AN12985

RT600 Hybrid Boot

Rev. 0 — 09/2020 Application Note

by: NXP Semiconductors

Contents

1 Introduction......................................1
2 RT600 boot overview...................... 1
3 Sample example application........... 3
4 Conclusion....................................17
5 References...................................17

1 Introduction

The RT600 family are RAM-based and M33-based MCUs with an internal DSP. The code must be either booted into the memory (from a host or non-volatile memory) or executed from an external flash memory directly without booting. A practical use case is code which boots and executes from the flash, so the boot image contains the code to boot internally into the RT685 RAM stored in the upper flash on powerup and the lower flash contains run-time code fetched/executed directly. The SDK does not provide this more practical use case, but the code that often resides in the internal RAM is desirable for performance reasons. The internal RAM size may be reserved mostly for data or code which may exceed the 4.5 MB of internal RAM provided on the RT685 MCUs.

This application note provides a project with a part of code booted from the external flash into the internal RT685 SRAM and the remaining code that resides in the flash is fetched/executed directly. Details on how to place the code as bootable into the SRAM or for execution directly from the flash via the assignments made in the linker script are provided. This application note also provides instructions on how to program the flash with both the bootable RAM portion of the code and the run-time code residing in the lower flash. Details such as secure boot or OTFAD decryption of flash data are out of scope.

2 RT600 boot overview

2.1 Boot features

Because the i.MX RT600 MCUs have no internal flash for code and data storage, the images must be stored elsewhere for loading upon reset or the CPU can execute them from the external memory (XIP). The images can be loaded into the on-chip SRAM from the external flash or downloaded via the serial ports (UART, SPI, I2C, USB). The code is then validated, and the boot ROM jumps to the on-chip SRAM.

Depending on the values of the OTP bits and ISP pins and the image header type definition, the bootloader decides whether to download the code into the on-chip SRAM or run it from an external memory. The bootloader checks the OTP bit settings first and then the ISP pins. If bit [3:0] in the OTP word BOOT_CFG [0] is not programmed (4b ' 0000), the boot source is determined by the states of the ISP boot pins (PIO1_15 , PIO1_16 , and PIO1_17).

2.2 Boot settings

In this application note, the FlexSPI boot mode is used. If the PRIMARY_BOOT_SRC bits in the OTP are not set, the i.MX RT600 reads the status of the ISP pins to determine the boot source. Table 1 describes the boot mode and the ISP downloader modes based on the ISP pins for the FlexSPI boot.

Table 1. Boot mode and ISP Downloader modes based on ISP pins

Boot mode	ISP2 pin `PIO1_17`	ISP1 pin `PIO1_16`	ISP0 pin `PIO1_15`	Description
—	Low	Low	Low	Reserved

Table continues on the next page...

Table 1. Boot mode and ISP Downloader modes based on ISP pins (continued)

Boot mode	ISP2 pin PIO1_17	ISP1 pin PIO1_16	ISP0 pin PIO1_15	Description
SDIO0 (SD Card)	Low	Low	High	Boot from an SD card device connected to SDIO 0 interface. The i.MXRT600 will look for a valid image in the SD card device. If there is no valid image found, the i.MXRT600 will enter the ISP boot mode based on OTP DEFAULT_ISP_MODE bits (6:4, BOOT_CFG [0])).
FlexSPI Boot from Port B	Low	High	Low	Boot from Quad or Octal SPI Flash devices connected to the FlexSPI interface 0 Port B. The i.MXRT600 will look for a valid image in external Quad/Octal SPI Flash device. If there is no valid image found, the i.MXRT600 will enter ISP boot mode.
FlexSPI Boot from Port A	Low	High	High	Boot from Quad/Octal SPI Flash devices connected to the FlexSPI interface 0 Port A. The i.MXRT600 will look for a valid image in external Quad/Octal SPI Flash device. If there is no valid image found, the i.MXRT600 will enter ISP boot mode.
SDIO 0 (eMMC)	High	Low	Low	Boot from an SD card device connected to SDIO 0 interface. The i.MXRT600 will look for a valid image in the SD card device. If there is no valid image found, the i.MXRT600 will enter the ISP boot mode based on OTP DEFAULT_ISP_MODE bits (6:4, BOOT_CFG [0]))
USB DFU (master boot)	High	Low	High	USB DFU class is used to download a boot image over the USB High-speed port into on-chip SRAM.
Serial ISP (UART, SPI, I^2C, USB-HID)	High	High	Low	The Serial Interface (UART, SPI, and I^2C,USB-HID) is used to program OTP, external Flash, SD or eMMC device.
Serial Master Boot(UART, SPI, I^2C, USB-HID)	High	High	High	Serial Master boot (SPI Slave, I^2C Slave, or UART, USB-HID) is used to download a boot image over the serial interface (SPI Slave, I^2C slave or UART,USB-HID).

2.3 Boot image offset

The bootloader looks for the boot image from a specified offset on a boot media. See the details in Table 2.

Table 2. Image offset on different boot media

Boot media	Image offset
FlexSPI Boot (Serial NOR Flash device)	0x1000
SD Boot (SD card)	0x1000
eMMC boot (eMMC memory)	0x1000
Recovery Boot (SPI NOR Flash device)	0x1000

2.4 Image link region

For the FlexSPI serial NOR flash boot, there are two possibilities: the Load-to-RAM boot and the XIP boot. For the Load-to-RAM boot, after the boot ROM runs, it initializes the FlexSPI module according to the external NOR flash type connected to the MCU device. The ROM loads the boot image from the NOR flash device with the 0x1000 offset to the MCU's internal SRAM. After that, the ROM jumps to the SRAM to run the boot image. For the XIP boot, the boot ROM only boots the image from the NOR flash device. The boot image header inside the boot image tells the ROM whether the boot image is the Load-to-RAM image or the XIP image. The ROM bootloader supports automated booting from the Serial NOR (Quad or Octal SPI Flash, HyperFlash) device and the eXecute-In-Place (XIP) from this Serial NOR flash. This is the main feature of the ROM bootloader. Figure 1 shows the various memory regions.

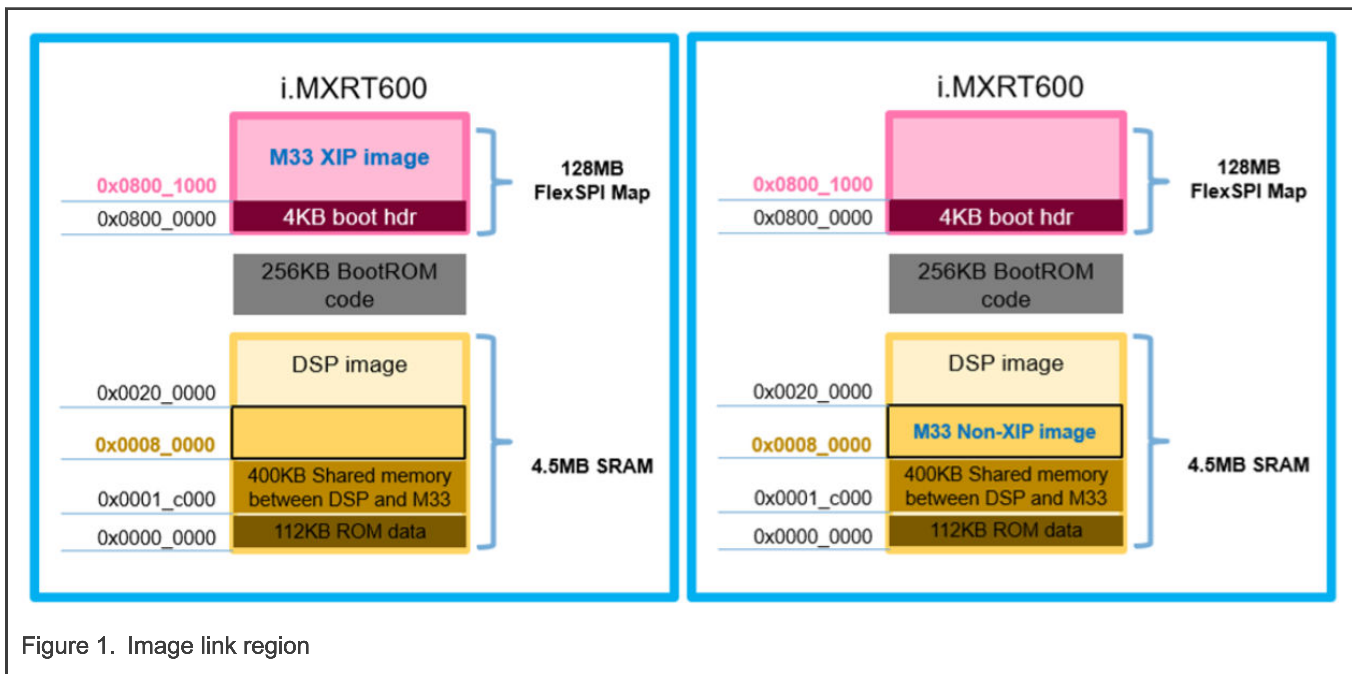

Figure 1. Image link region

For more details regarding the FlexSPI boot flow and process, see *How to Enable Master Boot from Serial NOR Flash* (document AN12773).

3 Sample example application

3.1 Environment

3.1.1 Hardware environment

- Board:
  - MIMXRT685EVK

- Debugger:
  - Integrated CMSIS-DAP debugger on the board
- Miscellaneous:
  - 1 micro USB cable
  - PC
- Board setup:
  - Connect the micro USB cable between the PC and the J5 link on the board to load and run a demo.

3.1.2 Software environment

- Tool chain:
  - MCUXpresso IDE 11.2 .0 or Keil 5.31 or IAR 8.50.5 IDEs
- Software package:
  - SDK_2.8.2_EVK-MIMRT685S

3.2 Steps

3.2.1 Steps for Keil IDE

1. Open the *hello_world.uvmpw* file (located in the *SDK_2.8.2_EVK-MIMXRT685S\boards\evkmimxrt685\demo_apps \ hello_world\mdk* folder) using the Keil IDE . This opens the Keil IDE with the example “hello_world” program.
2. Add a new target with the “hello_world_hybrid_debug” name, which should be based on the the “hello_world_debug” target which already exists in the project.
   a. Click the “Managed Project Item” icon, as shown in Figure 2.

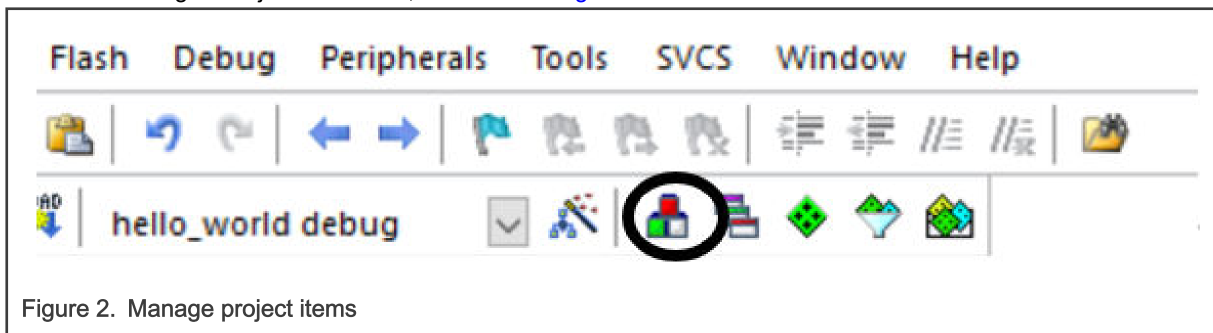

Figure 2. Manage project items

   b. This window gives you the option to add your own targets to the “ Project Targets ” list. Create a new target and select it as the current target as shown in Figure 3.

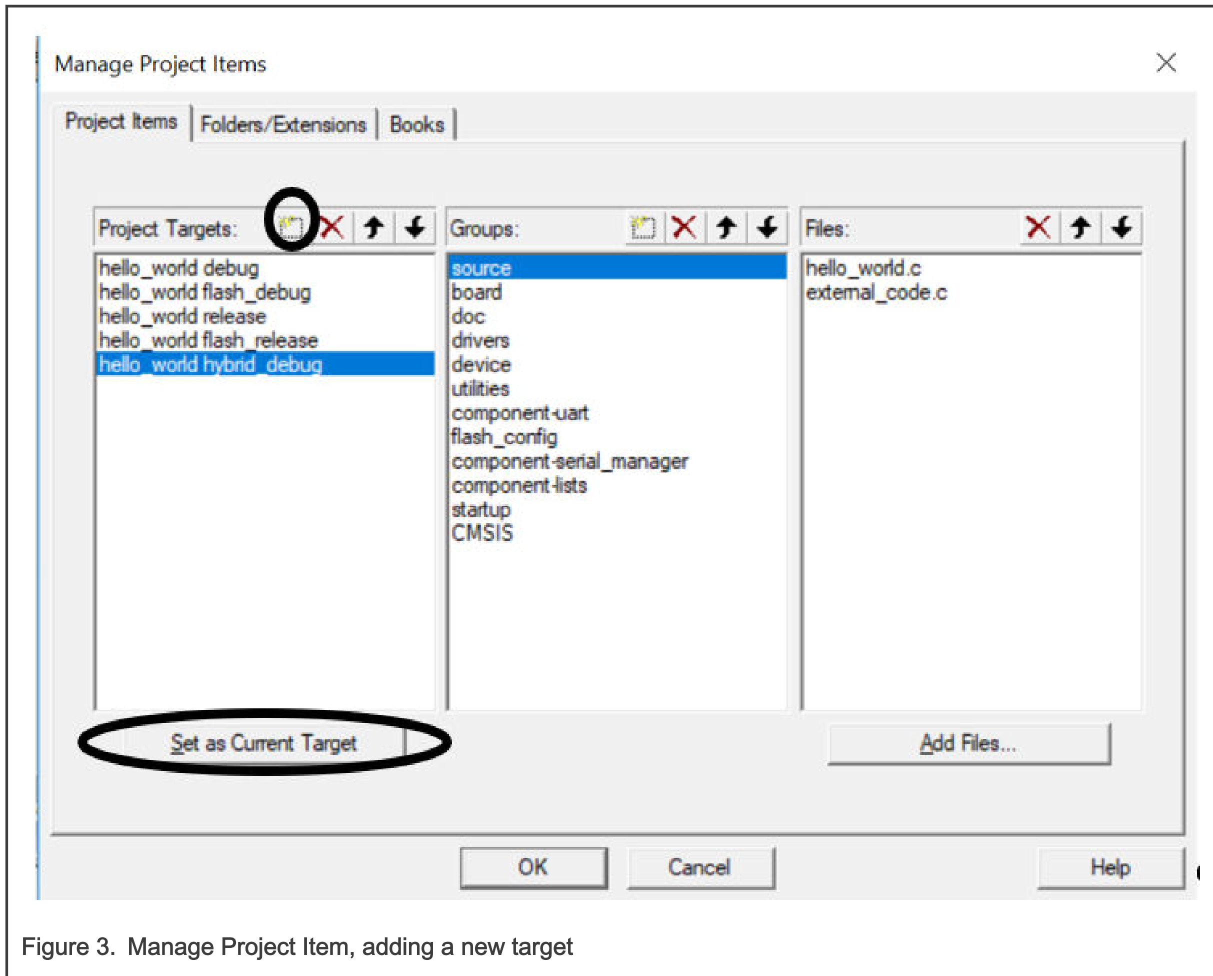

Figure 3. Manage Project Item, adding a new target

c. Add a new C file, which will be later placed to the external flash memory for XIP to the current project. In this example, a very short function with a for loop inside another for loop followed by a print statement is used. This function is then called from the main function in the *hello_world.c* file. Let's call this C file *external_code.c*. Extract the *hello_world_hybrid_mdk.zip* file and copy the *external_code.c*, *hello_world.c*, and *external_code.h* files in the "hello_world_hybrid_mdk" into the *SDK_2.8.2_EVK-MIMXRT685S\boards\evkmimxrt685\demo_apps\hello_world* folder. Now add the *external_code.c* file in the source group into the "hello_world_hybrid_debug" target, as shown in Figure 4.

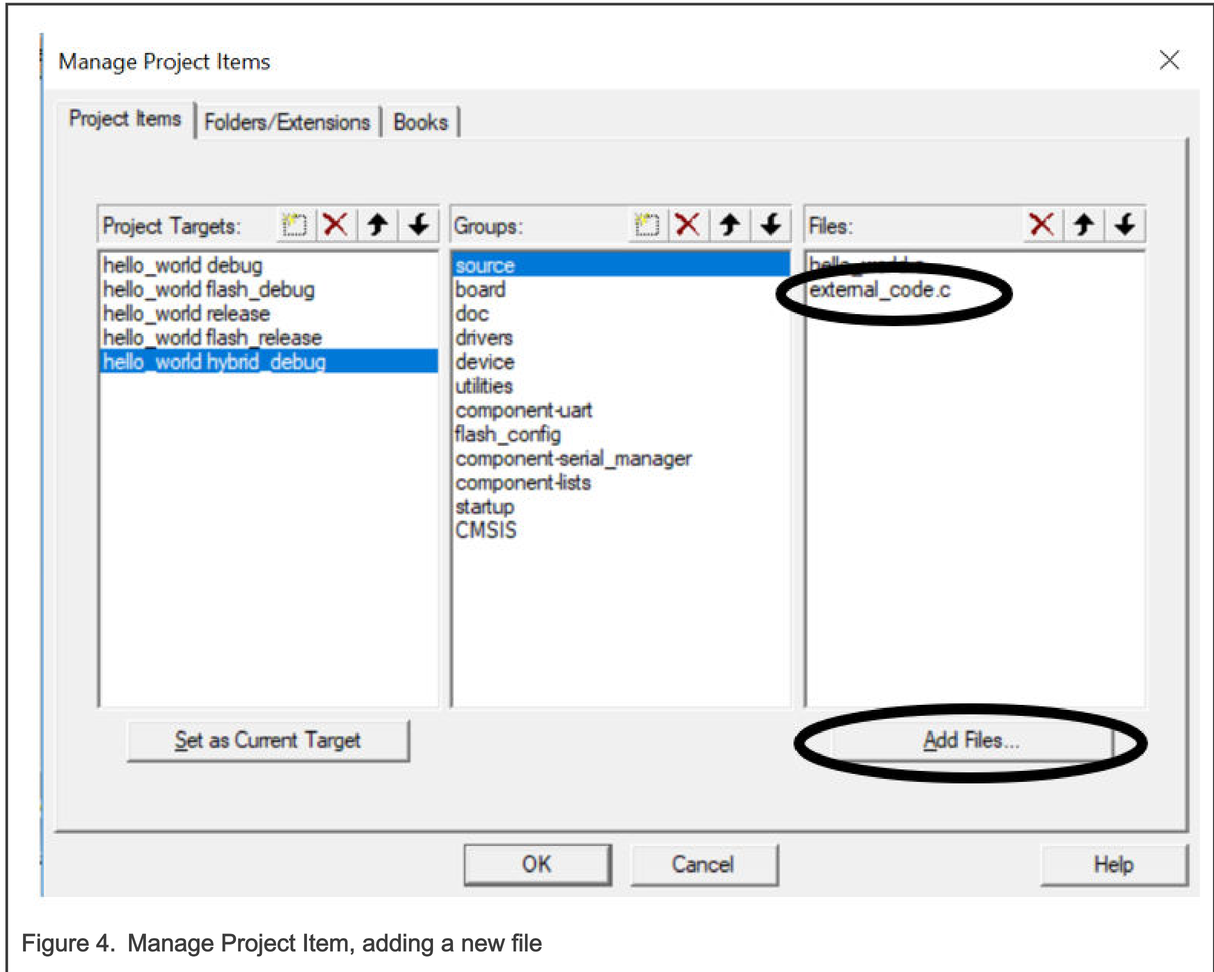

Figure 4. Manage Project Item, adding a new file

d. When the new C code is compiled, its object image (*.o*) can be placed in the external flash for XIP. This is achieved by modifying the *MIMXRT685Sxxxx_cm33_ram.scf* linker script file. Make a copy of this file and rename it to*MIMXRT685Sxxxx_cm33_hybrid.scf*. In the *MIMXRT685Sxxxx_cm33_hybrid.scf* file, make a few changes to program the flash.Firstly, allocate the starting address and size for the part of code which will be executed from the external flash. Note that the starting address for the text can only be after the interrupt vector table, as shown in Figure 5.

```
46 #define  m_flash_start               0x08000000
47
48 #define  m_boot_flash_conf_start     0x08000400
49 #define  m_boot_flash_conf_size      0x00000C00
50
51 #define  m_boot_interrupts_start     0x08001000
52
53 #define  m_interrupts_start          0x00080000
54 #define  m_interrupts_size           0x00000200
55
56 #define  m_text_start                0x00080200
57 #define  m_text_size                 0x000FFE00
58
59 #define  m_text_2_start              (m_boot_interrupts_start + m_interrupts_size)
60 #define  m_text_2_size               0x00000400
61
62 #define  m_data_start                0x20180000
63 #define  m_data_size                 0x00080000
```

Figure 5. Allocating the starting address and size

e. Secondly, add the following lines (line#85 - 87) of code inside the **.scf* file.

```
76 LR_m_interrupts m_boot_interrupts_start m_interrupts_size+m_text_size+m_text_2_size {
77   VECTOR_ROM m_interrupts_start m_interrupts_size {
78     * (.isr_vector,+FIRST)
79   }
80   VECTOR_RAM +0 FILL 0x0 m_text_start-ImageLimit(VECTOR_ROM) {
81   }
82   ER_m_text m_text_start m_text_size {
83     .ANY (+RO)
84   }
85   ER_m_text2 (m_text_2_start+ImageLimit(ER_m_text)-m_text_start) m_text_2_size {
86     external_code.o
87   }
88
89   RW_m_data m_data_start ALIGN 4 m_data_size-Stack_Size-Heap_Size { ; RW data
90     * (CodeQuickAccess)
91     * (DataQuickAccess)
92     .ANY (+RW +ZI)
```

Figure 6. Modifying linker file

Note that the file name used for the new C file is *external_code.c* and in line#86 , the object file is called *external_code.o*. By adding these lines, direct the linker to keep the execution and load the address for the *external_code.o* file at the same location. Because the *external_code.c* file is accessing “printf”, which is a part of the text portion, the linking should happen only after the text section is loaded. For this, find the exact address from where the *external_code.o* file should start and execute, which is just after the text region. Because the “ImageLimit” function gives the end address for an execution region, “ImageLimit(ER_m_text)-m_text_start” provides a location which is just after the text region.

f. Click the target options button, as shown in Figure 7.

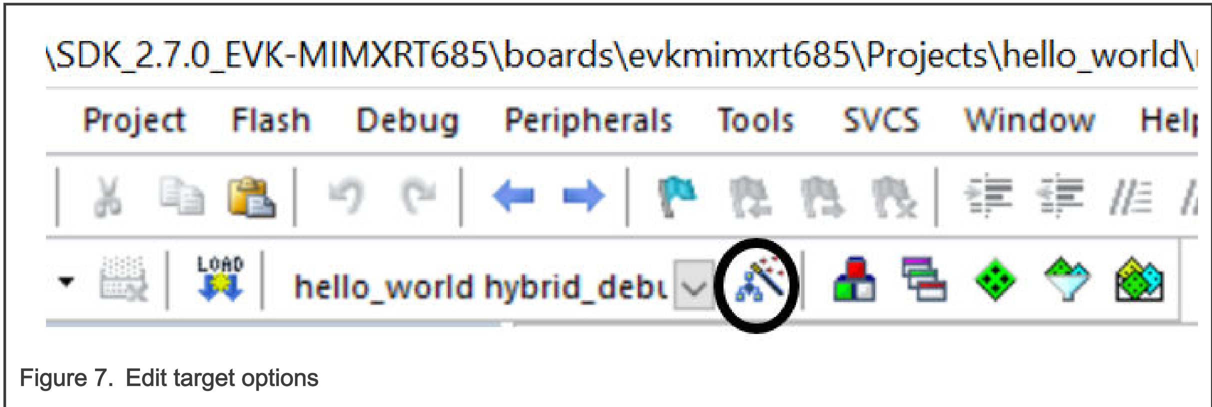

Figure 7. Edit target options

g. Now open the window and then open the “Linker” tab. Using the highlighted button, place the *MIMXRT685Sxxxx_cm33_hybrid.scf* file as the linker file:

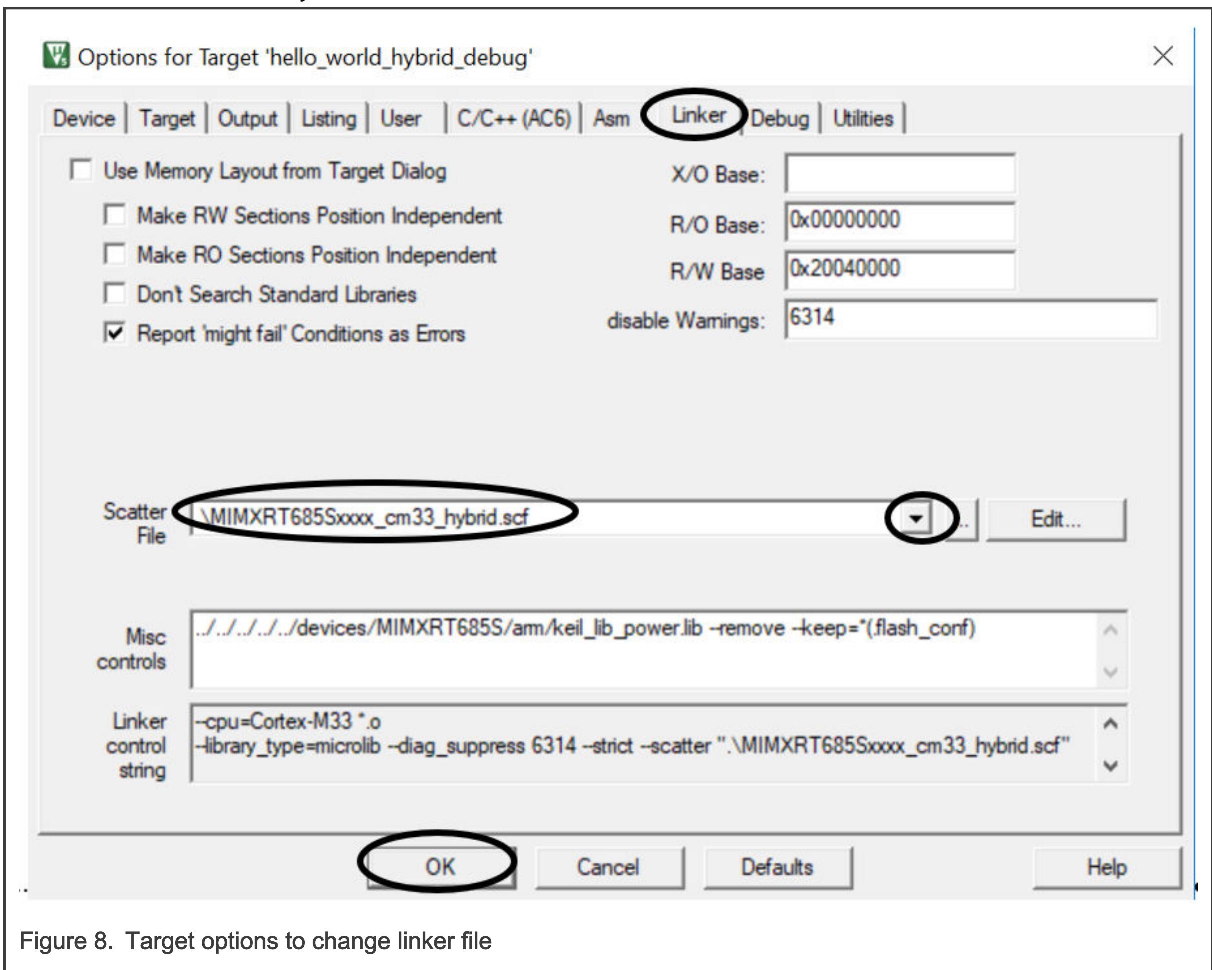

Figure 8. Target options to change linker file

h. See *Getting Started with MCUXpresso SDK for EVK-MIMXRT685* and perform the steps for running the “hello_world” demo using the Keil IDE. Make sure that the project target is the one which was just modified.

i. Place a breakpoint at the function call inside the main function of the *hello_world.c* file and debug. In the address window, the address of the function must be in the external flash.

3.2.2 Steps for MCUXPresso IDE

For the MCUXpresso IDE, modify the FreeMarker Linker script to relocate the code from the flash to the RAM.In this example, run the bulk of application code from the RAM, typically just by leaving the startup code and the vector table along with the

“external_code” object file in the flash. This is achieved by moving three linker script template files into the *linkscripts* folder within the “hello_world” project: *main_text.ldt*, *main_rodata.ldt*, and *main_data.ldt*. The above linker template scripts cause the main body of the code to be relocated into the main (first) RAM bank of the target MCU, which (by default) will also contain *data/bss*, as well as the stack and heap. The boot headers and vector tables must be in the flash, because the boot ROM needs them. The code that performs this relocation is executed early within the reset handler (within the *startup_xx* file). However, there is a potential for other critical functions to be called before this relocation is performed. For example, *SystemInit()* may be called first to perform essential operations, such as enabling the RAM. Any function that is called before the relocation must not be relocated. This is the reason for keeping the *startup_** and *system_** files in the flash in this example. For more details, see Section 17.14, “FreeMarker Linker Script Templates” in the *MCUXpresso IDE User Guide*.

In the *main_text.ldt* file, the following lines indicate the linker to pull the text section from the *startup_*.o*, *system_*.o*, and *external_code.o* object files:

```
*startup_*.o (.text.*)

*system_*.o (.text.*)

*external_code.o (.text. *)
```

In *main_rodata.ldt*, the following lines indicate the linker to pull in the “rodata” and “constdata” sections from the *startup_*.o*, *system_*.o*, and *external_code.o* object files:

```
*startup_*.o (.rodata .rodata.* .constdata .constdata.*)

*system_*.o (.rodata .rodata.* .constdata .constdata.*)

*external_code.o (.rodata .rodata.* .constdata .constdata.*)
```

In *main_data.ldt*, the following lines indicate the linker to pull in the “text”, “rodata”, “constdata”, and “data” sections:

```
*(.text*)

*(.rodata .rodata.* .constdata .constdata.*)

. = ALIGN(${text_align});

*(.data*)
```

The following are the required steps:

1. Follow *Getting Started with MCUXpresso SDK for EVK-MIMXRT685* to import the “ hello_world” project using the MCUXpresso IDE.
2. Add a new configuration called “hybrid_debug” by right clicking on the project , going to the “Manage” option in “Build Configurations”. Create a new configuration which should be based on the existing “Debug” configuration.

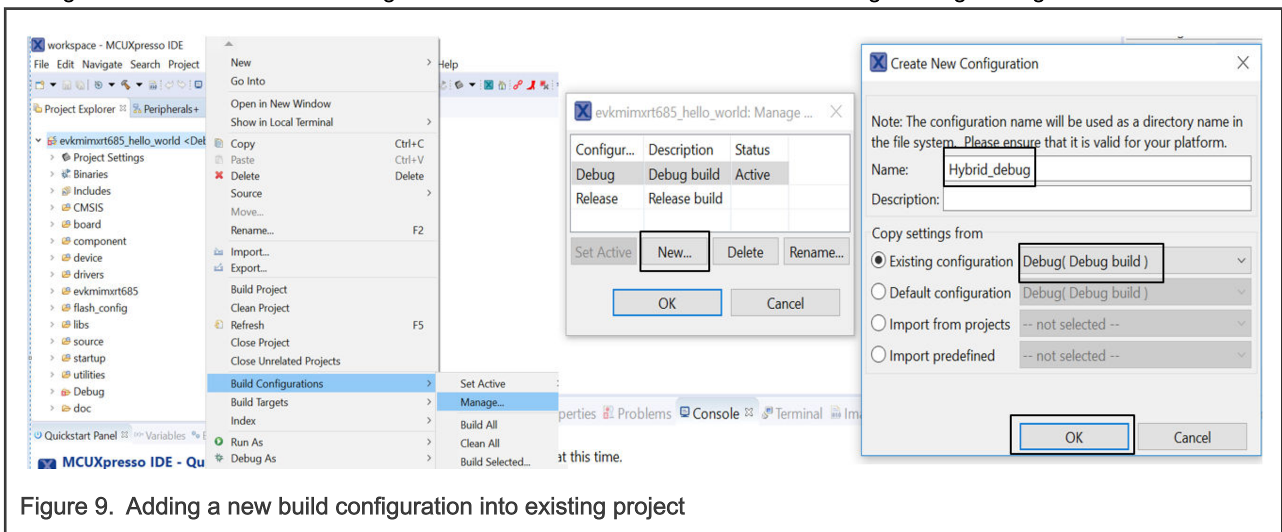

Figure 9. Adding a new build configuration into existing project

3. Set the new “Hybrid_debug” as the active configuration, as shown in Figure 10.

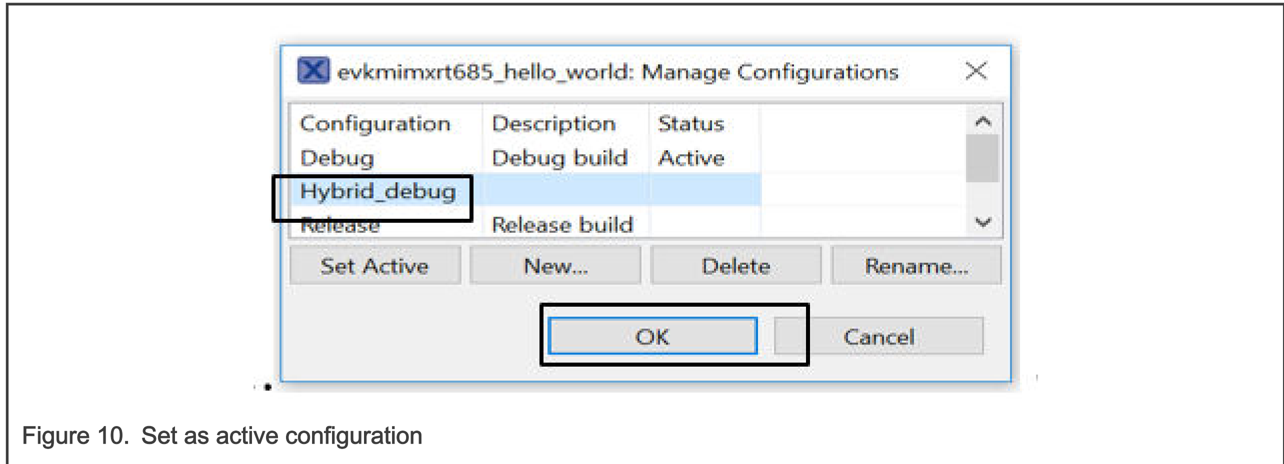

Figure 10. Set as active configuration

4. Extract the *hello_world_hybrid_mcux.zip* file and copy the *external_code.c*, *hello_world.c*, and *external_code.h* files in the *hello_world_hybrid_mcux\source* folder into your "hello_world" project source folder. The project folder location can be found by right clicking into the project in the MCUXpresso IDE and going to the "Resource" option and then selecting the "Show in System Explorer" option.

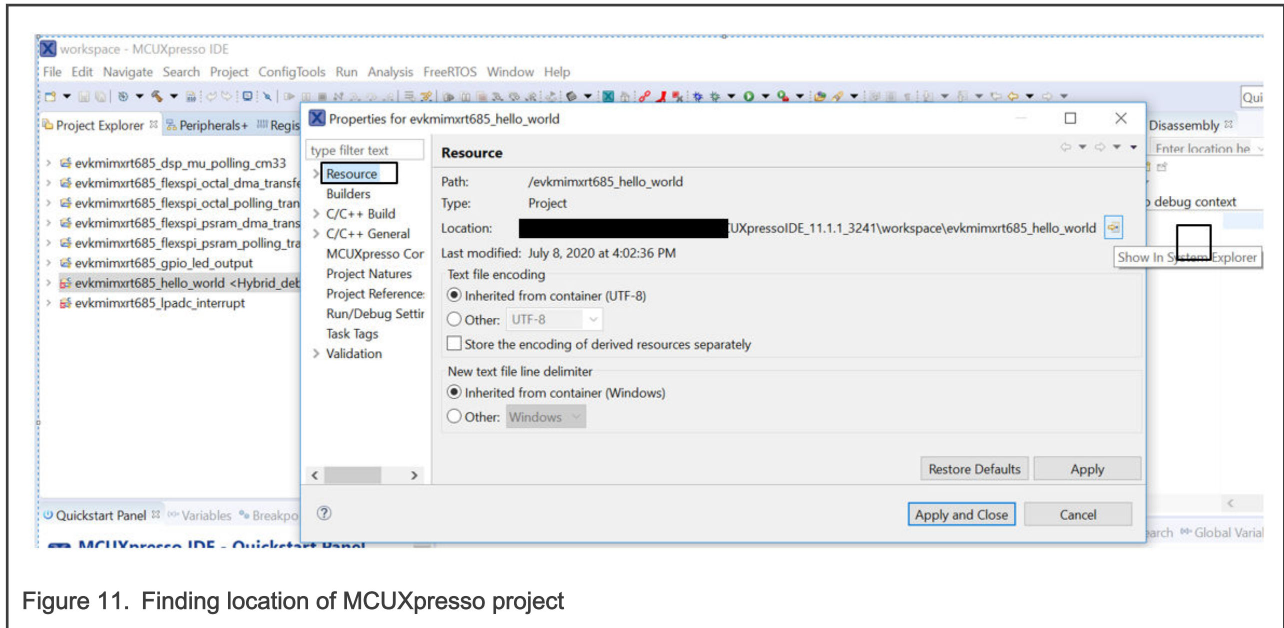

Figure 11. Finding location of MCUXpresso project

5. Copy the *linkscript* folder inside "hello_world_hybrid_mcux" and add it into the "hello_world" project. The folder should contain three files: *main_text.ldt*, *main_rodata.ldt*, and *main_data.ldt*.

6. See *Getting Started with MCUXpresso SDK for EVK-MIMXRT685* and perform the steps to build and run the " hello_wor ld" project using the MCUXpresso IDE.

7. Place a breakpoint inside *external_code.c* and you will see that the debugger moves from the RAM to the flash location.

3.2.3 Steps for IAR IDE

1. Open *hello_world.eww* (located in the *SDK_2.8.2_EVK-MIMXRT685S\boards\evkmimxrt685\demo_apps \ hello_world\iar* folder) using the IAR IDE. This opens the IAR IDE with the example "hello_world" application.

2. Add a new configuration called "hybrid_debug" by clicking on "Project->Edit Configurations…" and create a new configuration which is based on the existing "Debug" configuration.

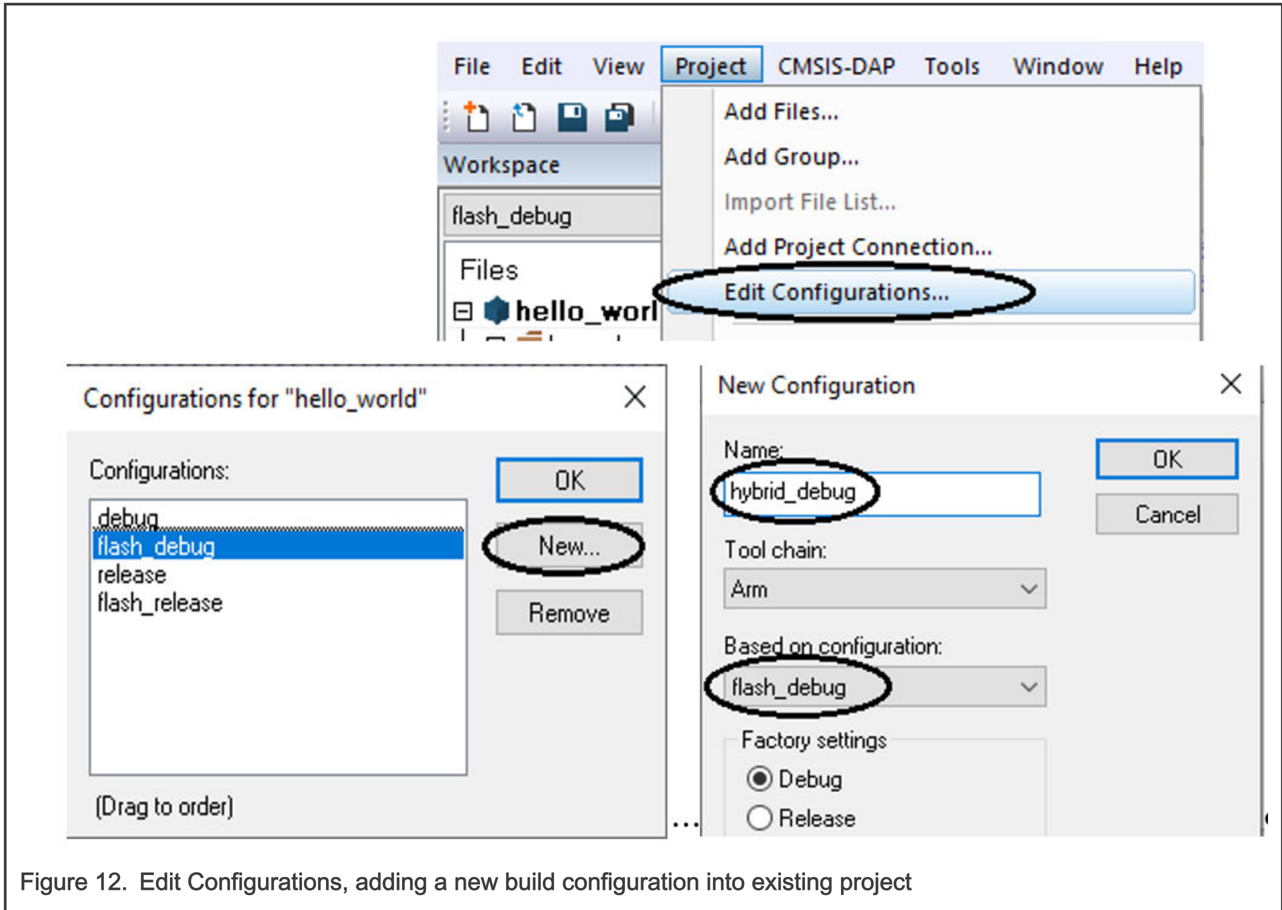

Figure 12. Edit Configurations, adding a new build configuration into existing project

3. Extract the *hello_world_hybrid_iar.zip* file and copy the *external_code.c*, *hello_world.c*, and *external_code.h* files in the *hello_world_hybrid_iar/source* folder into the *SDK_2.8.2_EVK-MIMXRT685S\boards\evkmimxrt685\demo_apps\hello_world* folder. Now add the *external_code.c* file in the source group by right clicking on the source folder of the project, as shown in Figure 13.

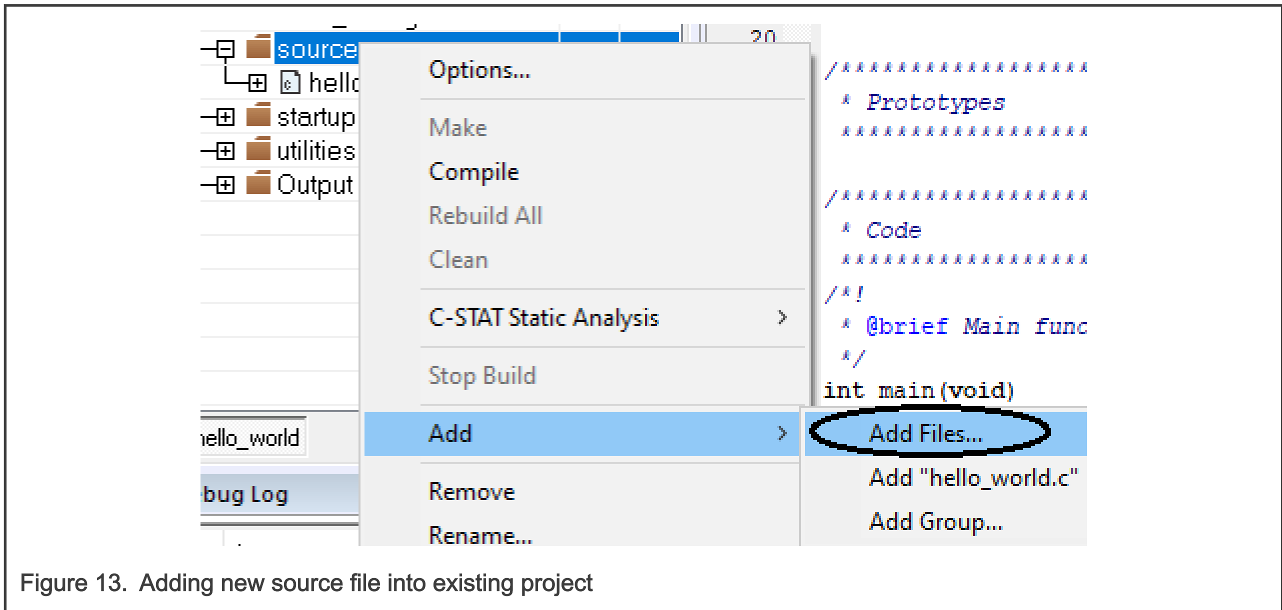

Figure 13. Adding new source file into existing project

4. When the new C code is compiled, its object image (*.o*) can be placed into the external flash for XIP. This can be achieved by modifying the *MIMXRT685Sxxxx_cm33_flash.scf* linker script file. Copy this file and rename it to

MIMXRT685Sxxxx_cm33_hybrid.scf. In the *MIMXRT685Sxxxx_cm33_hybrid.icf* file, make a few changes to program the flash:

```
48  define symbol m_interrupts_start              = 0x08001000;
49  define symbol m_interrupts_end                = 0x0800112F;
50
51  define symbol m_text_start                    = 0x08001130;
52  define symbol m_text_end                      = 0x081FFFFF;
53
54  define symbol m_interrupts_ram_start          = 0x00080000;
55  define symbol m_interrupts_ram_end            = 0x00080000 + __ram_vector_table_offset__;
56
57  define symbol m_data_start                    = m_interrupts_ram_start + __ram_vector_table_size__;
58  define symbol m_data_end                      = 0x001FFFFF;
59
60  define symbol m_usb_sram_start                = 0x40140000;
61  define symbol m_usb_sram_end                  = 0x40143FFF;
```

Figure 14. Changes to program the flash

Addresses 0x20080000 and 0x00080000 point to the same offset on the same SRAM (just the alias address). The only difference are the CM33 core access addresses below 0x20000000 with the code bus, and access the upper address by the system bus. For this use case, putting code on the code bus should be more efficient. Therefore, 0x00080000 is recommended in the linker file .

Now add the following lines (line# 87-89) of code into the *MIMXRT685Sxxxx_cm33_hybrid.icf* file to copy all of the code to the RAM excluding the *startup*, *system*, and *external_code.o* files.

```
82
83  /* regions for USB */
84  define region USB_BDT_region = mem:[from m_usb_sram_start to m_usb_sram_start + usb_bdt_size - 1];
85  define region USB_SRAM_region = mem:[from m_usb_sram_start + usb_bdt_size to m_usb_sram_end];
86
87  initialize by copy { readonly, readwrite, section .textrw, section CodeQuickAccess, section DataQuickAccess}
88              except { section .rodata, section .flash_conf, readonly object startup_MIMXRT685S_cm33.o, readonly object system_MIMXRT685S_cm33.o,
89                       readonly object external_code.o};
90
91  do not initialize  { section .noinit, section m_usb_bdt, section m_usb_global  };
```

Figure 15. Adding lines

The RAM code is copied in the last step by *iar_program_start()*. All code/data which is accessed before *iar_program_start()* must not be relocated. That is why you must keep the vector table, *Reset_Handler()*, and *SystemInit()* in the flash. Also, ".flash_config" is for ROM use and it should not be relocated as well.

Open the options window by right clicking the project and then open the "Linker" tab. Using the highlighted button, place the *MIMXRT685Sxxxx_cm33_hybrid.icf* file as the linker file, as shown in Figure 16.

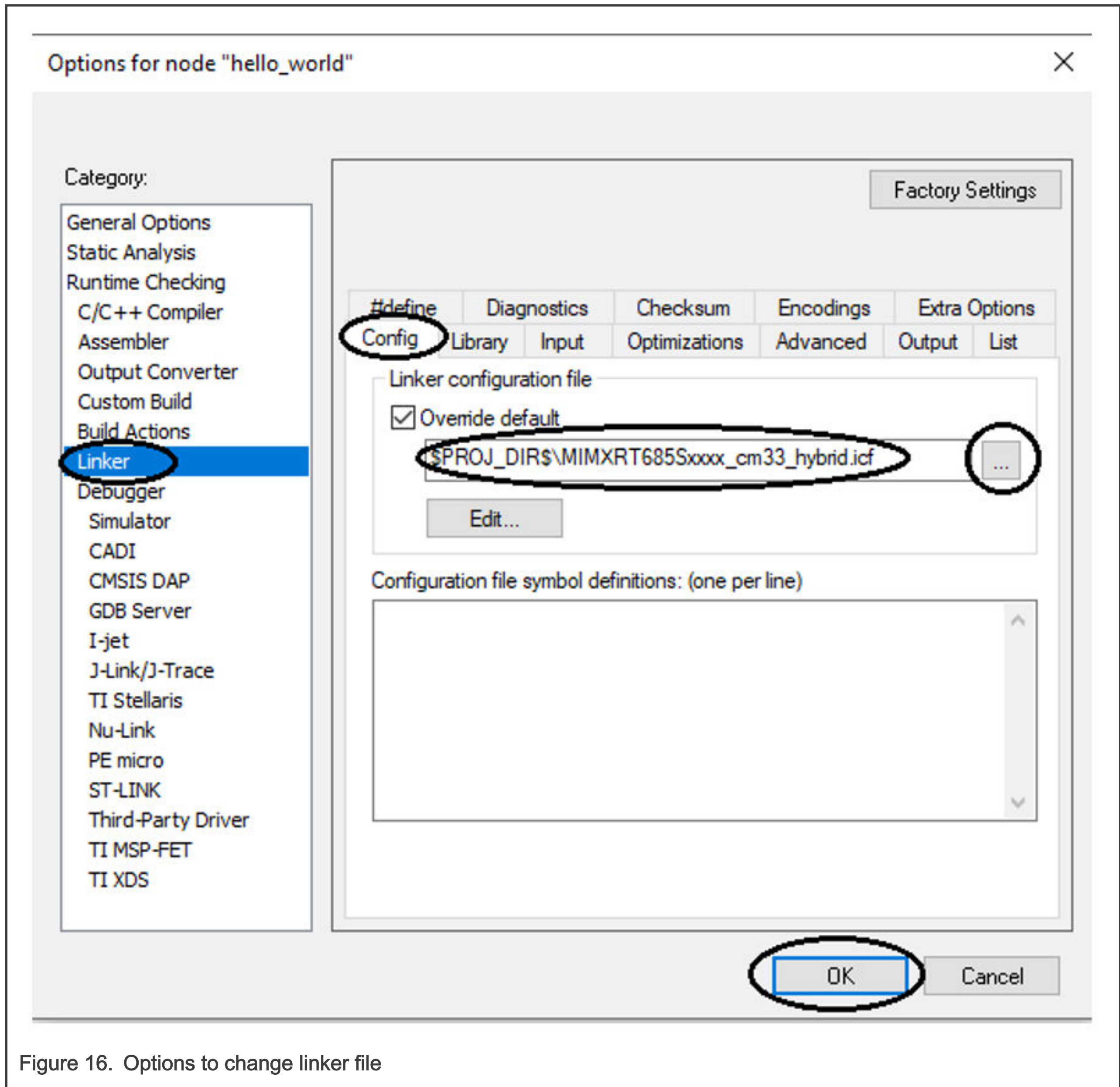

Figure 16. Options to change linker file

5. See *Getting Started with MCUXpresso SDK for EVK-MIMXRT685* for the steps to build and run the “hello_world” demo using the IAR IDE. Make sure that the project target is the one which was modified until now.
6. Place a breakpoint at the function call inside the main function of the *hello_world.c* file and you will see that the debugger moves from the RAM to the flash location from the address window.

NOTE

By default, the IDE decides which kind of breakpoint can be used. Because we have only eight hardware breakpoints, the IDE always tries to use the software breakpoint first. The software breakpoint is just a special instruction written in the RAM. During debugging, the IAR IDE firstly downloads the program into the flash,resets the system, and halts before the startup code. Then it sets the breakpoint at *main()* and continues to run. Because the breakpoint at *main()* is a software breakpoint, it will be overwritten after the startup code relocates (from the flash to the RAM in *iar_program_start()*). You can set the breakpoint and debug with either of the below configurations (a) or (b).

a. Force the IAR IDE to use hardware breakpoints with the limitation that only eight breakpoints are available in debugging.

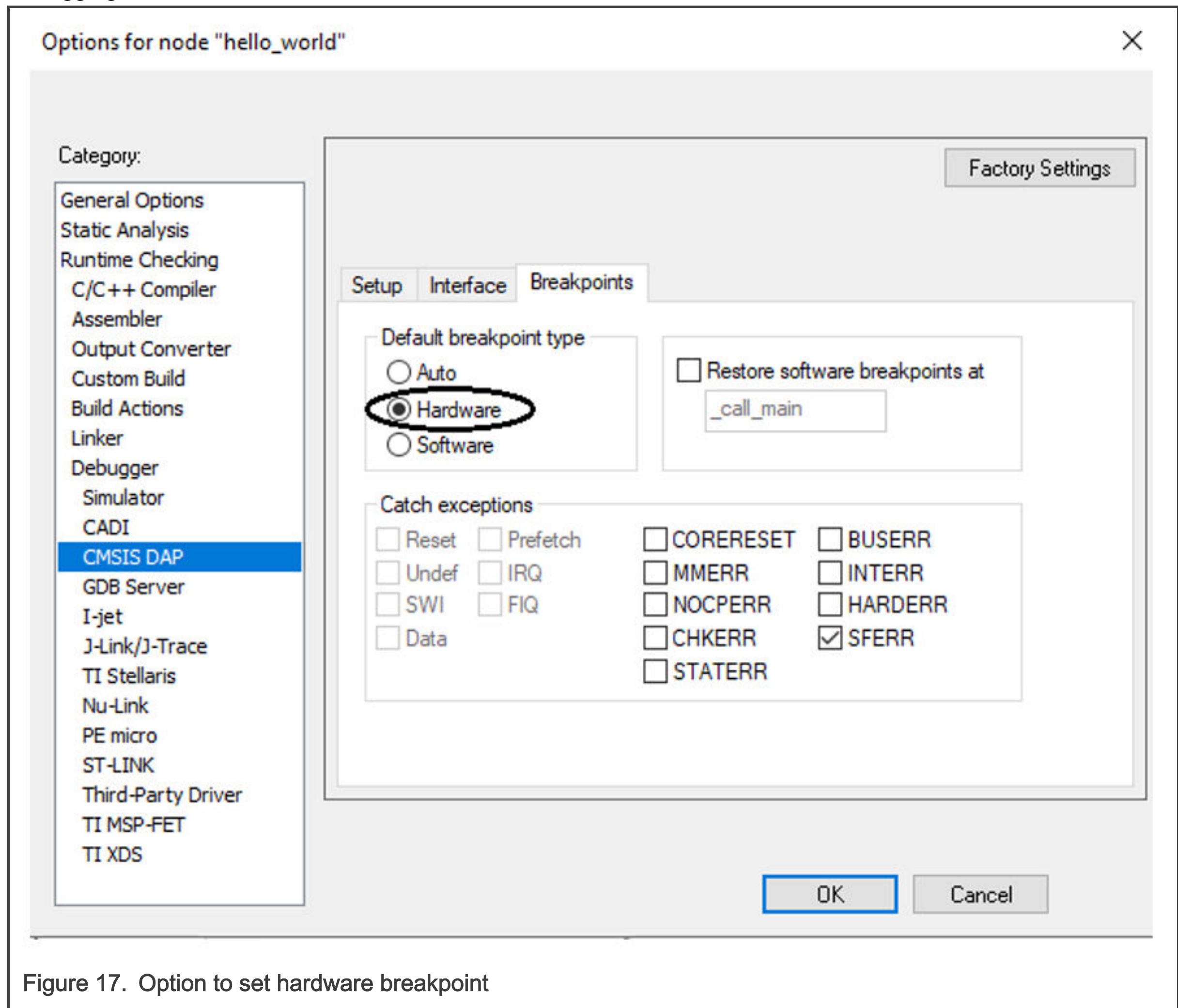

Figure 17. Option to set hardware breakpoint

b. Make the IAR IDE stop at *call_main()*. At *call_main()* , the code relocation/data copy completes, and then it is safe to set the software breakpoint.

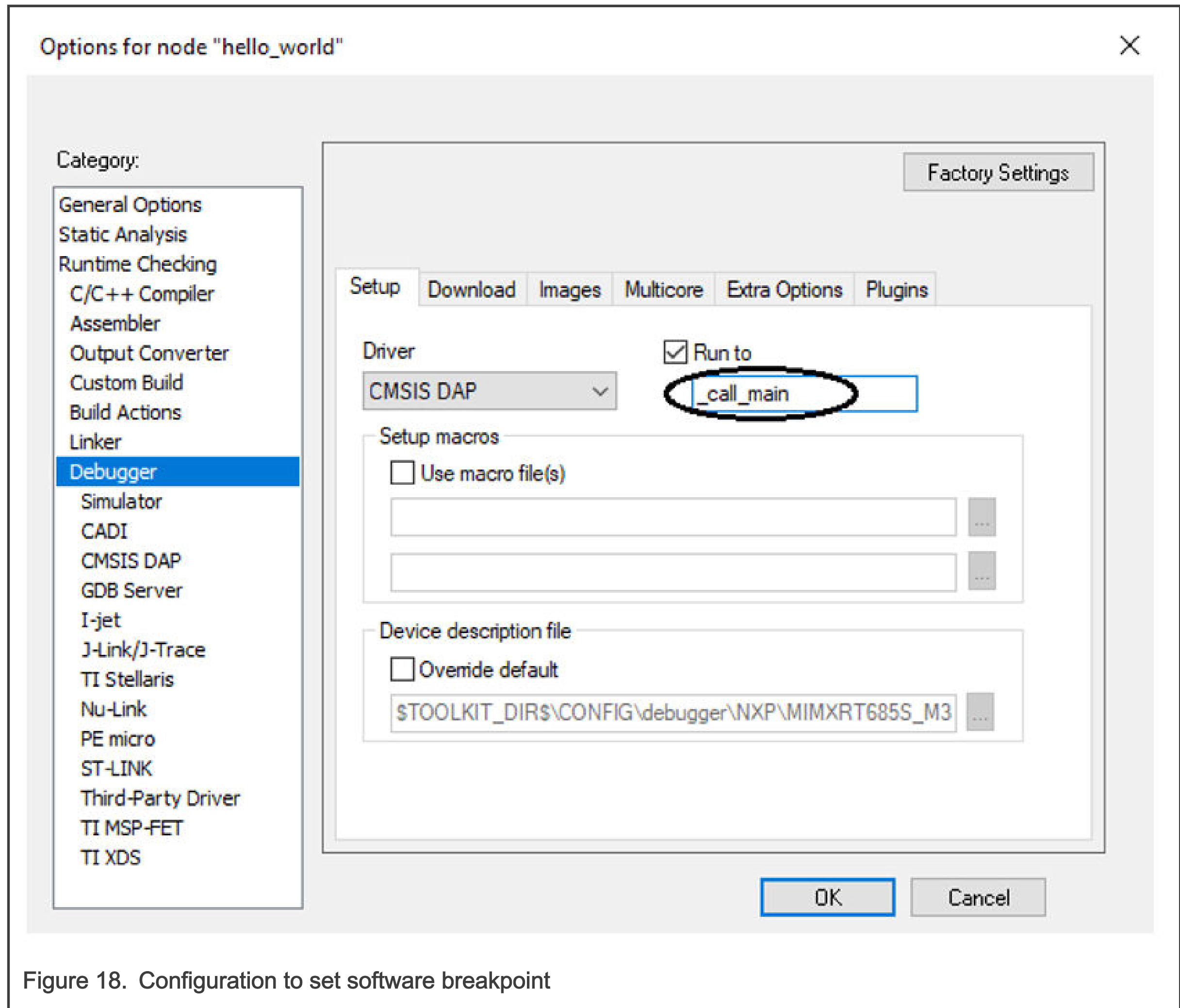

Figure 18. Configuration to set software breakpoint

3.3 Methodology for programming the flash

The idea is to place the complete image to the flash memory (non XIP) for it to be booted onto the SRAM. When the complete image is booted onto the SRAM, the execution starts. Because the linker script has already been modified to load some part of the code from the external flash memory, it will be executed on the flash only (XIP).

To program the external flash, NXP's "blhost" application is required. The *blhost.exe* (Windows OS host machine) file is present in the *SDK_2.8.2_EVK-MIMXRT685S\middleware\mcu-boot\bin\Tools\blhost\win* directory.

NOTE

It is recommended to use the "blhost" application with Windows Powershell.

See the *blhost User Guide* (document MCUBLHOSTUG) to get started with the "blhost" application.

The FlexSPI boot image can be either the XIP image or the Non-XIP image. The XIP image can only be linked at address 0x08001000 and the first 4 KB of the FlexSPI map region is used to store the flash config block.

A Non-XIP image should be linked into the internal 4.5 MB SRAM. As the first 112 KB of SRAM has been occupied by the ROM after the boot and the region 0x1C000 - 0x7FFFF is the shared memory between the DSP and Cortex-M33, it is better to link the Non-XIP image from 0x80000. For applications which do not use the DSP, the Non-XIP image can be linked starting from 0x1C000.

3.3.1 Steps for programming flash

1. See the following path to retrieve the complete binary image which will be loaded onto the flash using the “blhost” commands:
   - For the Keil IDE *SDK_2.8.2_EVK-MIMXRT685\boards\evkmimxrt685\Projects\hello_world\mdk\debug*
   - For the IAR IDE *SDK_2.8.2_EVK-MIMXRT685\boards\evkmimxrt685\Projects\hello_world\iar\flash_debug*

   NOTE

   For the MCUXpresso IDE, to convert the **.axf* file to the **.bin* file, right-click the project in the workspace and then select “Binary Utilites-> Create Binary” or open the project properties by right clicking. In the left-hand list of the “Properties” window, open “C/C++ Build” and select “Settings”. Select the “Build steps” tab and, in the “Post-build steps - Command” field, click “Edit...”. Uncomment the following line: arm-none-eabi-objcopy -v -O binary "${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.bin". Then click “OK” and “Apply and close”.

2. Set up the hardware to ensure booting from the FlexSPI-enabled NOR flash. For this, the settings for SW 5 on the RT685 EVK are as follows: The ISP0 is on/high, and ISP1 and ISP2 are both low, as shown in Figure 19.

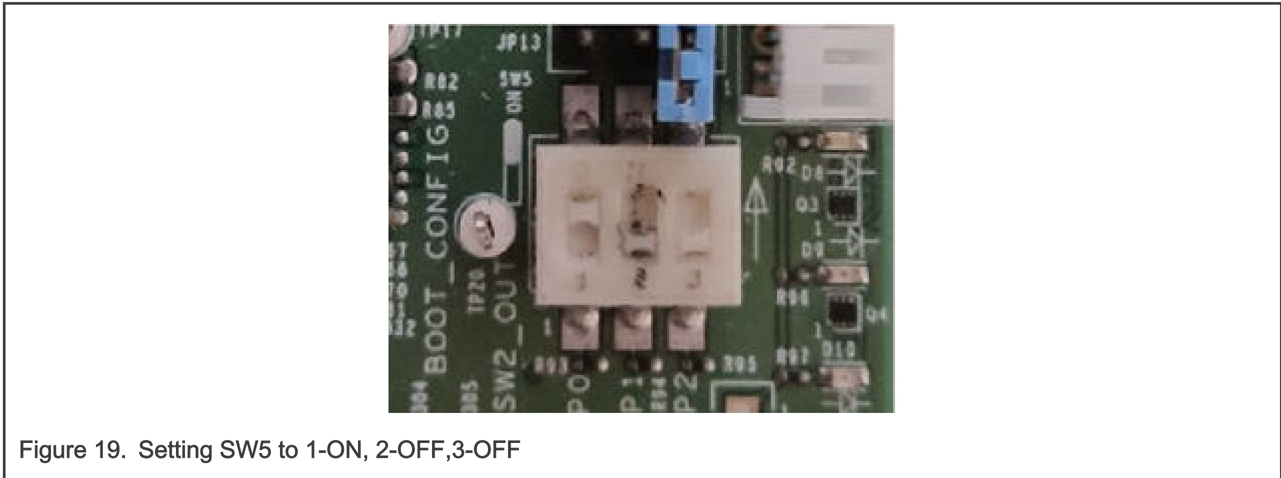

Figure 19. Setting SW5 to 1-ON, 2-OFF,3-OFF

3. Open the Powershell terminal in the “blhost” directory (*middleware/mcu-boot/bin/Tools/blhost/win*). Place the generated binary for the “hello_world” demo into this folder.
4. Connect a USB cable to the J7 USB port and issue the following “blhost” commands using Powershell:
   a. Configure the FlexSPI controller to program the flash:

      ./blhost -u 0x1fc9,0x0020 -- fill-memory 0x1c000 4 0xC1503051

      ./blhost -u 0x1fc9,0x0020 -- fill-memory 0x1c004 4 0x20000014

      ./blhost -u 0x1fc9,0x0020 -- configure-memory 9 0x1c000

   b. Erase the region to be programmed:

      ./blhost -u 0x1fc9,0x0020 -- flash-erase-region 0x08000000 0x6000

   c. Program the image to the flash at 0x08000000:

      ./blhost -u 0x1fc9,0x0020 -- write-memory 0x08000000 .\hello_world.bin

   NOTE

   When examining the *hello_world.bin* file in a HEX editor, the **.bin* starts from 0x08000000 and it is zero-filled from offset 0x0-0x400 for the MCUXpresso IDE. Therefore, the image should be programmed starting at 0x08000000 for steps b and c. For the Keil IDE, this address should be also 0x08000000. This address varies for other toolchains. In the IAR IDE, the **.bin* image starts from the FCB address at 0x08000400 (when BOOT_HEADER_ENABLE=1) and does not zero-fill from 0x08000000. For the IAR IDE, the generated binaries use 0x8000400 for programming image at steps b and c.

```
PS C:\nxp\blhost_2.6.2\bin\win> ./blhost -u 0x1fc9,0x0020 -- fill-memory 0x1c000 4 0xc1503051
Inject command 'fill-memory'
Successful generic response to command 'fill-memory'
Response status = 0 (0x0) Success.
PS C:\nxp\blhost_2.6.2\bin\win> ./blhost -u 0x1fc9,0x0020 -- fill-memory 0x1c004 4 0x20000014
Inject command 'fill-memory'
Successful generic response to command 'fill-memory'
Response status = 0 (0x0) Success.
PS C:\nxp\blhost_2.6.2\bin\win> ./blhost -u 0x1fc9,0x0020 -- configure-memory 9 0x1c000
Inject command 'configure-memory'
Successful generic response to command 'configure-memory'
Response status = 0 (0x0) Success.
PS C:\nxp\blhost_2.6.2\bin\win> ./blhost -u 0x1fc9,0x0020 -- flash-erase-region 0x08000400 0x6000
Inject command 'flash-erase-region'
Successful generic response to command 'flash-erase-region'
Response status = 0 (0x0) Success.
PS C:\nxp\blhost_2.6.2\bin\win> ./blhost -u 0x1fc9,0x0020 -- write-memory 0x08000400 .\hello_world.bin
Inject command 'write-memory'
Preparing to send 14944 (0x3a60) bytes to the target.
Successful generic response to command 'write-memory'
(1/1)100% Completed!
Successful generic response to command 'write-memory'
Response status = 0 (0x0) Success.
Wrote 14944 of 14944 bytes.
```

Figure 20. blhost commands sequence

The argument values 0xc1503051 and 0x20000014 in the fill-memory command is the FlexSPI boot configuration option block.

5. Switch the RT685-EVK board to the FlexSPI Port B boot mode by setting SW5 to 1-ON, 2-OFF, and 3-ON, as shown in Figure 21.

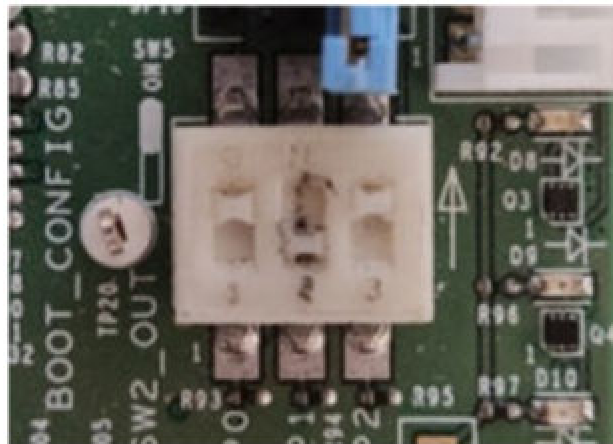

Figure 21. Setting SW5 to 1-ON, 2-OFF, 3-ON

6. Reset the board and connect the USB cable to the J5 port and the “hello_world” demo should run successfully.

4 Conclusion

This application note shows how some part of code can be booted from the external flash into the internal RT685 SRAM and how the remaining code can continue to reside in flash to be fetched/executed directly. The example explains how to change the linker file to do this hybrid booting in details.

5 References

1. *RT600 User Manual* (document UM11147)

2. *RT600 Data Sheet*

3. *MCUXpresso SDK Release Notes for EVK-MIMXRT685* (located inside the SDK)
4. *Getting Started with MCUXpresso SDK for EVK-MIMXRT685* (located inside the SDK)
5. *MCUXpresso IDE User Guide*
6. *MCU blhost User Guide* (document MCUBLHOSTUG)
7. *How to Enable Master Boot from Serial NOR Flash* (document AN12773)

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

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

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

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2020. All rights reserved.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 09/2020
Document identifier: AN12985