AN12830

Running application binary in SRAM with bootloader on LPC5500 MCU

Rev. 0 — 04/2020 Application Note

by: NXP Semiconductors

Contents

1 Introduction

In traditional MCU usage, the application executable image is downloaded into the FLASH inside the chip, and the code is run directly in FLASH. Per the hardware limitation, the FLASH access is much slower than the CPU clock. The CACHE technology can improve the performance in some cases, but it also involves additional issues, such as the data synchronous issue. Developers would like to move some code to the SRAM for better performance, because the speed of SRAM access is close to the CPU clock and much faster than the FLASH. However, the code in SRAM is lost when the power goes off, while only the FLASH can keep the code when the power is off.

There are already ways to debug the code in the SRAM within the IDE, but these ways can only control the MCU when the power is always on. When a Power-On-Reset (POR) occurs, the application loaded by an external debugger is lost. The way described in this document makes an improvement and resolves the issue where the code in the SRAM cannot be kept after the POR. This document shows a way of using a customized bootloader to move the application image from the FLASH to the SRAM in the booting phase and running it from SRAM, so that it can keep the image in the FLASH but run in the SRAM per every POR.

2 Principle

The idea of using a customized bootloader is from the dual-core usage on the LPC5500 MCU. When running a dual-core application, the image for CPU1 (the slave core) should be built firstly. CPU0 (the master core) is involved in the project during the compile phase. In the CPU0's application, when CPU1 is to be activated, it moves the original RAW image data for CPU1 from the preserved FLASH area to the SRAMX, feeds the base address of the image into the "CPU1 boot address register", and finally enables the clock and starts CPU1.

Even in the single-core usage, CPU0 can also be switched to the new running routine by setting a new base address to the PC register in its own Arm® Cortex®-M33 core. The stack pointer registers (PSP and MSP) should be updated and the vector table remapped, even if the new application would run in a pure and clean environment.

In the demo for this document, the whole project is divided into two standalone parts:

- Application project. This project is open to the application developer. The coding for the application project (as the previous work) runs in the FLASH and it can be built standalone and used to create an executable binary file. The only difference is using a customized linker file to relocate the whole image to the SRAM address space, because it would be there when running. The application code in this period uses the so-called "run-time address".
- Bootloader project. This project now has the most important role, because it is the default application running after the chip is powered on. It copies the RAW data of the application project's binary image to the indicated SRAM area manually. It performs important things (it sets up the PC, PSP, MSP, and VTOR registers in the Arm core) to activate CPU0 with the new environment for the application image. The application image is downloaded into the chip. It can be downloaded in several ways. The way presented in the demo is the easiest way. The flashloader integrated in the IDE (IAR) is used for the bootloader project, because the application image can be downloaded into the chip together with the bootloader project. The application image should be downloaded into the chip. The application code uses the so-called "download-time address" in this phase.

The boot process from the MCU reset to the application code has two steps, as shown in Figure 1.

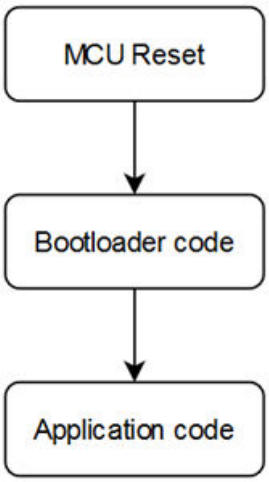

Figure 1. Two steps from MCU reset to application code

Another important thing is to arrange the memory area for each part. There are the following three parts:

- Application image in run-time
- Application image in download-time
- Bootloader image

The application image in the download-time and the bootloader code co-exist in the FLASH. The application image in the run-time and the bootloader RAM data co-exist in the SRAM. In the demo project, the memory areas are allocated according to Table 1.

Table 1. Memory allocation for demo project

Address	Usage	Comment
0x2000*0000* - *0x2002*FFFF, 192KB	application code in run-time	SRAM0/1/2
0x2003*0000* - *0x2003*FFFF, 64KB	application data in run-time	SRAM3
0x0000*0000* - *0x0000*FFFF, 64KB	bootloader code	FLASH
0x2003*0000* - *0x2003*FFFF, 64KB	bootloader data	SRAM3
0x0001*0000* - *0x0003*FFFF, 192KB	application binary in download-time	FLASH

In Table 1, the size of the reserved memory space for the application binary in download-time is the same as the application code in run-time, because the binary in download-time is copied to the SRAM for run-time. The data space for the application in run-time shares the same SRAM with the bootloader with no conflict. When the application is running, the bootloader is not running anymore.

3 Workflow

This section describes the creation of the whole demo project (including the application project and the bootloader project) step by step. In the demo project, it turns the LED on the LPCXpresso55s69 EVK board on and off. When the project runs normally, the red LED on the board is turned on and off.

3.1 Creating the application project to run in SRAM

The application project can be any user project with no special code changes to the bootloader. In this document, a project to turn the on-board red LED on and off is created in the MCUXpresso SDK IDE. The only and most critical change is to update the linker file according to Table 1.

The *LPC55S69cm33application_ram.icf* application project file contains the following code:

```
define symbol m_interrupts_start = 0x20000000 ; /* start from sram0. */
define symbol m_interrupts_end = 0x2000013F ;
define symbol m_text_start = 0x2000013F ;
define symbol m_text_end = 0x2002FFFF ; /* end within sram2. */
define symbol m_data_start = 0x20030000 ; /* start from sram3. */
define symbol m_data_end = 0x2003FFFF ; /* end within sram3. */
```

Make sure that the executable **.bin* file is created after the building, because it can be executed in the bootloader project directly after copying, without any extraction. Figure 2 shows this setting in the IAR IDE.

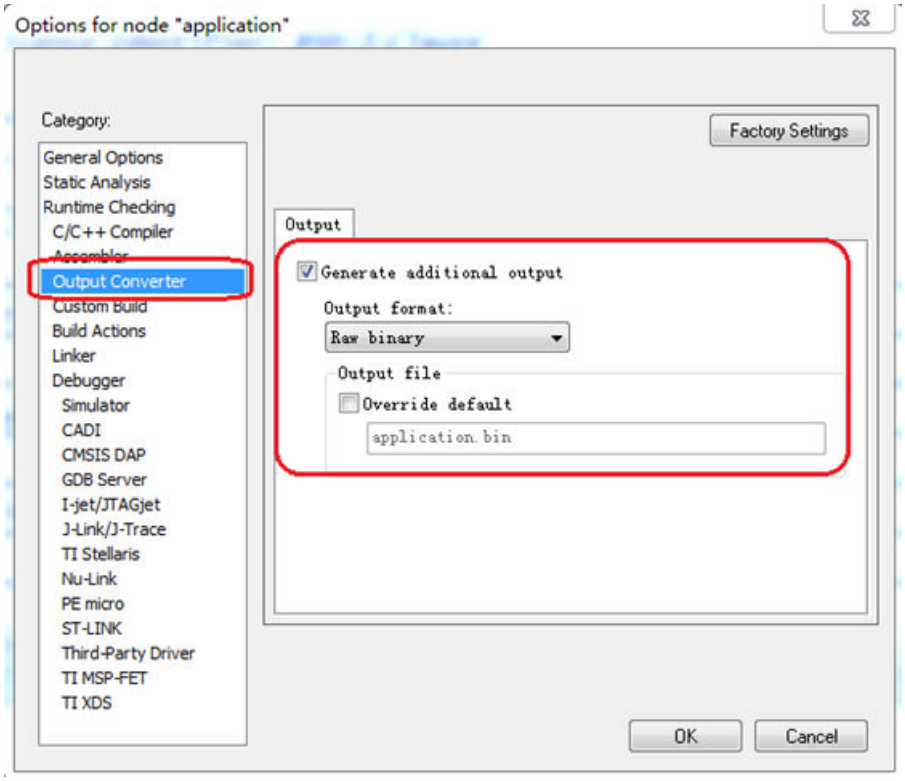

Figure 2. Creating binary file in application project

3.2 Creating the bootloader project to copy the application binary

The bootloader project is specially created to copy the application binary after the POR. It can simply do the copying with no operation to the peripheral (without even changing the clock system) using the default clock after the POR. However, it integrates the application project binary so that it can be downloaded to the FLASH inside the MCU through the IDE's flashloader. The application project binary can be downloaded in another way, for example; the ISP or *blhost.exe* (NXP bootloader client running on the PC to communicate with the on-chip ROM). However, using the IDE's flashloader is still the simplest way for the developers, because they download the project during the debugging phase.

3.2.1 Linker file

First of all, update the linker file according to Table 1 for the bootloader project. The following code is in the *LPC55S69cm33bootloader_flash.icf* file:

```
define symbol m_interrupts_start = 0x00000000 ;
define symbol m_interrupts_end = 0x0000013F ;
define symbol m_text_start = 0x00000140 ;
define symbol m_text_end = 0x0000FFFF ; /* end with first 64KB. */
define exported symbol application_image_start = 0x00010000 ; /* for application image. 192KB. */
define exported symbol application_image_end = 0x0003FFFF ;
define symbol m_data_start = 0x20030000 ;
define symbol m_data_end = 0x2003FFFF ; /* sram3. */
/* for user application binary. */
define region APPLICATION_region = mem : [ from application_image_start to application_image_end] ;
define block SEC_APPLICATION_IMAGE_BLOCK { section __sec_application };
place in APPLICATION_region { block SEC_APPLICATION_IMAGE_BLOCK };
```

The first part defines the memory areas for the bootloader project and the application project binary in download-time. The last part defines a memory block for the binary in the bootloader project, so that it can be included into the bootloader project binary (all in one) during the building. The section name "*_sec*application" quoted here is defined in the project option dialog.

Note that the application project's binary should still be visible to bootloader project, because the bootloader project must recognize it and move it from the FLASH to the SRAM.

3.2.2 Project option

The pre-build application binary must be included into the bootloader project in the project options dialog, as shown in Figure 3.

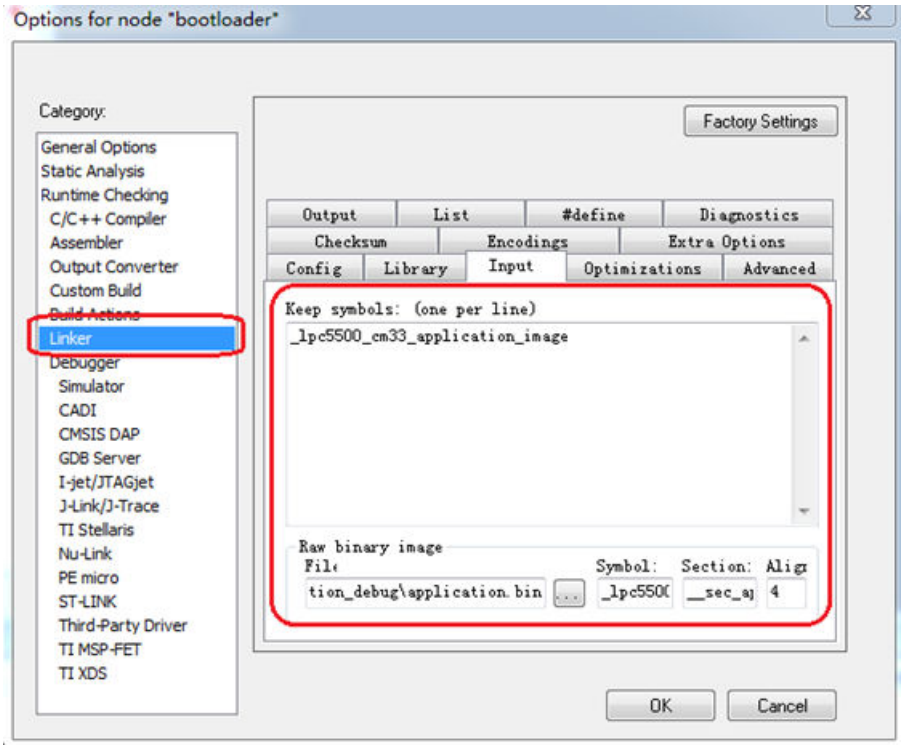

Figure 3. Linking the pre-build binary file into bootloader project

Put the following into the "Raw binary image" field:

- File: "*$PROJDIR$\applicationdebug\application.bin*"

- Symbol: "_lpc5500*cm33*application_image"
- Section: "_*sec*application"
- Alignment: "4"

The "Symbol" is used in the current project source code. The "Section" is defined here and it is quoted in the current project linker file.

3.2.3 Source code

In the SystemInit() function, which is called before the main() function in the boot routine, it enables all the RAM banks to make sure the SRAM0/1/2/3 is available for later work.

```
void SystemInit( void )
{
 /* enable RAM banks that may be off by default at reset */
 SYSCON->AHBCLKCTRLSET[ 0 ] = SYSCON_AHBCLKCTRL0_SRAM_CTRL1_MASK
 | SYSCON_AHBCLKCTRL0_SRAM_CTRL2_MASK
 | SYSCON_AHBCLKCTRL0_SRAM_CTRL3_MASK
 | SYSCON_AHBCLKCTRL0_SRAM_CTRL4_MASK;
}
```

In the main() function, which is the entry of the user application in a project, it copies the binary from the FLASH memory to the SRAM memory and jumps to run the new binary.

```
#define APPLICATION_BOOT_ADDRESS (void *)0x20000000
extern uint8_t application_image_start[]; /* defined in linker file. */
int main( void )
{
 /* copy application image to sram0/1/2. */
#pragma section = "__sec_application" /* this section is defined in project option dialog. */
 uint32_t application_image_size = ( uint32_t )__section_end( "__sec_application" ) -
( uint32_t )&application_image_start;
 /* copy the code for application from FLASH to the target memory. */
 memcpy(APPLICATION_BOOT_ADDRESS, ( void *)application_image_start, application_image_size);
 /* jump to the application image. */
 JumpToImage(APPLICATION_BOOT_ADDRESS);
}
```

Note that "APPLICATION*BOOT*ADDRESS" is defined as "0x2000_0000" to align with the linker file setting in the application project.

The "JumpToImage()" function has the most important role. It resets the running environment for the application code (vector table, stacks, and the PC pointer). When the PC pointer is changed, the CPU runs to the new routine.

```
typedef void (*func_0_t)( void ); /* define the type of function with no parameter. */
void JumpToImage( void * addr)
{
 uint32_t * vectorTable = ( uint32_t *)addr;
 uint32_t sp_base = vectorTable[ 0 ]; /* initial stack pointer. */
 func_0_t pc_func = (func_0_t)(vectorTable[ 1 ]); /* reset handler. */
 /* set new msp and psp. */
 __set_MSP(sp_base);
 __set_PSP(sp_base);
 /* remap the vector table. */
 SCB->VTOR = addr;
 /* jump to application. */
 pc_func();
 /* the code should never reach here. */
```

```
  while ( 1 )
  {}
}
```

3.3 Download and debug the application binary

To run the demo:

- Build the application project, and generate the *application.bin* binary file.
- Build the bootloader project to involve the pre-build *application.bin* file, and generate the all-in-one *bootloader.bin* binary file. The format of the generated executable file is not important, because it is downloaded using the tools integrated in the IDE.
- Download the final all-in-one binary file in the bootloader project using the download tool in the IDE.
- Terminate the debug window, reset the board by pressing the on-board reset button, and run the demo.
- The red LED light turns on and off and it is controlled by the application project. This means that the demo runs as expected.

For the first download, follow the above steps one by one to make sure that the bootloader and the application part are both available and running on the chip. When the bootloader part is already downloaded, the developer can load the RAM binary directly in the application project later for debugging purposes. Because the stack pointer registers (MSP, PSP) and the vector table register (VTOR) are already set up by the bootloader when loading the run-time application binary directly in the application project, the debugger writes the binary directly to the SRAM memory and starts the application in the debug window. However, if you want to run the final application project after the POR with no IDE and debugger help, follow the above steps again to keep the application binary stored in the FLASH memory, which is not erased when the power is off.

4 Benchmark

The EEMBC coremark test shows the performance improvement when the code runs in the SRAM. In the test project, the coremark thread is used as the application part, while the bootloader part is still the same as in the simple demo project. The LPC55S69 MCU core clock runs at 150 MHz with the IAR IDE compiler for Arm. The iteration number is set to 800, so that you don't wait too long (more than 10 seconds) for each test case.

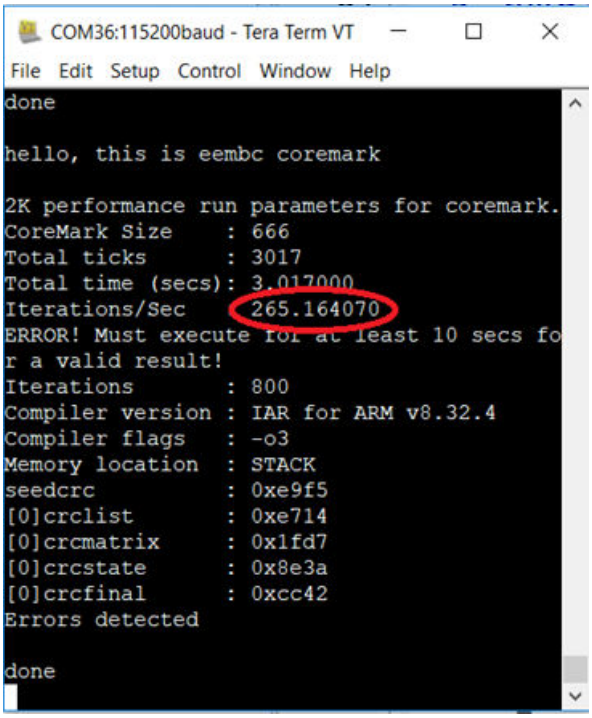

Figure 4. EEMBC coremark records for FLASH code

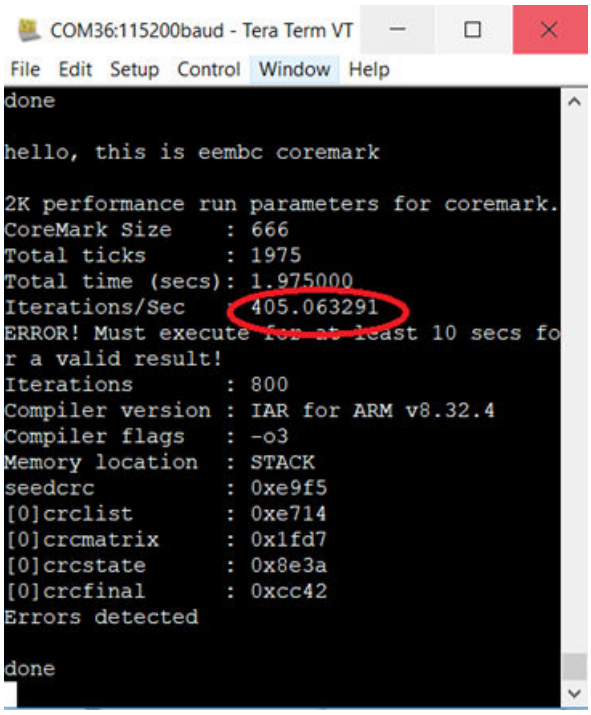

Figure 5. EEMBC coremark records for RAM code

All the records in different optimization levels are summarized in Table 2.

Table 2. Coremark records on various compiler optimizations

Optimization	FLASH code	RAM code
-o0 (none)	112.39	191.25
-o1 (low)	131.39	213.62
-o2 (medium)	182.23	280.11
-o3 (high speed)	265.16	405.06

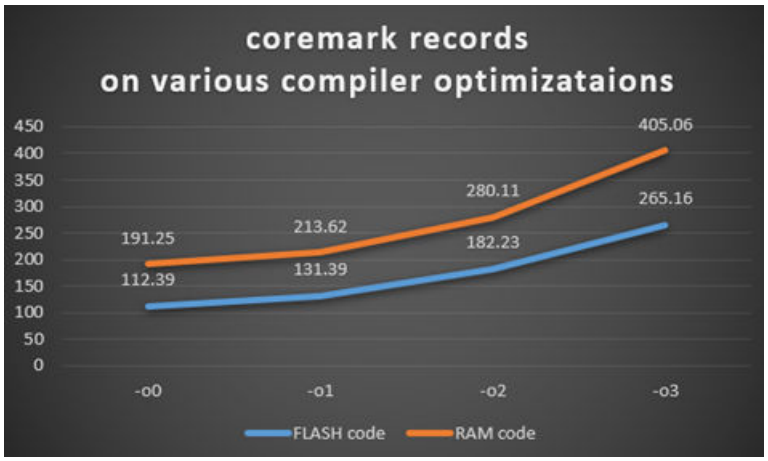

Figure 6. Coremark records on various compiler optimizations

The benchmark test shows that the image running in the SRAM runs much faster than when it is stored in the FLASH.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

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

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

Date of release: 04/2020

Document identifier: AN12830

arm