AN13113

A Kind of SD card based Second Bootloader on LPC54600 MCU

Rev. 0 — January 20, 2021 Application Note

by: NXP Semiconductors

Contents

1 Introduction

Sometimes, user would like to keep the MCU's application software (firmware) on the external memory media, so that various kinds of way of downloading the firmware can be supported. This feature is also useful for the engineers to demonstrate various examples project with only one hardware board. When the program cannot be downloaded to the chip directly in some reason (for example, the additional license requirements or the additional equipment requirements), switching the memory media with different firmware is more acceptable.

This application note describes a design of 2nd bootloader, using SD card as the external memory media to keep the firmware image files to be executed. The example projects, including the bootloader project and firmware project, are developed originally on the LPCXpresso54608 EVK board. The SD card is chosen as the external memory, as it sets up a file system where the PC can directly read and write the files.

After the POR, the bootloader software in the chip's internal flash runs first. It reads the SD card to detect the available firmware image files and tells the available options to users through the terminal, based on the UART. Users make the selection, and then the bootloader software loads the selected image files from SD card to the chip's internal memory, SRAM, and execute.

2 Hardware

2.1 LPC54608 MCU

The LPC54600 is a family of Arm® Cortex®-M4 based microcontrollers for embedded applications featuring a rich peripheral set with very low power consumption and enhanced debug features.

The LPC54600 family includes up to 512 KB of flash, 200 KB of on-chip SRAM, up to 16 KB of EEPROM memory, a quad SPI Flash Interface (SPIFI) for expanding program memory, one high-speed and one full-speed USB host and device controller, Ethernet AVB, LCD controller, Smart Card Interfaces, SD/MMC, CAN FD, an External Memory Controller (EMC), a DMIC subsystem with PDM microphone interface and I^2S, five general-purpose timers, SCTimer/PWM, RTC/alarm timer, Multi-Rate Timer (MRT), a Windowed Watchdog Timer (WWDT), ten flexible serial communication peripherals (USART, SPI, I^2S, I^2C interface), Secure Hash Algorithm (SHA), 12-bit 5.0 M samples/sec ADC, and a temperature sensor.

Type number	Package Name	Frequency/MHz	Flash/kB	SRAM/kB	FS USB	HS USB	Ethernet AVB	Classic CAN	CAN FD	LCD	Flexcomm Interface	EMC data bus width (bit)	GPIO	SHA
LPC54628 devices (HS/FS USB, Ethernet, CAN FD, CAN 2.0, LCD, SHA)														
LPC54628J512ET180	TFBGA180	220	512	200	yes	yes	yes	yes	yes	yes	10	8/16	145	yes
LPC54618 devices (HS/FS USB, Ethernet, CAN FD, CAN 2.0, LCD)														
LPC54618J512ET180	TFBGA180	180	512	200	yes	yes	yes	yes	yes	yes	10	8/16	145	no
LPC54618J512BD208	LQFP208	180	512	200	yes	yes	yes	yes	yes	yes	10	8/16/32	171	no
LPC54616 devices (HS/FS USB, Ethernet, CAN FD, CAN 2.0)														
LPC54616J256ET180	TFBGA180	180	256	136	yes	yes	yes	yes	yes	no	10	8/16	145	no
LPC54616J512BD208	LQFP208	180	512	200	yes	yes	yes	yes	yes	no	10	8/16/32	171	no
LPC54616J512ET100	TFBGA100	180	512	200	yes	yes	yes	yes	yes	no	9	8/16	64	no
LPC54616J512BD100	LQFP100	180	512	200	yes	yes	yes	yes	yes	no	9	8/16	64	no
LPC54608 devices (HS/FS USB, Ethernet, CAN 2.0, LCD)														
LPC54608J512ET180	TFBGA180	180	512	200	yes	yes	yes	yes	no	yes	10	8/16	145	no
LPC54608J512BD208	LQFP208	180	512	200	yes	yes	yes	yes	no	yes	10	8/16/32	171	no
LPC54607 devices (HS/FS USB, LCD)														

Figure 1. LPC54608 MCU in the LPC54600 family

In this application example, the LPC54608J512ET180 part is used.

Figure 2 shows the location of the internal memory.

Address range	General Use	Address range details and description	
0x0000 0000 to 0x1FFF FFFF	On-chip non-volatile memory	0x0000 0000 - 0x0007 FFFF	Flash memory (512 kB).
	Boot ROM	0x0300 0000 - 0x0300 FFFF	Boot ROM with flash services in a 64 kB space.
	SRAMX	0x0400 0000 - 0x0400 7FFF	I&D SRAM bank (32 kB).
	SPI Flash Interface (SPIFI)	0x1000 0000 - 0x17FF FFFF	SPIFI memory mapped access space (128 MB).
0x2000 0000 to 0x3FFF FFFF	SRAM Banks	0x2000 0000 - 0x2002 7FFF	SRAM banks (160 kB).
	SRAM bit band alias addressing	0x2200 0000 - 0x23FF FFFF	SRAM bit band alias addressing (32 MB)

Figure 2. LPC54608 internal memory for FLASH and SRAM (with SRAMX)

2.2 LPCXpresso54608 EVK board

The example projects, including the bootloader project and application projects, are developed originally on the LPCXpresso54608 EVK board. Actually, for the bootloader feature, the minimal requirement of hardware board is just with an SD card socket, as shown in Figure 3.

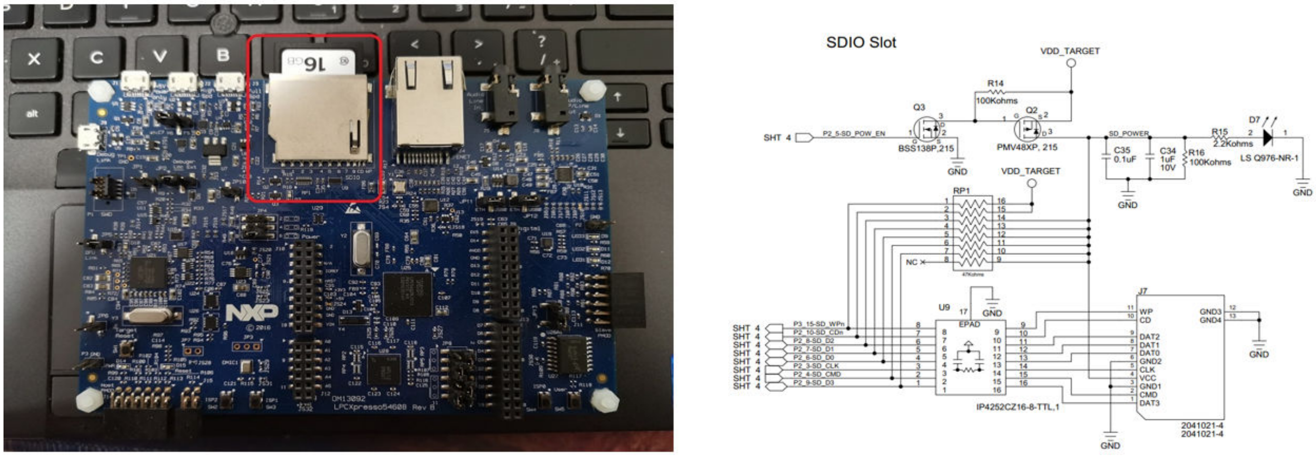

Figure 3. LPCXpresso54608 EVK board with SD card

To simplify the software design and focus on the implementation of bootloader feature, this application example does not enable the USB to SD card function with the integrated USB port on LPC54608 chip. An external SD card reader device can be used if your PC does not have the SD card socket, as shown in Figure 4. For future development, it will be a good trial to enable the on-chip USB feature. Then the user can drag the image files into SD card through the bootloader software directly, without an additional SD card reader.

Figure 4. Standalone SD card reader

3 Software

The whole example project includes two parts of projects:

- The bootloader project operates the SD card device with file system, loading image file from SD card to SRAM. Then the bootloader executes the image of firmware. During my development, I used the **sdcard_fatfs** example project from MCUXpresso SDK as the start of coding to create a **bootloader_sdcard_fatfs** project.
- The firmware runs the user application work. To create the image, build the firmware project. The firmware project is not an award of the existence of bootloader, It can be downloaded and debugged independently during the development. However, since the image file of firmware can be loaded to the SRAM, be careful that the code size of the image file is not so big to fill the SRAM. During my development, I used the **hello_world** example project from MCUXpresso SDK as the start of coding to create an **app_hello** project and created another **app_blinky** project based on it.

3.1 Edit the linker files to re-allocate the memory space

Just like other bootloader designs, the memory space allocation for each part of images is most important. Since the bootloader and the firmware are co-existing in a time inside the chip, the designer needs to separate them and make sure each part still has enough memory resource to run.

Generally, the bootloader's code is placed in the on-chip FLASH, and the data is placed in a piece of standalone internal SRAM (SRAMX, 32 KB, starting from 0x0400_0000). The firmware's code and data are all placed in the continuous memory space starting from 0x20000000, as shown in Table 1.

Table 1. Allocation of the memory space for bootloader and firmware project

Function	Address	Size
Bootloader code	0x0000_0000 - 0x0007FFFF	512 KB
Bootloader data	0x0400_0000 - 0x04007FFF	32 KB
Firmware code	0x2000_0000 - 0x2001FFFF	128 KB
Firmware data	0x2002_0000 - 0x20027FFF	32 KB

Actually, these settings take effect in the linker files for each project. In the current example, there are two linker files:

1. Bootloader project's linker file

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

define symbol m_text_start             = 0x00000400;
define symbol m_text_end               = 0x0007FFFF;

define symbol m_data_start             = 0x04000000;
define symbol m_data_end               = 0x04007FFF;

...
```

2. Firmware project's linker file

```
define symbol m_interrupts_start       = 0x20000000;
define symbol m_interrupts_end         = 0x200003FF;

define symbol m_text_start             = 0x20000400;
define symbol m_text_end               = 0x2001FFFF;

define symbol m_data_start             = 0x20020000;
define symbol m_data_end               = 0x20027FFF;

...
```

In both linker files, I enlarged the size of stack and heap for the default settings to provide enough resource for the following usage of cJSON. In the example projects, the stack size and the heap size are all with 4 KB.

```
/* Sizes */
if (isdefinedsymbol(__stack_size__)) {
  define symbol __size_cstack__        = __stack_size__;
} else {
  define symbol __size_cstack__        = 0x1000;
}

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

```
  define symbol __size_heap__          = 0x1000;
}
```

Actually, since the memory space is totally independent, the firmware project can be downloaded and debugged (into SRAM) directly without the regarding of bootloader. It can run normally under the control of the debugger.

3.2 Jump to execute the new image

To jump to execute the new image, manually fill the SPs (MSP and PSP) and the PC registers with the address for the new image. In this example, I used the traditional piece of code as following:

```
/* execute the firmware exists in sramx. */
typedef void(*func_0_t)(void);
void app_execute_ram_firmware(void * addr)
{
    uint32_t * vector_table = (uint32_t *)addr;
    uint32_t sp_base = vector_table[0];
    func_0_t pc_func = (func_0_t)(vector_table[1]);

    /* set new msp and psp. */
    __set_MSP(sp_base);
    __set_PSP(sp_base);
#if __VTOR_PRESENT == 1
    SCB->VTOR = addr;
#endif
    /* jump to application. */
    pc_func();
    /* the code should never reach here. */
    while (1)
    {}
}
```

3.3 Use the cJSON midware to read configuration file on SD card

cJSON is an open source middleware component of parsing the JSON text existing in GitHub - DaveGamble/cJSON: Ultralightweight JSON parser in ANSI C. It is coded with pure C to be integrated into any embedded system without any porting work related to the platform.

In this example, cJSON was used to parse the configuration file written in JSON format, which is a popular way used in Python programming. The *conf.json* file was placed in the SD card with the image files. It recorded the information about all the available image files. Then the bootloader read to find the real image file according to the configuration file. The contexnt of the *conf.json* file was as below:

```
{
        "default":
        {
              "idx":1,
              "num":2
        },
        "filepaths":
        [
             {
                   "idx":0,
                   "path":"/hello.bin"
              },
             {
                   "idx":1,
                   "path":"/blinky.bin"
```

```
                    }
            ]
}
```

It means there are two image files: *hello.bin* and *blinky.bin*, with their own index number. More image files can be added into the list if needed.

To integrate the cJSON into the bootloader project, add the *cJSON.h* and *cJSON.c* files, as shown in Figure 5.

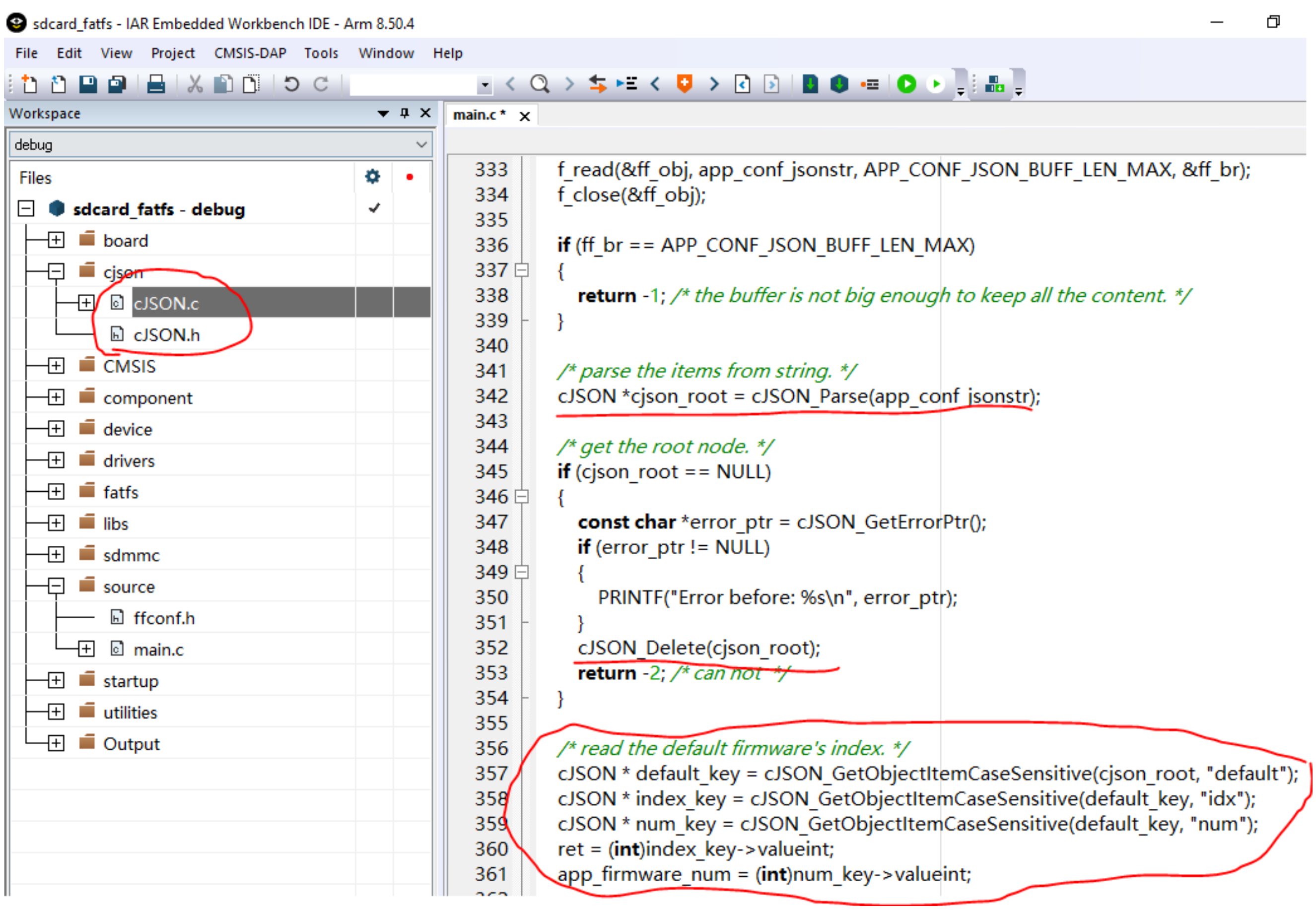

Figure 5. Use the cJSON in bootloader project

For the detailed usage about cJSON, see the *readme.md* file in the repository.

For the detailed coding work about the whole application demo project, see AN13113SW.

4 Run demo project

After the POR, the bootloader software in the chip's internal flash runs first. It reads the SD card to detect the available firmware image files and tells available options to users through the terminal, based on the UART. Users make the selection. Then the bootloader software loads the selected image files from SD card to internal memory of the chip, SRAM, and execute.

To run the demo project, please follow the steps:

1. Build the `app_hello` and `app_blinky` projects to create the *hello.bin* and *blinky.bin* files. Actually, there are also the pre-created ones in the attachment.
2. Plug in the SD card to PC, copy the *hello.bin*, *blinky.bin* and the *conf.json* files into the root directory of file system on SD card, and then unplug the SD card from PC. Now, the SD card is ready.

3. Build the `bootloader_sdcard_fatfs` project and download the image into LPC54608 EVK board.
4. Plug in the SD card to the LPC54608 EVK board and reset the board. Now the demo starts.
5. Watch the terminal and make the interaction with the board. The log is as seen in Figure 6.

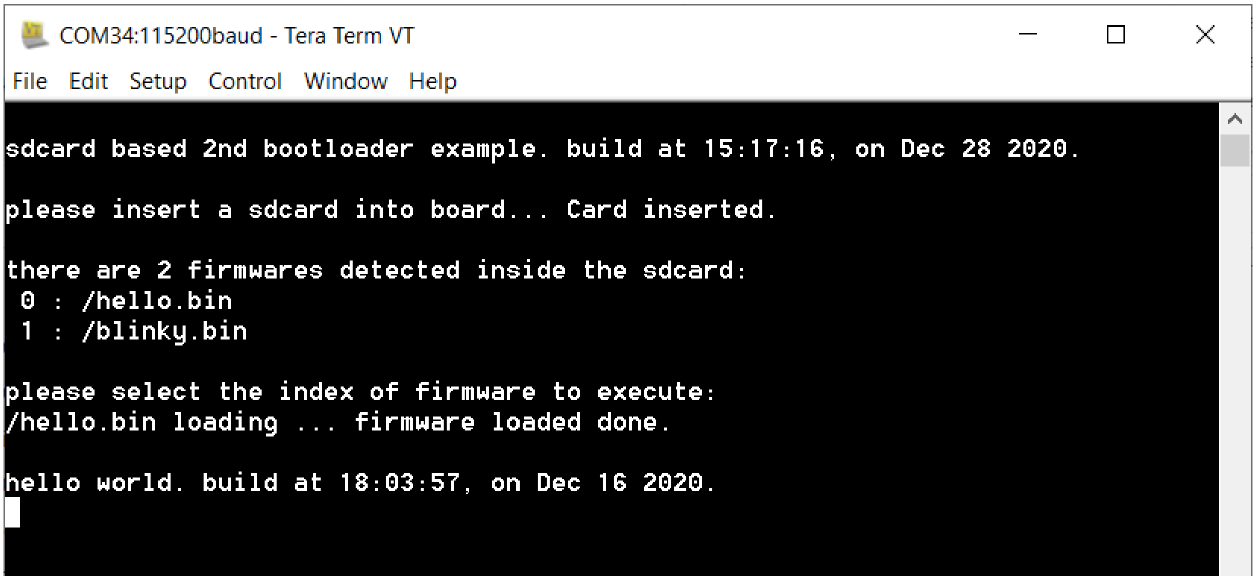

Figure 6. Terminal interaction with demo board

Date of release: January 20, 2021

Document identifier: AN13113