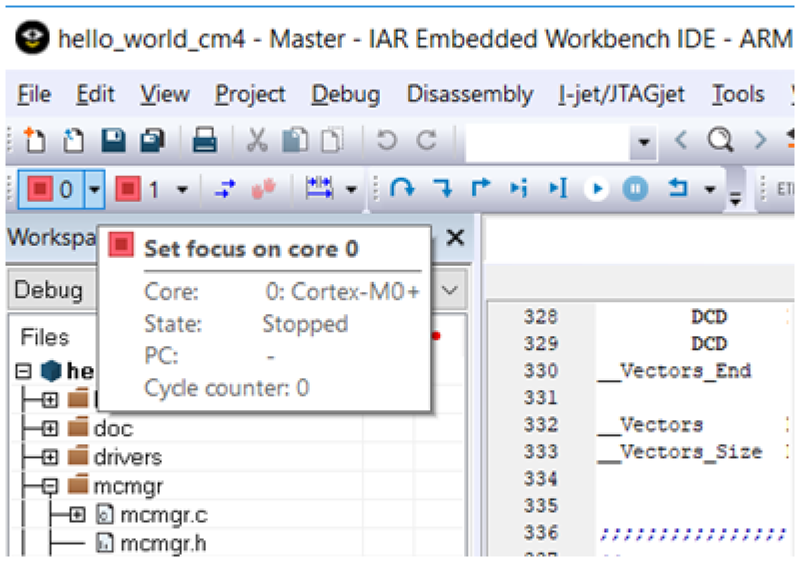


Figure 19. Multicore controls

4. If you mouse over the Core 0 or Core 1 status and control icons, an informational text box will give details of the core. Figure 20 and Figure 21 display an example.

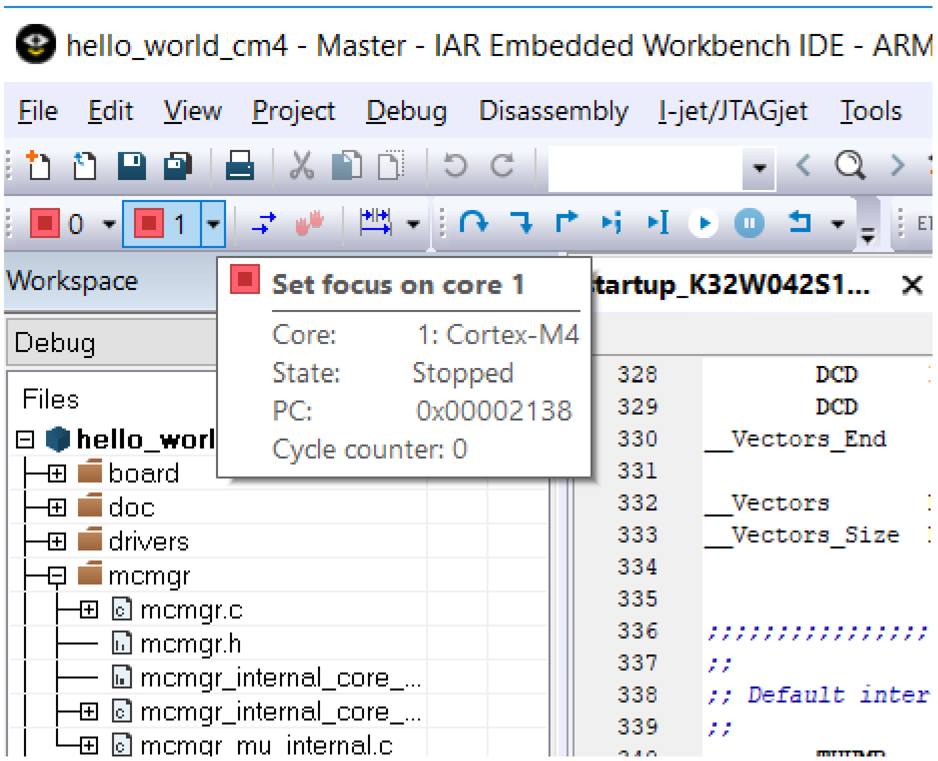

hello_world_cm4 - Master - IAR Embedded Workbench IDE - ARM
File Edit View Project Debug Disassembly I-jet/JTAGjet Tools
0
1
Workspa
Debug
Files
doc
drivers
mcmgr
mcmgr.c
mcmgr.h

Set focus on core 0

Core: 0: Cortex-M0+
State: Stopped
PC: -
Cycle counter: 0

```
DCD
DCD
__Vectors_End

__Vectors
__Vectors_Size
```

Figure 20. Core 0 status



Set focus on core 1
Core: 1: Cortex-M4
State: Stopped
PC: 0x00002138
Cycle counter: 0

```
        DCD
        DCD
__Vectors_End

__Vectors
__Vectors_Size


;;;;;;;;;;;;;;;;
;;
;; Default inter
;;
```

Figure 21. Core 1 status

Clicking the down arrow next to each icon will reveal control options for the cores.

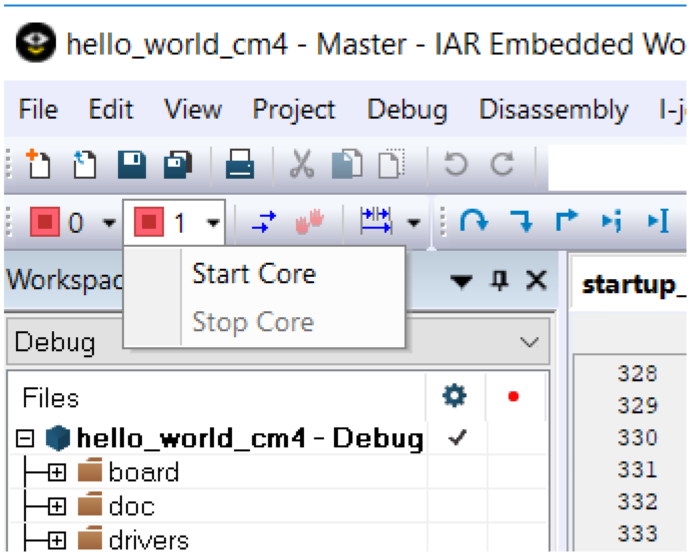

hello_world_cm4 - Master - IAR Embedded Wo
File Edit View Project Debug Disassembly I-j
0
1
Start Core
Stop Core
Workspac
Debug
Files
hello_world_cm4 - Debug
board
doc
drivers
startup_

Figure 22. Core controls

5. The toggle multicore execution mode will set how the other controls operate (Step Into, Step over, etc.,). Mousing over this button will display the current operation status.

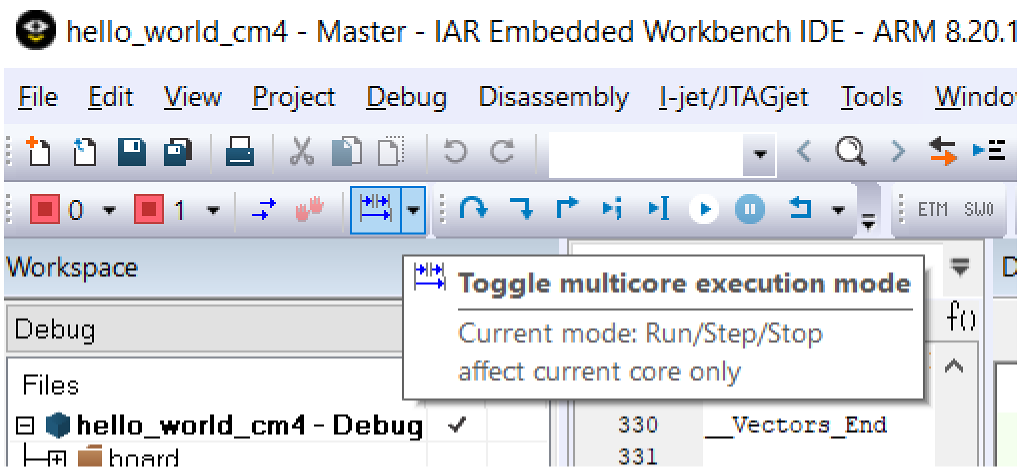

hello_world_cm4 - Master - IAR Embedded Workbench IDE - ARM 8.20.1
File Edit View Project Debug Disassembly I-jet/JTAGjet Tools Windo
Workspace
Debug
Files
hello_world_cm4 - Debug
board
Toggle multicore execution mode
Current mode: Run/Step/Stop affect current core only
330 __Vectors_End
331

Figure 23. Toggle multicore execution mode button

6. Clicking the down arrow next to this button will allow the user to change between single core operation and dual-core operation, as shown in Figure 24.

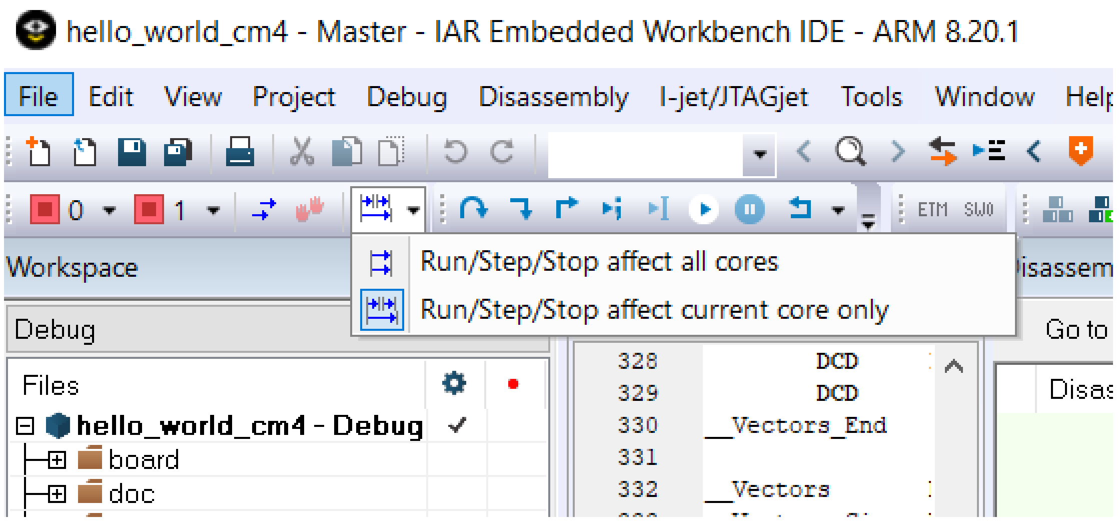

hello_world_cm4 - Master - IAR Embedded Workbench IDE - ARM 8.20.1
File Edit View Project Debug Disassembly I-jet/JTAGjet Tools Window Help
Workspace
Debug
Files
hello_world_cm4 - Debug
board
doc
Run/Step/Stop affect all cores
Run/Step/Stop affect current core only
328 DCD
329 DCD
330 __Vectors_End
331
332 __Vectors
Go to
Disas

Figure 24. Multicore mode selection options

5 Multicore projects in MCUXpresso

Multicore MCUs can be designed in many ways. However, within MCUXpresso IDE, there is an underlying expectation that one core (the Master) will control the execution (or at least the startup) of code running on other (Slave) core(s). This section describes:

- How to make a brand new multi-core project.
- Make a multi-core project from a pair of existing projects (slave and master pair).
- How to debug a multi-core project in MCUXpresso.

5.1 Multi-core project creation

Multicore application projects as described below consists of two linked projects:

- One project containing the Master code.

- One project containing the Slave code.

The **Master** project contains a link to the **Slave** project which will cause the output image from the **Slave** to be included into the **Master** image when the Master project is built.

NOTE

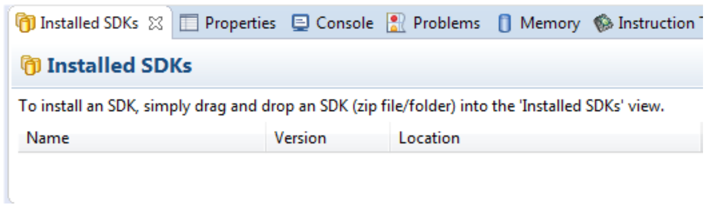
Building the Master project will trigger the Slave project to be built first.

5.1.1 Creating a master/slave project pair (using an SDK)

Since the Master project's configuration needs to reference the slave project, the slave project should be created first.

To create the slave project, perform the following operations:

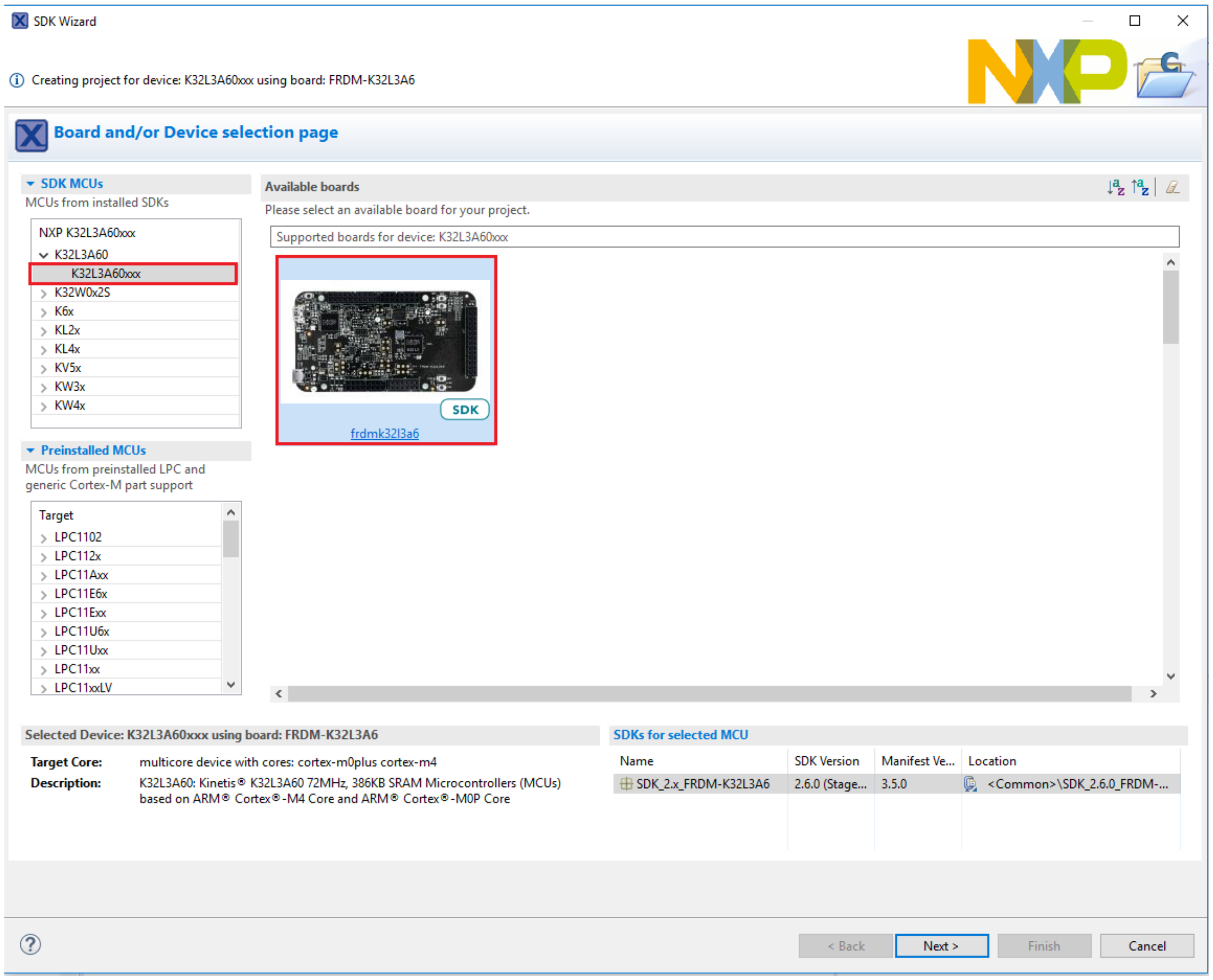
1. Drag and drop the SDK zip file into the **Installed SDKs** view (if the SDK has not already been installed) to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view.

Figure 25. Installed SDKs view in MCUXpresso IDE

2. Launch the New Project Wizard, select **frdmk32l3a6** and click **Next**.



SDK Wizard

Creating project for device: K32L3A60xxx using board: FRDM-K32L3A6

Board and/or Device selection page

SDK MCUs
MCUs from installed SDKs

NXP K32L3A60xxx
K32L3A60
K32L3A60xxx
K32W0x2S
K6x
KL2x
KL4x
KV5x
KW3x
KW4x

Preinstalled MCUs
MCUs from preinstalled LPC and generic Cortex-M part support

Target
LPC1102
LPC112x
LPC11Axx
LPC11E6x
LPC11Exx
LPC11U6x
LPC11Uxx
LPC11xx
LPC11xxLV

Available boards
Please select an available board for your project.
Supported boards for device: K32L3A60xxx
frdmk32l3a6

Selected Device: K32L3A60xxx using board: FRDM-K32L3A6

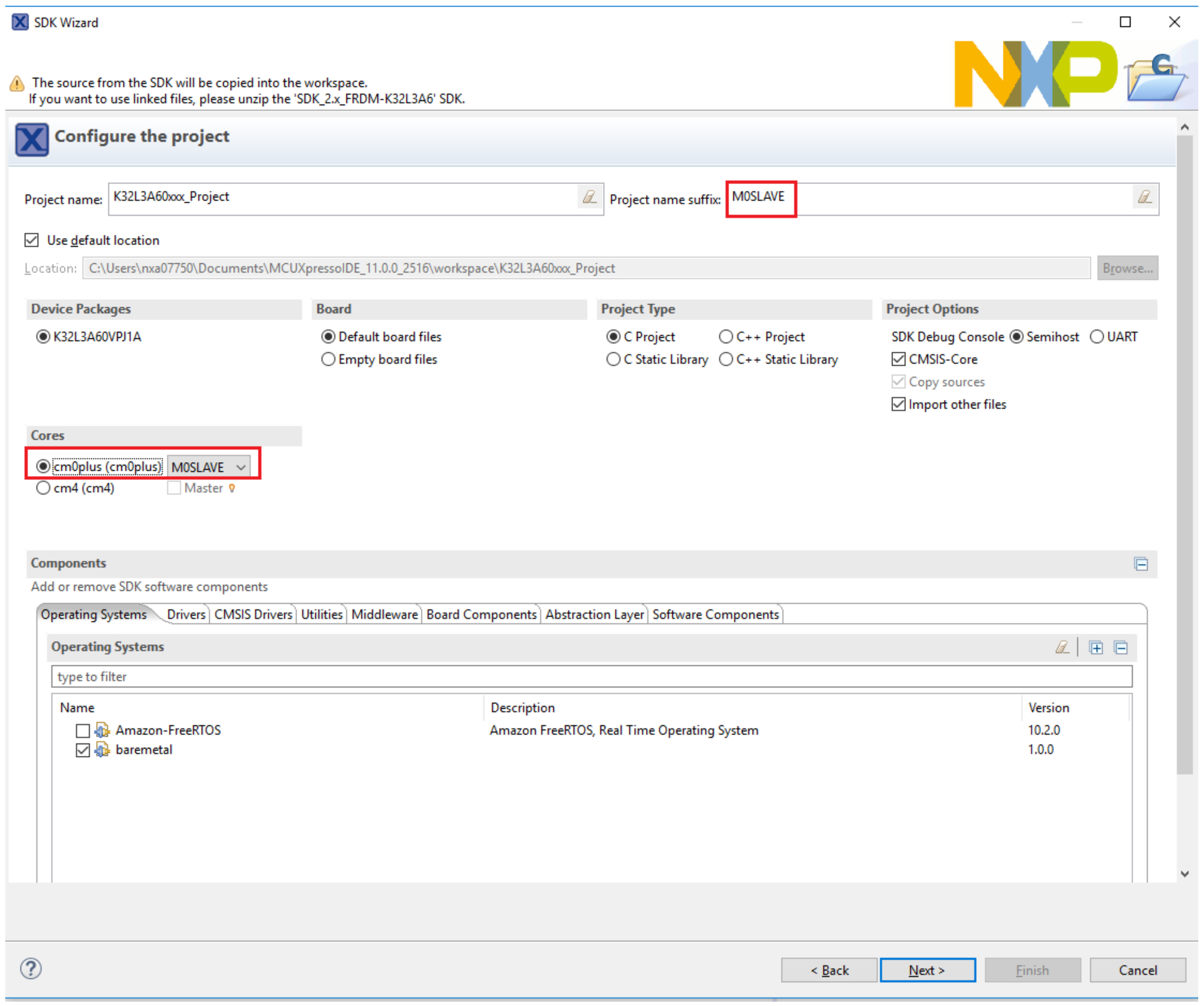
Target Core: multicore device with cores: cortex-m0plus cortex-m4
Description: K32L3A60: Kinetis® K32L3A60 72MHz, 386KB SRAM Microcontrollers (MCUs) based on ARM® Cortex®-M4 Core and ARM® Cortex®-M0P Core

SDKs for selected MCU

Name	SDK Version	Manifest Ve...	Location
SDK_2.x_FRDM-K32L3A6	2.6.0 (Stage...	3.5.0	<Common>\SDK_2.6.0_FRDM-...

Figure 26. New Project Wizard SDK MultiCore, Arm Cortex-M0+

3. On the **Confugure the project** page, select the **cm0plus** core. Make sure that **M0SLAVE** is selected in the core options (the Project will automatically be given the suffix of `M0SLAVE`). Drivers, utilities, etc. can be selected at this stage for the Slave project if required.


SDK Wizard

The source from the SDK will be copied into the workspace.
If you want to use linked files, please unzip the 'SDK_2.x_FRDM-K32L3A6' SDK.

Configure the project

Project name: K32L3A60xxx_Project   Project name suffix: M0SLAVE

☑ Use default location

Location: C:\Users\nxa07750\Documents\MCUXpressoIDE_11.0.0_2516\workspace\K32L3A60xxx_Project   Browse...

Device Packages: ● K32L3A60VPJ1A

Board: ● Default board files ○ Empty board files

Project Type: ● C Project ○ C++ Project ○ C Static Library ○ C++ Static Library

Project Options: SDK Debug Console ● Semihost ○ UART; ☑ CMSIS-Core; ☑ Copy sources; ☑ Import other files

Cores:
● cm0plus (cm0plus) M0SLAVE
○ cm4 (cm4) ☐ Master

Components

Add or remove SDK software components

Operating Systems | Drivers | CMSIS Drivers | Utilities | Middleware | Board Components | Abstraction Layer | Software Components

Operating Systems

type to filter

Name	Description	Version
☐ Amazon-FreeRTOS	Amazon FreeRTOS, Real Time Operating System	10.2.0
☑ baremetal		1.0.0

< Back | Next > | Finish | Cancel

Figure 27. SDK Wizard Arm Cortex-M0+ slave project configuration

4. Set the Arm Cortex-M0+ Slave memory configurations.

NOTE

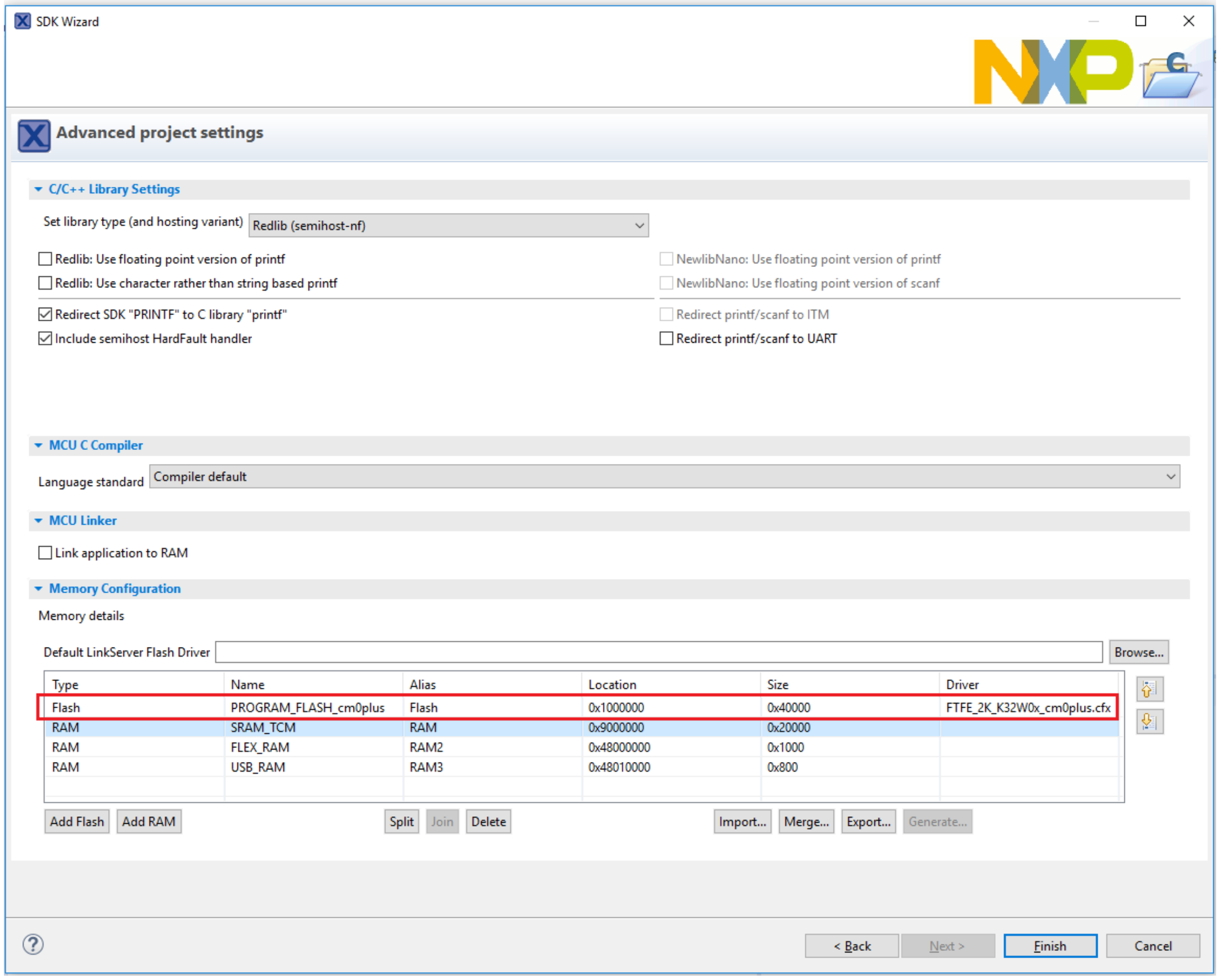
The MCUXpresso IDE's managed linker script mechanism will default to link code to the first Flash region in this view (if one exists) and use the first RAM region for data, heap and stack.

To force the Arm Cortex-M0+ code to link to a specific area of memory, ensure that the desired memory region is at the top of the memory configuration list.

NOTE

To place the project in RAM, from the previous note, ensure the Flash region is removed and the desired RAM bank is at the top of the memory configuration list.

In this example, we chose memory region starting at `0x1000000`, which corresponds to the Arm Cortex-M0+ flash space, for the Arm Cortex-M0+ code and RAM starting at `0x9000000` for the Arm Cortex-M0+ data.



SDK Wizard

Advanced project settings

C/C++ Library Settings

Set library type (and hosting variant): Redlib (semihost-nf)

- [ ] Redlib: Use floating point version of printf
- [ ] Redlib: Use character rather than string based printf
- [x] Redirect SDK "PRINTF" to C library "printf"
- [x] Include semihost HardFault handler
- [ ] NewlibNano: Use floating point version of printf
- [ ] NewlibNano: Use floating point version of scanf
- [ ] Redirect printf/scanf to ITM
- [ ] Redirect printf/scanf to UART

MCU C Compiler

Language standard: Compiler default

MCU Linker

- [ ] Link application to RAM

Memory Configuration

Memory details

Default LinkServer Flash Driver

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH_cm0plus	Flash	0x1000000	0x40000	FTFE_2K_K32W0x_cm0plus.cfx
RAM	SRAM_TCM	RAM	0x9000000	0x20000	
RAM	FLEX_RAM	RAM2	0x48000000	0x1000	
RAM	USB_RAM	RAM3	0x48010000	0x800	

Figure 28. New project wizard SDK Arm Cortex-M0+ slave memory

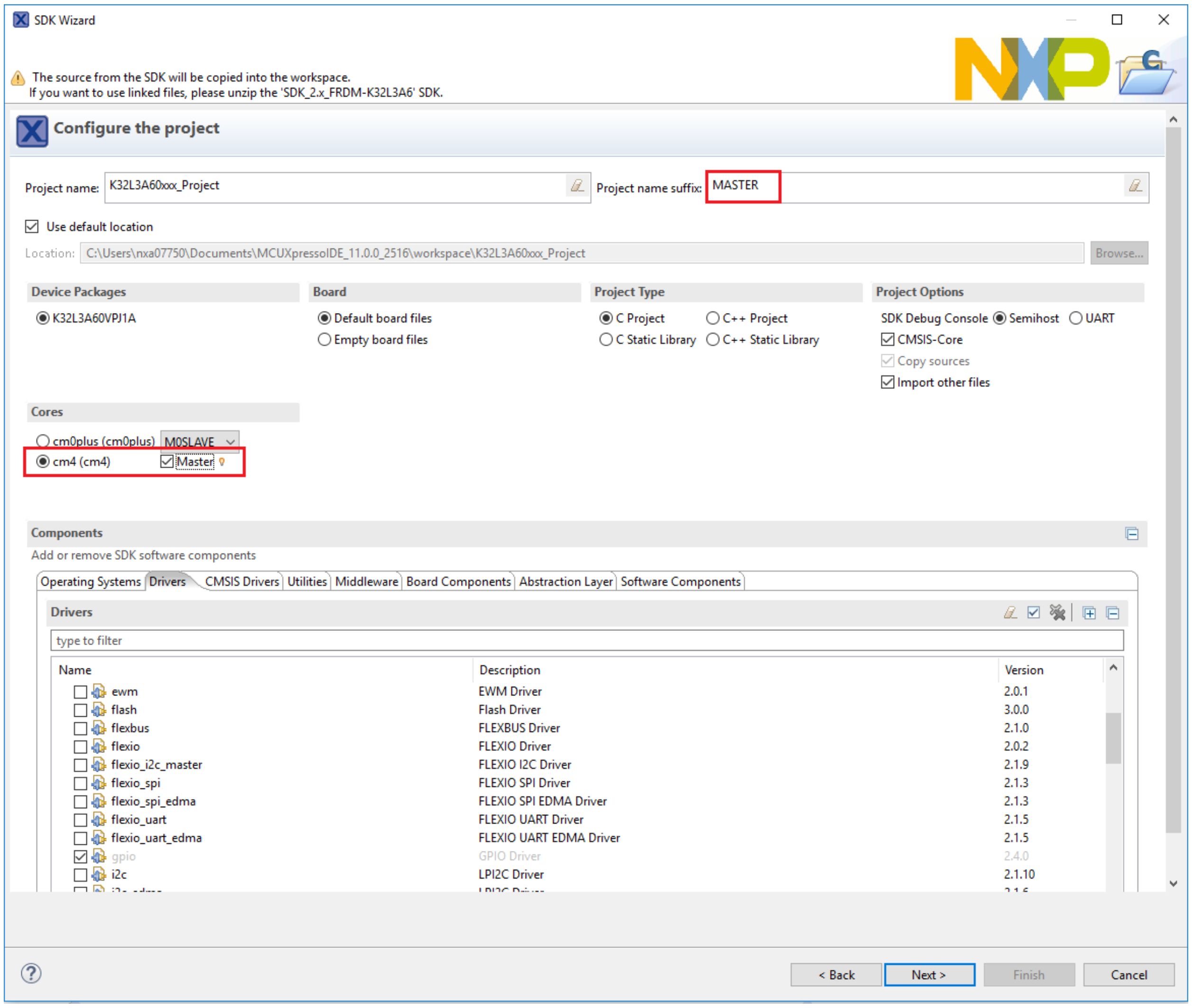
5. Click **Finish** to complete the creation of the Slave project.

To create the Master project, perform the following operations:

1. Launch the New Project Wizard, select **FRDMK32L3A6 SDK**, and click **Next**. Then, select **cm4 Core** and click the **MASTER** check box. This configures the wizard to create a Multicore project.

NOTE

The project will automatically be given the suffix MASTER. Drivers, utilities, etc. can be selected at this stage for the Master project if required.



SDK Wizard

The source from the SDK will be copied into the workspace.
If you want to use linked files, please unzip the 'SDK_2.x_FRDM-K32L3A6' SDK.

Configure the project

Project name: K32L3A60xxx_Project   Project name suffix: MASTER

☑ Use default location

Location: C:\Users\nxa07750\Documents\MCUXpressoIDE_11.0.0_2516\workspace\K32L3A60xxx_Project   Browse...

Device Packages
● K32L3A60VPJ1A

Board
● Default board files
○ Empty board files

Project Type
● C Project ○ C++ Project
○ C Static Library ○ C++ Static Library

Project Options
SDK Debug Console ● Semihost ○ UART
☑ CMSIS-Core
☑ Copy sources
☑ Import other files

Cores
○ cm0plus (cm0plus) M0SLAVE
● cm4 (cm4) ☑ Master

Components
Add or remove SDK software components

Operating Systems | Drivers | CMSIS Drivers | Utilities | Middleware | Board Components | Abstraction Layer | Software Components

Drivers

type to filter

Name	Description	Version
☐ ewm	EWM Driver	2.0.1
☐ flash	Flash Driver	3.0.0
☐ flexbus	FLEXBUS Driver	2.1.0
☐ flexio	FLEXIO Driver	2.0.2
☐ flexio_i2c_master	FLEXIO I2C Driver	2.1.9
☐ flexio_spi	FLEXIO SPI Driver	2.1.3
☐ flexio_spi_edma	FLEXIO SPI EDMA Driver	2.1.3
☐ flexio_uart	FLEXIO UART Driver	2.1.5
☐ flexio_uart_edma	FLEXIO UART EDMA Driver	2.1.5
☑ gpio	GPIO Driver	2.4.0
☐ i2c	LPI2C Driver	2.1.10

< Back | Next > | Finish | Cancel

Figure 29. SDK wizard Arm Cortex-M4 master project configurations

2. Set the Arm Cortex-M4 Master memory configurations.

NOTE

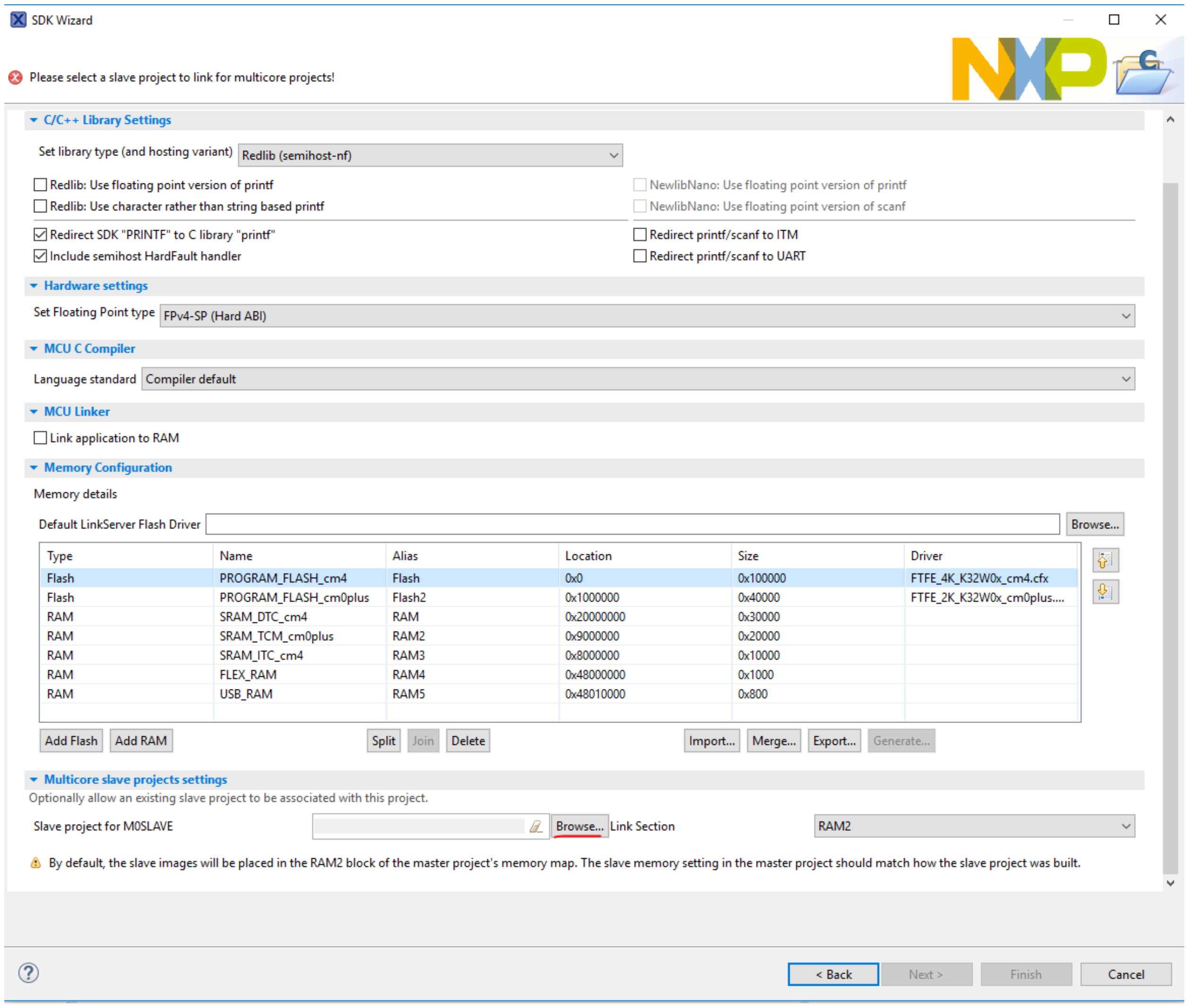
The MCUXpresso IDEs managed linker script mechanism will default to link code to the first Flash region in this view (if one exists) and use the first RAM region for data, heap and stack.

To place the Arm Cortex-M4 project code in a specific section of memory, ensure that the desired memory region is at the top of the memory configuration list. In this example, we are placing the Arm Cortex-M4 code in the Arm Cortex-M4 flash, which starts at address `0x0`, and the Arm Cortex M4 data in RAM starting at `0x20000000`.

NOTE

If we want to place the project in RAM, from the previous note, ensure the Flash regions are removed and the desired RAM bank is at the top of the memory configuration list.

3. Click **Browse** next to the Slave project for **M0SLAVE** selection box (as shown in Figure 30) to select the Slave project within the Workspace.



SDK Wizard

Please select a slave project to link for multicore projects!

C/C++ Library Settings

Set library type (and hosting variant): Redlib (semihost-nf)

- [ ] Redlib: Use floating point version of printf
- [ ] Redlib: Use character rather than string based printf
- [ ] NewlibNano: Use floating point version of printf
- [ ] NewlibNano: Use floating point version of scanf
- [x] Redirect SDK "PRINTF" to C library "printf"
- [x] Include semihost HardFault handler
- [ ] Redirect printf/scanf to ITM
- [ ] Redirect printf/scanf to UART

Hardware settings

Set Floating Point type: FPv4-SP (Hard ABI)

MCU C Compiler

Language standard: Compiler default

MCU Linker

- [ ] Link application to RAM

Memory Configuration

Memory details

Default LinkServer Flash Driver:

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH_cm4	Flash	0x0	0x100000	FTFE_4K_K32W0x_cm4.cfx
Flash	PROGRAM_FLASH_cm0plus	Flash2	0x1000000	0x40000	FTFE_2K_K32W0x_cm0plus....
RAM	SRAM_DTC_cm4	RAM	0x20000000	0x30000	
RAM	SRAM_TCM_cm0plus	RAM2	0x9000000	0x20000	
RAM	SRAM_ITC_cm4	RAM3	0x8000000	0x10000	
RAM	FLEX_RAM	RAM4	0x48000000	0x1000	
RAM	USB_RAM	RAM5	0x48010000	0x800	

Multicore slave projects settings

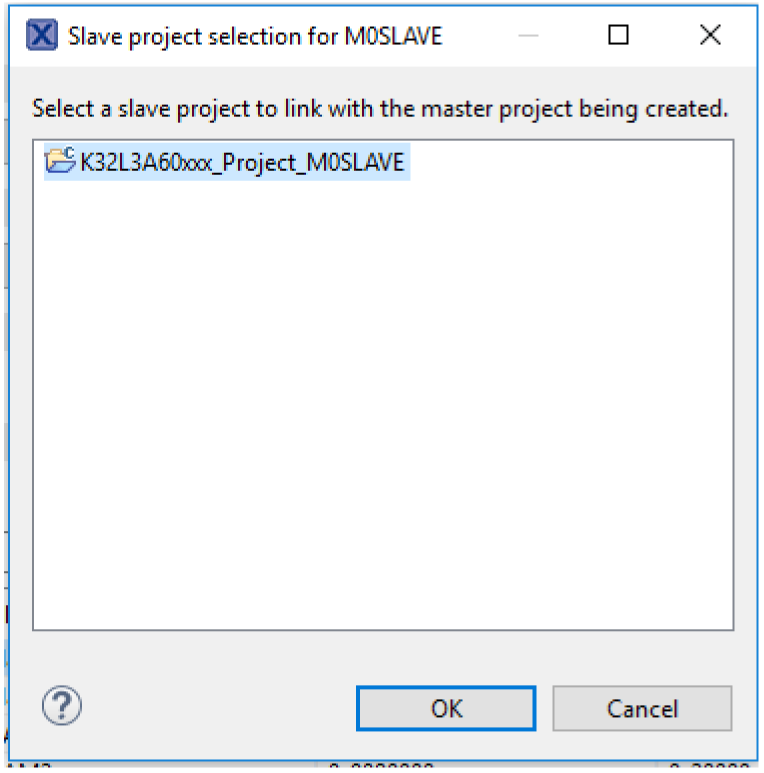
Optionally allow an existing slave project to be associated with this project.

Slave project for M0SLAVE: Link Section: RAM2

By default, the slave images will be placed in the RAM2 block of the master project's memory map. The slave memory setting in the master project should match how the slave project was built.

Figure 30. MCUXpresso memory selection dialog box

4. Select the previously created Slave project and click **OK**.

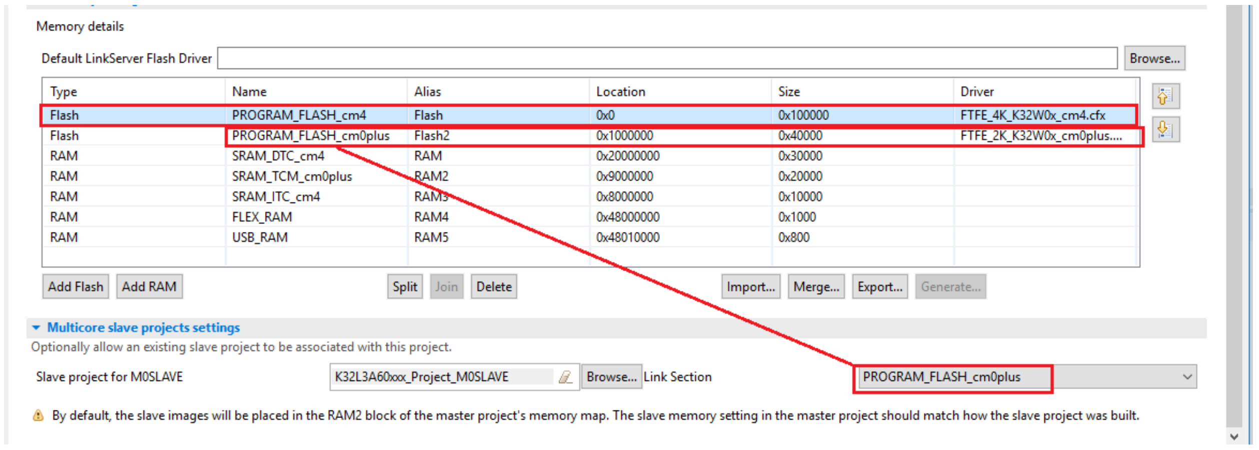


Select a slave project to link with the master project being created.

K32L3A60xxx_Project_M0SLAVE

Figure 31. New Project Wizard SDK Arm Cortex-M4 Master Slave Selection

5. Make sure that the Link Section name (the default value is **RAM2**) is a memory region that matches the linked address of the Slave project. In this case, we select **PROGRAM_FLASH_cm0plus** as it corresponds to address `0x1000000`.



Memory details

Default LinkServer Flash Driver

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH_cm4	Flash	0x0	0x100000	FTFE_4K_K32W0x_cm4.cfx
Flash	PROGRAM_FLASH_cm0plus	Flash2	0x1000000	0x40000	FTFE_2K_K32W0x_cm0plus....
RAM	SRAM_DTC_cm4	RAM	0x20000000	0x30000	
RAM	SRAM_TCM_cm0plus	RAM2	0x9000000	0x20000	
RAM	SRAM_ITC_cm4	RAM3	0x8000000	0x10000	
RAM	FLEX_RAM	RAM4	0x48000000	0x1000	
RAM	USB_RAM	RAM5	0x48010000	0x800	

Multicore slave projects settings

Optionally allow an existing slave project to be associated with this project.

Slave project for M0SLAVE: K32L3A60xxx_Project_M0SLAVE   Link Section: PROGRAM_FLASH_cm0plus

By default, the slave images will be placed in the RAM2 block of the master project's memory map. The slave memory setting in the master project should match how the slave project was built.

Figure 32. New project wizard SDK Arm Cortex-M4 master memory and slave selection

6. Click **Finish** to generate the Master project.

5.1.2 Editing existing project settings for Multicore

If you wish to make a multi-core project from two existing projects (one for the Arm Cortex-M4 and the other one for the Arm Cortex-M0+), perform the following modifications to the project settings. The multi-core `hello_world` example located in the FRDMK32L3A6 SDK will be used as a guiding example.

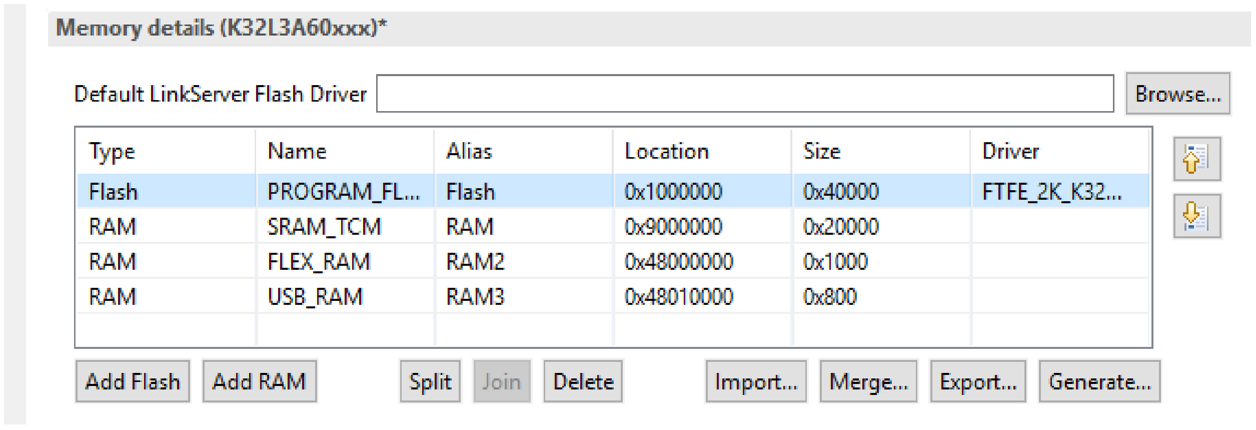
1. Modify the slave project settings.

Go to **Project Properties -> C/C++ Build -> MCU Settings**

In the memory details, edit the list so that the memory region you want the Arm Cortex-M0+ code to be placed in is at the top of the list. For this example, we chose memory region starting at `0x1000000`.

NOTE

The MCUXpresso IDE's managed linker script mechanism will default to link code to the first Flash region in this view (if one exists) and use the first RAM region for data, heap and stack.



Memory details (K32L3A60xxx)*

Default LinkServer Flash Driver

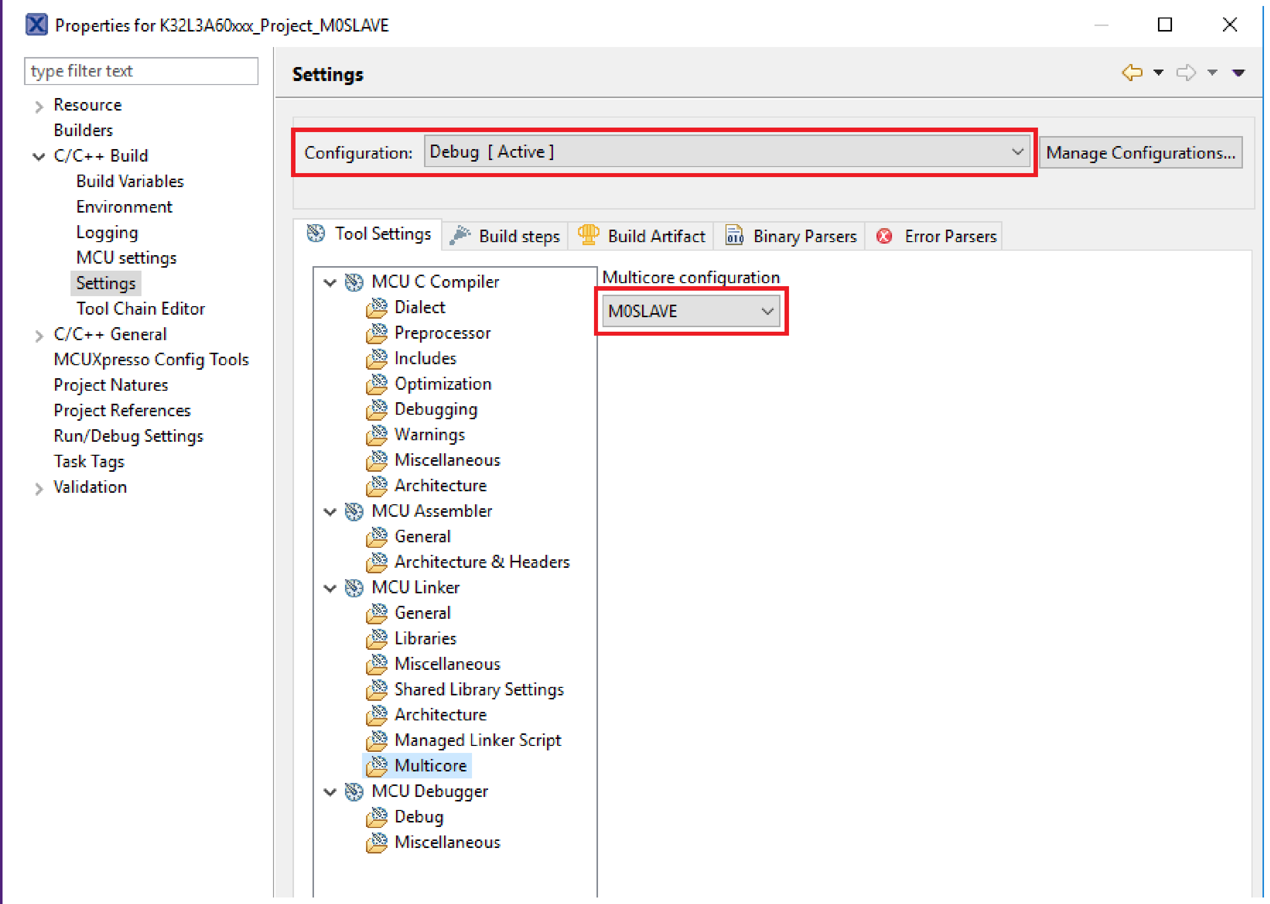
Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FL...	Flash	0x1000000	0x40000	FTFE_2K_K32...
RAM	SRAM_TCM	RAM	0x9000000	0x20000	
RAM	FLEX_RAM	RAM2	0x48000000	0x1000	
RAM	USB_RAM	RAM3	0x48010000	0x800	

Figure 33. Existing Arm Cortex-M0+ slave memory settings

NOTE

To place the project in RAM, from the previous note, ensure the Flash regions are removed and the desired RAM bank is at the top of the memory configuration list.

2. Go to **Project Properties -> C/C++ Build -> Settings -> Tool Settings -> MCU Linker -> Multicore**, select the desired configuration, and ensure that the project is configured as **M0SLAVE**.



Configuration: Debug [Active]

Multicore configuration: M0SLAVE

Figure 34. Existing Arm Cortex-M0+ Slave Multicore Settings

NOTE

The same build configuration must be selected for both the Arm Cortex-M4 and Arm Cortex-M0+ projects.

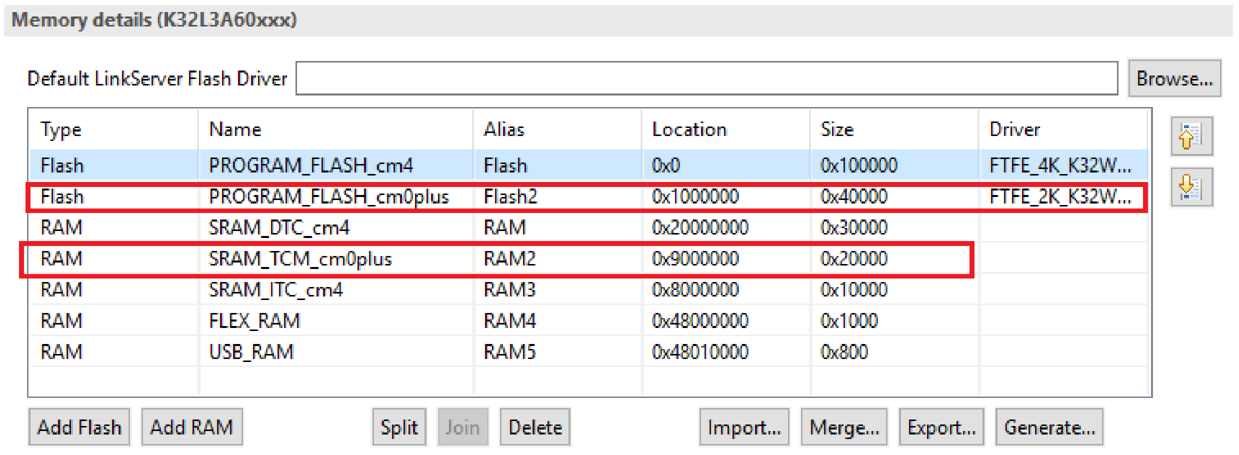
3. Click **OK** to save the changes and close the project properties window.

Now it’s time to edit the Master project settings.

1. Go to **Project Properties -> C/C++ Build -> MCU Settings**.
2. In the memory details, edit the list so that the memory region you want the Arm Cortex-M4 code to be placed in is at the top of the list. For this example, we chose memory region starting at `0x0`. Additionally, ensure that the memory region that you chose for the Arm Cortex-M0+ code is also in the list.

NOTE

The MCUXpresso IDE’s managed linker script mechanism will default to link code to the first Flash region in this view (if one exists) and use the first RAM region for data, heap and stack.



Memory details (K32L3A60xxx)

Default LinkServer Flash Driver

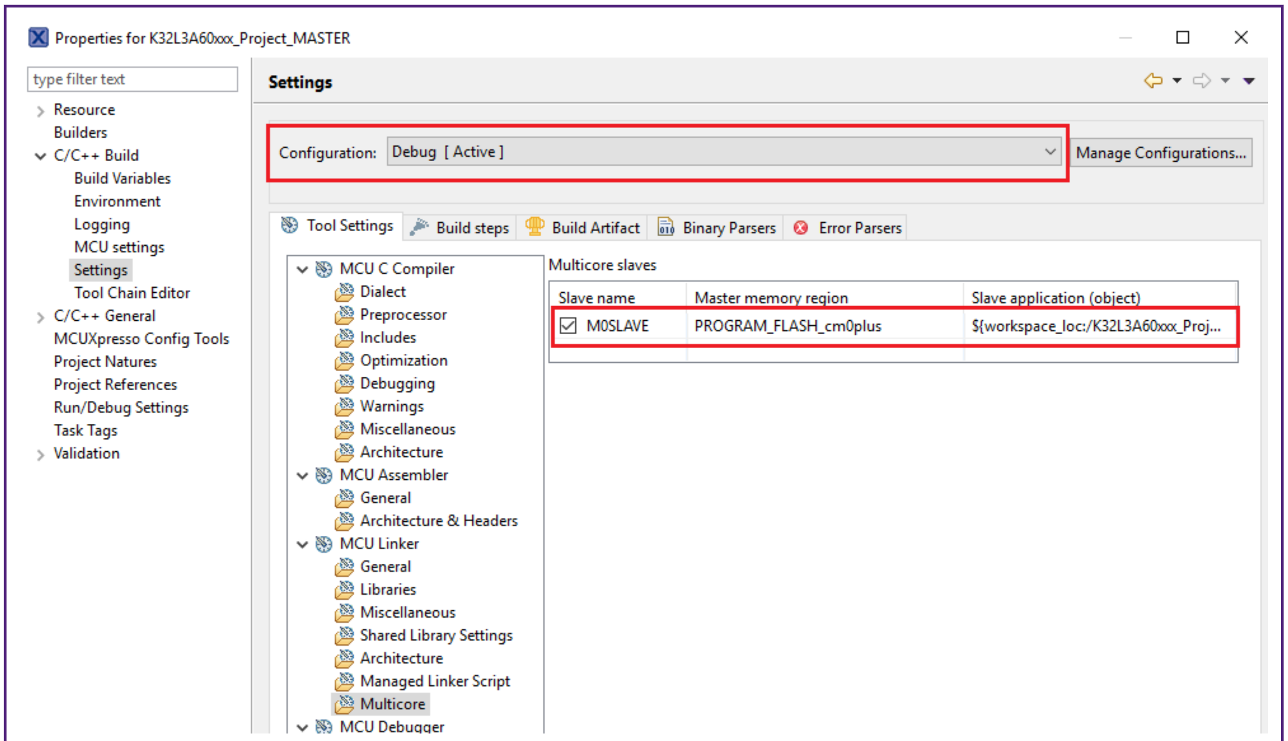
Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH_cm4	Flash	0x0	0x100000	FTFE_4K_K32W...
Flash	PROGRAM_FLASH_cm0plus	Flash2	0x1000000	0x40000	FTFE_2K_K32W...
RAM	SRAM_DTC_cm4	RAM	0x20000000	0x30000	
RAM	SRAM_TCM_cm0plus	RAM2	0x9000000	0x20000	
RAM	SRAM_ITC_cm4	RAM3	0x8000000	0x10000	
RAM	FLEX_RAM	RAM4	0x48000000	0x1000	
RAM	USB_RAM	RAM5	0x48010000	0x800	

Figure 35. Existing Arm Cortex-M4 Master Memory Settings

NOTE

To place the project in RAM, from the previous note, ensure the Flash regions are removed and the desired RAM bank is at the top of the memory configuration list.

3. Go to **Project Properties** -> **C/C++ Build** -> **Settings** -> **Tool Settings** -> **MCU Linker** -> **Multicore** and select the desired configuration. The selected configuration should be the same as that of the Slave project. In this case, we chose the Debug configuration for both.
4. Select the checkbox for M0SLAVE to indicate that there is a slave project that should be linked to the Arm Cortex-M4 project. Then, select the appropriate master memory region in which the Arm Cortex-M0+ code will be placed. This should correspond to the memory region chosen in the Arm Cortex-M0+ project properties. In this example, we chose `PROGRAM_FLASH_cm0plus`, which corresponds to memory starting at `0x1000000`.


Properties for K32L3A60xxx_Project_MASTER

Settings

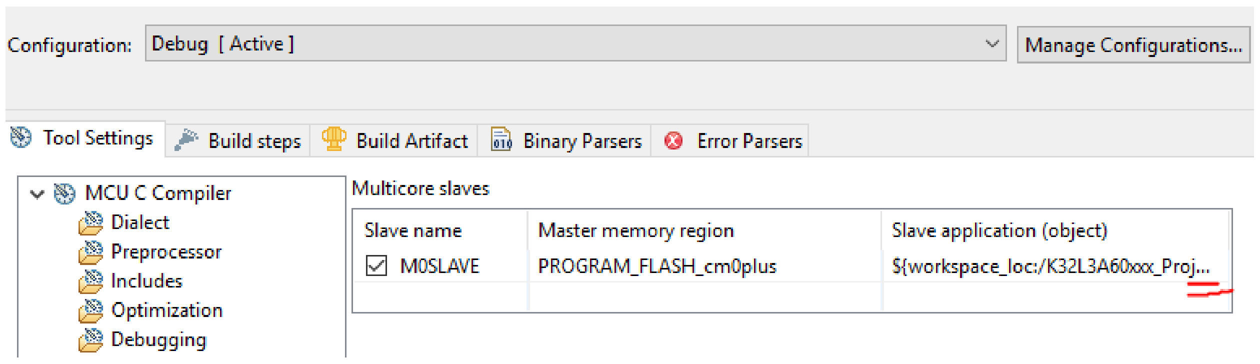
Configuration: Debug [Active]

Multicore slaves

Slave name	Master memory region	Slave application (object)
✓ M0SLAVE	PROGRAM_FLASH_cm0plus	${workspace_loc:/K32L3A60xxx_Proj...

Figure 36. Existing Arm Cortex-M4 Master Multicore settings

5. Under the Slave application, click on the ellipsis to open a window to select the object file for the slave project.

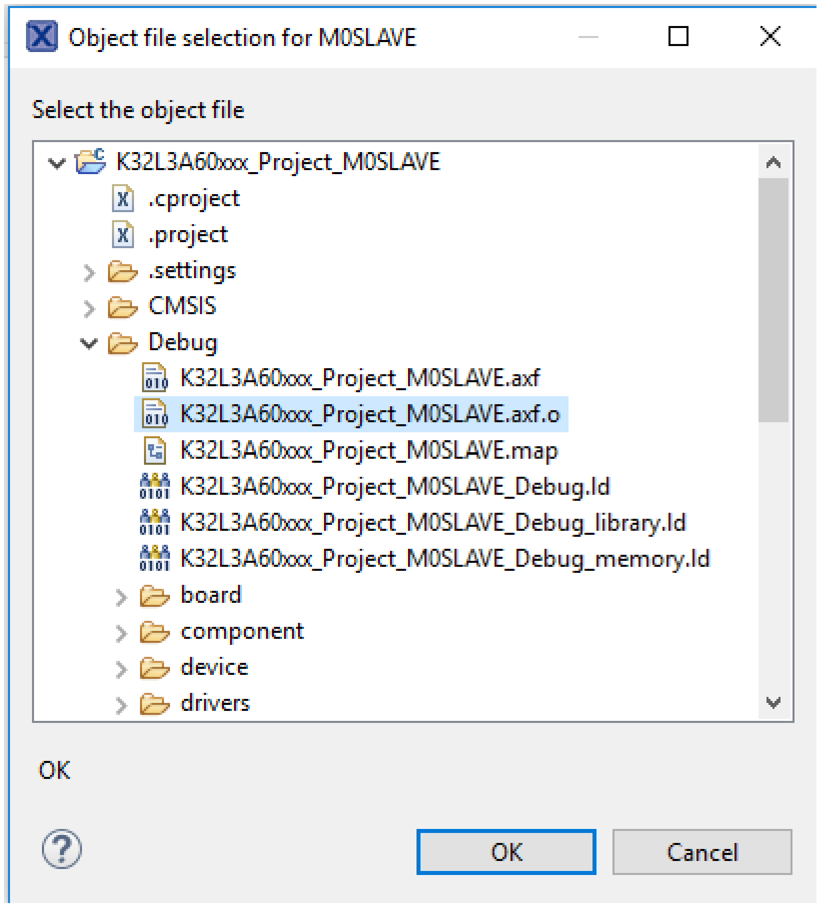

Configuration: Debug [Active]

Multicore slaves

Slave name	Master memory region	Slave application (object)
✓ M0SLAVE	PROGRAM_FLASH_cm0plus	${workspace_loc:/K32L3A60xxx_Proj...

Figure 37. Ellipsis to click on to select slave project

6. This file should be in the selected configuration folder after building the slave project.


Object file selection for M0SLAVE

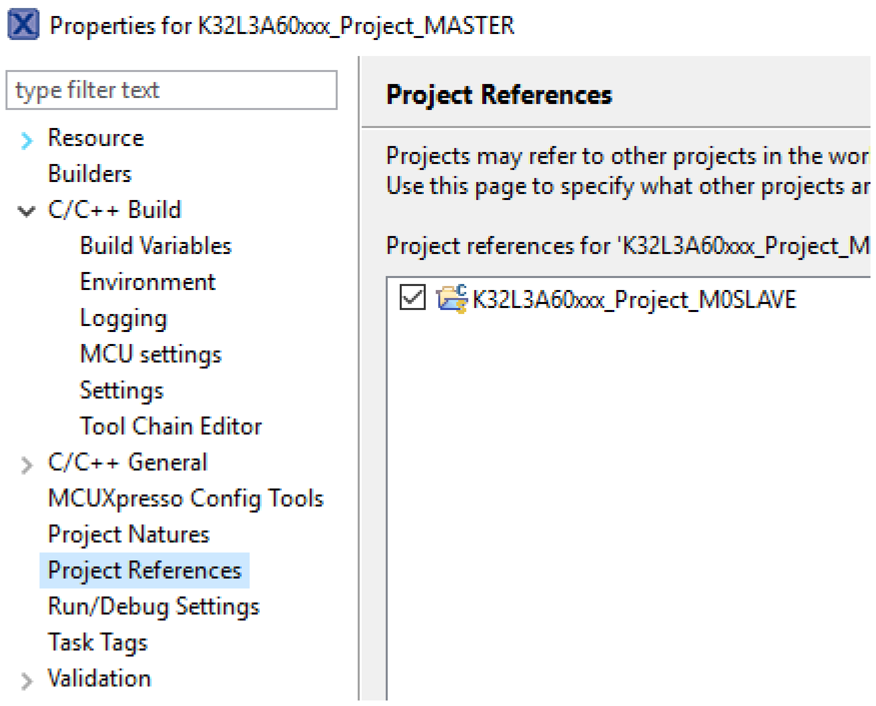
Select the object file

K32L3A60xxx_Project_M0SLAVE
- .cproject
- .project
- .settings
- CMSIS
- Debug
  - K32L3A60xxx_Project_M0SLAVE.axf
  - K32L3A60xxx_Project_M0SLAVE.axf.o
  - K32L3A60xxx_Project_M0SLAVE.map
  - K32L3A60xxx_Project_M0SLAVE_Debug.ld
  - K32L3A60xxx_Project_M0SLAVE_Debug_library.ld
  - K32L3A60xxx_Project_M0SLAVE_Debug_memory.ld
- board
- component
- device
- drivers

OK

Figure 38. Existing Arm Cortex-M4 Master Slave selection

7. Go to *Project Properties* -> **Project References** and select the checkbox for the slave project.


Properties for K32L3A60xxx_Project_MASTER

type filter text

- Resource
- Builders
- C/C++ Build
  - Build Variables
  - Environment
  - Logging
  - MCU settings
  - Settings
  - Tool Chain Editor
- C/C++ General
- MCUXpresso Config Tools
- Project Natures
- Project References
- Run/Debug Settings
- Task Tags
- Validation

Project References

Projects may refer to other projects in the wor
Use this page to specify what other projects ar

Project references for 'K32L3A60xxx_Project_M

☑ K32L3A60xxx_Project_M0SLAVE

Figure 39. Existing Arm Cortex-M4 Master Slave Project Reference

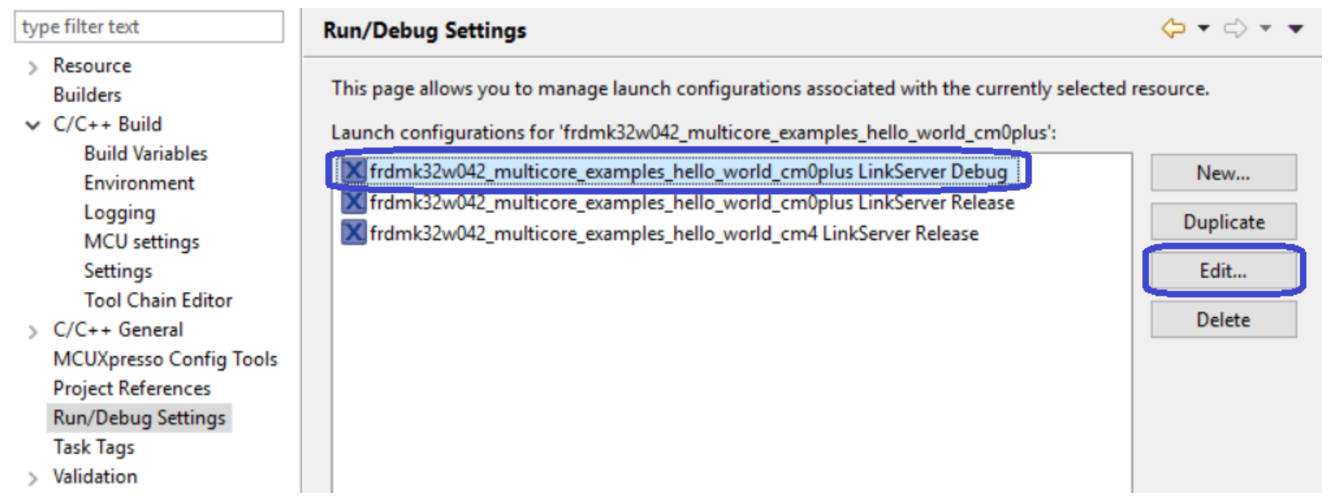
8. Click **OK** to save the changes and close the project properties window.

6 Multicore debug in MCUXpresso

The Master core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory; therefore, the Master project should be run (debugged) first. However, before debugging, you must ensure that the slave project debug settings are configured correctly for multi-core debugging.

6.1 Configuring slave debug settings

With the slave project selected, go to **Project Properties** -> **Run/Debug Settings**. Select the debug configuration and click **Edit**, as shown in Figure 40.



Run/Debug Settings

This page allows you to manage launch configurations associated with the currently selected resource.

Launch configurations for 'frdmk32w042_multicore_examples_hello_world_cm0plus':

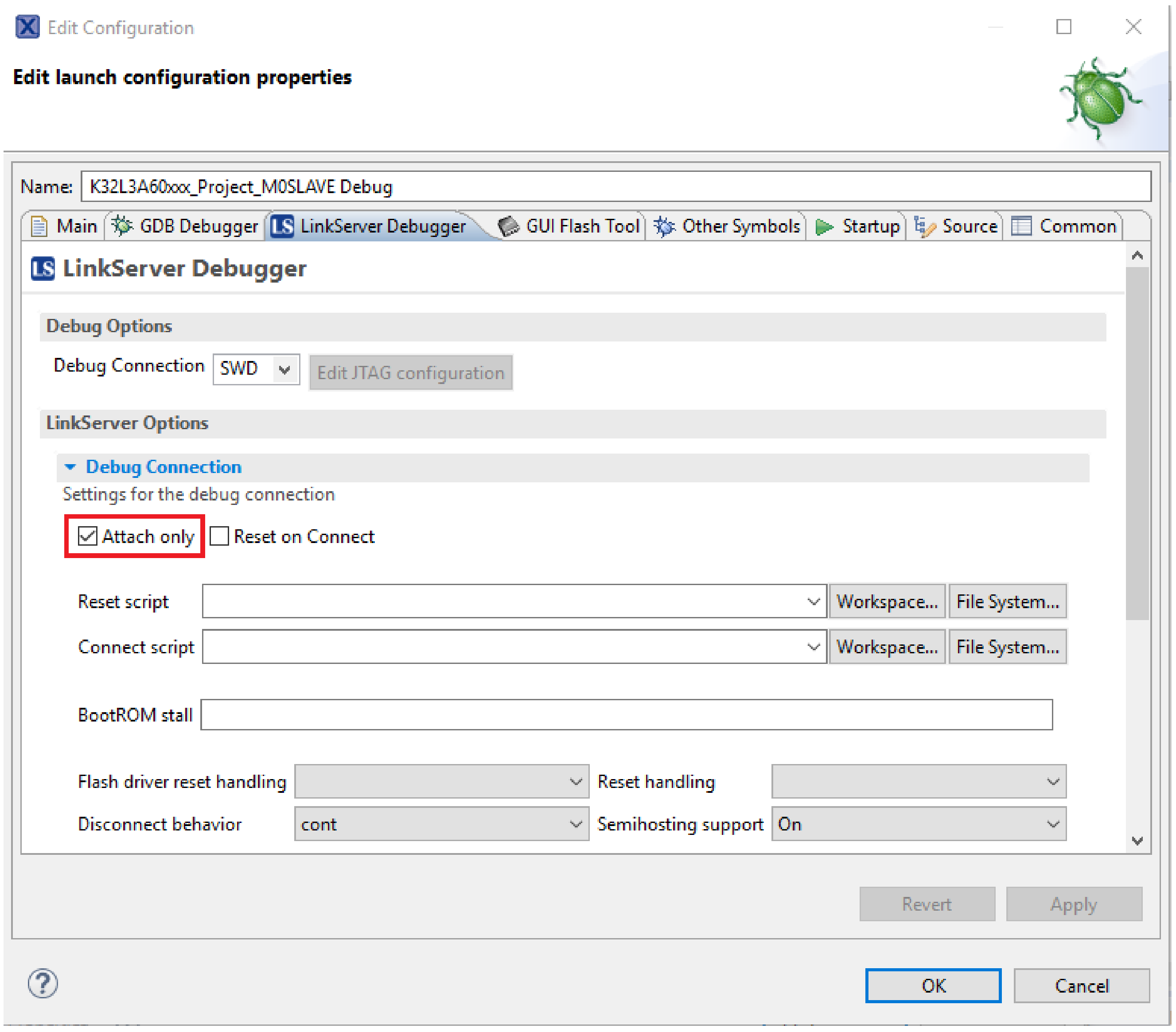
- frdmk32w042_multicore_examples_hello_world_cm0plus LinkServer Debug
- frdmk32w042_multicore_examples_hello_world_cm0plus LinkServer Release
- frdmk32w042_multicore_examples_hello_world_cm4 LinkServer Release

Figure 40. Arm Cortex-M0+ slave edit run/debug settings

NOTE

If no configurations are available, click new to create a new configuration. Be sure to click **Search project** and select the binary from the slave project. You may also need to build the slave project before executing this step.

In the **Edit** launch configuration properties window, click on the **LinkServer Debugger** tab and check **Attach only**, as shown in Figure 41.



Edit launch configuration properties

Name: K32L3A60xxx_Project_M0SLAVE Debug

LinkServer Debugger

Debug Options

Debug Connection: SWD

LinkServer Options

Debug Connection

Settings for the debug connection

[✓] Attach only [ ] Reset on Connect

Reset script:

Connect script:

BootROM stall:

Flash driver reset handling:

Reset handling:

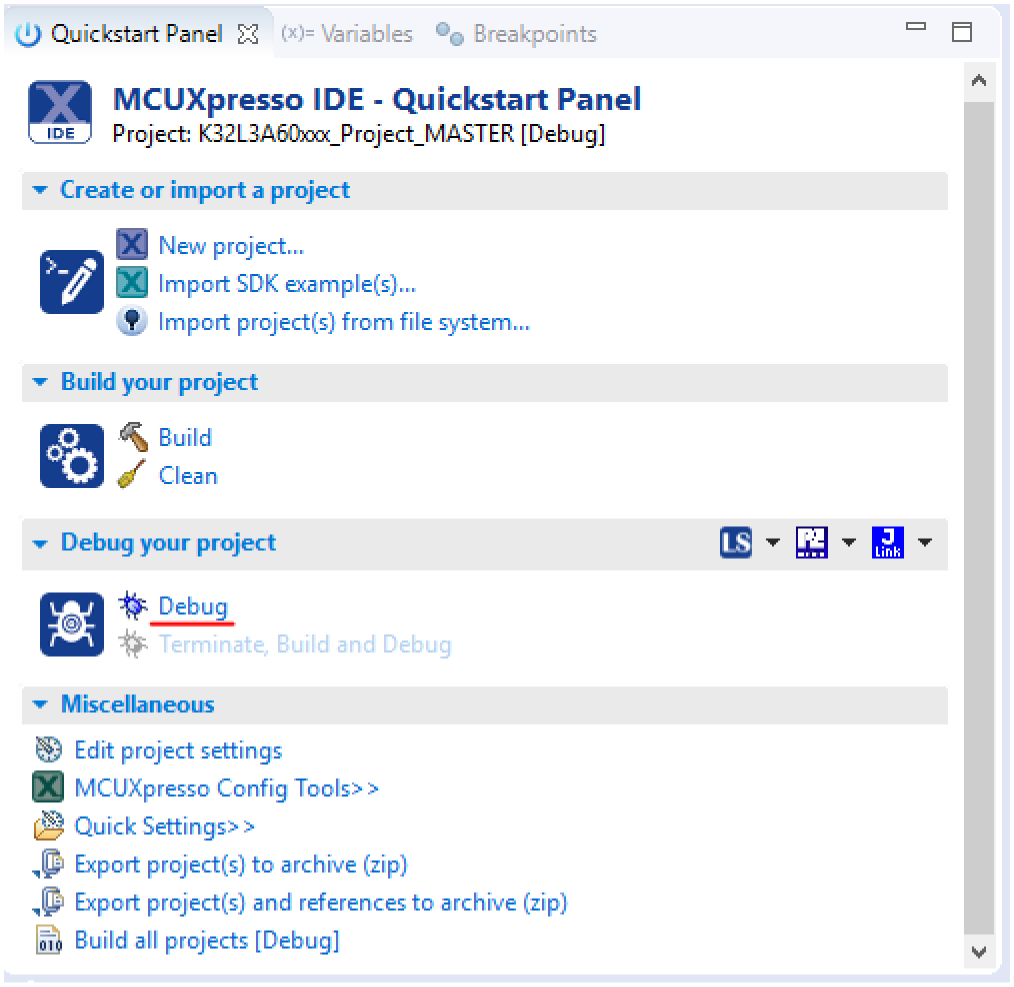
Disconnect behavior: cont

Semihosting support: On

Figure 41. Arm Cortex-M0+ slave attach only settings

6.2 Primary core debugging

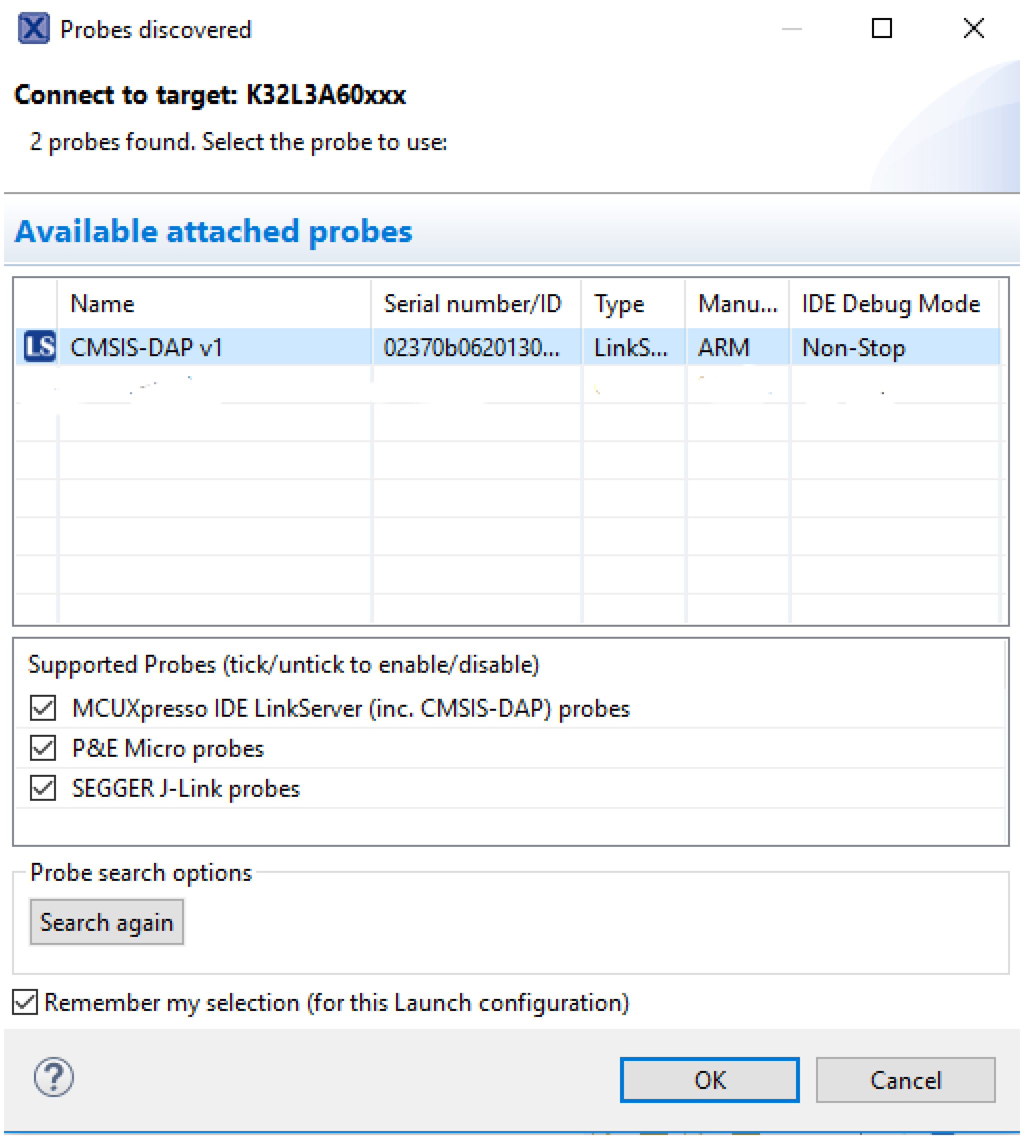
To download and run the multicore application, switch to the Master application project and perform all steps as described in Section 7.3 Run an example application in *MCUXpresso IDE User Guide* (found in `<K32L SDK root>\docs`). These steps are common for both single core applications and the Master side of dual-core applications, to ensure that both sides of the multicore application are properly loaded and started.


Quickstart Panel
(x)= Variables
Breakpoints
IDE

MCUXpresso IDE - Quickstart Panel
Project: K32L3A60xxx_Project_MASTER [Debug]

Create or import a project
New project...
Import SDK example(s)...
Import project(s) from file system...
Build your project
Build
Clean
Debug your project
LS
Link
Debug
Terminate, Build and Debug
Miscellaneous
Edit project settings
MCUXpresso Config Tools>>
Quick Settings>>
Export project(s) to archive (zip)
Export project(s) and references to archive (zip)
Build all projects [Debug]

Figure 42. Debug `K32L3A60xxx_Project_MASTER` case



Probes discovered

Connect to target: K32L3A60xxx

2 probes found. Select the probe to use:

Available attached probes

	Name	Serial number/ID	Type	Manu...	IDE Debug Mode
LS	CMSIS-DAP v1	02370b0620130...	LinkS...	ARM	Non-Stop

Supported Probes (tick/untick to enable/disable)

- [x] MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes
- [x] P&E Micro probes
- [x] SEGGER J-Link probes

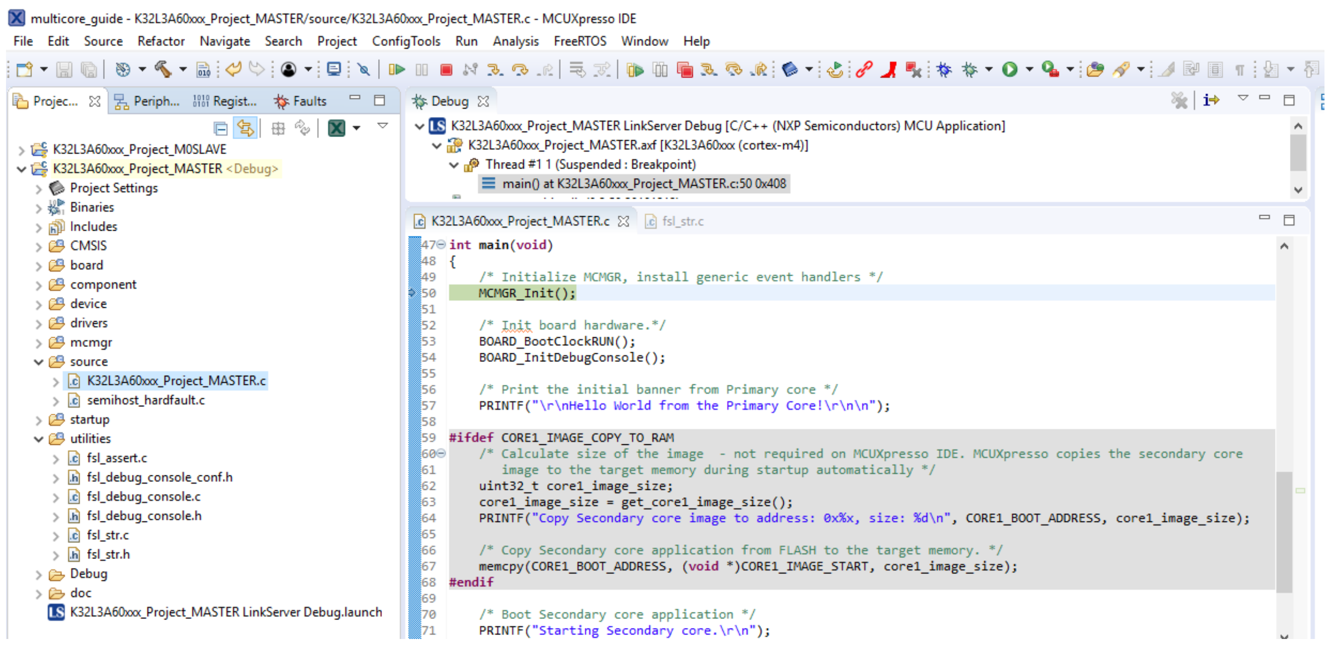
Probe search options

Search again

- [x] Remember my selection (for this Launch configuration)

OK Cancel

Figure 43. Attached probes: debug emulator selection



multicore_guide - K32L3A60xxx_Project_MASTER/source/K32L3A60xxx_Project_MASTER.c - MCUXpresso IDE

File Edit Source Refactor Navigate Search Project ConfigTools Run Analysis FreeRTOS Window Help
Projec... Periph... Regist... Faults
Debug

```
K32L3A60xxx_Project_MOSLAVE
K32L3A60xxx_Project_MASTER <Debug>
  Project Settings
  Binaries
  Includes
  CMSIS
  board
  component
  device
  drivers
  mcmgr
  source
    K32L3A60xxx_Project_MASTER.c
    semihost_hardfault.c
  startup
  utilities
    fsl_assert.c
    fsl_debug_console_conf.h
    fsl_debug_console.c
    fsl_debug_console.h
    fsl_str.c
    fsl_str.h
  Debug
  doc
  K32L3A60xxx_Project_MASTER LinkServer Debug.launch
```

```
K32L3A60xxx_Project_MASTER LinkServer Debug [C/C++ (NXP Semiconductors) MCU Application]
  K32L3A60xxx_Project_MASTER.axf [K32L3A60xxx (cortex-m4)]
    Thread #1 1 (Suspended : Breakpoint)
      main() at K32L3A60xxx_Project_MASTER.c:50 0x408
```

K32L3A60xxx_Project_MASTER.c   fsl_str.c

```
int main(void)
{
    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();

    /* Init board hardware.*/
    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");
```

Figure 44. Stop master core application at `main()` when running debugging

From here, you can continue to debug the application as if it were a single core device. Or you could continue on to learn how to debug the secondary core at the same time.

6.3 Secondary core debugging

It is possible to debug both sides of the multicore application in parallel by attaching to the running application of the slave core. After creating and running the debug session for the master core, perform the same steps for the slave core application. Highlight the multicore slave project in the Project Explorer and click **Debug** in the **Quickstart Panel**, as shown in Figure 45.

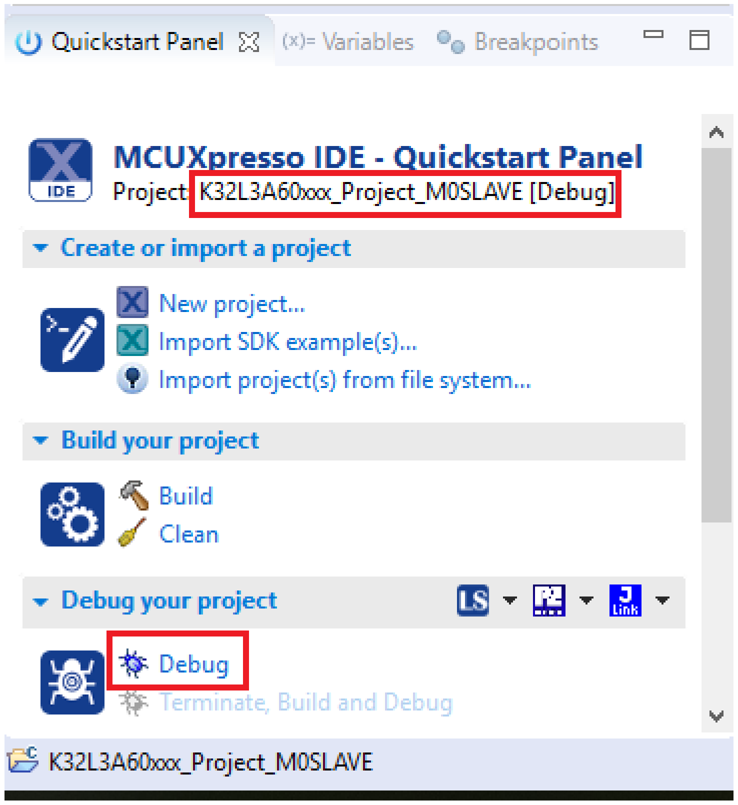
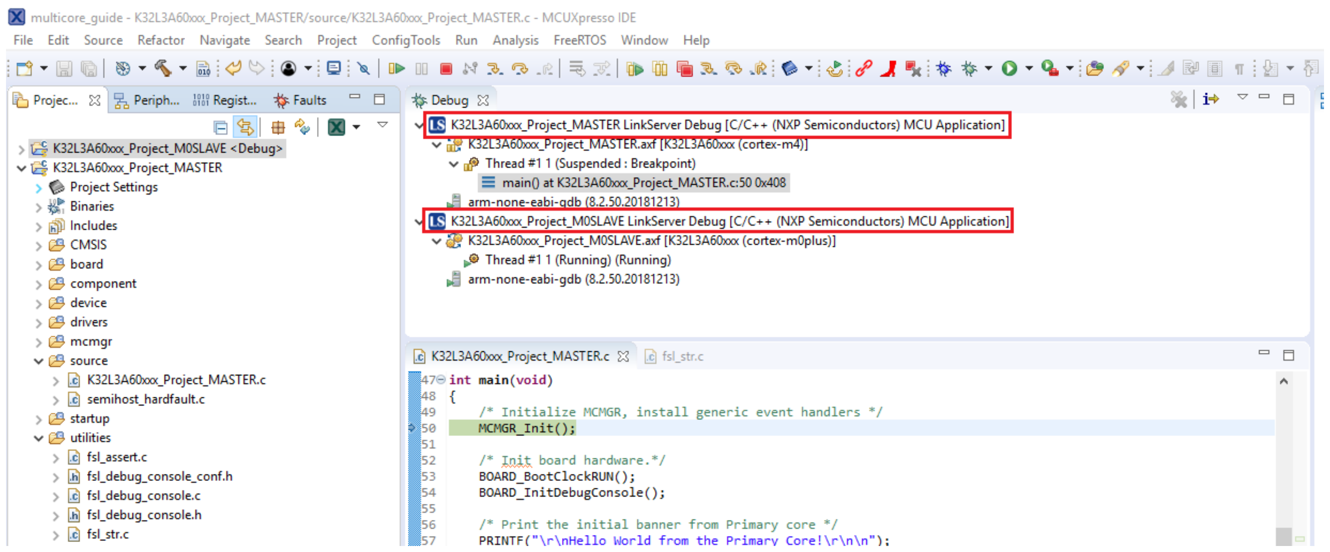

Quickstart Panel
(x)= Variables
Breakpoints
MCUXpresso IDE - Quickstart Panel
Project: K32L3A60xxx_Project_M0SLAVE [Debug]
Create or import a project
New project...
Import SDK example(s)...
Import project(s) from file system...
Build your project
Build
Clean
Debug your project
Debug
Terminate, Build and Debug
K32L3A60xxx_Project_M0SLAVE

Figure 45. Debug `K32L3A60xxx_Project_M0SLAVE` case

After initializing the slave debug session, you can see two separate threads that you can control from the single IDE, as shown in Figure 46. You can synchronize suspension/resumption of both cores using **Suspend All Debug sessions** and **Resume All Debug sessions** controls.


multicore_guide - K32L3A60xxx_Project_MASTER/source/K32L3A60xxx_Project_MASTER.c - MCUXpresso IDE
File Edit Source Refactor Navigate Search Project ConfigTools Run Analysis FreeRTOS Window Help
Projec... Periph... Regist... Faults
K32L3A60xxx_Project_M0SLAVE <Debug>
K32L3A60xxx_Project_MASTER
Project Settings
Binaries
Includes
CMSIS
board
component
device
drivers
mcmgr
source
K32L3A60xxx_Project_MASTER.c
semihost_hardfault.c
startup
utilities
fsl_assert.c
fsl_debug_console_conf.h
fsl_debug_console.c
fsl_debug_console.h
fsl_str.c
Debug
K32L3A60xxx_Project_MASTER.c
fsl_str.c

```
K32L3A60xxx_Project_MASTER LinkServer Debug [C/C++ (NXP Semiconductors) MCU Application]
  K32L3A60xxx_Project_MASTER.axf [K32L3A60xxx (cortex-m4)]
    Thread #1 1 (Suspended : Breakpoint)
      main() at K32L3A60xxx_Project_MASTER.c:50 0x408
  arm-none-eabi-gdb (8.2.50.20181213)
K32L3A60xxx_Project_M0SLAVE LinkServer Debug [C/C++ (NXP Semiconductors) MCU Application]
  K32L3A60xxx_Project_M0SLAVE.axf [K32L3A60xxx (cortex-m0plus)]
    Thread #1 1 (Running) (Running)
  arm-none-eabi-gdb (8.2.50.20181213)
```

```
int main(void)
{
    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();

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

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

Figure 46. Two opened debug sessions

Now, the two debug sessions are opened, and the debug controls can be used for both debug sessions depending on the debug session selection. At this point, it is possible to suspend and resume individual cores independently. It is also possible to make synchronous suspension and resumption of both cores with either following methods:

- select both opened debug sessions (multiple selection) and click **Suspend**/**Resume** (highlighted in Figure 47).

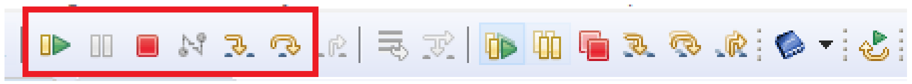


Figure 47. MCUXpress IDE single thread controls

- use the **Suspend All Debug sessions** and **Resume All Debug sessions** buttons (highlighted in Figure 48).

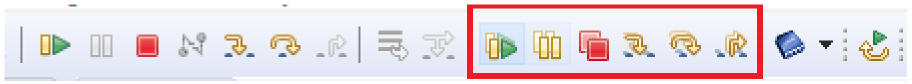


Figure 48. MCUXpresso IDE multicore synchronous controls

7 Multicore code

The project settings for the primary core and secondary core are only part of the dual-core project aspect. The source code for each project must be written to take into account the operations of the other core. In general, the primary core is expected to configure the common clocks, peripherals, and memory. In doing this, the secondary core will have to do minimal work once it is released from reset, but there is absolutely nothing preventing the secondary core from setting up its own areas.

As there is no mechanism for the secondary core to release on its own, the primary core must release the secondary core from reset with any of the following methods:

1. using the MCMGR (Multicore Manager) high-level drivers
2. using the MU (Messaging Unit) unit low-level drivers
3. using raw register accesses

The MCMGR high-level drivers provide flexibility and control for complex applications and can also register and trigger events between the cores. The MU unit low-level drivers simply give you access to the registers in a portable manner and would likely only be used if you have a simple program that needs to perform a limited number of actions. When using this option, it's important that the user understands what the target application is doing and how the interaction of the cores is happening. Raw register accesses should only be used for simple programs or when the user needs a custom function, driver, or has an advanced level of understanding of the MU and inter-core interaction. This application note only focuses on the use of the MCMGR high-level drivers as this is recommended. The MCMGR code is part of the MCUXpresso SDK package (see the `middleware/multicore/mcmgr` folder).

The project (both master and slave) needs the following files:

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

NOTE

The files are already added to the `hello_world multicore` project, but you may need to add them manually to your own project(s).

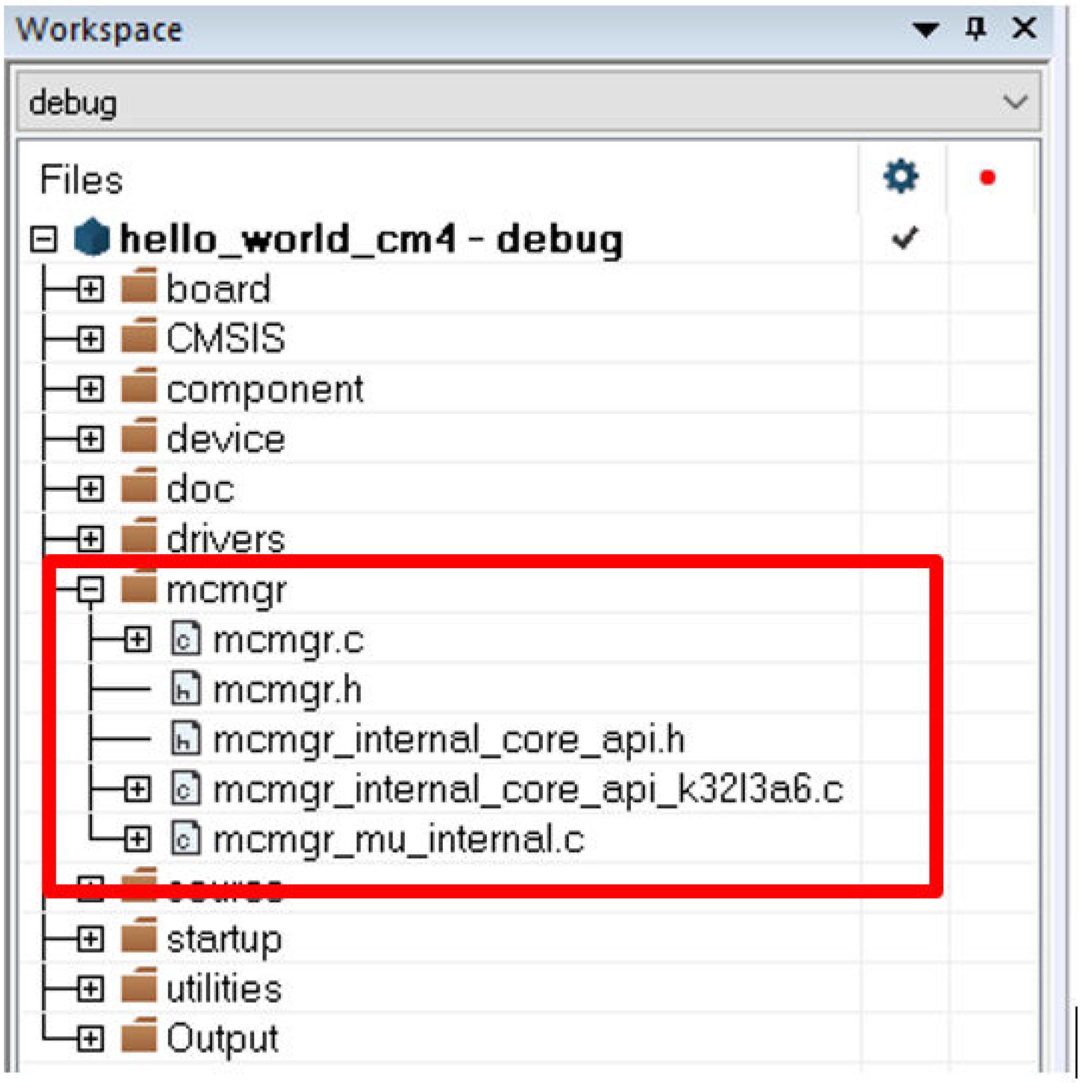
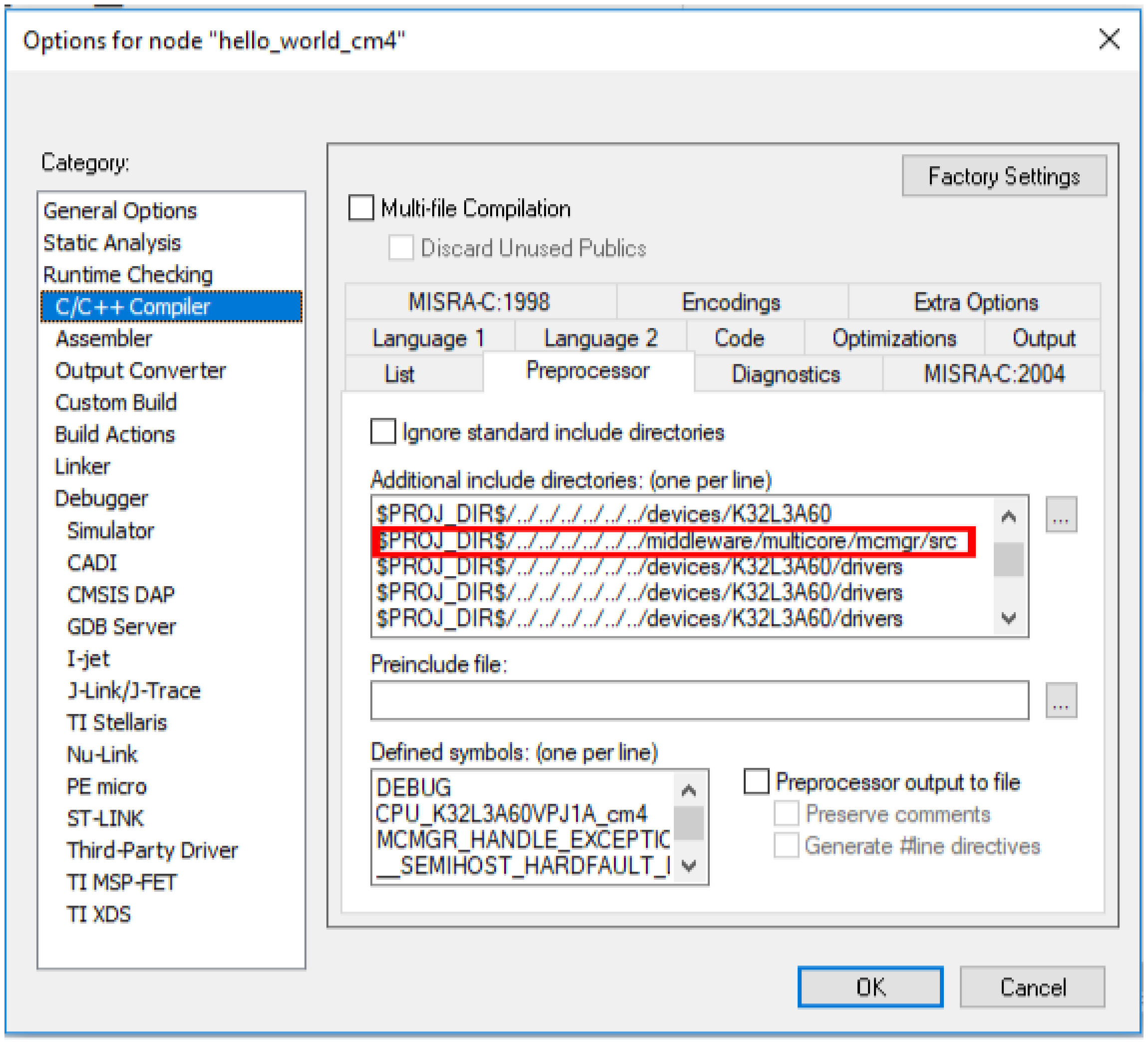


Figure 49. MCMGR files included in multicore projects

In addition, the path to the header files must be made known to the compiler. The path shown in Figure 50 should be included in your preprocessor search path.

NOTE

The path is already added to the `hello_world` multicore project, but you may need to add it manually for project(s) you are converting.



Additional include directories: (one per line)

```
$PROJ_DIR$/../../../../../../devices/K32L3A60
$PROJ_DIR$/../../../../../../middleware/multicore/mcmgr/src
$PROJ_DIR$/../../../../../../devices/K32L3A60/drivers
$PROJ_DIR$/../../../../../../devices/K32L3A60/drivers
$PROJ_DIR$/../../../../../../devices/K32L3A60/drivers
```

Preinclude file:

Defined symbols: (one per line)

```
DEBUG
CPU_K32L3A60VPJ1A_cm4
MCMGR_HANDLE_EXCEPTIC
__SEMIHOST_HARDFAULT_I
```

Figure 50. File path to be added to Preprocessor search path for Multicore Manager files

In addition, the MU drivers will also be needed. They are `fsl_mu.c` and `fsl_mu.h`.

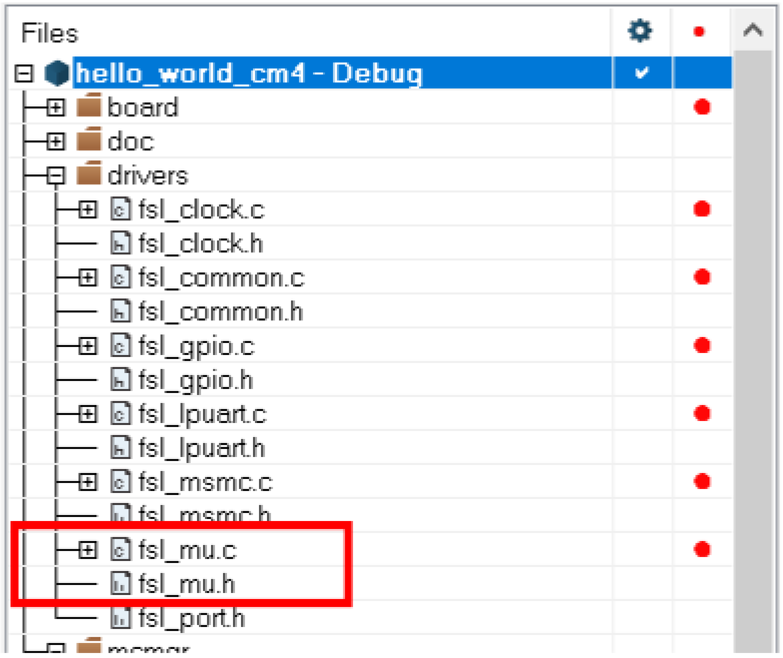
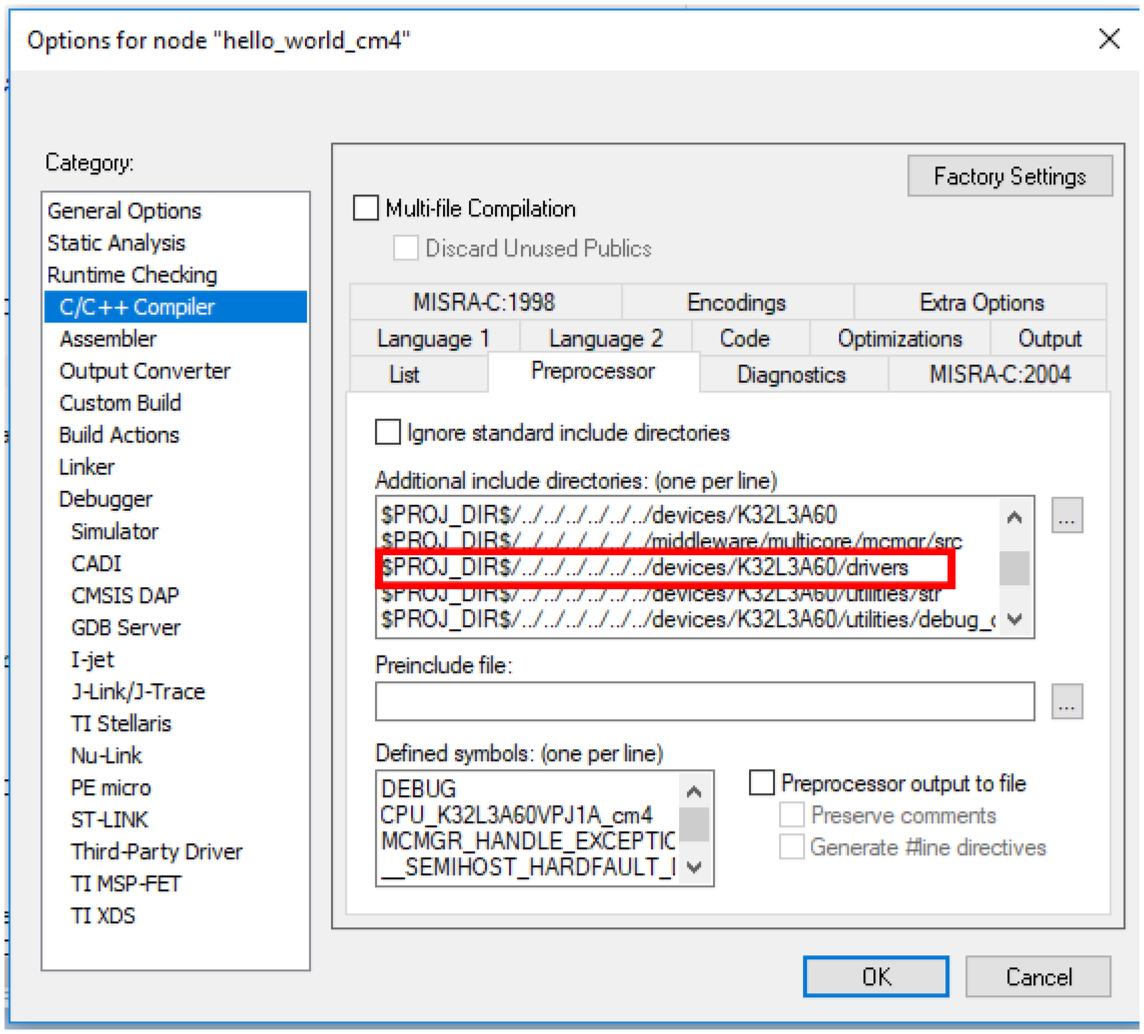


Figure 51. MU drivers added to the `hello_world` multicore project

NOTE

The preprocessor search path should include the file path to these driver files.



Additional include directories: (one per line)

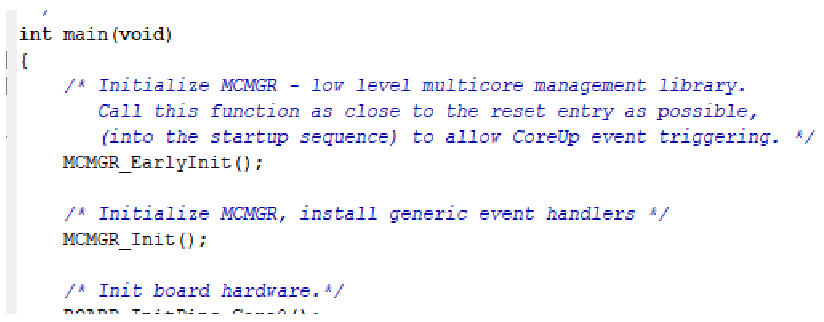
```
$PROJ_DIR$/../../../../../../devices/K32L3A60
$PROJ_DIR$/../../../../../../middleware/multicore/mcmgr/src
$PROJ_DIR$/../../../../../../devices/K32L3A60/drivers
$PROJ_DIR$/../../../../../../devices/K32L3A60/utilities/str
$PROJ_DIR$/../../../../../../devices/K32L3A60/utilities/debug_c
```

Defined symbols: (one per line)

```
DEBUG
CPU_K32L3A60VPJ1A_cm4
MCMGR_HANDLE_EXCEPTIC
__SEMIHOST_HARDFAULT_I
```

Figure 52. MU driver file path

To use the MCMGR high-level drivers, there are two initialization functions that need to be called. These calls apply to both master and slave project(s), as shown in Figure 53.

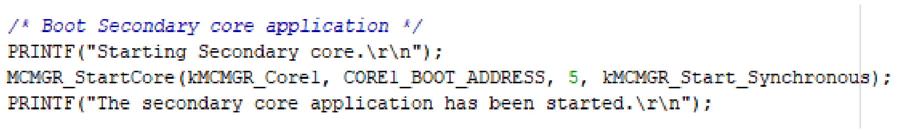


```
int main(void)
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

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

    /* Init board hardware.*/
```

Figure 53. MCMGR initialization function calls

The `MCMGR_EarlyInit` function must be called as close to the reset entry as possible. This function enables the clock gate to the MU unit and triggers the core up event that is propagated to the counterpart core. The `MCMGR_Init` function must be called from main and enables all of the other MCMGR API to be used.

Once these functions have been called, the master software should then call the `MCMGR_StartCore` function to start the secondary core. This function call is as shown in Figure 54.



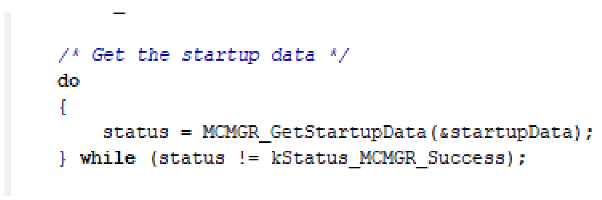
```
/* Boot Secondary core application */
PRINTF("Starting Secondary core.\r\n");
MCMGR_StartCore(kMCMGR_Core1, CORE1_BOOT_ADDRESS, 5, kMCMGR_Start_Synchronous);
PRINTF("The secondary core application has been started.\r\n");
```

Figure 54. MCMGR function to start the secondary core

The `MCMGR_StartCore` function requires the following arguments:

- `mcmgr_core_t coreNum`: This should be either `kMCMGR_Core1` or `MCMGR_Core0`. **Core0** is the Arm Cortex-M4 core and **Core1** is the Arm Cortex-M0+ core.
- `void *bootAddress`: This is a pointer to the beginning of the secondary core's application. The only valid options are the start of the secondary core's flash memory space or the RAM memory space.
- `uint32_t startupdata`: This is some user/application data (variable, or array, or function) to be passed from the primary core to the secondary core during the startup. For instance, it can be leveraged in case of rpmsg inter-core communication for passing the shared memory base address from the master side to the remote side.
- `mcmgr_start_mode_t` mode: This is the mode with which you want the cores to start. Valid options are `kMCMGR_Start_Synchronous` or `kMCMGR_Start_Asynchronous`. If `kMCMGR_Start_Synchronous` is used, the primary core will start the secondary core and then wait for an event from the secondary core before continuing operation. Otherwise, the primary core will start the secondary core and immediately continue operation.

The slave core may either just start and do nothing, or retrieve any startup data that it was passed. Figure 55 shows an example.



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

Figure 55. Slave core implementation retrieving startup data

How the slave core processes this data depends on the specifics of your application.

8 Conclusion

This application note has shown, through a simple example, how multicore projects are created, compiled, and debugged. The SDK package for the K32L series devices contains many multicore examples to aid in your project development. These examples include Embedded Remote Procedure Call examples, Remote Processor Messaging examples, demonstrations of the SDK's Multicore Manager drivers, and Resource Domain Manager examples. More information on all of these resources can be found at `<KSDK_ROOT>/multicore` in the respective resources folder.

Date of release: 01/2020

Document identifier: AN12673