How to write flash resident bootloader for MPC55xx/56xx

by: Lukas Zadrapa
Technical Information & Commercial Support
Roznov pod Radhostem
Czech Republic

Table Of Contents

1 Introduction

All MPC55xx/56xx devices feature Boot Assist Module (BAM) that supports internal flash boot mode and serial boot mode.

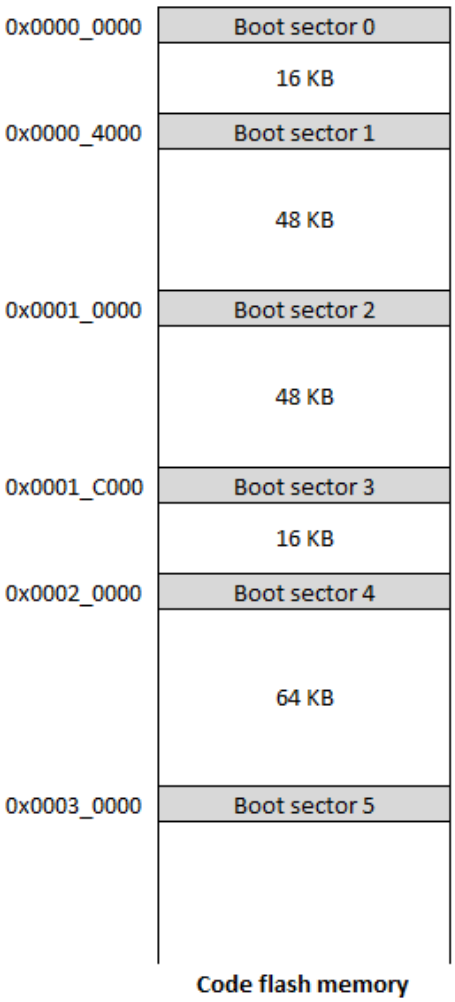
- Internal flash boot mode requires the user to set RCHW (Reset configuration Half Word) and reset vector to one of several possible locations in flash memory. The BAM searches for valid RCHW after reset. Once valid RCHW is found, code at address of reset vector is starting executed. If RCHW is not found, the device is put into static mode.
- Serial boot mode provides a mechanism to download and then execute code into the microcontroller SRAM. Code may be downloaded using either FlexCAN or LINFlex/eSCI.

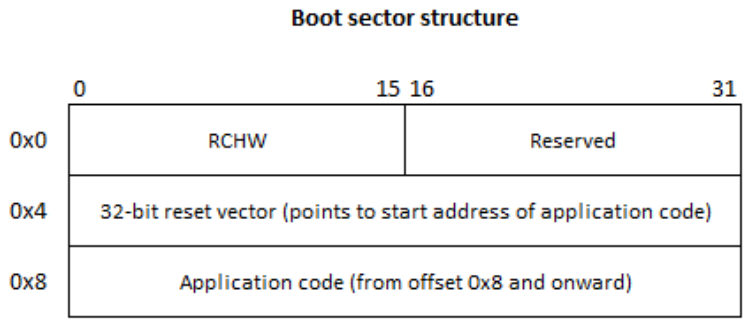
Boot mode is selected by hardware setting – by FAB and ABS pins (or BOOTCFG pin on some devices). Change of this hardware setting could not be always possible in the application. If we want to upgrade firmware using only software intervention, we have to use flash resisdent bootloader which is covered by this document. This kind of bootloader uses internal flash boot mode and the bootloader is executed after each reset. Bootloader makes a decision if user application is supposed to be executed or if the bootloader should start downloading new firmware. The decision can be made upon any conditon selected by user: state of word in Flash/EEPROM memory, state of any pins, on the basis of data received via any communication interface…

This application note will use microcontroller MPC5634M and CodeWarrior Development Studio for MPC55xx/MPC56xx (Classic IDE) version 2.10 to demonstrate all necessary steps. The principle is the same for all other devices and IDEs.

2 Creating of projects

It is recommended to use two independent projects for bootloader and user application in order to avoid possible cross references (typically when using libraries). The projects must be configured to use only selected flash address range without overlapping. Both projects will take an advantage of several possible locations of RCHW. Both projects will have own RCHW and reset vector placed at different location, so projects can be developed, debugged and executed independetly without any modifications. Once the bootloader is ready, it can be linked to user application in a binary form, so only one image is downloaded to flash in production.



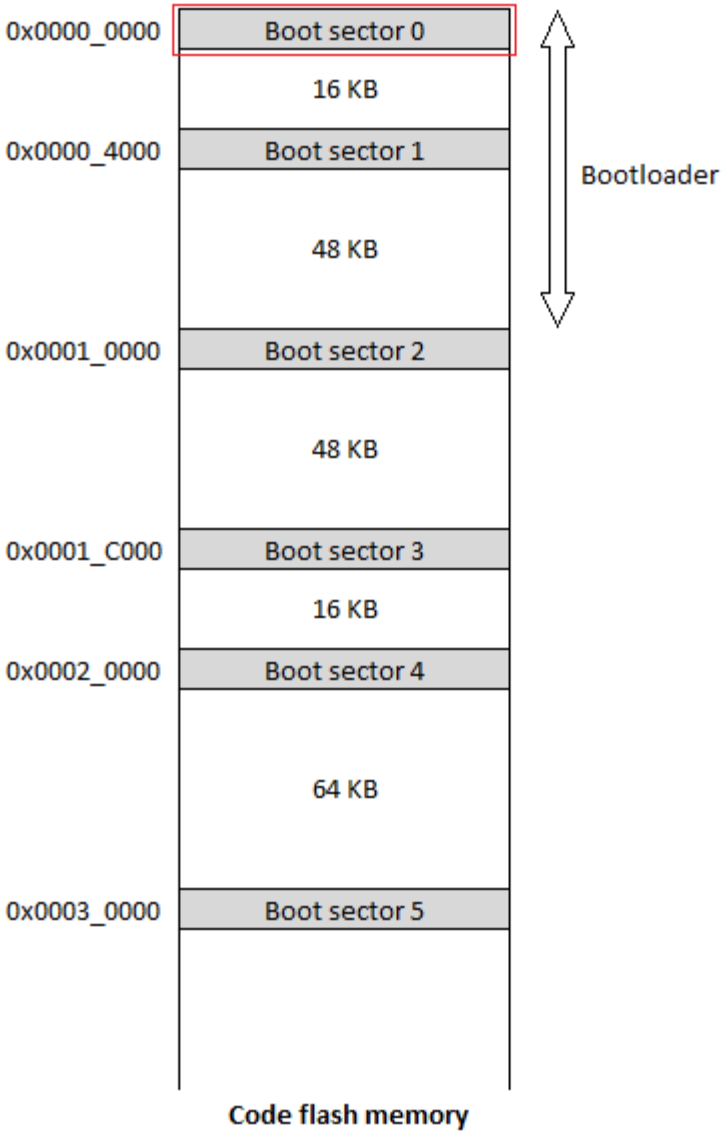


2.1 Project for bootloader

Create a project for bootloader and open its linker file. These are default memory segments on MPC5634M:

```
MEMORY
{
    resetvector:          org = 0x00000000,   len = 0x00000008
    init:                 org = 0x00000020,   len = 0x00000FE0
    exception_handlers:   org = 0x00001000,   len = 0x00001000
    internal_flash:       org = 0x00002000,   len = 0x0017E000
    internal_ram:         org = 0x40000000,   len = 0x00014000
    heap  :               org = 0x40014000,   len = 0x00002000
    stack :               org = 0x40016000,   len = 0x00001800
}
```

Now it is necessary to allocate flash memory that will be used only for bootloader. Let's say that we will use first 64kB: 0x0000_0000 – 0x0000_FFFF. The bootloader will use boot sector 0:



Change the memory segments as follows (only internal_flash segment in this case):

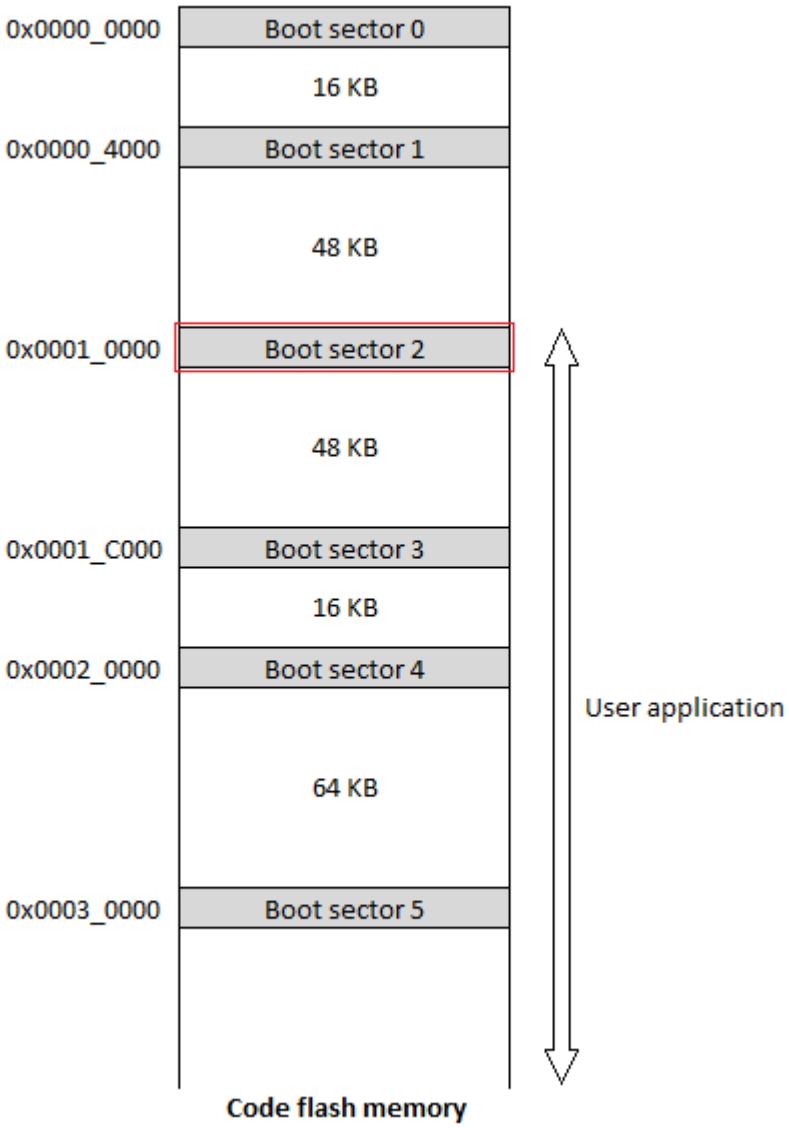
```
MEMORY
{
    resetvector:            org = 0x00000000,    len = 0x00000008
    init:                   org = 0x00000020,    len = 0x00000FE0
```

```
    exception_handlers:    org = 0x00001000,   len = 0x00001000
    internal_flash:        org = 0x00002000,   len = 0x0000E000   //use 64KB only
    internal_ram:          org = 0x40000000,   len = 0x00014000
    heap  :                org = 0x40014000,   len = 0x00002000
    stack :                org = 0x40016000,   len = 0x00001800
}
```

The rest of flash memory (0x0001_0000 – 0x0007_FFFF) will be used for user application.

2.2 Project for user application

Create a project for user application and open its linker file. User application will use flash memory 0x0001_0000 – 0x0017_FFFF and boot sector 2:

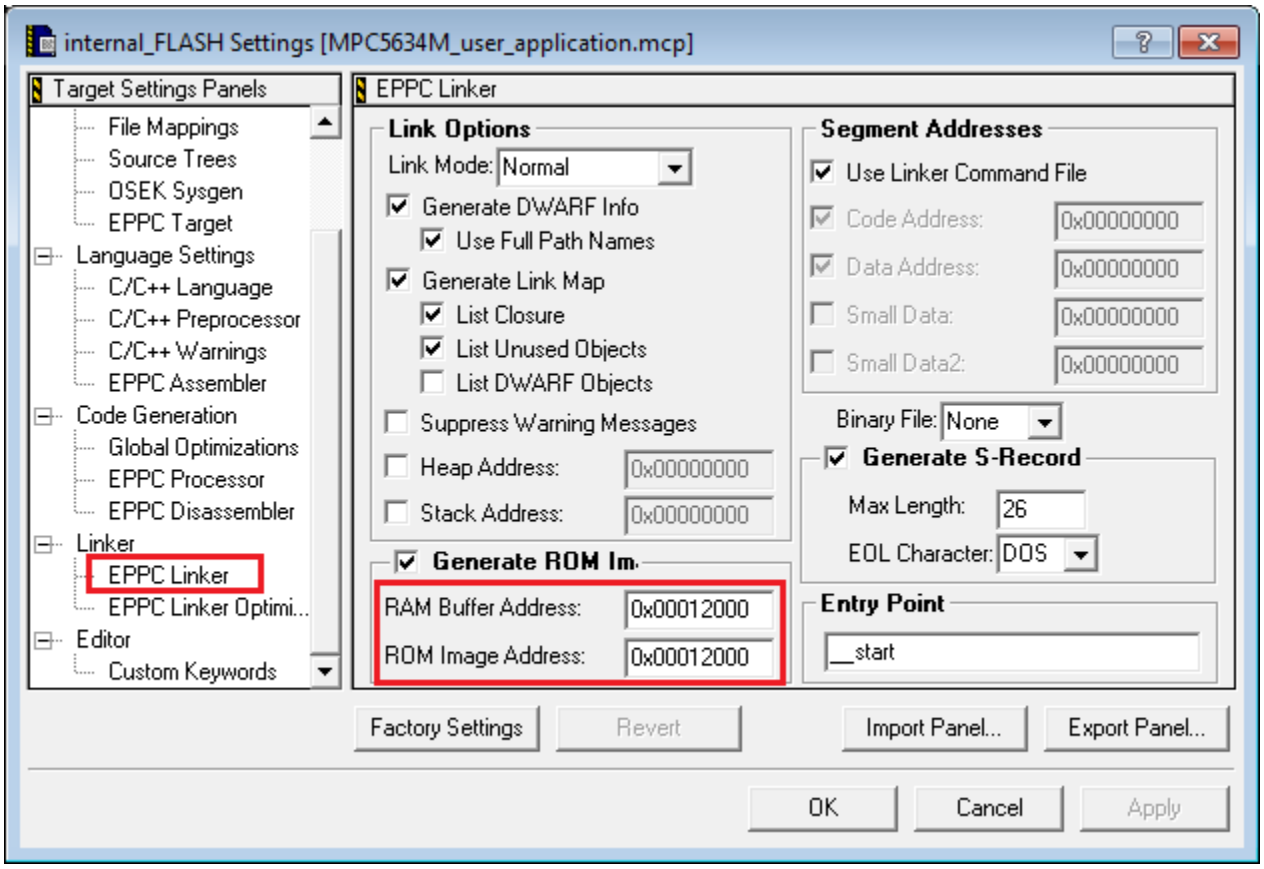


Change the memory segments as follows (move all flash segments above 0x0001_0000):

```
MEMORY
{
    resetvector:          org = 0x00010000,   len = 0x00000008
    init:                 org = 0x00010020,   len = 0x00000FE0
    exception_handlers:   org = 0x00011000,   len = 0x00001000
    internal_flash:       org = 0x00012000,   len = 0x0016E000
    internal_ram:         org = 0x40000000,   len = 0x00014000
    heap  :               org = 0x40014000,   len = 0x00002000
    stack :               org = 0x40016000,   len = 0x00001800
}
```

If we move the base address of internal_flash segment, CodeWarrior IDE requires following setting in order to generate correct ROM image:

Press alt+F7 in CodeWarrior IDE, select “EPPC Linker” and change RAM Buffer Address and ROM Image Address to base address of internal_flash segment. Both parameters must be set to the same value (0x00012000 in our example):



Next step is to adjust the LOAD addresses in linker file. LOAD directive specifies the address of section in ROM image:

```
SECTIONS
{
     .__bam_bootarea LOAD (0x00010000) : {} > resetvector

     GROUP : {
        .init LOAD (0x00010020) : {}
        .init_vle (VLECODE) LOAD (_e_init) : {
          *(.init)
          *(.init_vle)
        }
     } > init

     .__exception_handlers  (VLECODE) LOAD (0x00011000) : {} > exception_handlers

     GROUP  : {
        .text : {}
        .text_vle (VLECODE) ALIGN(0x08): {
           *(.text)
           *(.text_vle)
         }
         .rodata (CONST) : {
           *(.rdata)
           *(.rodata)
         }
         .ctors : {}
         .dtors : {}
        extab : {}
        extabindex : {}
     } > internal_flash

     GROUP : {
         .__uninitialized_intc_handlertable ALIGN(2048) : {}
         .data    : {}
         .sdata   : {}
         .sbss    : {}
         .sdata2  : {}
         .sbss2   : {}
         .bss     : {}
     } > internal_ram
}
```

Now we have ready projects for bootloader and user application.

3 Bootloader

Bootloaders are usually customer specific, so this is just one of possible solutions. This example will use serial interface (RS232) to load user application and certain double-word in flash memory will be used to select if bootloader or user application is supposed to be executed on startup.

Following text is just for short guidance, user is supposed to study the source files for more details.

3.1 Startup condition

If fast start up of user application is required, the startup condition should be checked immediately after reset by asm code before initialization of all resources of bootloader project. Jumping at reset vector of user application can take only a few cycles.

As an entry point, CodeWarrior uses function __start() located in file:

c:\Program Files\Freescale\CW for MPC55xx and MPC56xx 2.10\PowerPC_EABI_Support\Runtime\Src__start.c

Copy this file to bootloader project and add this file to sources, so we can modify this file as per our needs. CodeWarrior will return warnings because of overloading of this source file but we can ommit these warnings.

We will use the last double-word in flash for startup condition. Advantage of this soluiton is that this double-word will be programmed as a last one by bootloader when new firmware is being programmed. In case of unexpected reset or power-down, the double-word will not be programmed. That means the bootloader will be executed after next reset and there's no risk that bootloader calls corrupted user application. Bootloader could also check the CRC of user application but this is out of scope of this application note.

MPC5634M has 1.5MB of flash memory, so the last double-word in flash is at address 0x0017_FFF8 – 0x0017_FFFF. We will use key 0xFFFF_0000_FFFF_0000 to start the user application. This value will be compared with content of last double-word in flash. All other values will start the bootloader.

It is recommended to use value for the key from this table (except all 'F' and all '0'):

0xFFFF_FFFF_FFFF_FFFF
0xFFFF_FFFF_FFFF_0000
0xFFFF_FFFF_0000_FFFF
0xFFFF_0000_FFFF_FFFF
0x0000_FFFF_FFFF_FFFF
0xFFFF_FFFF_0000_0000
0xFFFF_0000_FFFF_0000
0x0000_FFFF_FFFF_0000
0xFFFF_0000_0000_FFFF
0x0000_FFFF_0000_FFFF
0x0000_0000_FFFF_FFFF
0xFFFF_0000_0000_0000
0x0000_FFFF_0000_0000
0x0000_0000_0000_0000

The reason is that all these double words share the same ECC value (0xFF). User application will overprogram this value to zero in order to start the bootloader. This will ensure that no ECC error will occur in flash and no exception will be triggered when reading the double word after overprogramming.

Here is an example of startup code:

```
asm void __start(register int argc, register char **argv, register char **envp)
{
  nofralloc          /* MWERKS: explicitly no stack frame allocation */

  /*************************************************************************/
  /* Startup condition of bootloader */
  /*************************************************************************/

  /* On some devices, it may be necessary to initialize MMU before accessing
     the flash */

  /* Load double-word from flash address 0x0017_FFF8 to r10 and r11 */
  lis     r9,0x0017
  ori     r9,r9,0xFFF8
  lwz     r10,0(r9)

  lis     r9,0x0017
  ori     r9,r9,0xFFFC
  lwz     r11,0(r9)

  /* load "key" 0xFFFF_0000_FFFF_0000 to r8 and r9  */
  lis     r8,0xFFFF
  ori     r8,r8,0x0000
  lis     r9,0xFFFF
  ori     r9,r9,0x0000

  /* compare if flash content at address 0x0017_FFF8 is equal to key */
  xor     r3,r11,r9
  xor     r4,r10,r8
  or      r0,r4,r3
  cmpwi   r0,0

  /* if not equal, continue in executing of the bootloader */
  bne   bootloader

  /* if equal, start the user application */
  /* load address of reset vector of user application from 0x0001_0004 */
  lis   r3,0x0001
  ori   r3,r3,0x0004
  lwz   r3,0(r3)
  /* move address to link register */
  mtlr  r3
  /* branch link register - start the user application */
  blrl

  /* continue executing of the bootloader */
  bootloader:

  /*************************************************************************/
  /* End of startup condition of bootloader */
  /*************************************************************************/
```

3.2 Initialization, serial communicaiton interface and user interface

It is worth considering to set the system frequency to its maximum to achive fast execution of code and also to be able to set higher communication speed of serial interface. Higher communication speed

makes the downloading of code much faster. MPC5634M features eSCI module. Some other devices feature LINFlex module. Notice that the RS-232 level shifter is necessary to communicate with PC.

Serial communication is set to format:

- 8-data bit
- one start bit
- one stop bit
- no parity
- baud rate 115200bps

As an user interface, we can use any terminal emulation program like Microsoft HyperTerminal that supports format described above, Xon / Xoff flow control and that is able to send a text file.

3.3 Flash programming

It is recommended to use Standard Software Drivers (SSD) for flash that are available on website www.freescale.com. MPC5xxx devices implement several versions of flash module. SDD drivers for selected device can be found on its summary page under "Sofware & Tools" tab.

Flash memory on MPC5634M is composed of two banks: 512kB and 1MB. Read-while-write operations are supported only between banks. The code for flash programming must be executed from RAM memory in order to be able to program whole flash.

Used C-array drivers are defined as constants by default. C-array drivers are compiled as position independent and they can be copied to any location. We can either copy the drivers to RAM manually or we can simply re-define them as variables (just delete "const") and they will be automatically copied to RAM by startup code.

3.4 S-record format

All MPC5xxx devices implement Error Correction Code (ECC) in flash memory. ECC is handled on a 64 bit boundary. Thus, if only 1 word in any given 64 bit ECC segment is programmed, the adjoining word (in that segment) should not be programmed since ECC calculation has already completed for that 64-bit segment. Attempts to program the adjoining word results in an operation failure (most likely).Due to ECC, flash memory must be programmed by double-words (64 bits) in a single step.

S-record files generated by CodeWarrior or other IDEs may not be always aligned to 64 bit boundary. It is worth considering to convert the s-record file of user application to format where all s-records line are aligned to 64 bit boundary and where the length of each line is constant. This will make the flash programming much easier and safer.

Once the user application is ready to be downloaded by bootloadder, user can use attached tool SRECCONV.exe to convert the file. It will align all lines to 64 bit boundary and the length of all data fields will be constant 16 bytes. The tool accepts two parameters:

SRECCONV.exe <input file> <output file>

Example:

SRECCONV.exe internal_FLASH.mot output.mot

Output.mot file then can be downloaded by bootloader.

4 User application

There are only two requirements for user application:

- User application must contain a key that is placed as a constant at the last double-word in flash. This allows the bootloader to start user application after download.
- There must be a mechanism that erases the key in flash memory and that triggers software reset when upgrade of firmware is requested. This example will use command received through serial interface. Once 3 consecutive hashes (“#”) are received through the serial interface, the user application will erase the key and reset the device which results in execution of bootloader.

In order to place the key as a constant to flash, one more change is necessary in linker file. Make the internal_flash segment smaller and create new segment for the key:

Original internal_flash segment:

```
internal_flash:     org = 0x00012000, len = 0x0006E000
```

Change it to:

```
internal_flash:     org = 0x00012000, len = 0x0006DFF8
key_segment:        org = 0x0007FFF8, len = 0x00000008
```

Create a section:

```
.__key_segment (VLECODE) LOAD (0x0007FFF8): {} > key_segment
```

Create a constact in main.c:

```
#pragma push
#pragma section all_types ".__key_segment" ".__key_segment"
unsigned long long key = 0xFFFF0000FFFF0000;
#pragma pop
```

And ensure that the constant will not be stripped by the linker. Place the “key” to FORCEACTIVE list in the linker file:

```
FORCEACTIVE { "bam_rchw" "bam_resetvector" "key"}
```

5 User interface guide