Freescale Semiconductor

Merging applications using Kinetis Design Studio

By: Carlos Mendoza / Technical Information Center

About this document

Usually a user application can be developed independently, that is without the bootloader, and it can be loaded into the microcontroller and debugged directly. However, for production purposes it is worth merging the user application and bootloader together, so it can be downloaded into the microcontroller all at once as a single file and reduce manufacture time and cost.

This document shows two methods of merging the applications.

1. Merging applications using linker commands.
2. Merging applications using the P&E Advanced Flash Programming options.

The steps described in the document were done using the MK64FN1M0VLL12 MCU like the one in the FRDM-K64F board, but the same principles are applicable to any Kinetis MCU.

Software versions

The steps described in this document are valid for the following versions of the software tools:

- KDS v3.0.0

Contents

1. Glossary

KDS ***Kinetis Design Studio*****:** Integrated Development Environment (IDE) software for Kinetis MCUs.

KSDK ***Kinetis Software Development Kit*****:** Set of peripheral drivers, stacks and middleware layers for Kinetis microcontrollers.

2. Overview and concepts

2.1 Linker File (.ld)

The Linker file (.ld) combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run ld.

2.1.1 Memory Segment

The memory segment is used to divide the microcontroller memory into segments. Each segment can have read, write and execute attributes. The address and the length of each segment are defined as well. An example is shown in listing 1.

```
MEMORY
{
  m_interrupts          (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000400
  m_flash_config        (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x00000410, LENGTH = 0x000FFBF0
  m_data                (RW)  : ORIGIN = 0x1FFF0000, LENGTH = 0x00010000
  m_data_2              (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00030000
}
```

Listing 1 – K64 Memory segment

2.1.2 Sections Segment

In sections segment are defined the contents of target-memory sections. In other words, a section indicates which parts of your application will be allocated in each memory segment. Main sections are '.text' which contains all the code and the constants of an application, '.data' which contains all initialized data, and '.bss' which contains all non-initialized data.

Below you can see section '.text' of an application using K64. As you can notice it is contained in segment 'm_text'.

```
.text :
{
  . = ALIGN(4);
  *(.text)                  /* .text sections (code) */
  *(.text*)                 /* .text* sections (code) */
  *(.rodata)                /* .rodata sections (constants, strings, etc.) */
  *(.rodata*)               /* .rodata* sections (constants, strings, etc.) */
  *(.glue_7)                /* glue arm to thumb code */
  *(.glue_7t)               /* glue thumb to arm code */
  *(.eh_frame)
  KEEP (*(.init))
  KEEP (*(.fini))
  . = ALIGN(4);
} > m_text
```

Listing 2 – K64 .text section

3. Merging two applications

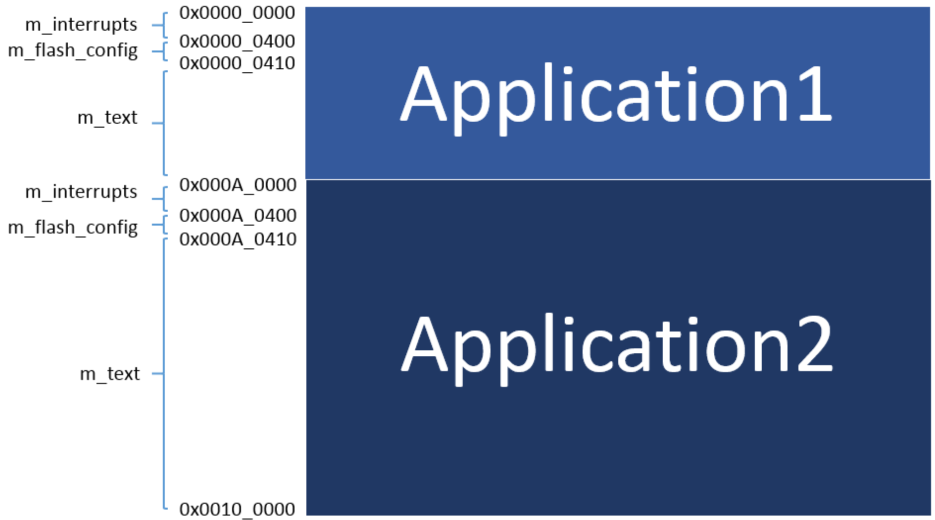
3.1 Creating applications

- Before we start merging the projects we need to create 2 new baremetal or KSDK projects (No Processor Expert) using KDS, select MK64FN1M0xxx1 as device. Let's call them **Application1** and **Application2**.

- **Application1** is supposed to be the Bootloader, as example it will just toggle the blue LED 5 times. It will be linked to the default memory address which is 0x0000 and will end at 0xA000 (the flash space reserved for the bootloader will depend on the size of your bootloader).

- **Application2** is supposed to be the user Application, as example it will toggle the red LED forever. It will be linked to the address 0xA000 and will end at 0x0010_0000.

- Here is the Flash memory layout of how the bootloader and application will be programmed:



- In a default project the memory segments of MK64FN1M0xxx1 are defined in the linker file as next:

```
MEMORY
{
  m_interrupts          (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000400
  m_flash_config        (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x00000410, LENGTH = 0x000FFBF0
  m_data                (RW)  : ORIGIN = 0x1FFF0000, LENGTH = 0x00010000
  m_data_2              (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00030000
}
```

Listing 3 – Default Memory segments

- First we need to reduce the memory size in **Application1** to leave some space and create a new memory segment for **Application2** which will start in address 0xA000. Open MK64FN1M0xxx12_flash.ld located in "${ProjDirPath}/Project_Settings/Linker_Files", the MEMORY segment must look as below after being edited.

```
MEMORY
{
  m_interrupts          (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000400
  m_flash_config        (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x00000410, LENGTH = 0x00009BF0
  app2_text             (RX)  : ORIGIN = 0x0000A000, LENGTH = 0x000F6000
  m_data                (RW)  : ORIGIN = 0x1FFF0000, LENGTH = 0x00010000
  m_data_2              (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00030000
}
```

Listing 4 – Application1 modified Memory segments

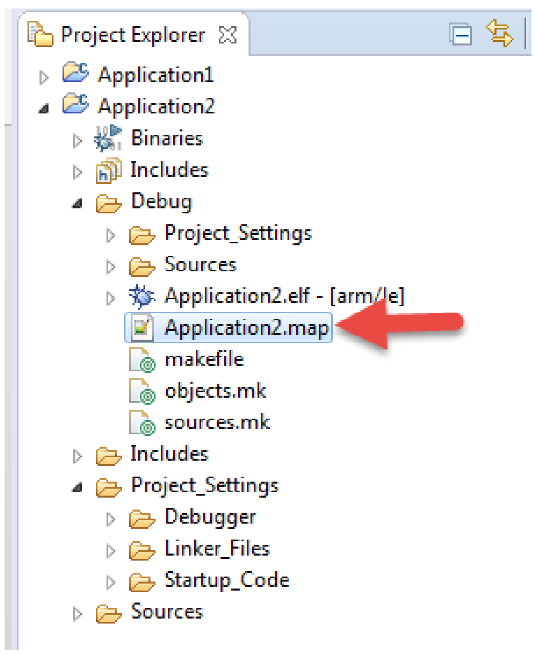
[m_text] *Bootloader space*
[app2_text] *Application space*

- As it was mentioned before, the **Application2** must be linked in address 0xA000, therefore we must add an offset of 0xA000 to all the flash segments. Open MK64FN1M0xxx12_flash.ld located in "${ProjDirPath}/Project_Settings/Linker_Files", the MEMORY segment must be edited as follows:

```
MEMORY
{
  m_interrupts          (RX)  : ORIGIN = 0x0000A000, LENGTH = 0x00000400
  m_flash_config        (RX)  : ORIGIN = 0x0000A400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x0000A410, LENGTH = 0x000F5BF0
  m_data                (RW)  : ORIGIN = 0x1FFF0000, LENGTH = 0x00010000
  m_data_2              (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00030000
}
```

Listing 5 – Application2 modified Memory segments

- Build **Application2**, then look for Application2.map which you will find "${ProjDirPath}/Debug":



- Open the Application2.map file and search for the Reset_Handler which is the application entry point address. As you can see in this case it is 0x0000a4d8:

```
.text     0x0000a4d8    0x30  ./Project_Settings/Startup_Code/startup_MK64F12.o
          0x0000a4d8          Reset_Handler
```

Listing 6 – Application2.map Entry Point

- Now go back to **Application1**. We will make this application to toggle the onboard blue LED 5 times and then jump to the entry code of **Application2** (Reset_Handler). You can use the following code:

```
/* include peripheral declarations "fsl_device_registers.h" if it is a KSDK project or
"MK64F12.h" if it is a baremetal project */

#define GPIO_PIN_MASK            0x1Fu
#define GPIO_PIN(x)              (((1)<<(x & GPIO_PIN_MASK)))

void delay();

int main(void){

	int i;
	/* Turn on all port clocks */
	SIM_SCGC5 = SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTB_MASK |
SIM_SCGC5_PORTC_MASK | SIM_SCGC5_PORTD_MASK | SIM_SCGC5_PORTE_MASK;
	/*Set PTB21 (connected to BLUE LED) for GPIO functionality*/
	PORTB_PCR21=(0|PORT_PCR_MUX(1));
	/*Change PTB21 to output*/
	GPIOB_PDDR=GPIO_PDDR_PDD(GPIO_PIN(21));

	for(i = 0; i < 10; i++){
		/*Toggle the blue LED on PTB21*/
		GPIOB_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(21));
		delay();
	}

	__asm("bl 0x0000a4d8"); //Jump to Application2 entry point

	return 0;
}


void delay(){
  unsigned int i, n;
  for(i=0;i<10000;i++){
	for(n=0;n<200;n++){
		__asm("nop");
	}
  }
}
```

Listing 7 – Application1 main.c

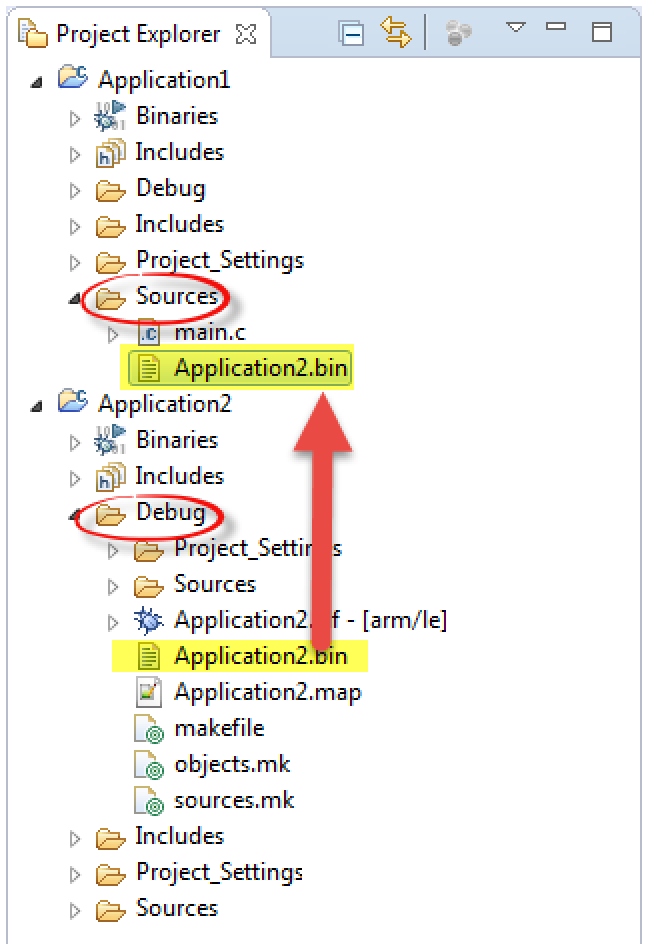
- The next step is to make **Application2** toggle the Red LED forever. You can use the following code:

```
/* include peripheral declarations "fsl_device_registers.h" if it is a KSDK project or
"MK64F12.h" if it is a baremetal project */


#define GPIO_PIN_MASK            0x1Fu
#define GPIO_PIN(x)              (((1)<<(x & GPIO_PIN_MASK)))

void delay();

int main(void){

       /* Turn on all port clocks */
       SIM_SCGC5 = SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTB_MASK |
SIM_SCGC5_PORTC_MASK | SIM_SCGC5_PORTD_MASK | SIM_SCGC5_PORTE_MASK;
       /*Set PTB22 (connected to RED LED) for GPIO functionality*/
       PORTB_PCR22=(0|PORT_PCR_MUX(1));
       /*Change PTB22 to output*/
       GPIOB_PDDR=GPIO_PDDR_PDD(GPIO_PIN(22));

       while(1){
              /*Toggle the RED LED on PTB22*/
              GPIOB_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(22));
              delay();
       }

       return 0;
}


void delay(){
  unsigned int i, n;
  for(i=0;i<10000;i++){
         for(n=0;n<200;n++){
                __asm("nop");
         }
  }
}
```

Listing 8 – Application2 main.c

3.2 Merging applications using linker commands

- Now that the two applications have been created we will merge them using linker commands, first we need to generate a binary file of the **Application2** to insert it in the **Application1**, to do this follow the steps described in the Appendix A. After this, copy the binary file into the 'Sources' folder in **Application1**.



- The next step is to tell the **Application1** linker to include the **Application2** binary file. First you need to use TARGET, INPUT and OUTPUT_FORMAT commands. You can do this just after MEMORY segment and before SECTIONS segment:

```
/* Specify the memory areas */
MEMORY
{
  m_interrupts          (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000400
  m_flash_config        (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x00000410, LENGTH = 0x00009BF0
  my_text               (RX)  : ORIGIN = 0x0000A000, LENGTH = 0x000F6000
  m_data                (RW)  : ORIGIN = 0x1FFF0000, LENGTH = 0x00010000
  m_data_2              (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00030000
}

TARGET(binary)                  /* specify the file format of binary file */
INPUT (Application2.bin)        /* provide the file name */
OUTPUT_FORMAT(default)          /* restore the out file format */


/* Define output sections */
SECTIONS
{

…
```

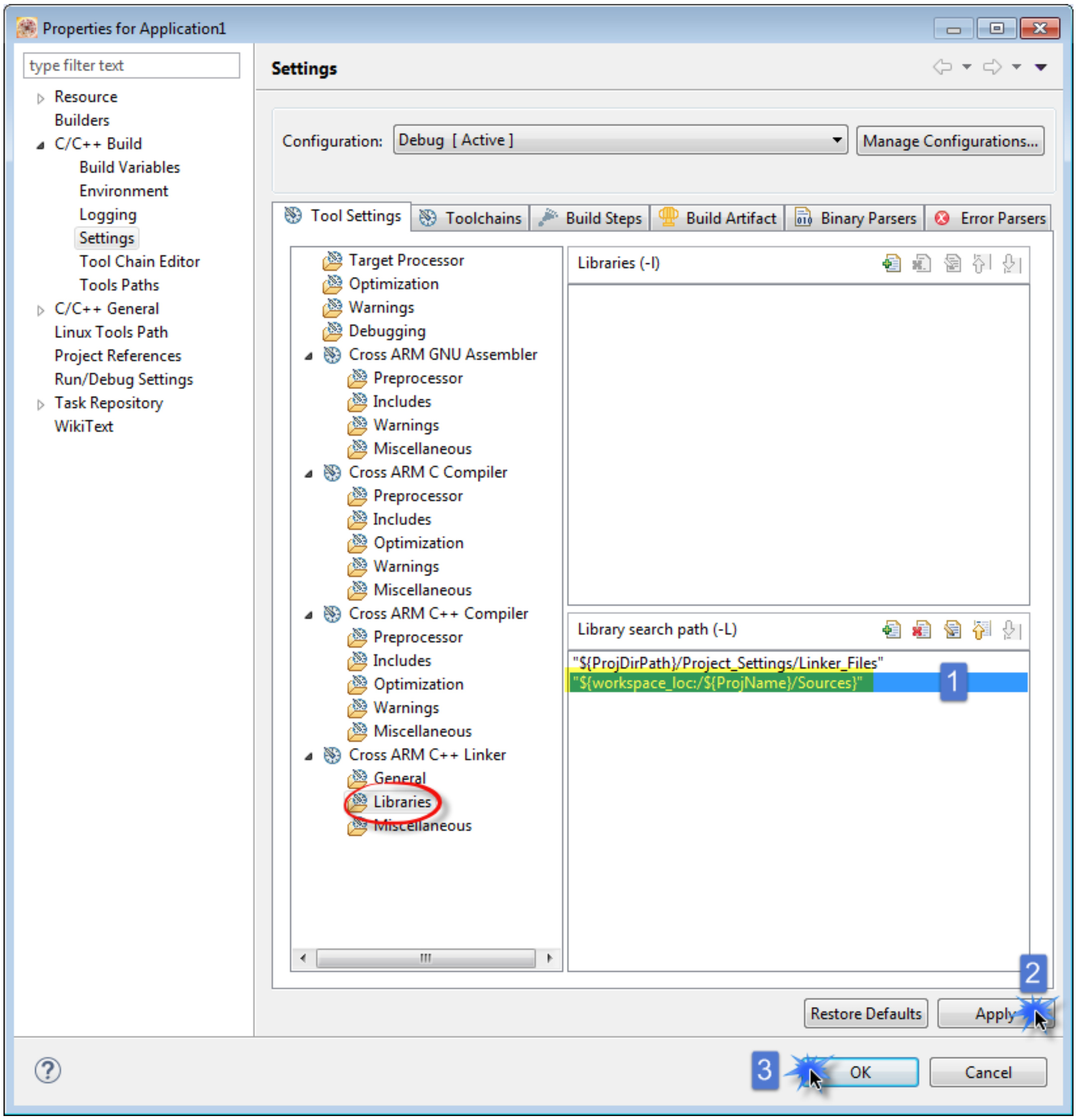
Listing 9 – Application1 modified linker file

- Then add a new section inside SECTIONS segment to tell the linker where to allocate this binary file. You can call this section '.app2' and put it just before section '.data'. Notice this section is contained in segment 'my_text'.

```
.app2 :
{
      Application2.bin (.data)
      . = ALIGN (0x4);
} > app2_text


.data : AT(__DATA_ROM)
{

…
```

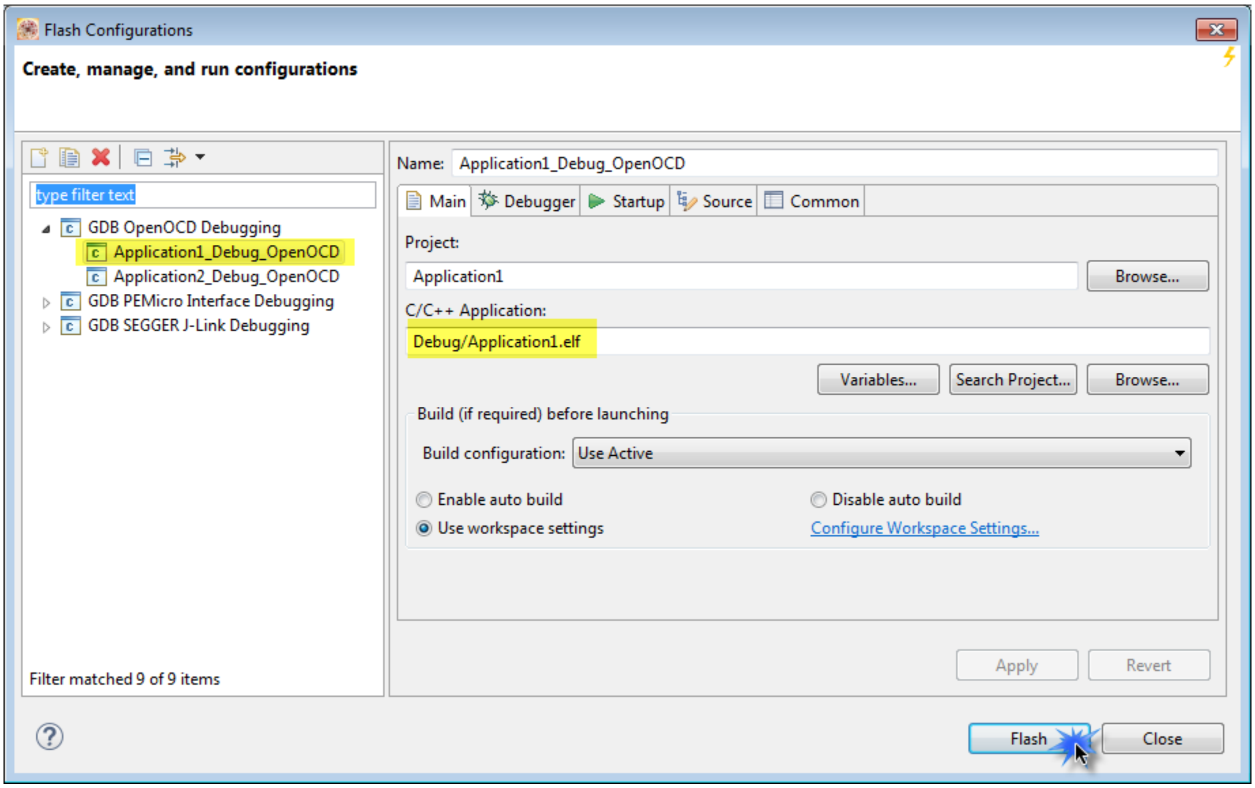
Listing 10 – Application1 new app2 section

– Finally go to menu Project > Properties > C/C++ Build > Settings > ARM Ltd Windows GCC C Linker > Libraries and add under ‘Library search path (-L)’ the Sources folder path "${workspace_loc:/${ProjName}/Sources}", this way the linker will be able to find the **Application2** binary file.



–

– Now you can build and program the application using ‘Flash from file…’ option. Just click ‘Flash from file…’ icon to get the Flash Configurations menu, choose your connection and the .elf file generated by **Application1** and click on the ‘Flash’ button.



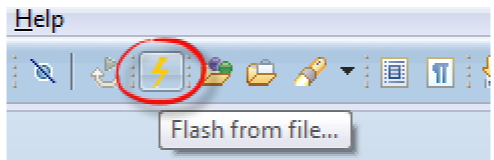
– Reset your board and you should see the blue LED toggle 5 times which indicates the **Application1** is being executed after this the red LED will start to toggle indicating that the program jumped to the **Application2**.

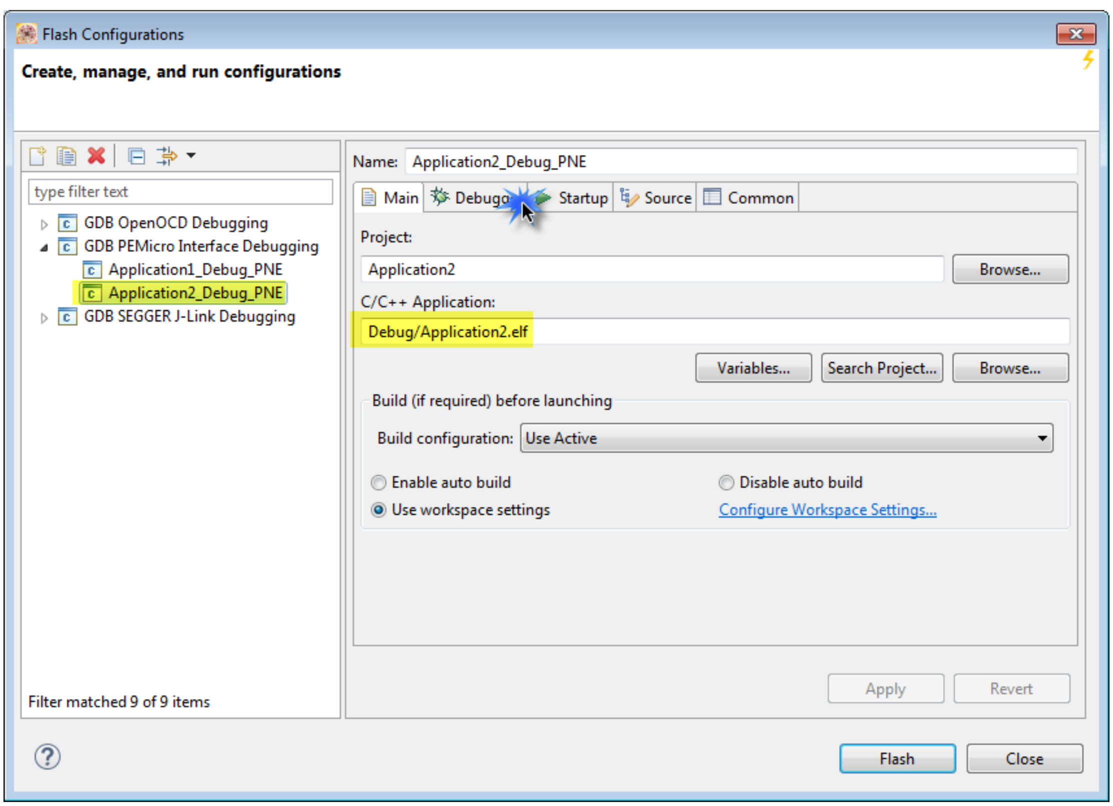
3.3 Merging applications using the P&E Advanced Flash Programming options

- After creating the two applications following the steps described in the section 3.1 we will now proceed to merge them using the P&E Advanced Flash Programming options flash the Application2 to the MCU.

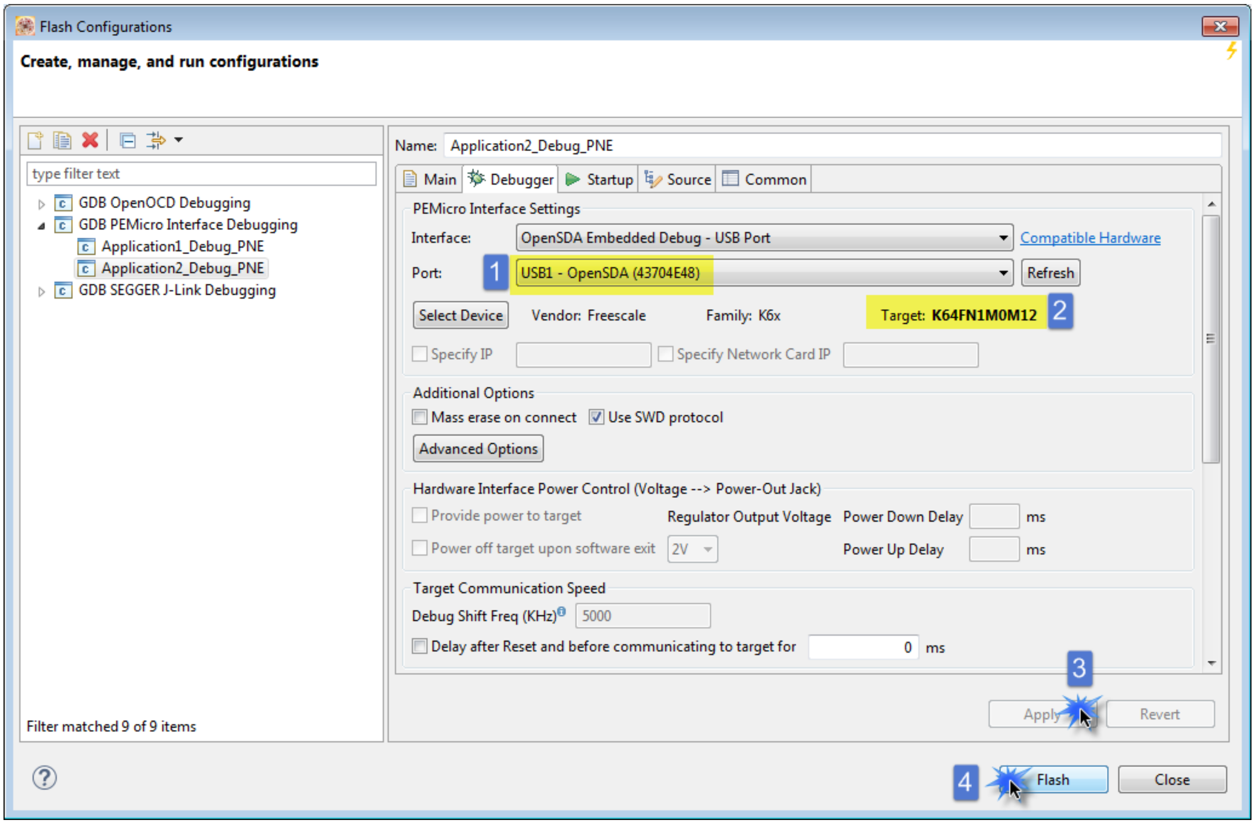
- For this section we will need our board to have the P&E OpenSDA firmware, you can find more information on how to load this firmware to your board on this link: https://community.freescale.com/docs/DOC-105199

- Build and program the Application2 using 'Flash from file…' option. Just click 'Flash from file…' icon to get the Flash Configurations menu, choose the "GDB PEMicro Interface Debugging" connection, and select the .elf file generated by **Application2**.



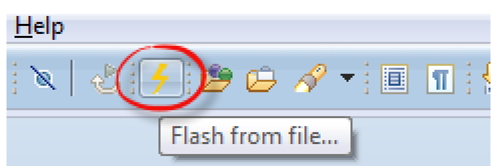


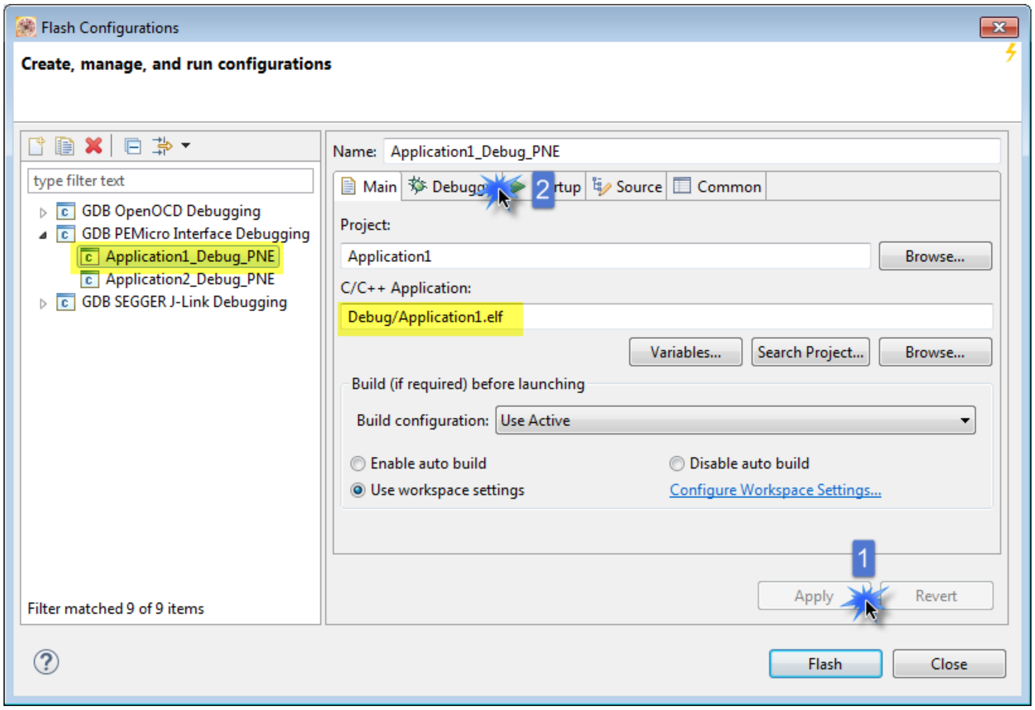
– Configure the PEMicro interface settings on the "Debugger" tab, click on Apply then on the 'Flash' button.



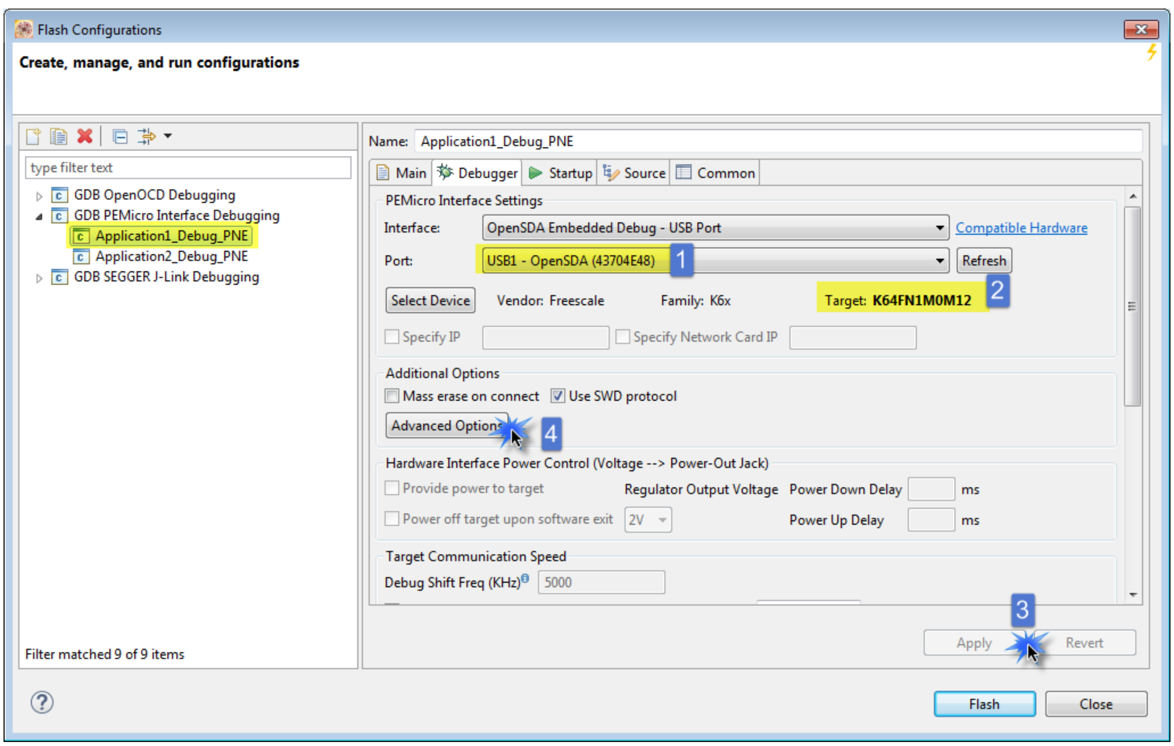
– The next step is to flash the **Application1** to the board making sure that the **Application2** is not erased, to do this we use the Advanced Flash Programming options from P&E.

– Click on the 'Flash from file…' icon to get the Flash Configurations menu, choose the "GDB PEMicro Interface Debugging" connection and select the .elf file generated by **Application1**.

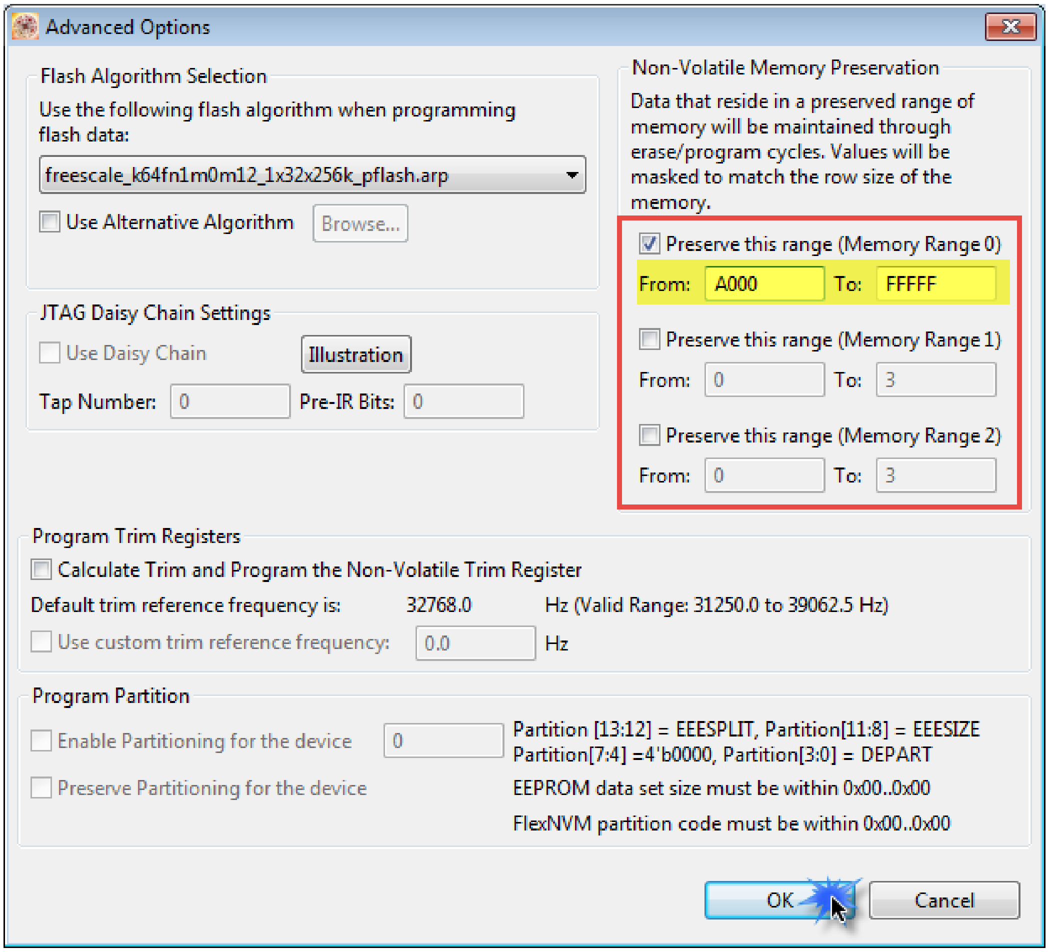




− Configure the PEMicro interface settings on the “Debugger” tab and open the “Advanced Options”.



– On the Advanced options window we will preserve the memory range where the **Application2** is located.

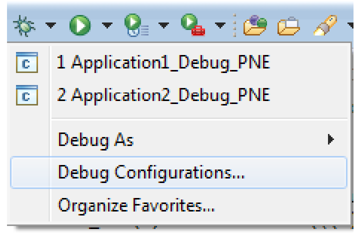


– Click on OK, then Apply and finally on Flash.

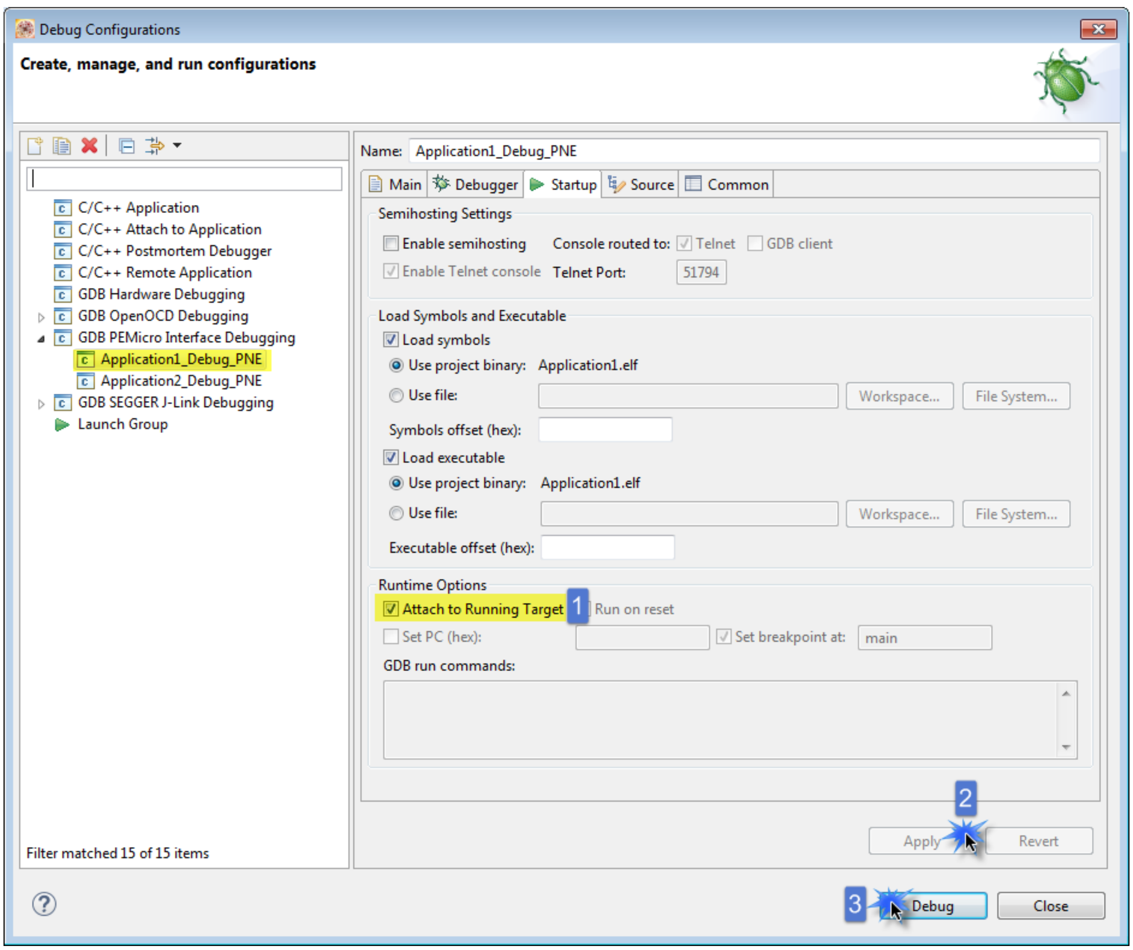
– Reset your board and you should see the blue LED toggle 5 times which indicates the **Application1** is being executed after this the red LED will start to toggle indicating that the program jumped to the **Application2**.

– Now that the MCU has both applications flashed we need to generate a single binary file that contains both of the applications so it can be used for factory programming, this can be done by performing a memory dump of the MCU.

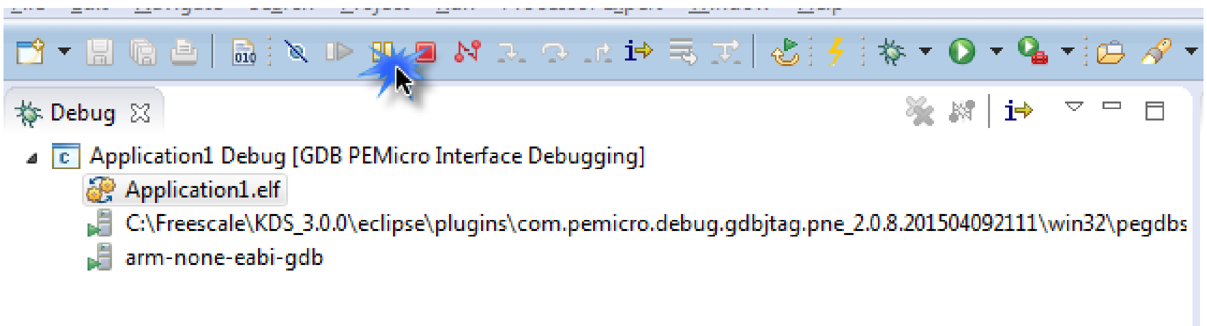
– Click on the arrow next to the Debug icon and select the "Debug Configurations…" option.



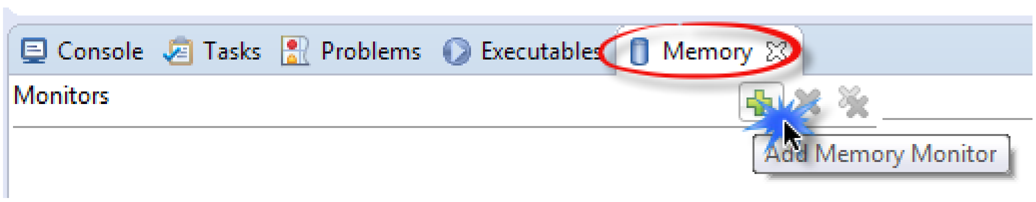
– On the Debug Configurations window choose the "GDB PEMicro Interface Debugging" connection, go to the "Startup" tab and enable the "Attach to Running Target" option, click on Apply then on Debug.

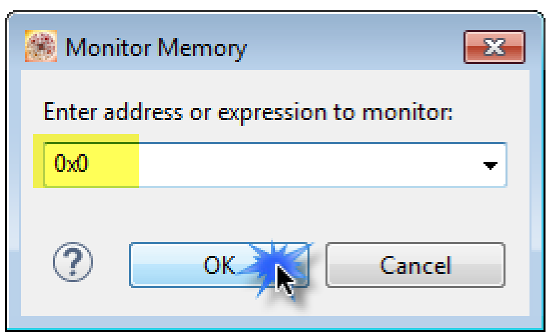


– The Debug perspective will open, suspend the execution of the program.

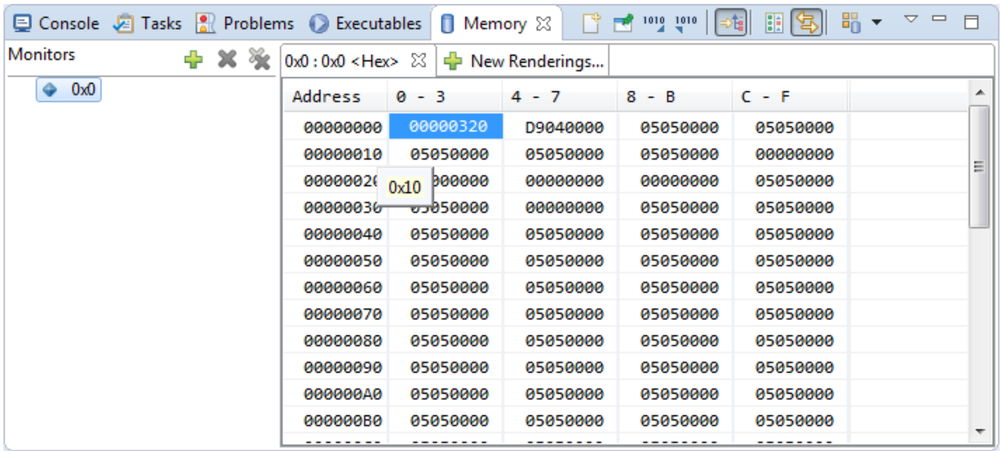


– Go to the Memory view (Window > Show View > Memory) and add a new address to monitor:

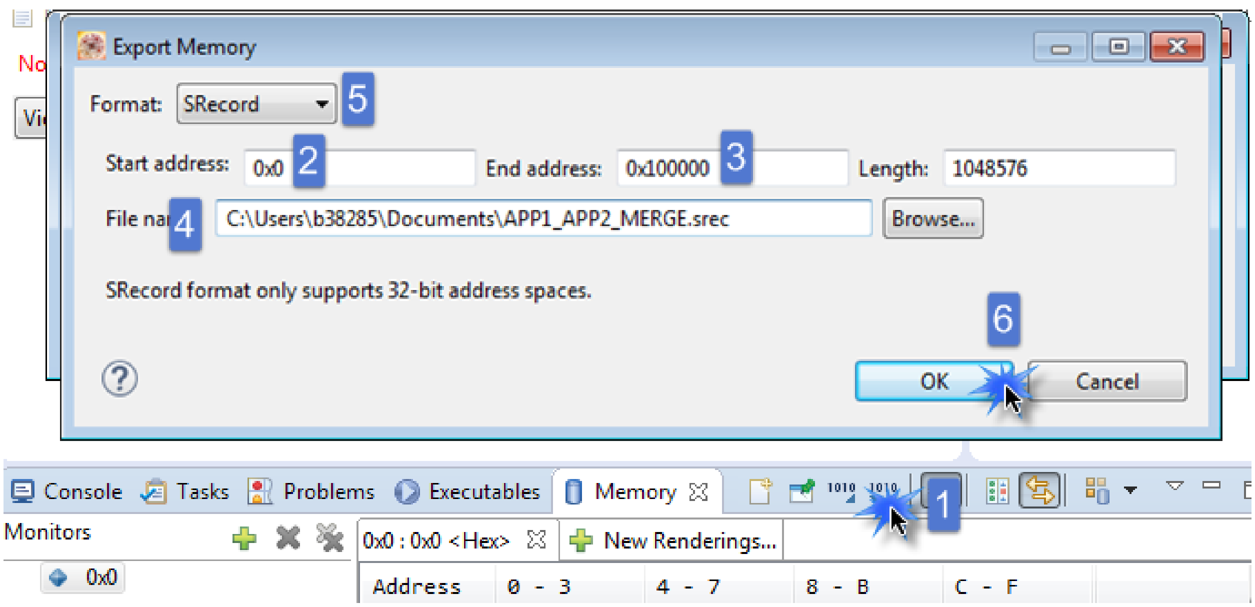




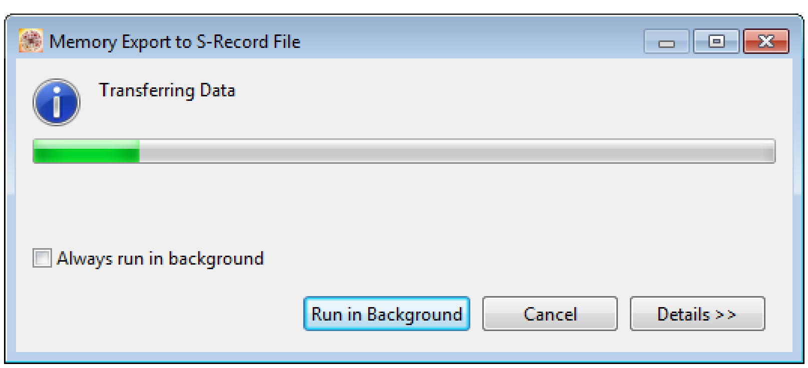
– The memory information will show up:



– Click on the "Export" button, set the start and end address of the memory, set a file name, select the format of the output file and click on OK:



– A window will show up indicating that the memory content is being exported:



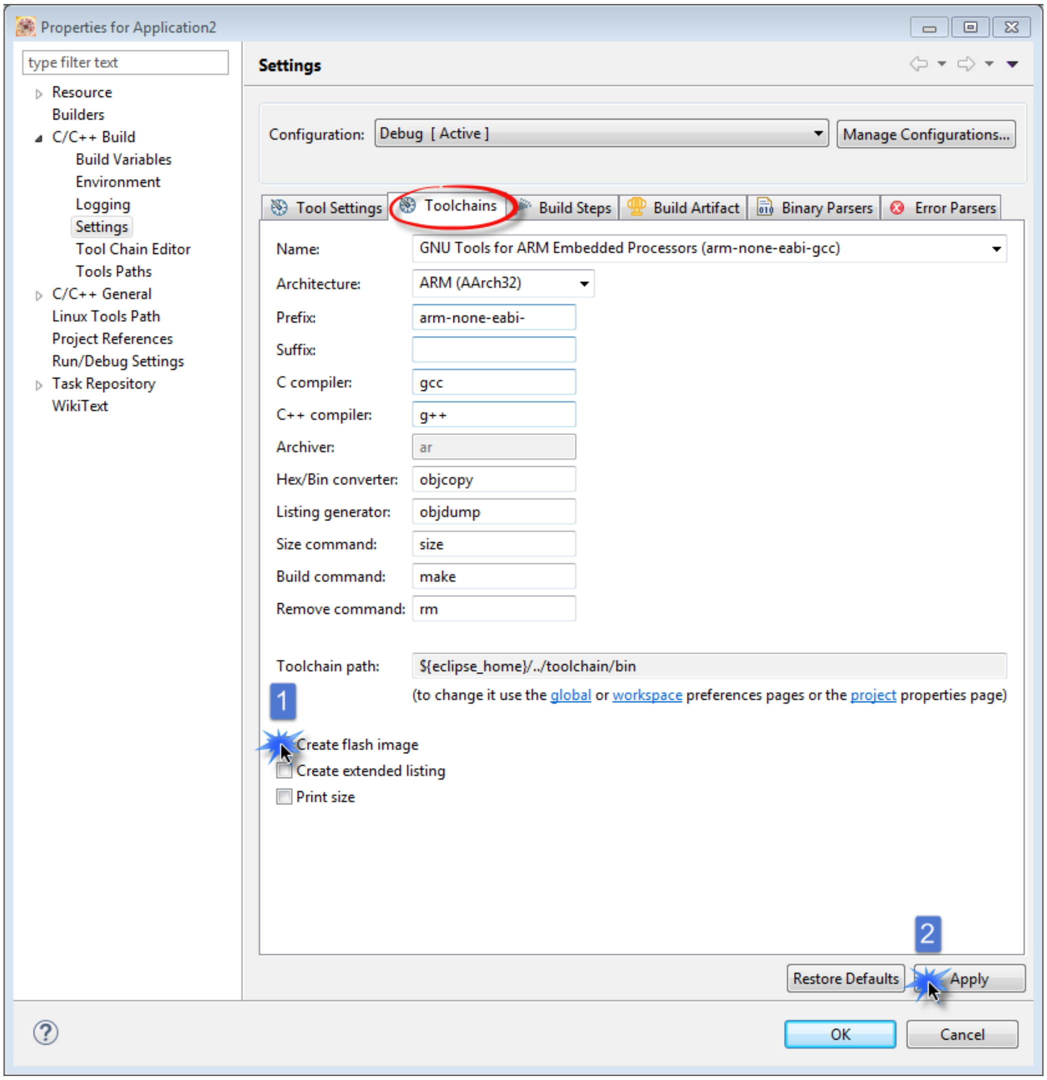
– Finally you get a single file with both applications that can be used for production purposes:

APP1_APP2_MERGE.srec

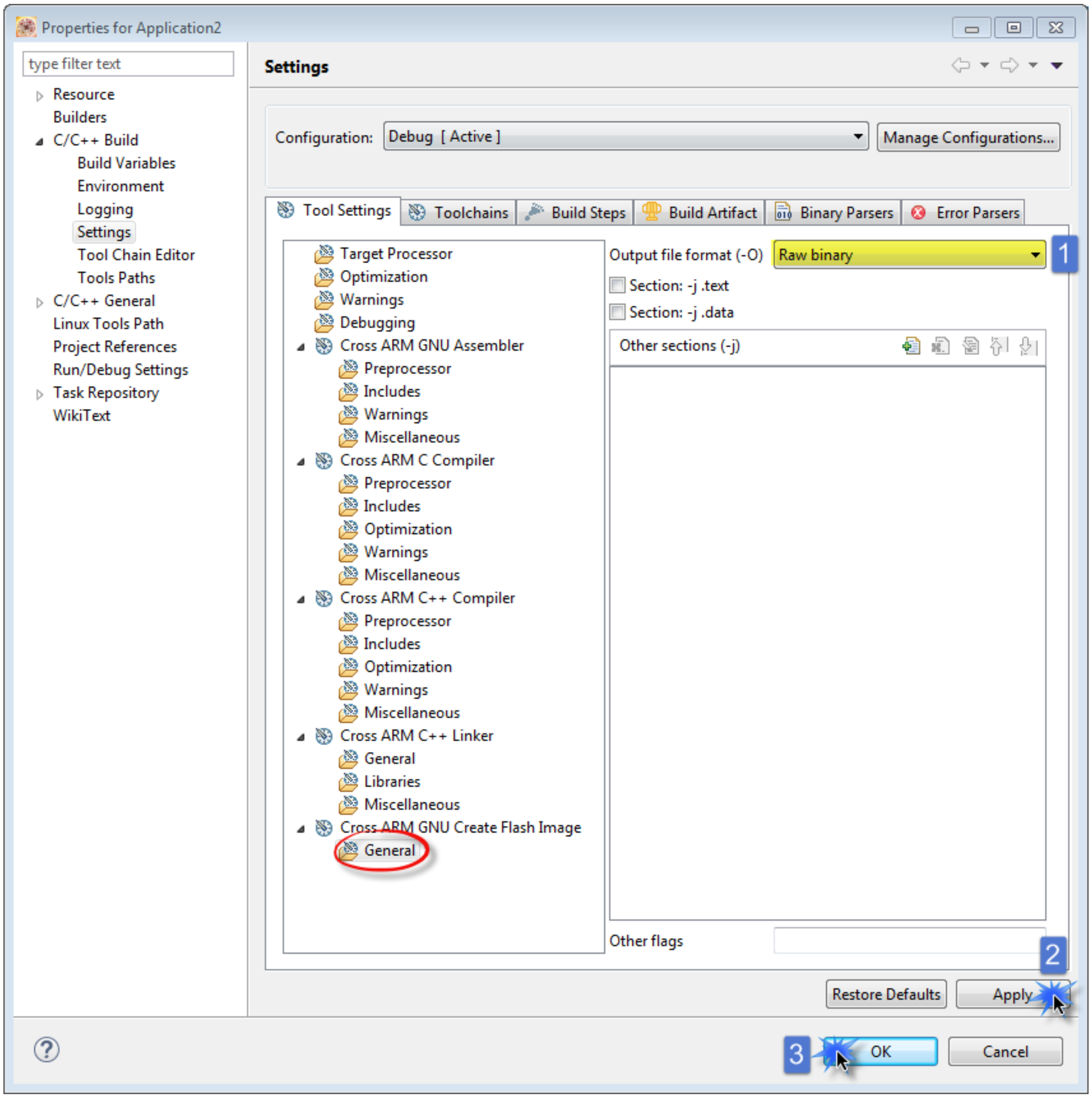
```
S31500000000000003 20D904000005050000050500 00D6
```

Appendix A – Binary file generation in KDS

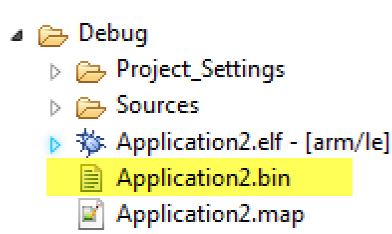
1) Open the project in KDS, go to Project -> Properties -> C/C++ Build -> Settings -> Toolchains. Enable the checkbox for "**Create flash image**" and click on Apply:



2) Go to Tool Settings -> Cross ARM GNU Create Flash Image -> General. In the "Output file format (-O) option select **Raw binary**. Click on Apply and then OK or close the Properties window.



3) Build the project. Once the build process is over, you should find the generated binary file (**.bin** extension) inside of the build folder called “Debug” by default:



Appendix B - References

- KDS webpage:
  www.freescale.com/kds

- Relocating Code and Data Using the KDS GCC Linker File (.ld) for Kinetis:
  https://community.freescale.com/docs/DOC-104433

- Kinetis Design Studio videos:

  - Installation of KDS and Kinetis SDK: https://community.freescale.com/videos/3281
  - Installation of OpenSDA Firmware: https://community.freescale.com/videos/3282
  - Debugging with KDS: https://community.freescale.com/videos/3283
  - Building the KSDK demo applications: https://community.freescale.com/videos/3378