AN12124

LPC5411x dual core debugging guide

Rev.1.0 — 8 March 2018 Application note

Document information

Info	Content
Keywords	LPC5411x, LPC54102, LPC54114, dual core, debug
Abstract	This application note introduces the basic dual core development process using three integrated IDEs. They are MCUXpresso IDE, IAR and Keil. The focus will be on the debugging process and techniques within these IDEs.

Revision history

Rev	Date	Description
1.0	20180308	Initial version

Contact information

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

1. Introduction

The LPC541xx is a power efficient and high performance MCU series. The system frequency can be up to 100 MHz. Both LPC54102 and LPC54114 integrate two cores, Cortex-M4 and Cortex-M0+. The asymmetric architecture provides flexible options for the user in complex design applications with high performance. This application note will help the user to tackle the challenges during dual core development process and debugging. It focuses on dual core debugging techniques and the support from three commonly used IDEs - MCUXpress IDE, IAR and Keil. LPCXpresso54114 board and the related SDK is used as an example target platform to discuss the topic. For basic dual core introduction and its development process, see AN12123 - LPC541xx dual core starting guide.

Since the topic is based on advanced debugging, user should have a basic dual core knowledge and debugging skills. Relative user guide of IDE can be used as a starting reference. The application note introduces the debugging interface SWD and later will analyze the booting process of the two cores. At the end, support on dual core debugging of IDE is discussed.

MCUXpresso IDE is the NXP in-house development tool dedicated for supporting MCU series. It is a free edition with no code size limit, and can be downloaded from NXP website: MCUXpresso IDE.

IAR is also a professional MCU development tool provided by IAR systems. License is required for the usage and it can be downloaded from the IAR website. A 30 days free trail with 32 kB code size limit is available for the user at IAR .

ARM provides Keil MDK for Cortex and ARM devices development. Evaluation edition is available for code size below 32 kB and can be downloaded from Keil website: Keil MDK.

1.1 Folder structure

LPC54114 SDK projects are used as examples to elaborate the dual core debugging topic. This application note is available with a zip-folder containing three project files that include two “hello world” example projects. One is for Cortex-M4 master project and the other is for Cortex-M0+ slave project. See Fig 1 for the package contents.

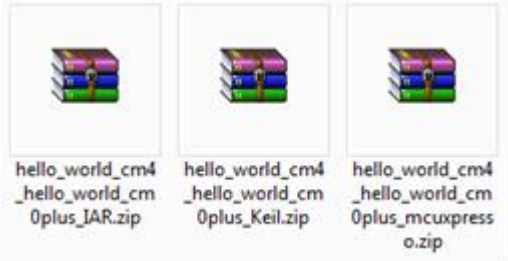


Fig 1. AN package contents

File name	Description
hello_world_cm4_hello_world_cm0plus_IAR	IAR dual core example file containing Cortex-M4 and Cortex-M0+ “hello world” projects. Unzip it and open the projects with IAR
hello_world_cm4_hello_world_cm0plus_Keil	Keil dual core example file containing Cortex-M4 and Cortex-M0+ “hello world” projects. Unzip it and open the projects with Keil
hello_world_cm4_hello_world_cm0plus_mcuxpresso	MCUXpresso IDE dual core example file containing Cortex-M4 and Cortex-M0+ “hello world” projects. Import the archive file with MCUXPresso IDE

2. SWD interface

Serial Wire Debug (SWD) is a simple two-wire interface used for ARM core debugging. The physical layer of SWD consists of two lines:

- SWDIO: a bidirectional data line
- SWCLK: a clock driven by the host

LPC541xx uses multiple access port to implement dual core debugging capabilities. If the IDE and debugger supports this feature, both master and slave projects can be debugged simultaneously. See Fig 2 for the debug access port architecture of the LPC541xx MCU. It has two AHB-AP connections, Cortex-M4 AP and Cortex-M0+ AP. For simplicity, the JTAG-DP is not included, which cannot be used for debugging for LPC541xx family.

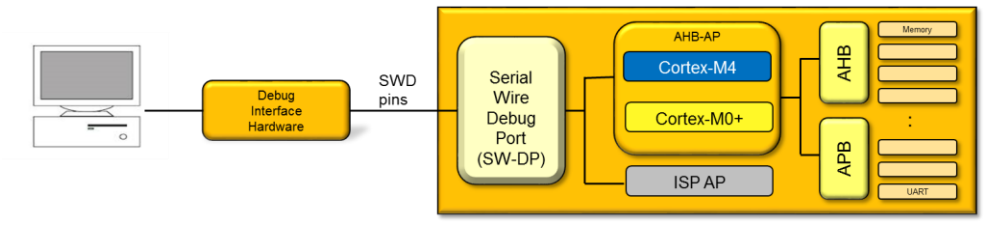


Fig 2. LPC541xx debug access port architecture

3. Dual core booting analysis

LPC541xx family has an asymmetric dual core architecture integrating both Cortex-M4 and Cortex-M0+. It is necessary to understand and setup the booting behavior of the two cores. Both will boot from default reset interrupt vector when powered ON. The application should configure and setup the booting sequence, which is usually written in assembly language in the startup file. Because the reset interrupt service routine should be executed by both the cores, it should be coded with set of instructions common for both the cores.

According to the dual core implementation of LPC541xx, there are master and slave roles for the asymmetric architecture. Upon power ON, Cortex-M4 is the default master and Cortex-M0+ is the slave. Master has the ability to enable or reset the slave core.

Slave core goes to sleep directly upon powering up and waits for the master to set up a proper booting environment for it and boot again.

Details of the LPC541xx dual core implementation is available in Chapterxx of AN12123 <LPC541xx dual core starting guide>.

The analysis of LPC541xx series MCU startup file for MCUXpress IDE, IAR and Keil is discussed in following sections.

3.1 MCUXpresso IDE startup file

Following Fig 3 shows the code snippet for ResetISR () function from the MCUXpresso IDE startup file:

```
void ResetISR(void) {
    asm volatile(
        ".set   cpu_ctrl,       0x40000800\t\n"
        ".set   coproc_boot,    0x40000804\t\n"
        ".set   coproc_stack,   0x40000808\t\n"
        "MOVS   R5, #1\t\n"
        "LDR    R0, =0xE000ED00\t\n"
        "LDR    R1, [R0]\t\n"           // READ CPUID register
        "LDR    R2,=0x410CC601\t\n"     // CM0 R0p1 identifier
        "EORS   R1,R1,R2\t\n"           // XOR to see if we are C0
        "LDR    R3, =cpu_ctrl\t\n"      // get address of CPU_CTRL
        "LDR    R1,[R3]\t\n"            // read cpu_ctrl reg into R1
        "BEQ.N  cm0_boot\t\n"
    "cm4_boot:\t\n"
        "LDR    R0,=coproc_boot\t\n"    // coproc boot address
        "LDR    R0,[R0]\t\n"            // get address to branch to
        "MOVS   R0,R0\t\n"              // Check if 0
        "BEQ.N  check_master_m4\t\n"    // if zero in boot reg, we just branch to  real reset
        "BX     R0\t\n"                 // otherwise, we branch to boot address
    "commonboot:\t\n"
        "LDR    R0, =ResetISR2\t\n"     // Jump to 'real' reset handler
        "BX     R0\t\n"
    "cm0_boot:\t\n"
        "LDR    R0,=coproc_boot\t\n"    // coproc boot address
        "LDR    R0,[R0]\t\n"            // get address to branch to
        "MOVS   R0,R0\t\n"              // Check if 0
        "BEQ.N  check_master_m0\t\n"    // if zero in boot reg, we just branch to  real reset
        "LDR    R1,=coproc_stack\t\n"   // pickup coprocesor stackpointer (from syscon CPSTACK)
        "LDR    R1,[R1]\t\n"
        "MOV    SP,R1\t\n"
        "BX     R0\t\n"                 // goto boot address
    "check_master_m0:\t\n"
        "ANDS   R1,R1,R5\t\n"           // bit test bit0
        "BEQ.N  commonboot\t\n"         // if we get 0, that means we are masters
        "B.N    goto_sleep_pending_reset\t\n"   // Otherwise, there is no startup vector for slave, so we go to sleep
    "check_master_m4:\t\n"
        "ANDS   R1,R1,R5\t\n"           // bit test bit0
        "BNE.N  commonboot\t\n"         // if we get 1, that means we are masters
    "goto_sleep_pending_reset:\t\n"
        "MOV    SP,R5\t\n"              // load 0x1 into SP so that any stacking (eg on NMI) will not cause us to wakeup
            // and write to uninitialised Stack area (instead it will LOCK us up before we cause damage)
            // this code should only be reached if debugger bypassed ROM or we changed master without giving
            // correct start address, the only way out of this is through a debugger change of SP and PC
    "sleepo:\t\n"
        "WFI\t\n"                       // go to sleep
        "B.N    sleepo\t\n"
    );
}
```

Fig 3. Startup file for MCUXpresso IDE

Fig 4 shows the flow chart for the ResetISR () function.

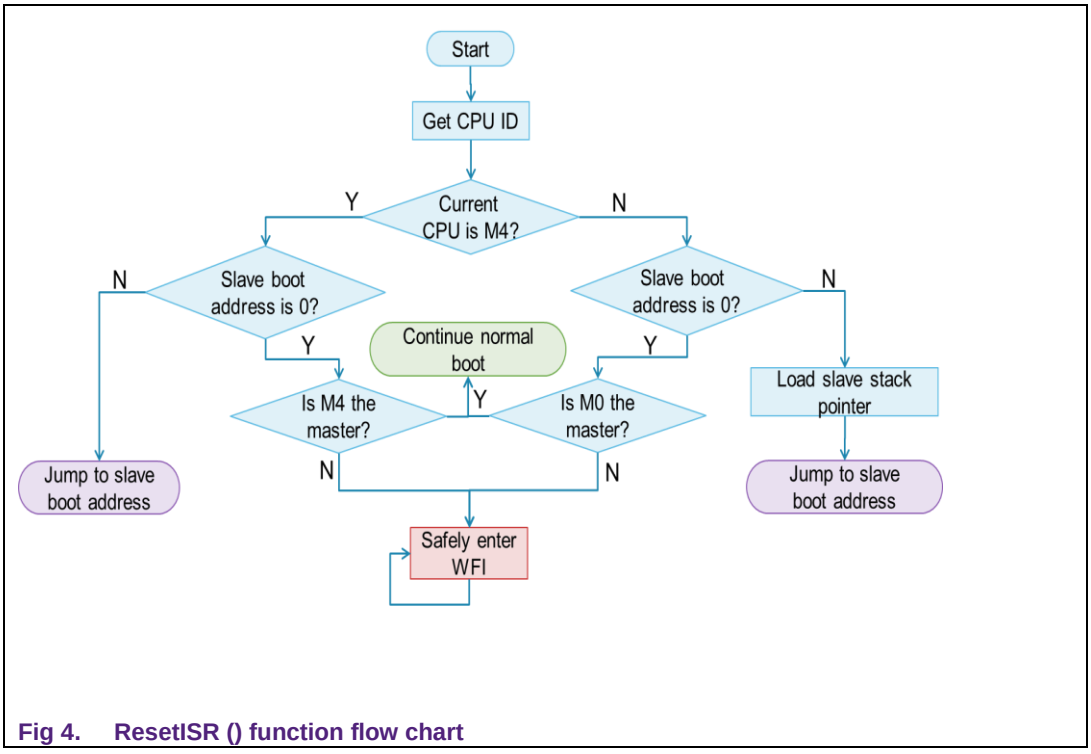


Fig 4. ResetISR () function flow chart

- Cortex-M4 booting up analysis

  If the Cortex-M4 is the default master and the slave boot address is 0 upon reset, it follows normal booting process to execute its application code. Details of slave boot address is available in LPC5410x/LPC5411x user manual: Co-processor boot register.

  The initialization code of the application must set up the Cortex-M0+ slave core booting up environment. It includes copying the slave image into its destination execution address, configuring the slave boot address and stack pointer, and then reset the slave.

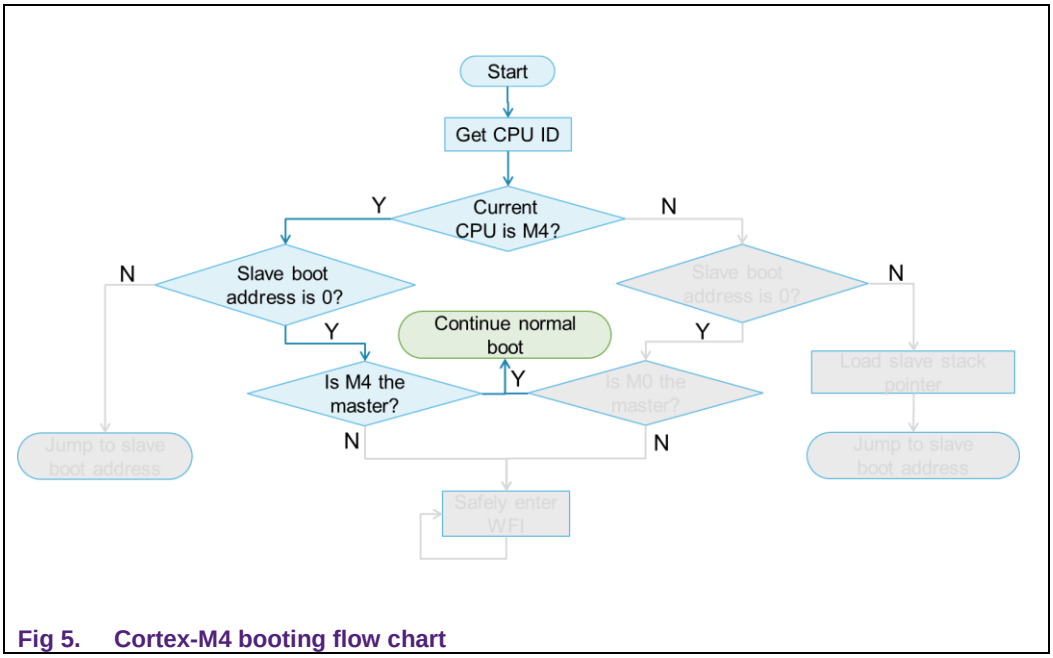


Fig 5. Cortex-M4 booting flow chart

- Cortex-M0+ first booting up analysis

  Because Cortex-M0+ is the default slave and the initial slave boot address is 0, it executes WFI to enter the sleep mode.

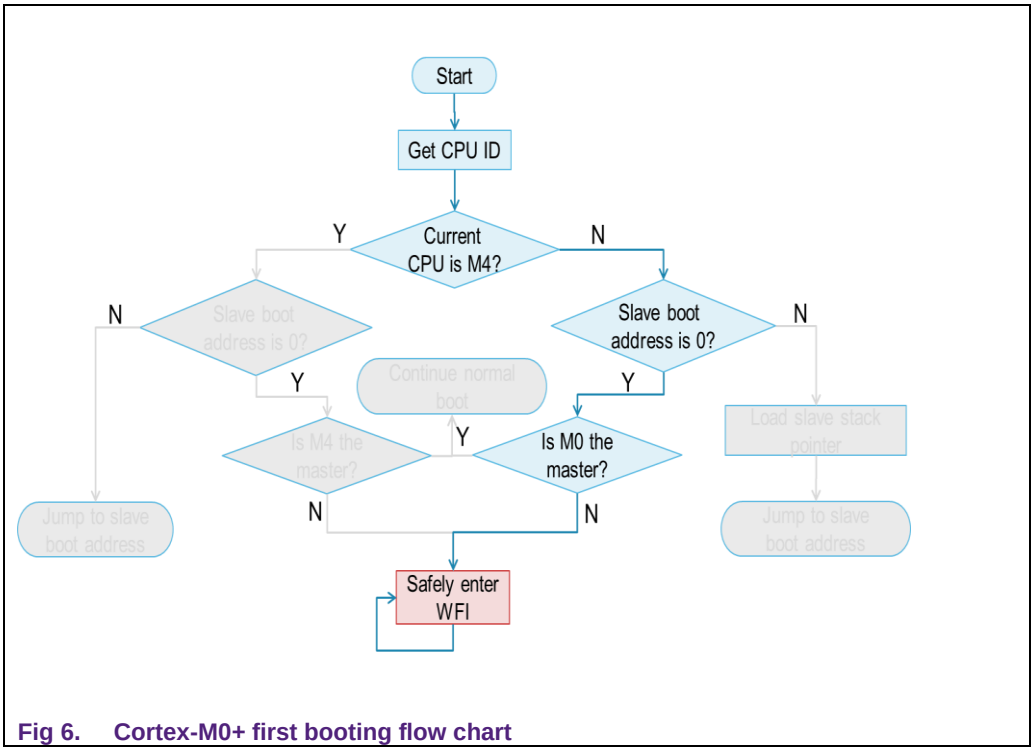


Fig 6. Cortex-M0+ first booting flow chart

- Cortex-M0+ second booting up after master reset

After master core sets up the slave booting up environment and resets the slave, Cortex-M0+ will boot again. This time, the slave boot address will not be 0. Cortex-M0+ will jump to this address and execute its application code.

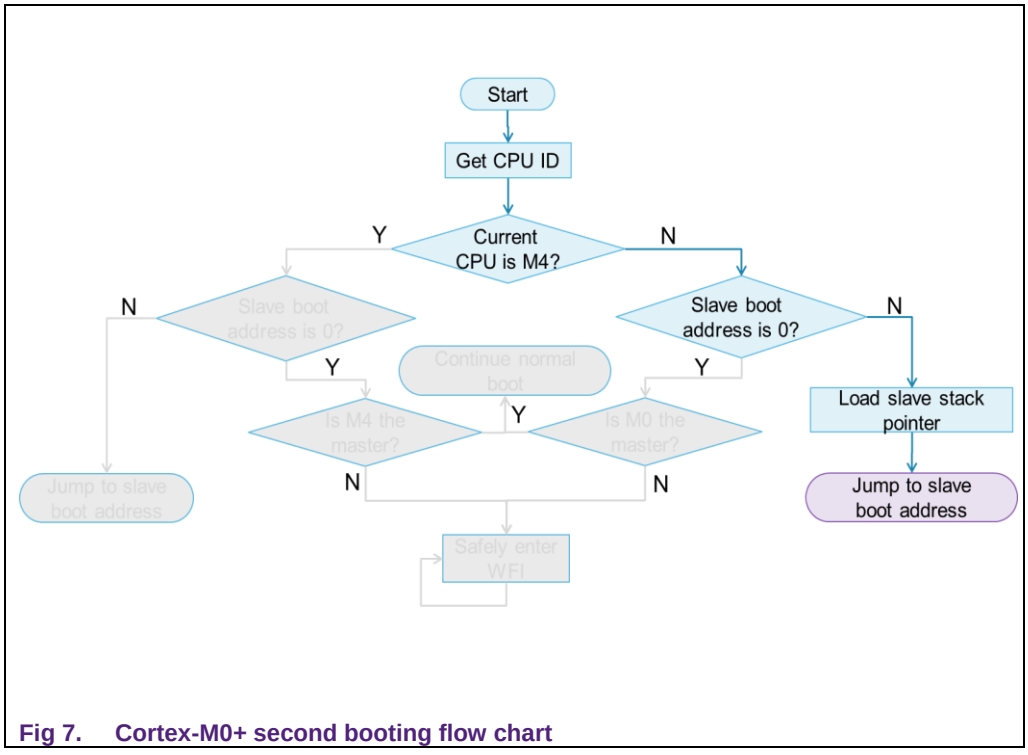


Fig 7. Cortex-M0+ second booting flow chart

3.2 IAR and Keil startup file

The startup file of IAR and Keil is slightly different from the MCUXpresso IDE, but the overall process is the same. See Fig 8 for the reset handler flow chart in the startup file of Keil and IAR.

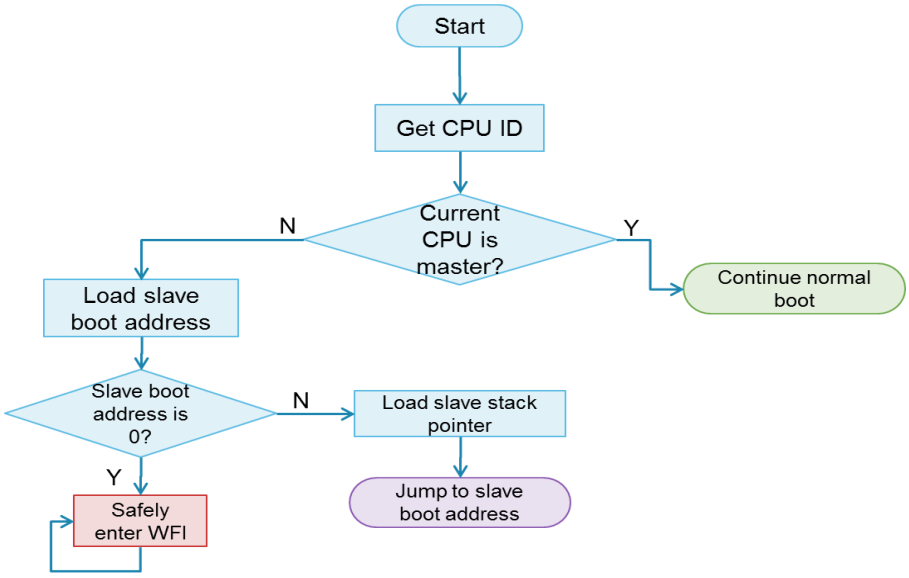


Fig 8. Reset interrupt handler flow chart

Fig 9 shows the Cortex-M4 booting flow chart, which is the default master upon power ON.

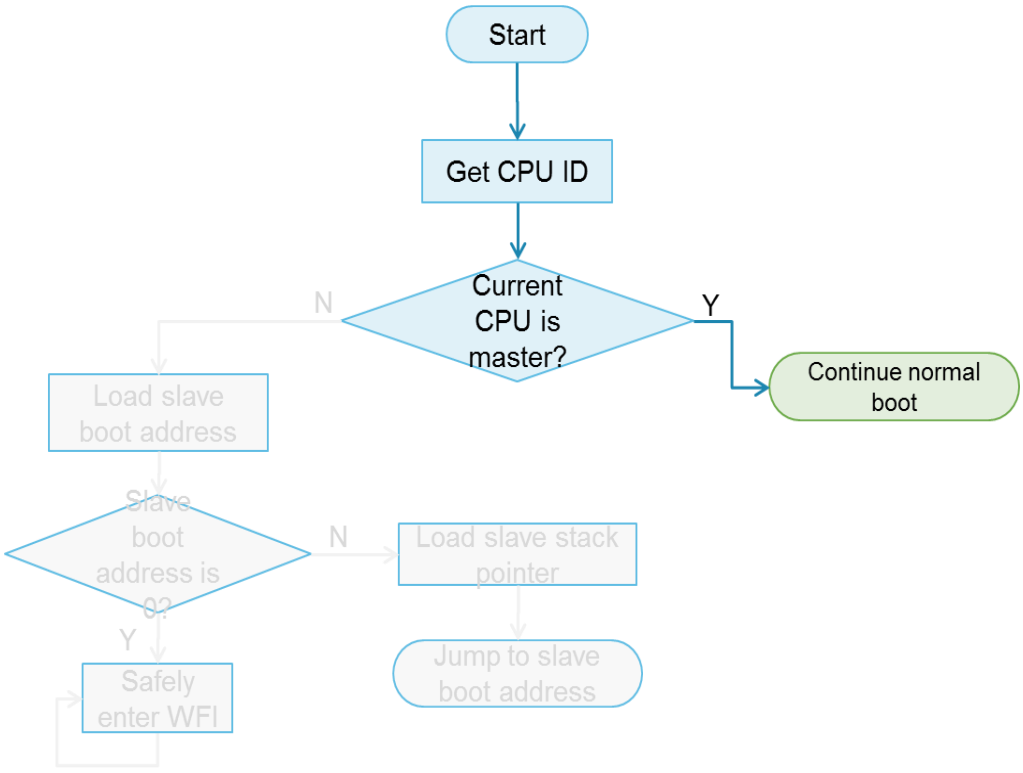


Fig 9. Cortex-M4 booting flow chart

Fig 10 shows the Cortex-M0+ first booting flow chart when the initial startup environment is not configured by the master core.

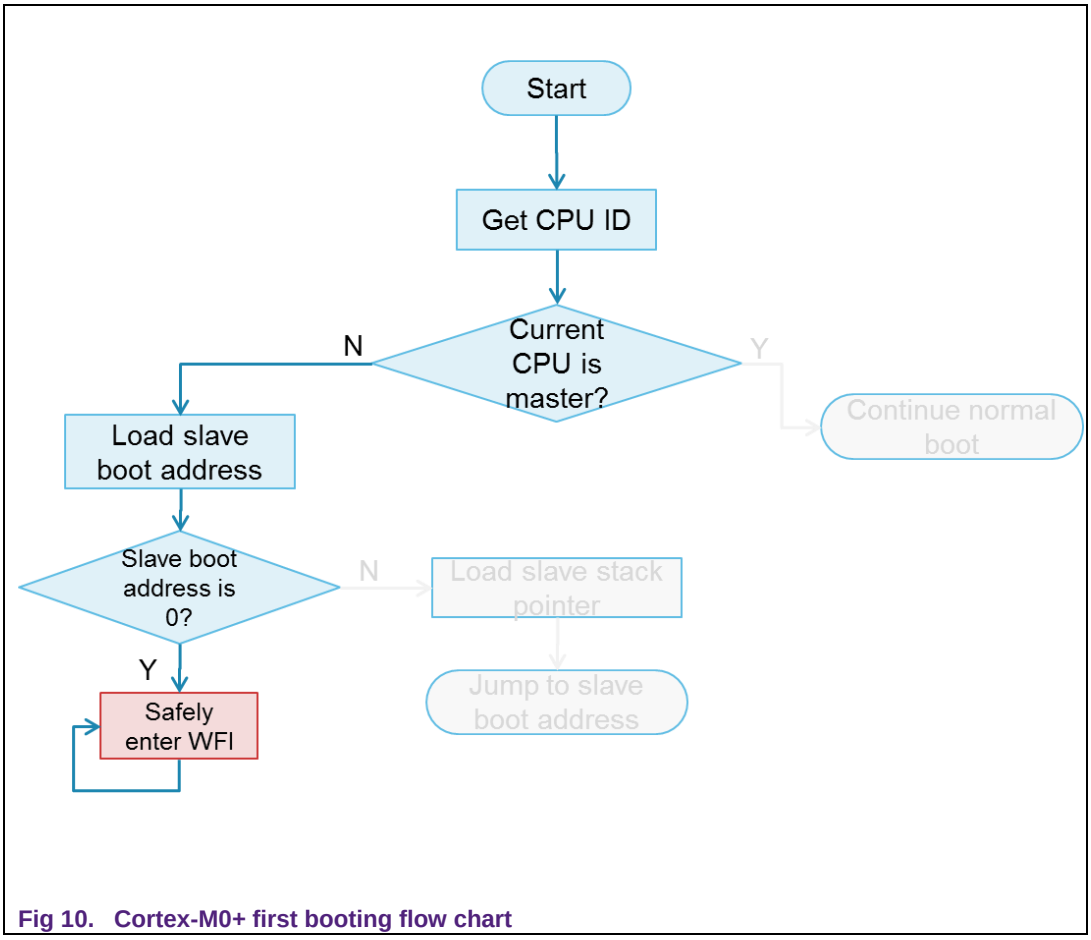


Fig 10. Cortex-M0+ first booting flow chart

Fig 11 shows the Cortex-M0+ second booting flow chart, after master setting up the startup environment and resetting it.

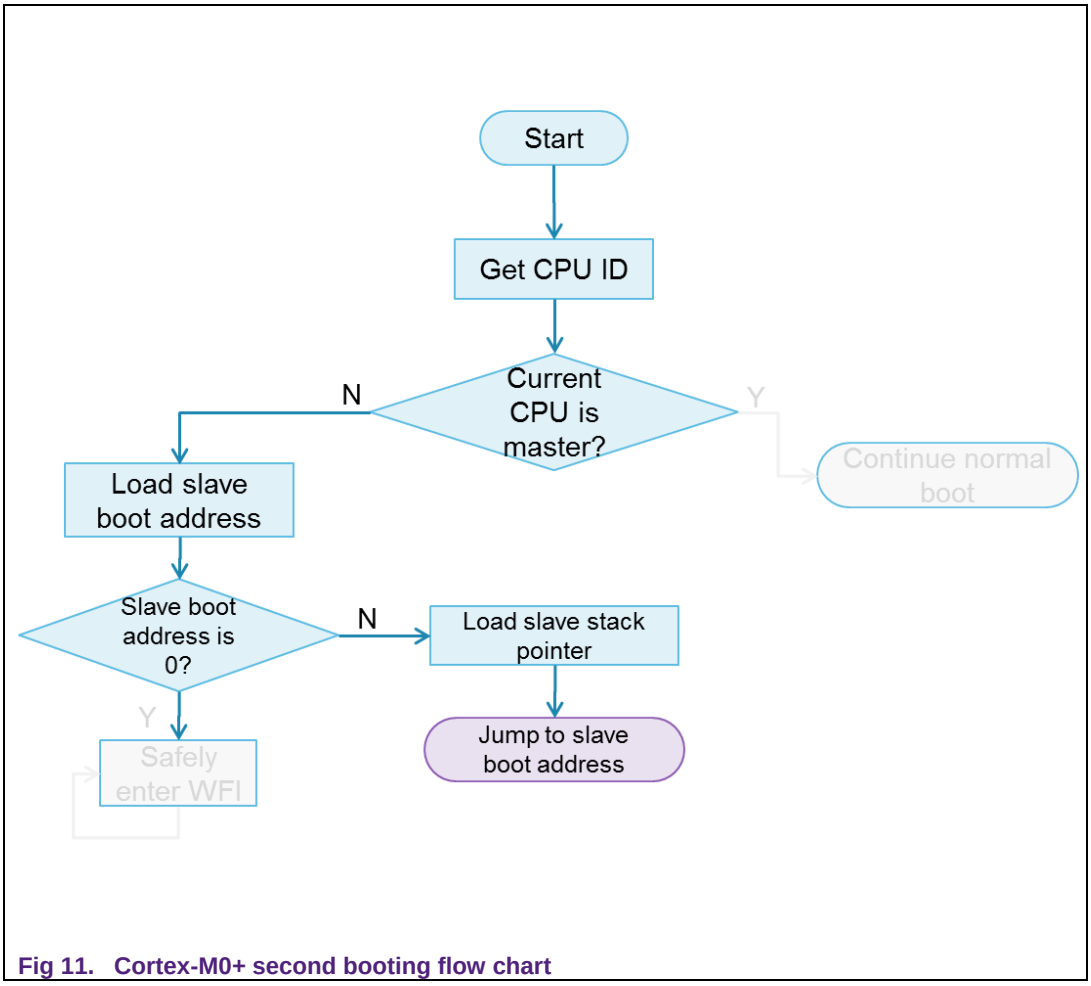


Fig 11. Cortex-M0+ second booting flow chart

4. Debugging with MCUXpresso IDE

MCUXpresso IDE has a good support on dual core configuration and debugging. The slave core project can be linked with the master project by the application configuration, and their image integrated together by the automatic linker script, which can then be downloaded into internal flash memory. The two projects can be debugged together within the same workspace, which is convenient and a useful feature for the developers.

4.1 Project configuration

Open MCUXpresso IDE and import the "hello_world_cm4_hello_world_cm0plus_mcuxpresso.zip" in the AN package. There are two projects in the workspace, "lpcxpresso54114_multicore_examples_hello_world_cm0plus" is the slave Cortex-M0+ project and "lpcxpresso54114_multicore_examples_hello_world_cm4" is the master Cortex-M4 project.

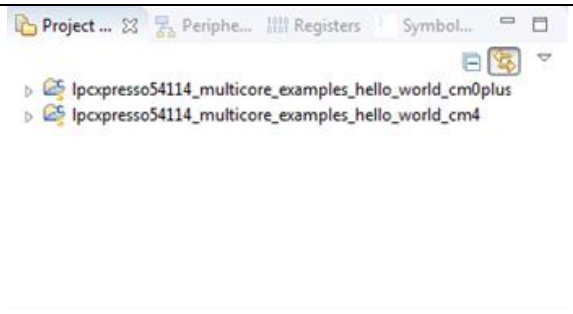


Fig 12. MCUXpresso IDE dual core projects

Because the slave project is already linked together with the master project, compiling the master project will automatically invoke the slave compilation first. The generated image file includes binary of both the projects. So, it is not required to specifically compile the slave project.

Fig 13 shows the memory setting of the Cortex-M4 master project. Flash (or MFlash256) is assigned for master code storage and execution. Slave code is integrated together with the master code, so it will also be downloaded into flash memory space. But the code will later be copied into RAM2 (or Ram1_90) which is the designated code space in slave project and is executed. RAM (or Ram0_64) is the master data space.

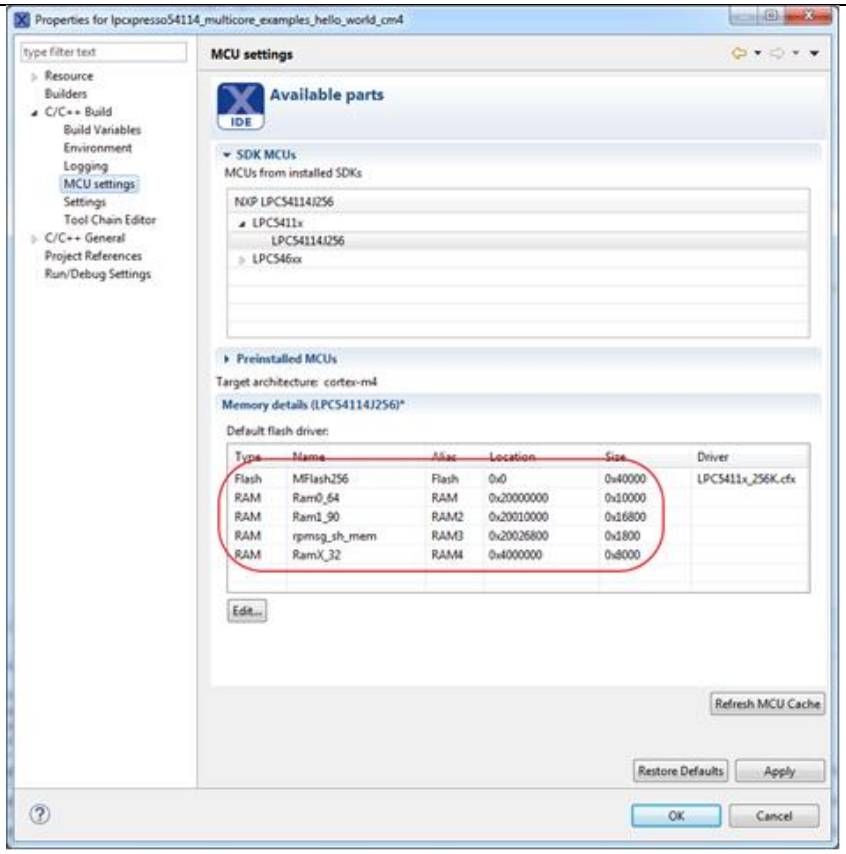


Fig 13. Master project memory map

Fig 14 shows the memory setting of slave Cortex-M0+ project, RAM (or Ram1_90) is assigned as the code and data space.

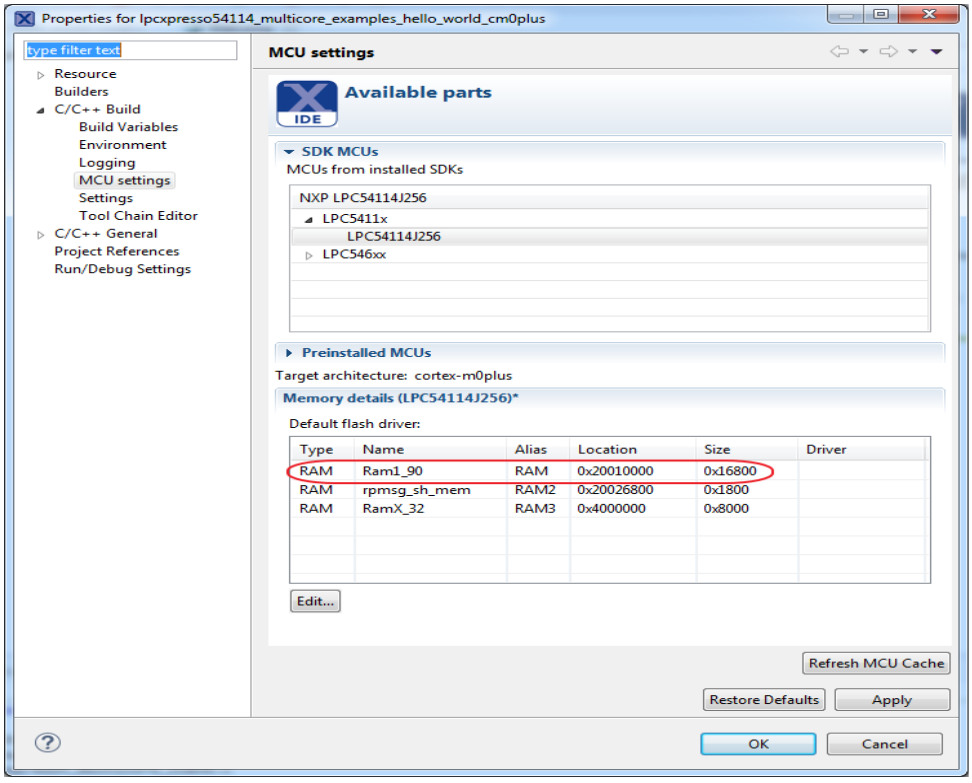


Fig 14. Slave project memory map

The slave project linkage with the master is configured in the *Multicore* option of the master project. The slave project and its execution memory space are specified here. The memory space designated should be in accordance with the memory setting in slave project, which is Ram1_90 in this case. Then the master linker script will treat this area as the slave code and data space, and it will be initialized upon master startup.

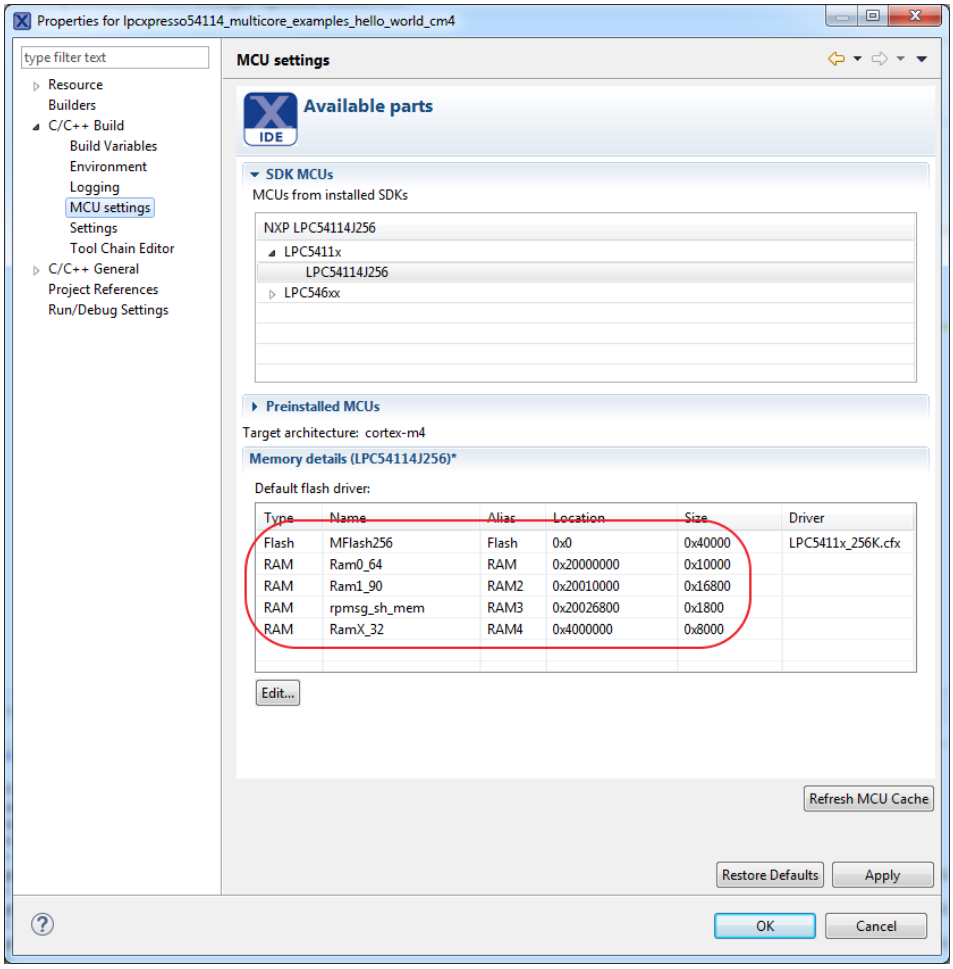
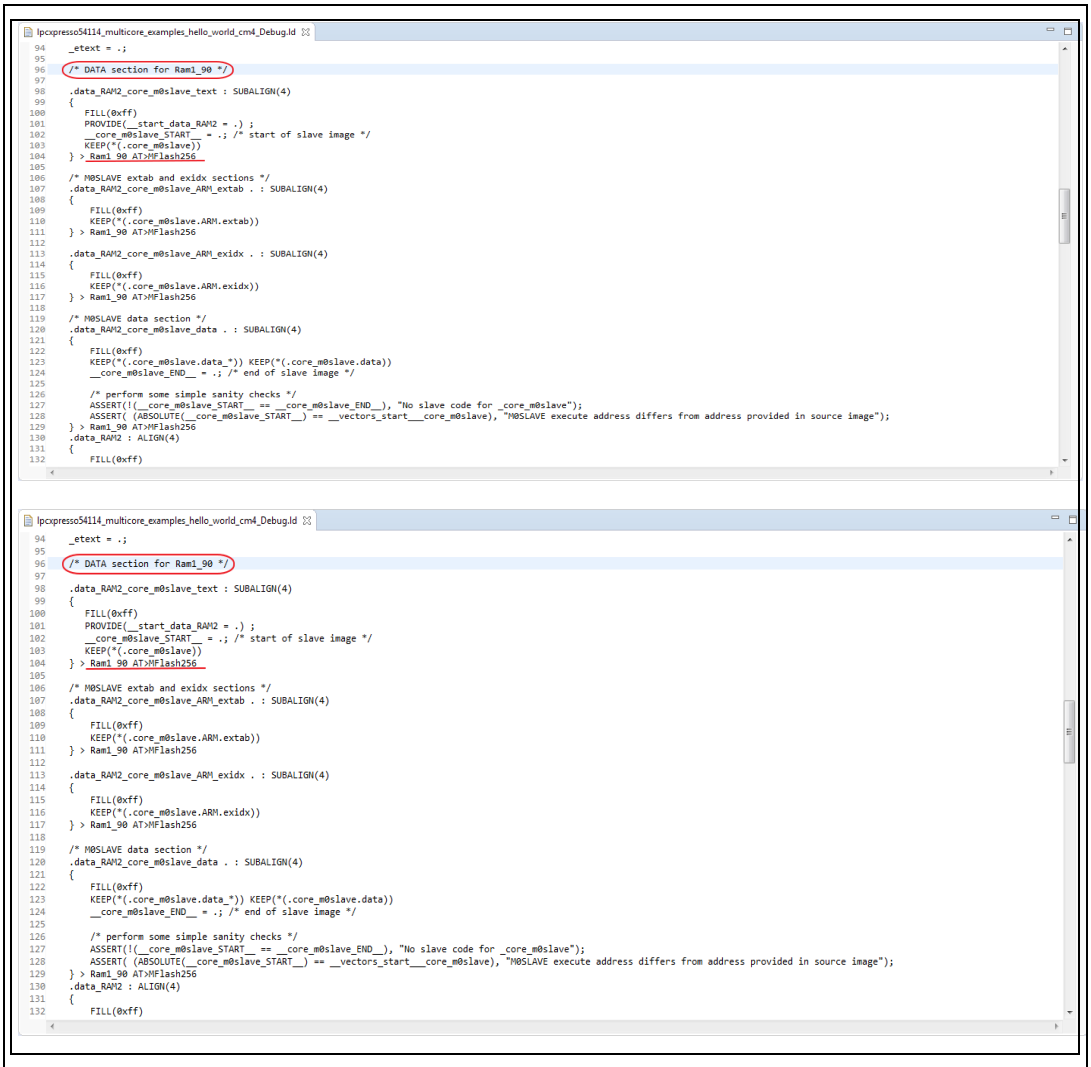


Fig 15. Master project multicore option for slave linkage

Fig 16 shows the master project linker file on slave code and data relocation, which is automatically generated by the linker script.



lpcxpresso54114_multicore_examples_hello_world_cm4_Debug.ld

```
94  _etext = .;
95
96  /* DATA section for Ram1_90 */
97
98  .data_RAM2_core_m0slave_text : SUBALIGN(4)
99  {
100     FILL(0xff)
101     PROVIDE(__start_data_RAM2 = .) ;
102     __core_m0slave_START__ = .; /* start of slave image */
103     KEEP(*(.core_m0slave))
104 } > Ram1_90 AT>MFlash256
105
106 /* M0SLAVE extab and exidx sections */
107 .data_RAM2_core_m0slave_ARM_extab . : SUBALIGN(4)
108 {
109     FILL(0xff)
110     KEEP(*(.core_m0slave.ARM.extab))
111 } > Ram1_90 AT>MFlash256
112
113 .data_RAM2_core_m0slave_ARM_exidx . : SUBALIGN(4)
114 {
115     FILL(0xff)
116     KEEP(*(.core_m0slave.ARM.exidx))
117 } > Ram1_90 AT>MFlash256
118
119 /* M0SLAVE data section */
120 .data_RAM2_core_m0slave_data . : SUBALIGN(4)
121 {
122     FILL(0xff)
123     KEEP(*(.core_m0slave.data_*)) KEEP(*(.core_m0slave.data))
124     __core_m0slave_END__ = .; /* end of slave image */
125
126     /* perform some simple sanity checks */
127     ASSERT(!(__core_m0slave_START__ == __core_m0slave_END__), "No slave code for _core_m0slave");
128     ASSERT( (ABSOLUTE(__core_m0slave_START__) == __vectors_start___core_m0slave), "M0SLAVE execute address differs from address provided in source image");
129 } > Ram1_90 AT>MFlash256
130 .data_RAM2 : ALIGN(4)
131 {
132     FILL(0xff)
```

lpcxpresso54114_multicore_examples_hello_world_cm4_Debug.ld

```
94  _etext = .;
95
96  /* DATA section for Ram1_90 */
97
98  .data_RAM2_core_m0slave_text : SUBALIGN(4)
99  {
100     FILL(0xff)
101     PROVIDE(__start_data_RAM2 = .) ;
102     __core_m0slave_START__ = .; /* start of slave image */
103     KEEP(*(.core_m0slave))
104 } > Ram1_90 AT>MFlash256
105
106 /* M0SLAVE extab and exidx sections */
107 .data_RAM2_core_m0slave_ARM_extab . : SUBALIGN(4)
108 {
109     FILL(0xff)
110     KEEP(*(.core_m0slave.ARM.extab))
111 } > Ram1_90 AT>MFlash256
112
113 .data_RAM2_core_m0slave_ARM_exidx . : SUBALIGN(4)
114 {
115     FILL(0xff)
116     KEEP(*(.core_m0slave.ARM.exidx))
117 } > Ram1_90 AT>MFlash256
118
119 /* M0SLAVE data section */
120 .data_RAM2_core_m0slave_data . : SUBALIGN(4)
121 {
122     FILL(0xff)
123     KEEP(*(.core_m0slave.data_*)) KEEP(*(.core_m0slave.data))
124     __core_m0slave_END__ = .; /* end of slave image */
125
126     /* perform some simple sanity checks */
127     ASSERT(!(__core_m0slave_START__ == __core_m0slave_END__), "No slave code for _core_m0slave");
128     ASSERT( (ABSOLUTE(__core_m0slave_START__) == __vectors_start___core_m0slave), "M0SLAVE execute address differs from address provided in source image");
129 } > Ram1_90 AT>MFlash256
130 .data_RAM2 : ALIGN(4)
131 {
132     FILL(0xff)
```

Fig 16. Master project linker file for slave code and data relocation

After building the dual core projects, the slave project has 6864 Byte code and data size. They are all allocated to Ram1_90.

```
Finished building: ../board/pin_mux.c

Finished building: ../CMSIS/system_LPC54114_cm0plus.c

Building target: lpcxpresso54114_multicore_examples_hello_world_cm0plus.axf
Invoking: MCU Linker
arm-none-eabi-gcc -nostdlib -L"D:\App LPC5411x\AN\dualcore\sdk_proj\hello_world_mcuxpresso\lpcxpresso54114_multicore_e
Memory region         Used Size  Region Size  %age Used
        Ram1_90:       15320 B        90 KB     16.62%
    rpmsg_sh_mem:          0 GB         6 KB      0.00%
          RamX_32:         0 GB        32 KB      0.00%
copy from `lpcxpresso54114_multicore_examples_hello_world_cm0plus.axf' [elf32-littlearm] to `lpcxpresso54114_multicore
Finished building target: lpcxpresso54114_multicore_examples_hello_world_cm0plus.axf

make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "lpcxpresso54114_multicore_examples_hello_world_cm0plus.axf"; # arm-none-eabi-objcopy -v -O binary
   text    data     bss     dec     hex filename
   6864       0    8456   15320    3bd8 lpcxpresso54114_multicore_examples_hello_world_cm0plus.axf
```

Fig 17. Slave project code and data size

After building the master project, it is observed that 6864 Bytes of Ram1_90 is used, which is in accordance with slave code and data size.

```
Building target: lpcxpresso54114_multicore_examples_hello_world_cm4.axf
Invoking: MCU Linker
arm-none-eabi-gcc -nostdlib -L"D:\App LPC5411x\AN\dualcore\sdk_proj\hello_world_mcuxpresso\lpcxpresso54114_multico
Memory region         Used Size  Region Size  %age Used
       MFlash256:       20496 B       256 KB      7.82%
         Ram0_64:        8472 B        64 KB     12.93%
         Ram1_90:        6864 B        90 KB      7.45%
    rpmsg_sh_mem:           0 GB         6 KB      0.00%
         RamX_32:           0 GB        32 KB      0.00%
Finished building target: lpcxpresso54114_multicore_examples_hello_world_cm4.axf

make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "lpcxpresso54114_multicore_examples_hello_world_cm4.axf"; # arm-none-eabi-objcopy -v -O binary
   text    data     bss     dec     hex filename
  20492       4    8468   28964    7124 lpcxpresso54114_multicore_examples_hello_world_cm4.axf
```

Fig 18. Master project Ram1_90 memory used size

4.2 Project debugging

After compiling the projects, connect the LPCXpresso54114 board to the PC USB interface and start debugging. To start with, choose master Cortex-M4 project and enter the debugging state. The debugger will stop at the entry of the main () function. At this point, the reset handler in the startup file is executed and the slave code and data are copied to the designated Ram1_90 area.

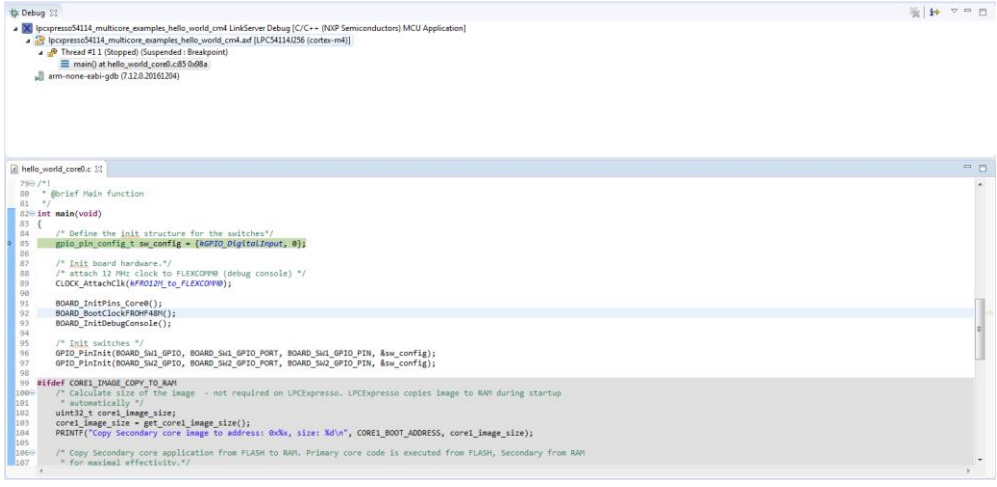


Fig 19. Launch master project debugging

Then choose slave project and start debugging it. The two projects will be in debugging state at the same time. Since the slave booting address is not setup, slave cannot execute its code.

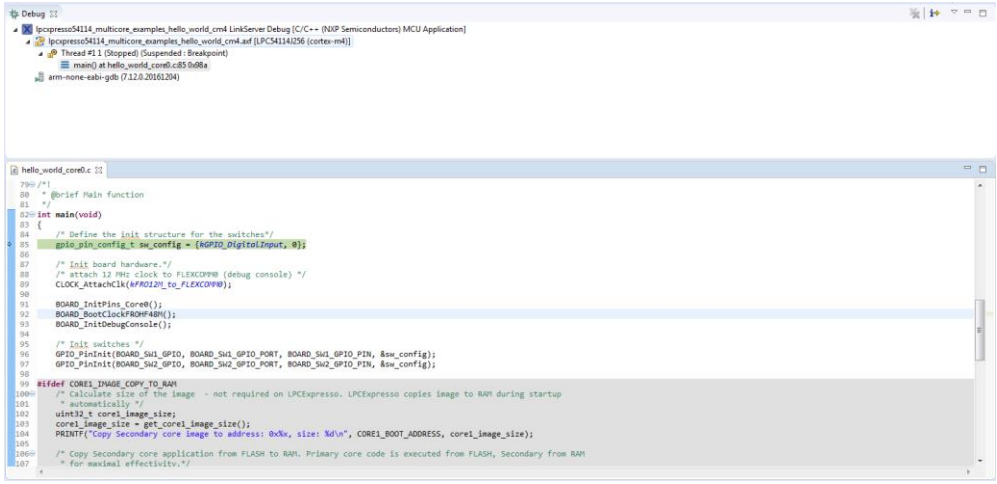


Fig 20. Both projects are in debugging state

The MCMGR_StartCore () function configures the slave core booting address and initializes the stack pointer, then resets it. Per previous booting analysis, the slave core starts up normally this time and begin to execute the application code. The debugger will automatically jump and stop at the entry of main () function in the slave project.

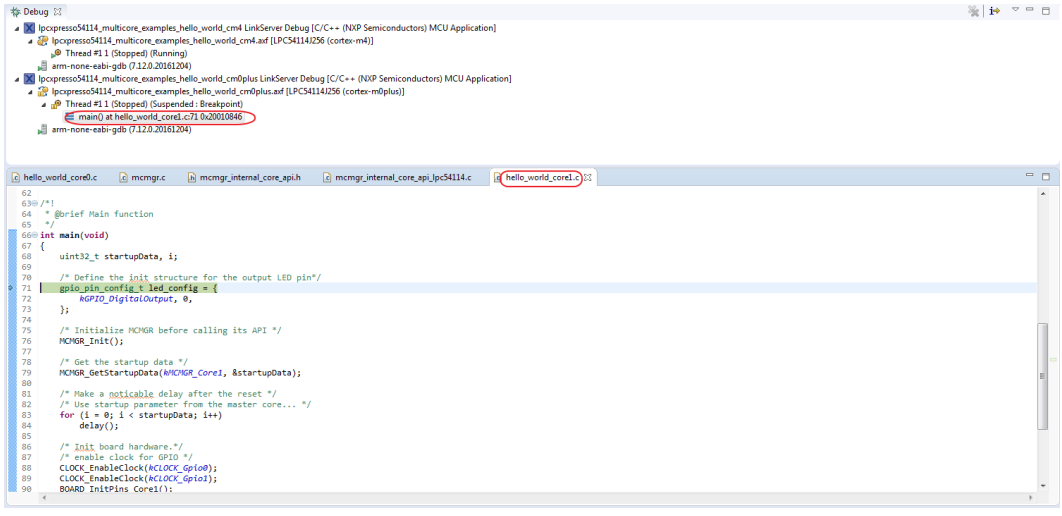


Fig 21. Debugger stops at slave project main () function entry

Then the dual core projects can be debugged as normal single project.

5. Debugging with IAR

IAR also supports dual core architecture feature, which is convenient for the users to debug their application project. The slave project can be linked with the master project via linker option, where the slave binary image and loading section are designated. Then, the linker will integrate both the image files together during compilation. So, the slave project should be built first, otherwise the master project will report compilation error because the slave image data cannot be found. This process in IAR is similar to the MCUXpresso IDE, except that the slave image load address in the flash should be specified in the master linker file and the image should be copied to its execution memory space by the master application.

5.1 Project configuration

Unzip the “hello_world_cm4_hello_world_cm0plus_IAR.zip” file in the AN package. There are two folders containing Cortex-M4 master project and Cortex-M0+ slave project respectively.

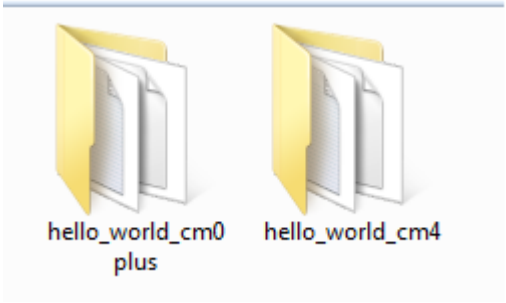


Fig 22. IAR dual core projects folders

Open both the two projects and compile “hello_world_cm0plus” slave project. Then the master “hello_world_cm4” project can be built successfully.

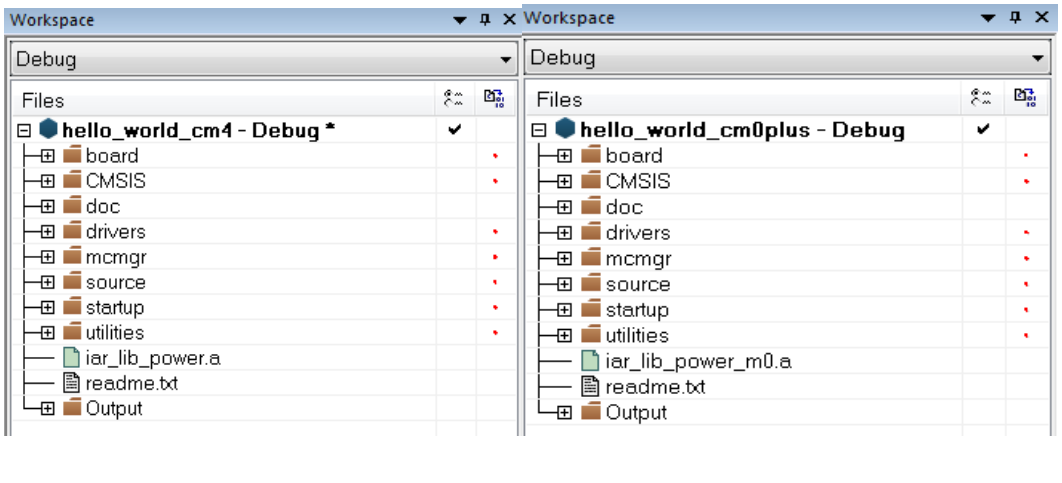


Fig 23. Master and slave projects

Most of the project configurations for the dual core is set on the master project side including both the images and debugging linkage. See Fig 24 for the linker script file “LPC54114J256_cm4.icf” which controls the memory layout for the master project. It also defines the slave core image load region from 0X30000 to 0X3FFFF in the flash space.

```
define symbol __ram_vector_table_size__ =  isdefinedsymbol(__ram_vector_table__) ? 0x000000E0 : 0;
define symbol __ram_vector_table_offset__ =  isdefinedsymbol(__ram_vector_table__) ? 0x000000DF : 0;

define symbol m_interrupts_start             = 0x00000000;
define symbol m_interrupts_end               = 0x000000DF;

define symbol m_text_start                   = 0x000000E0;
define symbol m_text_end                     = 0x0002FFFF;

define symbol m_interrupts_ram_start         = 0x20000000;
define symbol m_interrupts_ram_end           = 0x20000000 + __ram_vector_table_offset__;

define symbol m_data_start                   = m_interrupts_ram_start + __ram_vector_table_size__;
define symbol m_data_end                     = 0x2000FFFF;

define exported symbol rpmsg_sh_mem_start    = 0x20026800;
define exported symbol rpmsg_sh_mem_end      = 0x20027FFF;

define symbol m_sramx_start                  = 0x04000000;
define symbol m_sramx_end                    = 0x04007FFF;

define exported symbol core1_image_start     = 0x00030000;
define exported symbol core1_image_end       = 0x0003FFFF;

define symbol __crp_start__                  = 0x000002FC;
define symbol __crp_end__                    = 0x000002FF;

define symbol __ram_iap_start__              = 0x2000FFE0;
define symbol __ram_iap_end__                = 0x2000FFFF;

/* Sizes */
if (isdefinedsymbol(__stack_size__)) {
  define symbol __size_cstack__        = __stack_size__;
} else {
  define symbol __size_cstack__        = 0x0400;
}

if (isdefinedsymbol(__heap_size__)) {
  define symbol __size_heap__          = __heap_size__;
} else {
  define symbol __size_heap__          = 0x0800;
}

define exported symbol __VECTOR_TABLE  = m_interrupts_start;
define exported symbol __VECTOR_RAM    = isdefinedsymbol(__ram_vector_table__) ? m_interrupts_ram_start : m_interrupts_start;
define exported symbol __RAM_VECTOR_TABLE_SIZE = __ram_vector_table_size__;


define memory mem with size = 4G;
define region TEXT_region             = mem:[from m_interrupts_start to m_interrupts_end]
                                      | mem:[from m_text_start to m_text_end]
                                      - mem:[from  __crp_start__ to __crp_end__];
define region DATA_region             = mem:[from m_data_start to m_data_end]
                                      - mem:[from __ram_iap_start__ to __ram_iap_end__];
define region CSTACK_region           = mem:[from m_sramx_start to m_sramx_end];
define region CRP_region              = mem:[from  __crp_start__ to __crp_end__];
define region m_interrupts_ram_region = mem:[from m_interrupts_ram_start to m_interrupts_ram_end];

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_WBLOCK          { section  __sec_core };

initialize by copy { readwrite };

if (isdefinedsymbol(__USE_DLIB_PERTHREAD))
{
  // Required in a multi-threaded application
  initialize by copy with packing = none { section __DLIB_PERTHREAD };
}

do not initialize  { section .noinit };
do not initialize  { section rpmsg_sh_mem_section };
```

Fig 24. Master project linker script file

See Fig 25 for the slave project linker script file. The code execution region starts from SRAM1 address 0X20010000. The master application relocates the slave image from the flash load region to the SRAM1 execution space.

```
define symbol __ram_vector_table_size__ =  isdefinedsymbol(__ram_vector_table__) ? 0x000000C0 : 0;
define symbol __ram_vector_table_offset__ =  isdefinedsymbol(__ram_vector_table__) ? 0x000000BF : 0;

define symbol m_interrupts_start             = 0x20010000;
define symbol m_interrupts_end               = 0x200100BF;

define symbol m_text_start                   = 0x200100C0;
define symbol m_text_end                     = 0x2001FFFF;

define symbol m_interrupts_ram_start         = 0x20020000;
define symbol m_interrupts_ram_end           = 0x20020000 + __ram_vector_table_offset__;

define symbol m_data_start                   = m_interrupts_ram_start + __ram_vector_table_size__;
define symbol m_data_end                     = 0x200267FF;

define exported symbol rpmsg_sh_mem_start    = 0x20026800;
define exported symbol rpmsg_sh_mem_end      = 0x20027FFF;

define symbol m_sramx_start                  = 0x04000000;
define symbol m_sramx_end                    = 0x04007FFF;

/* Sizes */
if (isdefinedsymbol(__stack_size__)) {
  define symbol __size_cstack__        = __stack_size__;
} else {
  define symbol __size_cstack__        = 0x0400;
}

if (isdefinedsymbol(__heap_size__)) {
  define symbol __size_heap__          = __heap_size__;
} else {
  define symbol __size_heap__          = 0x0800;
}

define exported symbol __VECTOR_TABLE  = m_interrupts_start;
define exported symbol __VECTOR_RAM    = isdefinedsymbol(__ram_vector_table__) ? m_interrupts_ram_start : m_interrupts_start;
define exported symbol __RAM_VECTOR_TABLE_SIZE = __ram_vector_table_size__;
```

Fig 25. Slave project linker script file

The *input* tab in the linker option links the slave image file with the load region defined in linker script file. The slave binary image file path and its load section is designated. In current case, “$PROJ_DIR$/../hello_world_cm0plus/debug/hello_world_cm0plus.bin” is the raw binary image file for slave core. *__sec_core* is its load section, which is defined in the linker script file. IAR will search this image file during master project compilation and will link it together with the master image.

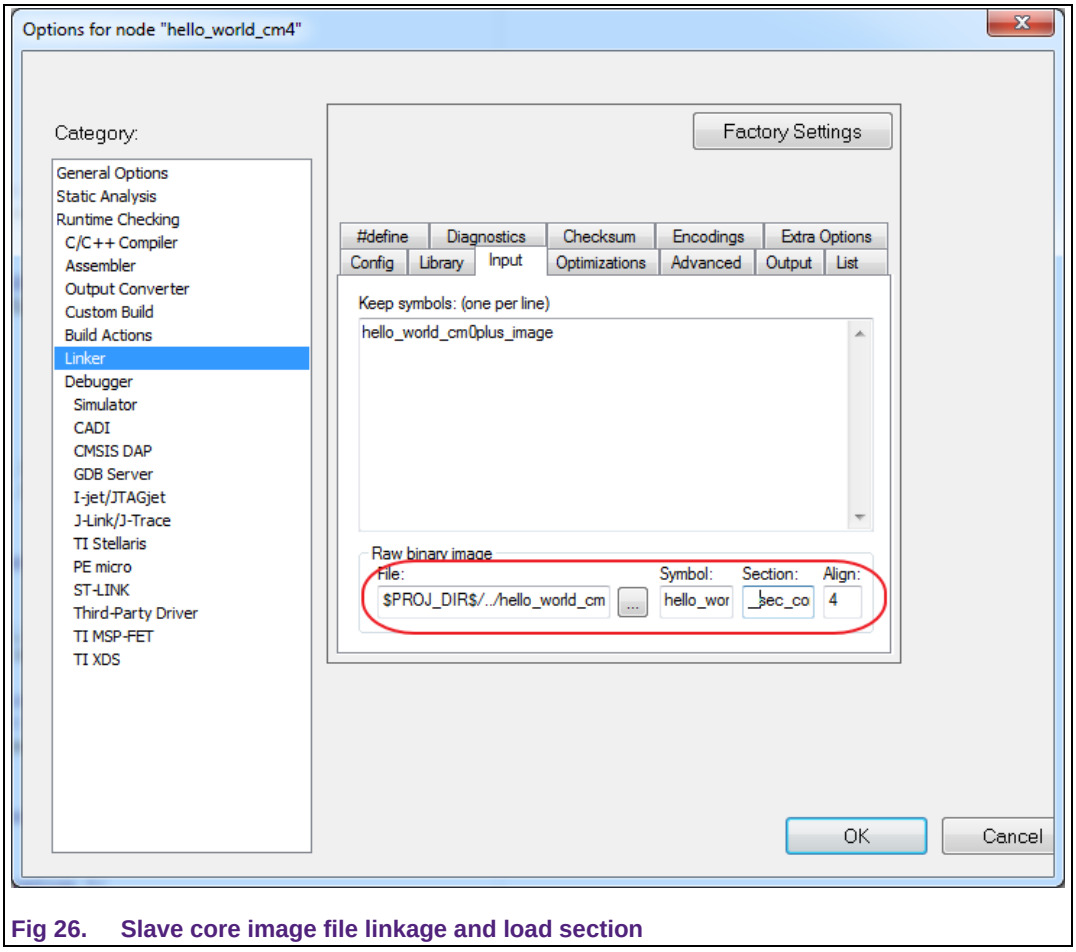


Fig 26. Slave core image file linkage and load section

The dual core debugging in IAR for the master and slave projects can also be connected via *Multicore* tag in debugger option. After checking the *Enable multicore master mode* and choosing the slave project path, the slave project can be automatically opened and placed into debugging mode when the master project is debugged.

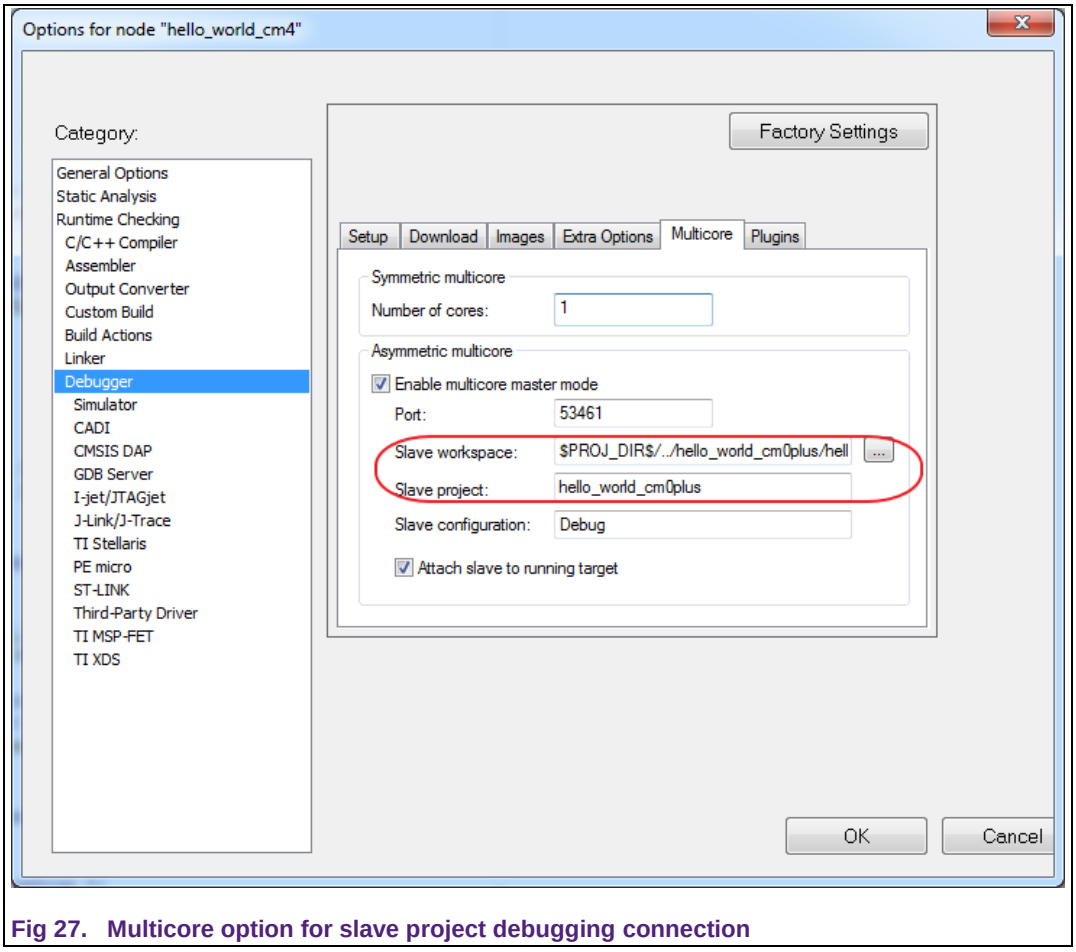


Fig 27. Multicore option for slave project debugging connection

For both the projects, the reset setting of the debugger should be configured properly. Usually, the master project should be configured to hardware reset and the slave project should not use hardware or software reset. This method ensures that the master debugging session is not affected.

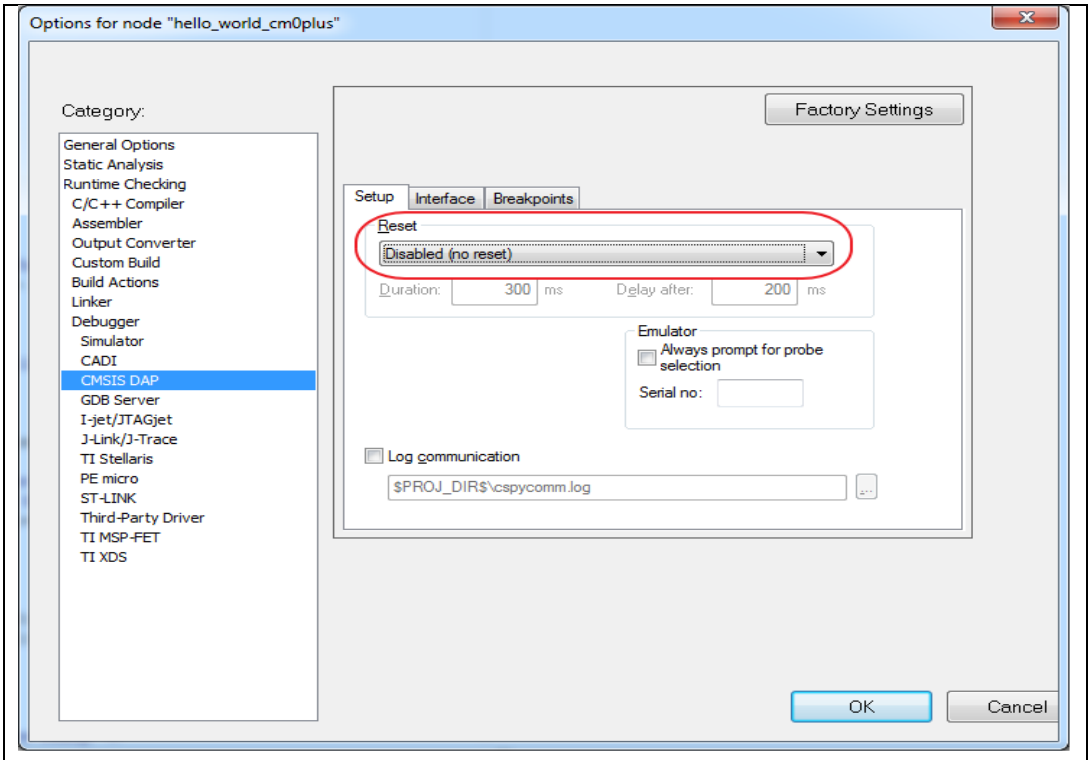


Fig 28. Reset setting for master project

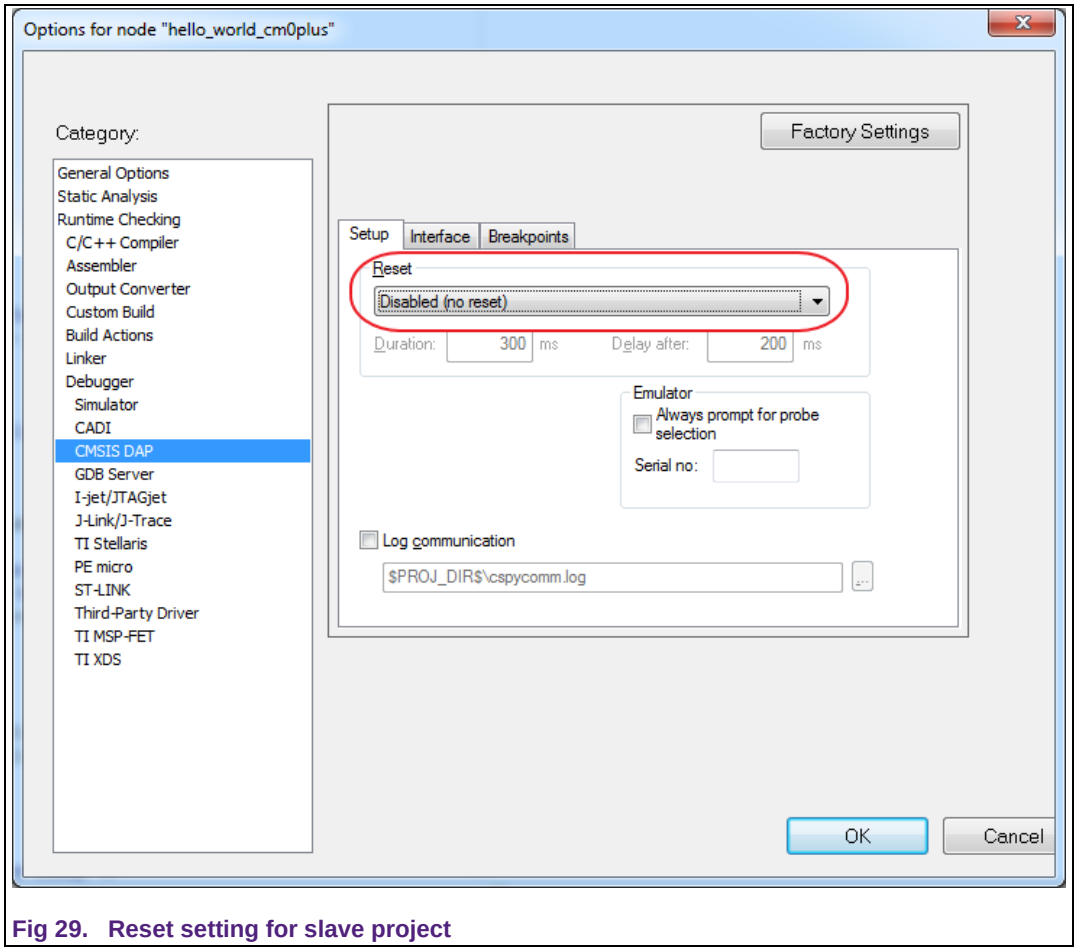


Fig 29. Reset setting for slave project

5.2 Project debugging

After successfully building the master project, click the *Download and Debug* button to enter the debugging mode. IAR opens the corresponding slave project and starts debugging. The master core will be stopped at the entry point of main () function and slave will be in a sleep state.

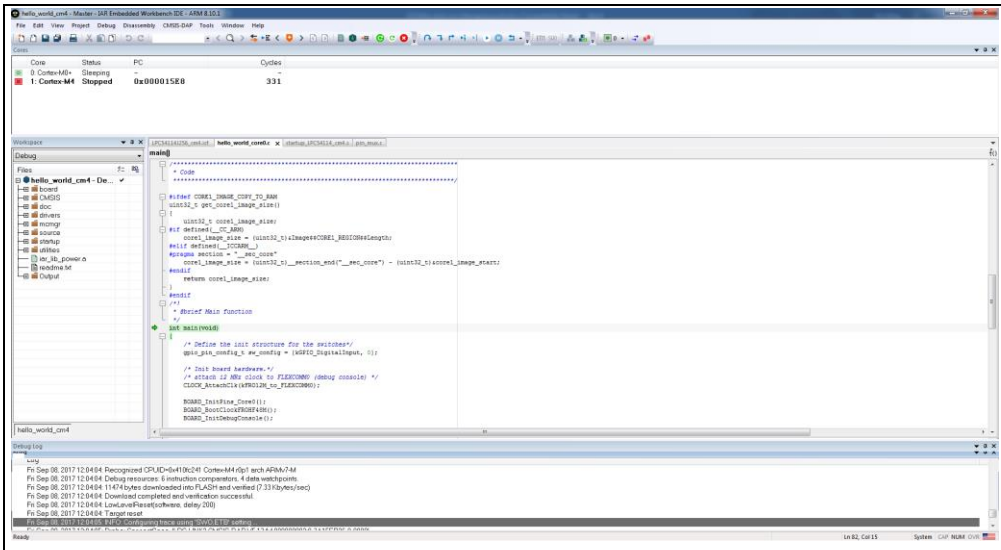


Fig 30. Master project in debugging state

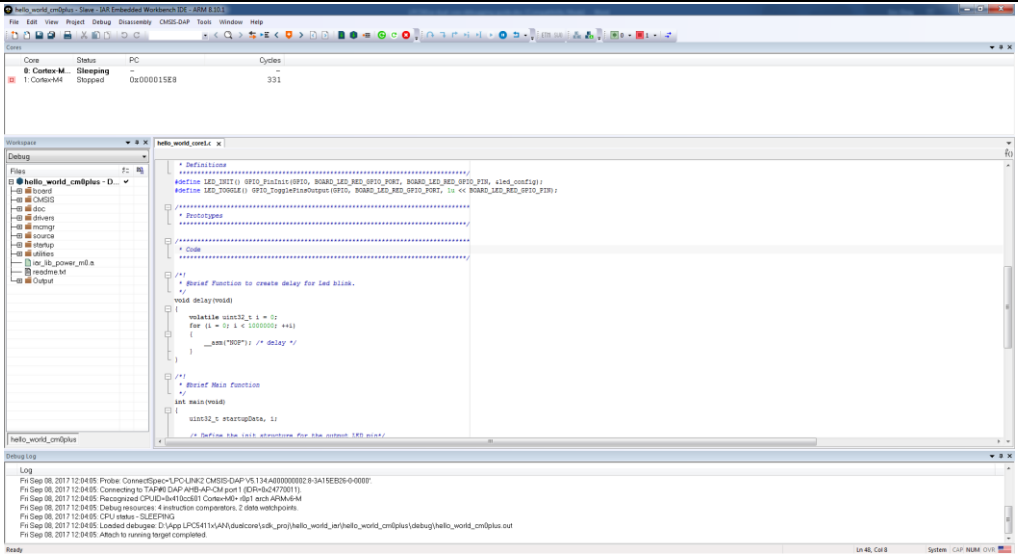


Fig 31. Slave project in debugging state

There is a *cores* window in *view* menu available for checking current states of both the cores, such as their execution status.

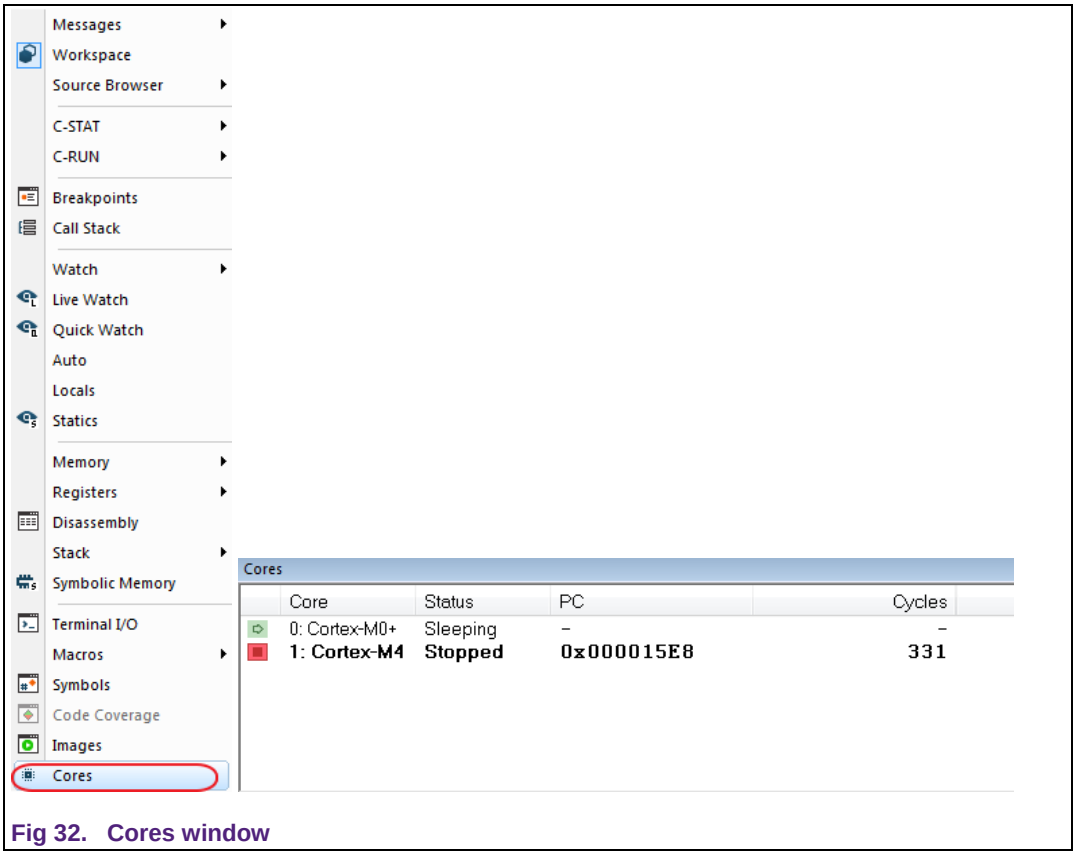


Fig 32. Cores window

Although the slave code has been loaded into the designated flash address, it should also be copied to its execution memory space. This task and the slave booting environment setup are all conducted by the master application code. The copy destination address should be in accordance with the slave project memory layout setting.

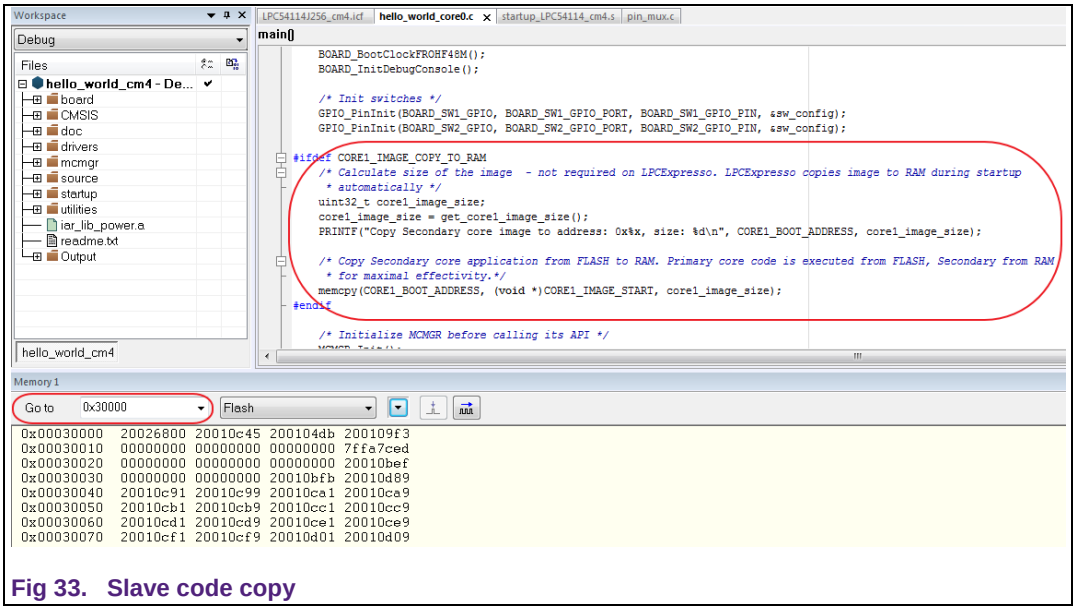


Fig 33. Slave code copy

Following is the memory copy destination address, which is the same as the slave code execution region starting address.

#define CORE1_BOOT_ADDRESS (void *)0x20010000

As illustrated in the chapter *dual core booting analysis*, the master core needs to setup the slave booting up environment and reset it again. Then, the slave will jump to and execute its application code. MCMGR_StartCore () function will do this task after the slave code copy. When this function is called, the slave core status will change from sleeping to running.

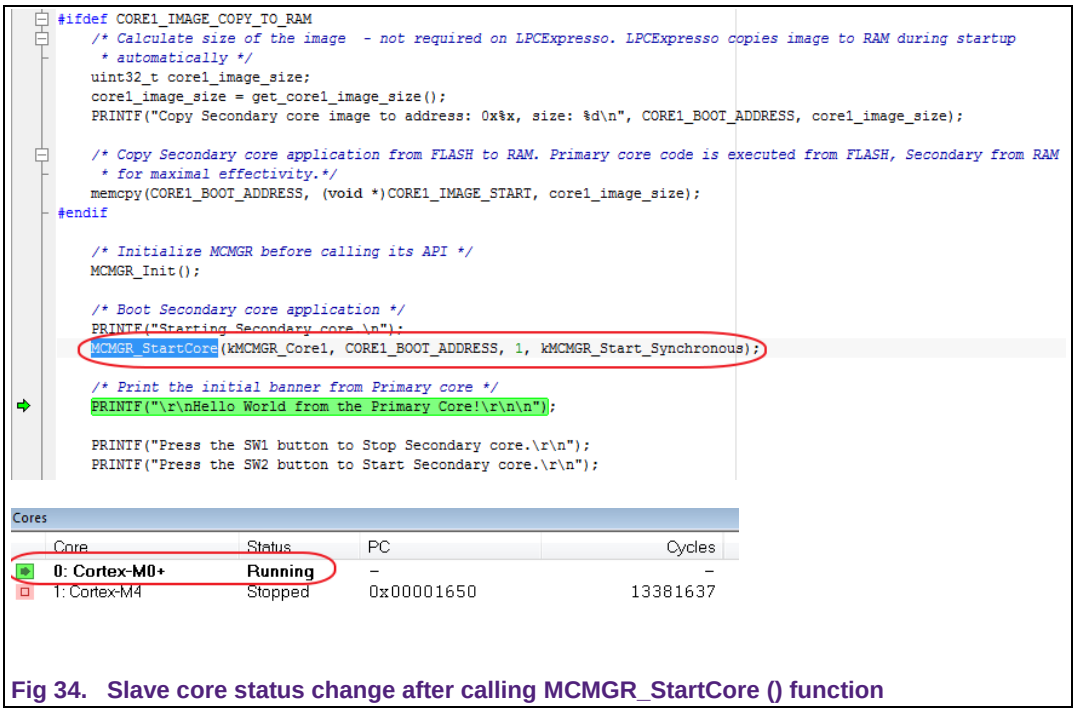


Fig 34. Slave core status change after calling MCMGR_StartCore () function

Then the two projects can be debugged like other usual projects.

6. Debugging with Keil

Keil MDK cannot link the two dual core projects together and debug them at the same time as MCUXPresso IDE or IAR. They are treated as two independent projects. The master project should include the slave binary image into its own executable file and download them into flash memory together. The application is also responsible for slave code relocation and booting environment setup.

6.1 Project configuration

Unzip the *hello_world_cm4_hello_world_cm0plus_Keil.zip*. There are two folders containing both the Cortex-M4 master and Cortex-M0+ slave projects.

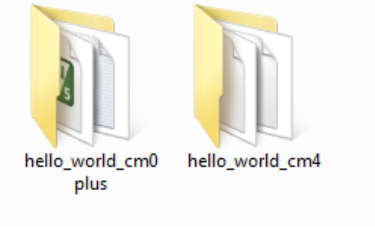


Fig 35. Keil dual core projects folders

Because the master project needs to search and include the slave image binary during compilation, open the two projects and build the slave project *hello_world_cm0plus*. Fig 36 shows picture of the two Keil projects.

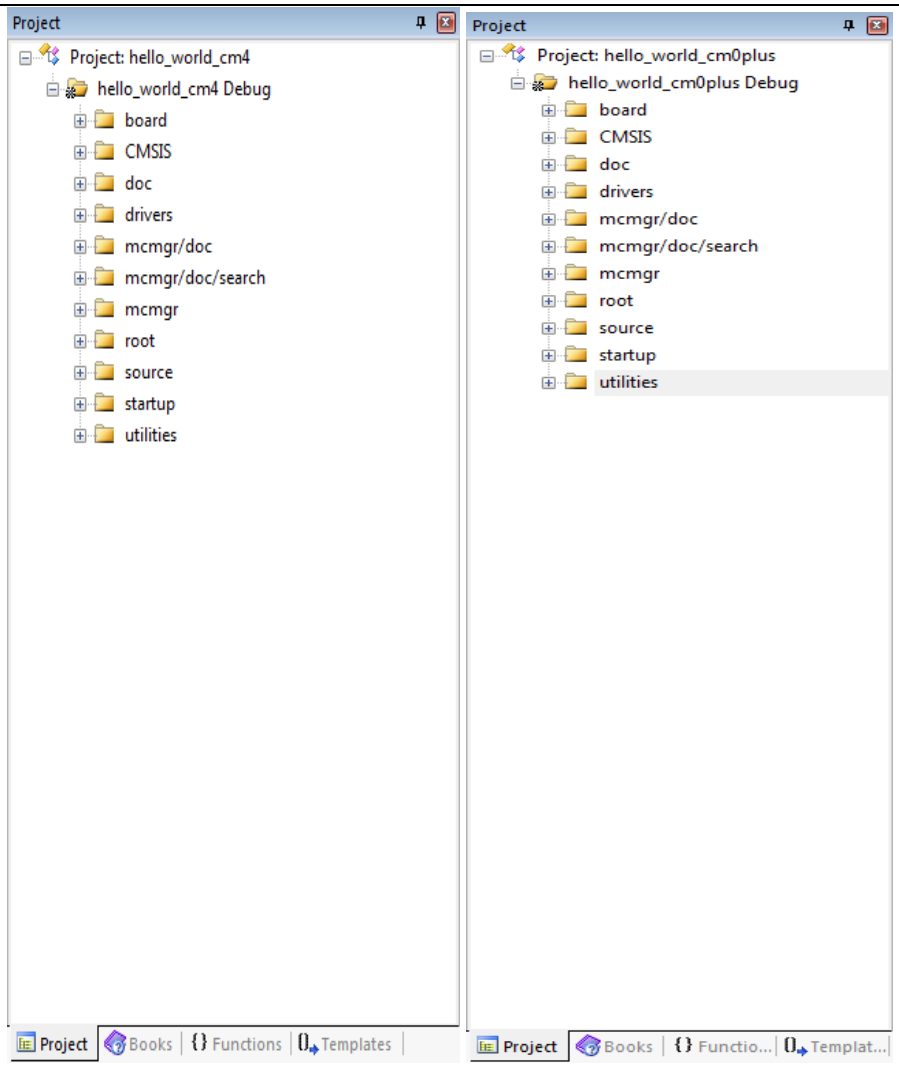


Fig 36. Keil dual core projects

There is an assembly file *incbin.s* in the master project which includes the slave binary file and allocates it into the slave code load region defined by the master linker script file. The search path of the binary file is configured in the *Asm* tab of the master project options.

incbin.s

```
;/*
; * Copyright (c) 2016, Freescale Semiconductor, Inc.
; * All rights reserved.
; *
; * Redistribution and use in source and binary forms, with or without modification,
; * are permitted provided that the following conditions are met:
; *
; * o Redistributions of source code must retain the above copyright notice, this list
; *   of conditions and the following disclaimer.
; *
; * o Redistributions in binary form must reproduce the above copyright notice, this
; *   list of conditions and the following disclaimer in the documentation and/or
; *   other materials provided with the distribution.
; *
; * o Neither the name of Freescale Semiconductor, Inc. nor the names of its
; *   contributors may be used to endorse or promote products derived from this
; *   software without specific prior written permission.
; *
; * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
; * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
; * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
; * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
; * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
; * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
; * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
; * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
; * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
; * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
; */

        AREA M0CODE, DATA, READONLY, PREINIT_ARRAY, ALIGN=3
        EXPORT m0_image_start
        EXPORT m0_image_end
m0_image_start
        INCBIN core1_image.bin
m0_image_end
        END
```

Fig 37. Slave image including file

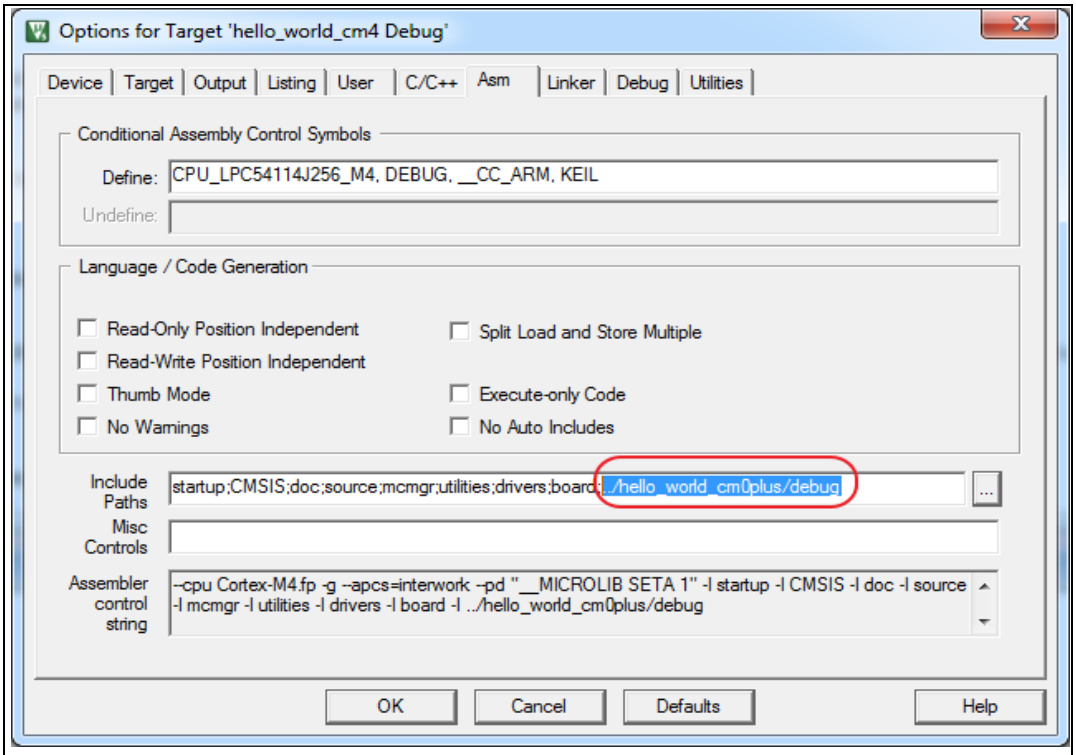


Fig 38. Slave binary file search path

The Keil project should set the project configuration, to generate the binary file required. See Fig 39 for the slave project setting on the binary image output. The After-Build command used is “$K\ARM\ARMCC\bin\fromelf.exe --bincombined --bincombined_base=0x20010000 --output=$Lcore1_image.bin !L”.

The binary file name(core1_image.bin) should be in accordance with the file included in the master project *incbin.s*.

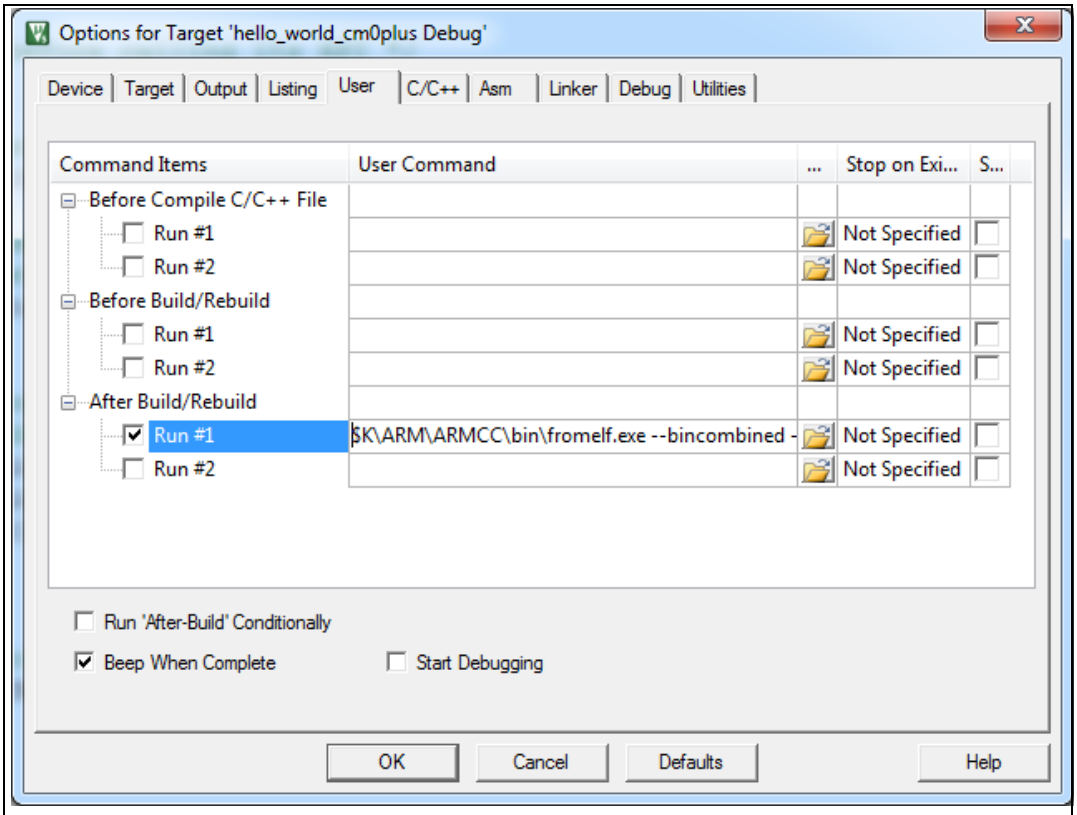


Fig 39. Slave binary file generation configuration

See Fig 40 for the slave code load region definition in the master project linker script file.

LPC54114J256_cm4.scf

```
44  **     http:                 www.nxp.com
45  **     mail:                 support@nxp.com
46  **
47  ** ###################################################################
48  */
49
50  #if (defined(__ram_vector_table__))
51    #define __ram_vector_table_size__    0x000000E0
52  #else
53    #define __ram_vector_table_size__    0x00000000
54  #endif
55
56  #define m_interrupts_start             0x00000000
57  #define m_interrupts_size              0x000000E0
58
59  #define m_text_start                   0x000000E0
60  #define m_text_size                    0x0002FF20
61
62  #define m_core1_image_start            0x00030000
63  #define m_core1_image_size             0x00010000
64
65  #define m_interrupts_ram_start         0x20000000
66  #define m_interrupts_ram_size          __ram_vector_table_size__
67
68  #define m_rpmsg_sh_mem_start           0x20026800
69  #define m_rpmsg_sh_mem_size            0x00001800
70
71  #define m_data_start                   (m_interrupts_ram_start + m_interrupts_ram_size)
72  #define m_data_size                    (0x00010000 - m_interrupts_ram_size - m_rpmsg_sh_mem_size)
73
74  #define m_sramx_start                  0x04000000
75  #define m_sramx_size                   0x00008000
```

```
120  LR_CORE1_IMAGE m_core1_image_start {
121    CORE1_REGION m_core1_image_start m_core1_image_size {
122      *(M0CODE)
123    }
124  }
125
126
```

Fig 40. Slave image load region definition

Slave code execution region is defined in the slave project linker script file. So, the master application should relocate it from flash to SRAM1.

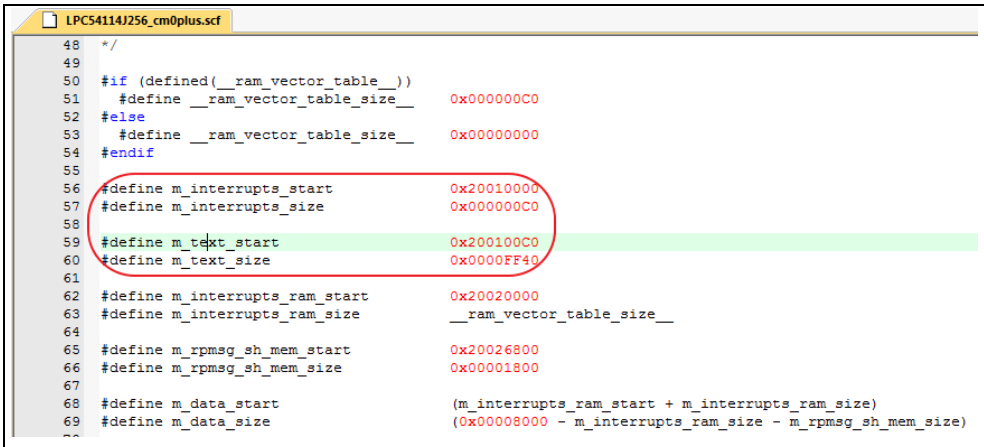


Fig 41. Slave code execution region definition

6.2 Project debugging

Keil dual core projects debugging is the same as in general projects. There is no linkage relationship between the two projects. The master core still needs to copy the slave code from flash to the execution RAM region in explicit code and setup the booting environment.

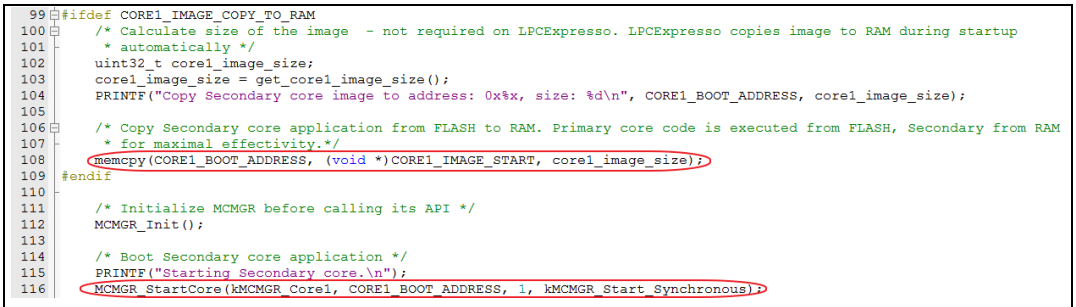


Fig 42. Slave code copy and booting address setup

7. Conclusion

The dual core booting up and project debugging on three IDEs are introduced in this application note. MCUXpresso IDE and IAR have better support on the dual core development and debugging, compared to Keil MDK. This application note analyzes the project configuration and debugging procedure with all three IDEs. The user can choose one of the IDEs for application development.

8. Legal information

8.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

8.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

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.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Translations — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

8.3 Licenses

Purchase of NXP <xxx> components
<License statement text>

8.4 Patents

Notice is herewith given that the subject device uses one or more of the following patents and that each of these patents may have corresponding patents in other jurisdictions.

<Patent ID> — owned by <Company name>

8.5 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

<Name> — is a trademark of NXP Semiconductors N.V.

9. List of figures

10. Contents

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

Date of release: 8 March 2018
Document identifier: AN12124