Improving Execution Performance on SPI Flash of NUC505

Application Note for 32-bit NuMicro® Family

Document Information

Abstract	This document instructs how to improve execution performance on SPI Flash through moving critical code/data to SRAM for faster execution. The `BootTemplate` samples in the BSP will be taken as examples for explanation.
Apply to	NuMicro® family NUC505 series.

Table of Contents

1 Introduction

As to the NUC505 series, through the SPIM (SPI master), user can regard the SPI Flash as a ROM module and place program on it for execution. The SPI Flash, compared to built-in SRAM, has large memory size but very slow speed. With space and speed taken into consideration, this document describes how to improve program performance on SPI Flash through moving critical code/data to SRAM. For easy explanation, the `BootTemplate` samples in the BSP will be taken as examples.

1.1 Intended Audience

This document is written for users who would write programs on the NUC505 series but at the same time must take into consideration of execution performance on SPI Flash.

1.2 Associated Registers

Associated registers include:

- **VTOR**[1] register. Associated with the Critical on SRAM memory model, by which user can relocate the vector table start address to a different memory location.
- **SYS_LVMPADDR** and **SYS_LVMPLEN** registers. Associated with the Full on SRAM memory model, which are used to set VECMAP[2] to map SRAM to 0x00000000.
- **SYS_IPRST0** register. Associated with the Full on SRAM memory model, which is used to reset CPU and make VECMAP setting become effective at the same time.

[1] Located in ARM® Cortex® -M4 system control block.

[2] Mapping mechanism to map one memory block, e.g. IBR or SRAM to 0x00000000.

2 Memory Models

With space and speed taken into consideration, user may apply memory models introduced here to place critical code/data on SRAM for better performance: Typical, Critical on SRAM, main() on SRAM, Full on SRAM, and Overlay. The Typical memory model arranges read-only code/data on SPI Flash and the remaining on SRAM, with read/write data differently handled. Based on the Typical memory model, the Critical on SRAM memory model moves critical code/data from SPI Flash to SRAM for better performance. The main() on SRAM memory model makes the idea further by moving all code/data to SRAM except unmovable part. Same as the main() on SRAM memory model, the Full on SRAM memory model moves all code/data to SRAM with another approach. The Overlay memory model divides a large program into multiple pieces of code/data which are loaded into SRAM when required.

2.1 Nature of SPI Flash as ROM

For which memory model to apply, user must first take the nature of SPI Flash and SRAM into consideration. The SPI Flash has larger size but much slower speed than SRAM. The two factors are described below.

2.1.1 Large Space Capable of Placing Large Program

If the SPI Flash device embedded in an MCP chip[3] has 2 MB, the SPI Flash has 16 times capacity higher than 128 KB built-in SRAM. Code or read-only data which does not have performance requirements can be located on SPI Flash.

2.1.2 Slow Speed Unsuitable to Run Performance-Critical Task

The SPI Flash access is non-zero wait state. Depending on SPI bus clock and bit mode, execution performance on SRAM is ten times faster than on SPI Flash. Code or data which has performance requirements must be located on SRAM rather than on SPI Flash.

2.2 Typical (All Programs Located on SPI Flash)

This model is a typical ROM memory model, which is listed here as a basis and for comparison with other improved models.

2.2.1 Memory Address Space

Refer to Figure 2-1 Typical Memory Model for illustration of the typical memory layout.

[3] For the type of SPI Flash in an MCP chip, refer to the NUC505 Base Series Naming Rule section in the *NUC505 Series Technical Reference Manual*.

- Read-only section is located on SPI Flash.
- Read/Write section is placed on SPI Flash at the start and then copied to SRAM at run-time for being writable.
- Uninitialized section and reserved regions (heap and stack) are located on SRAM.

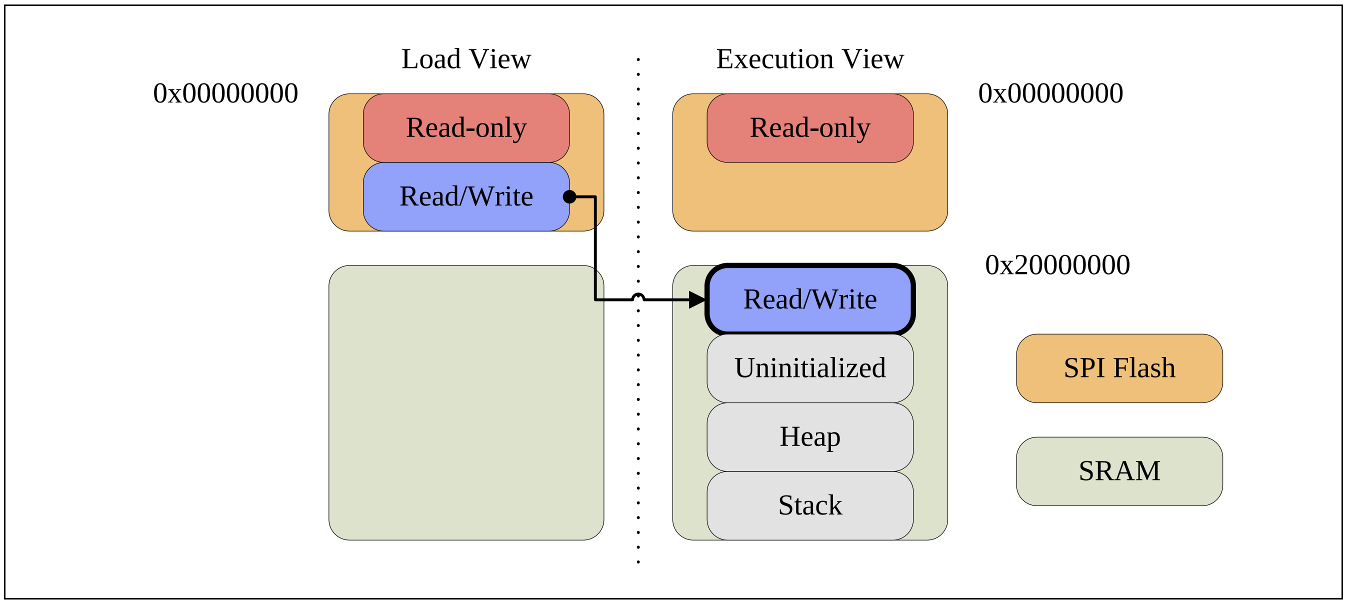


Figure 2-1 Typical Memory Model

2.3 Critical on SRAM (Performance-Critical Task Located on SRAM)

This model is applied when the whole program is too large to fit into 128 KB SRAM. Only small, critical code/data can be moved to SRAM for faster speed. To apply this model, user first needs to analyze his program and find out bottleneck of execution performance. Note that the analysis approach is not within the scope of this document.

2.3.1 Memory Address Space

Based on the Typical model, critical code/data is moved to SRAM for faster speed (refer to Figure 2-2 Critical on SRAM Memory Model).

- Original read-only section is split into two parts: non-critical and critical.
  - Non-critical part is located on SPI Flash.
  - Critical part is placed on SPI Flash at the start and then copied to SRAM at run-time for faster speed.
- Read/Write section is placed on SPI Flash at the start and then copied to SRAM at run-time for being writable.
- Uninitialized section and reserved regions (heap and stack) are located on SRAM.

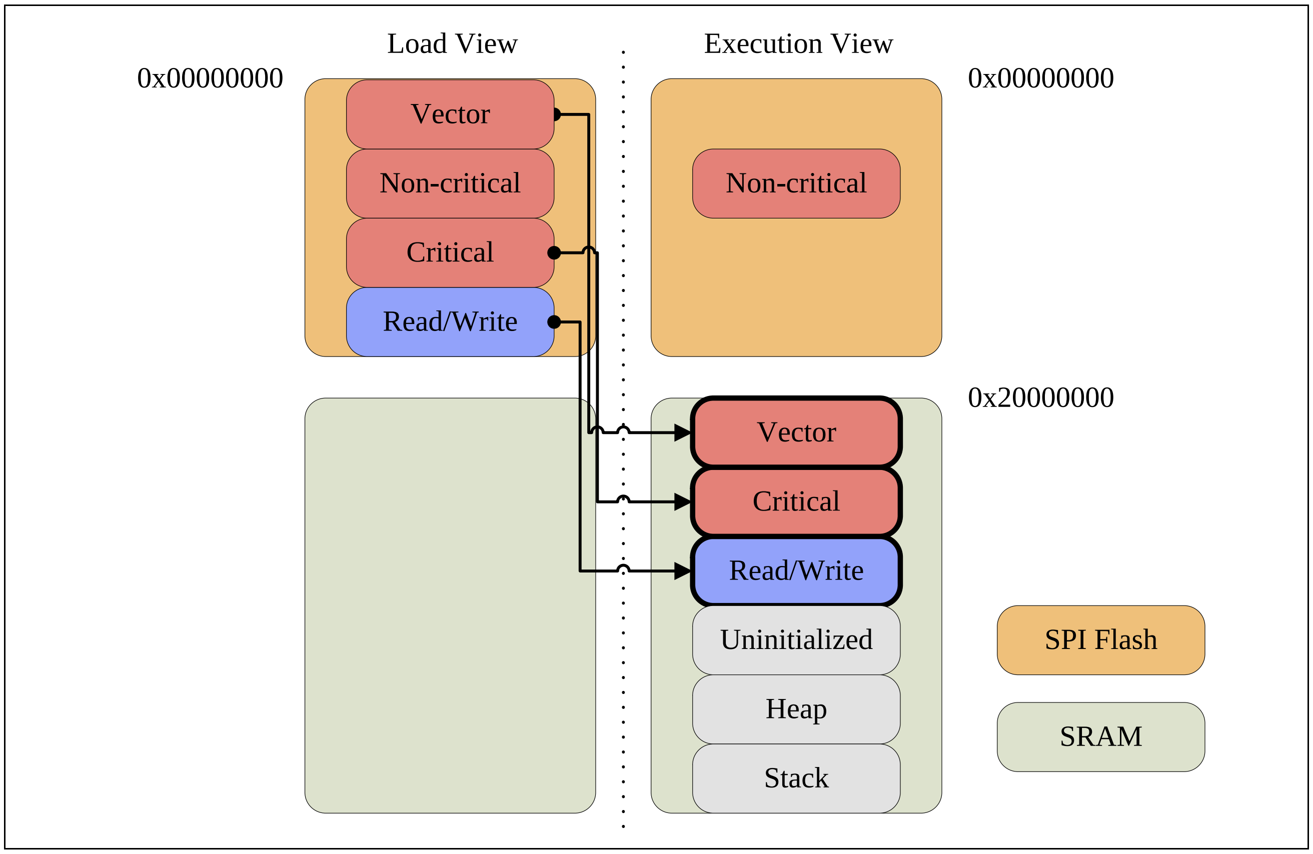
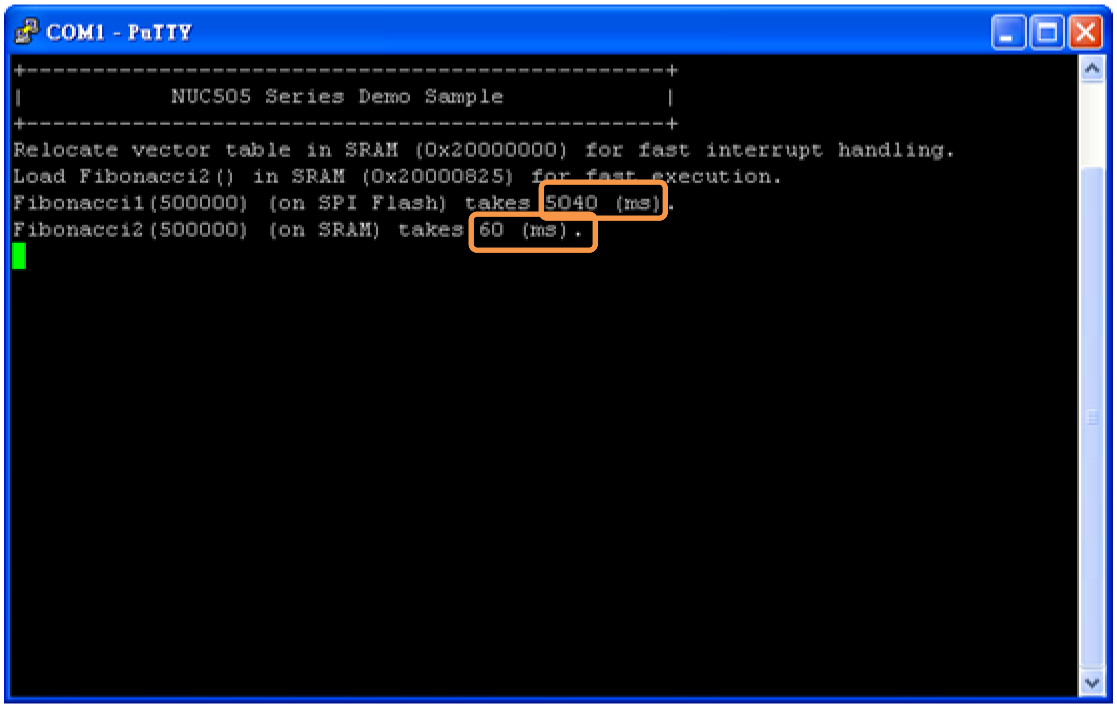


Figure 2-2 Critical on SRAM Memory Model

2.3.2 Improvement after applying this model

In the `BootTemplate/CriticalOnSRAM` sample, the two functions `Fibonacci1()` and `Fibonacci2()` are created to compute Fibonacci sequence. They are located at SPI Flash and SRAM respectively, and their speed difference matches that of SPI Flash and SRAM. For better overall improvement, it is necessary to find out critical part through another means.



```
+-----------------------------------------------+
|            NUC505 Series Demo Sample            |
+-----------------------------------------------+
Relocate vector table in SRAM (0x20000000) for fast interrupt handling.
Load Fibonacci2() in SRAM (0x20000825) for fast execution.
Fibonacci1(500000) (on SPI Flash) takes 5040 (ms).
Fibonacci2(500000) (on SRAM) takes 60 (ms).
```

Figure 2-3 Compute Fibonacci Sequence in `BootTemplate/CriticalOnSRAM`

2.3.3 Sample in BSP

The `BootTemplate/CriticalOnSRAM` sample gives a complete example of this model.

2.3.3.1 Relocate Vector Table to SRAM

The Vector table is a critical part for speed. At the start, it is located on SPI Flash. For faster speed, user can relocate it to SRAM by following the steps below:

1. Reserve one memory region[4] size of which is equal to the vector table.
2. Copy the vector table from SPI Flash to the reserved region on SRAM.
3. Set the VTOR register with start address of the reserved region.

```
{
#if defined ( __CC_ARM )
    extern uint32_t __Vectors[];
    extern uint32_t __Vectors_Size[];
    extern uint32_t Image$$ER_VECTOR2$$ZI$$Base[];
    memcpy((void *) Image$$ER_VECTOR2$$ZI$$Base, (void *) __Vectors, (unsigned int)
__Vectors_Size);
    SCB->VTOR = (uint32_t) Image$$ER_VECTOR2$$ZI$$Base;
```

[4] Refer to Keil scatter-loading description file and IAR linker configuration file for Keil and IAR projects respectively.

```
#elif defined (__ICCARM__)
    #pragma section = "VECTOR2"
    extern uint32_t __Vectors[];
    extern uint32_t __Vectors_Size[];
    memcpy((void *) __section_begin("VECTOR2"), (void *) __Vectors, (unsigned int)
__Vectors_Size);
    SCB->VTOR = (uint32_t) __section_begin("VECTOR2");
#endif
 }
```

2.3.3.2 Keil scatter-loading description file

The following is an example of Keil scatter-loading description file for such model. Portions related to critical code/data, including vector table are highlighted.

LR_ROM   0x00000000 0x00200000 ; 2MB (SPI FLash)

```
{
    ER_STARTUP +0
    {
         startup_nuc505Series.o(RESET, +First)
     }
    ER_RO +0
    {
           *(+RO)
    }
    ; Relocate vector table in SRAM for fast interrupt handling.
    ER_VECTOR2      0x20000000   EMPTY   0x00000400
    {
     }
    ; Critical code in SRAM for fast execution. Loaded by ARM C library at startup.
     ER_FASTCODE_INIT    0x20000400
    {
       clk.o(+RO); CLK_SetCoreClock() may take a long time if it is run on SPI Flash.
    }
    ER_RW   +0
    {
           *(+RW)
    }
    ; Critical code in SRAM for fast execution. Loaded by user.
    ER_FASTCODE_UNINIT  +0  OVERLAY
    {
       *(fastcode)
    }
}

LR_RAM      0x20010000  0x00010000
{
      ER_ZI +0
      {
            *(+ZI)
      }
}
```

2.3.3.3 IAR linker configuration file

The following is an example of IAR linker configuration file for such model. Portions related to critical code/data including vector table are highlighted.

```
define symbol __ICFEDIT_intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__   = 0x00000000;
define symbol __ICFEDIT_region_ROM_end__     = 0x001FFFFF;
define symbol __ICFEDIT_region_RAM_start__   = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__     = 0x2001FFFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__   = 0x400;
define symbol __ICFEDIT_size_heap__     = 0x800;

define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to
__ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to
__ICFEDIT_region_RAM_end__];

define block CSTACK        with alignment = 8, size = __ICFEDIT_size_cstack__  { };
define block HEAP          with alignment = 8, size = __ICFEDIT_size_heap__    { };
define block VECTOR2       with alignment = 8, size = 0x400                    { };

initialize by copy      { readwrite, readonly object clk.o };
initialize manually     { section fastcode };
do not initialize       { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region             { readonly };
place at start of RAM_region          { block VECTOR2 };
place in RAM_region             { readwrite, block CSTACK, block HEAP };
```

2.4 main() on SRAM (Main Program Located on SRAM)

This model is applied when the whole program can fit into 128 KB SRAM. Because program must boot on SPI Flash, initialization code must still be arranged on SPI Flash. To this end, the whole program is split into two parts with the call to `main()` as the dividing line: `init/cstartup` and `post-init/cstartup`. The `init/cstartup` part is still arranged on SPI Flash but the `post-init/cstartup` part is moved to SRAM for faster speed.

2.4.1 Memory Address Space

Based on the Critical on SRAM model, the whole program except the `init/cstart` part is moved to SRAM for faster speed (refer to Figure 2-4 main() on SRAM Memory Model).

- Original read-only section is split into two parts: `init/cstartup` and `post-init/cstartup.`
  - `Init/cstartup` part is located on SPI Flash.
  - `Post-init/cstartup` part is placed on SPI Flash at the start and then copied to SRAM at run-time for faster speed.
- Read/Write section is placed on SPI Flash at the start and then copied to SRAM at run-time for being writable.
- Uninitialized section and reserved sections (heap and stack) are located on SRAM.

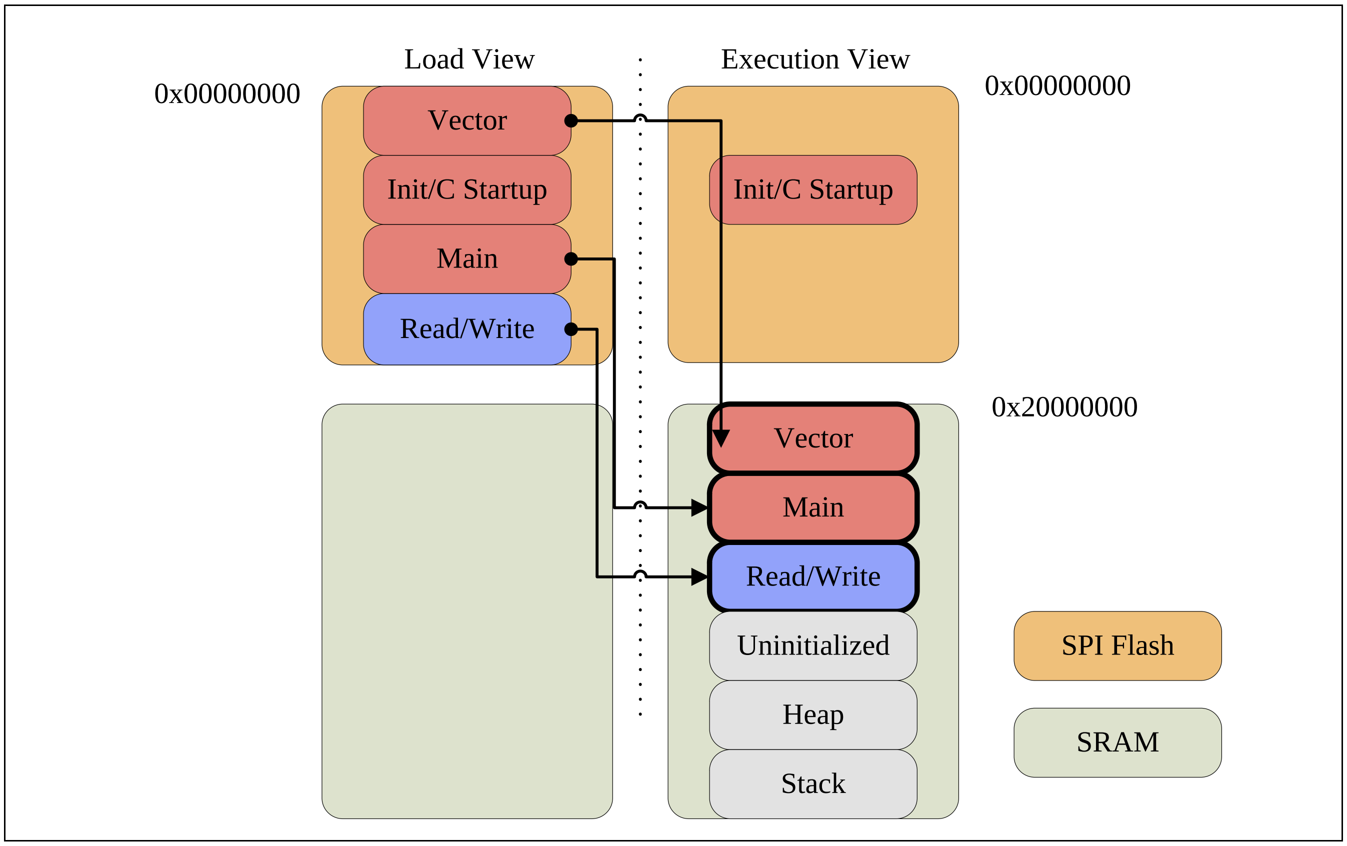


Figure 2-4 main() on SRAM Memory Model

2.4.2 Improvement after applying this model

This model needs a longer initialization time to copy the post-initialization part to SRAM. After that, speed will be the best because code/data runs on SRAM.

2.4.3 Sample in BSP

The `BootTemplate/MainOnSRAM` sample gives a complete example of this model.

2.4.3.1 Keil scatter-loading description file

The following is an example of Keil scatter-loading description file for such model. Portions related to `init/cstartup` and `post-init/cstartup` are highlighted.

```
LR_ROM      0x00000000
{
     ; Code/data at his point belongs to init/cstartup and must be located on SPI Flash.
     ER_STARTUP +0
    {
        startup_nuc505Series.o(RESET, +First)   ; vector table
        *(InRoot$$Sections)                     ; library init
        startup_nuc505Series.o                  ; startup
        system_nuc505Series.o(i.SystemInit)
     }

     ; Code/data from this point belongs to post-init/cstartup and is located on SRAM
     ; for fastest speed.
    ; Relocate vector table in SRAM for fast interrupt handling.
    ER_VECTOR2  0x20000000  EMPTY   0x00000400
    {
    }

    ER_RO       +0
    {
           *(+RO)
     }

    ER_RW       +0
     {
           *(+RW)
     }

    ER_ZI       +0
    {
        *(+ZI)
    }
}
```

2.4.3.2 IAR linker configuration file

The following is an example of IAR linker configuration file for such model. Portions related to critical `init/cstartup` and `post-init/cstartup` are highlighted.

```
define symbol __ICFEDIT_intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__    = 0x00000000;
define symbol __ICFEDIT_region_ROM_end__      = 0x001FFFFF;
define symbol __ICFEDIT_region_RAM_start__    = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__      = 0x2001FFFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__    = 0x400;
define symbol __ICFEDIT_size_heap__      = 0x800;

define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to
__ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to
__ICFEDIT_region_RAM_end__];

define block CSTACK        with alignment = 8, size = __ICFEDIT_size_cstack__  { };
define block HEAP          with alignment = 8, size = __ICFEDIT_size_heap__    { };
define block VECTOR2  with alignment = 8, size = 0x400                    { };

/* Split read-only code/data into init/cstartup and post-init/cstartup here. */
initialize by copy      { readonly, readwrite }    except  { readonly object
startup_nuc505Series.o };

place at address mem:__ICFEDIT_intvec_start__    { readonly section .intvec };

place in ROM_region              { readonly };
place at start of RAM_region     { block VECTOR2 };
place in RAM_region              { readwrite, block CSTACK, block HEAP };
```

2.5 Full on SRAM (Loader Program Located on SPI Flash and User Program Fully Located on SRAM)

Same as the main() on SRAM model, this model is applied when the whole program can fit into 128 KB SRAM. Because program must boot on SPI Flash, rather than one-program solution of the main() on SRAM model, there are two programs in this model: loader program and user program. The loader program is responsible for loading the user program into SRAM for faster speed.

2.5.1 Memory Address Space

In this model, there are two programs: the user program and loader program.

The user program has all code/data directly arranged on SRAM (refer to Figure 2-5 Full on SRAM Memory Model).

- Read-only section is located on SRAM.
- Read/Write section is located on SRAM.
- Uninitialized section and reserved regions (heap and stack) are located on SRAM.

Note: In this model, user application must still be located on from 0x00000000[5] instead of 0x20000000 even on SRAM. Before execution of this program, SRAM will remap from 0x20000000 to 0x00000000 by the loader program as illustrated below.

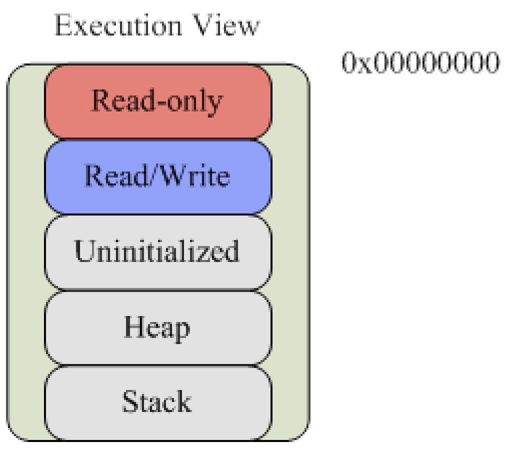


Figure 2-5 Full on SRAM Memory Model

The loader program usually has the following memory address space (refer to Figure 2-6 Loader Memory Model):

- Read-only section is located on SPI Flash.

[5] Mean system address, not SPI Flash address.

- Read/write section is placed on SPI Flash and then copied to SRAM at run-time for being writable.
- User application image is located on SPI Flash at the start and then copied to SRAM at run-time for execution.
- Uninitialized section and reserved sections (heap and stack) are located on SRAM.

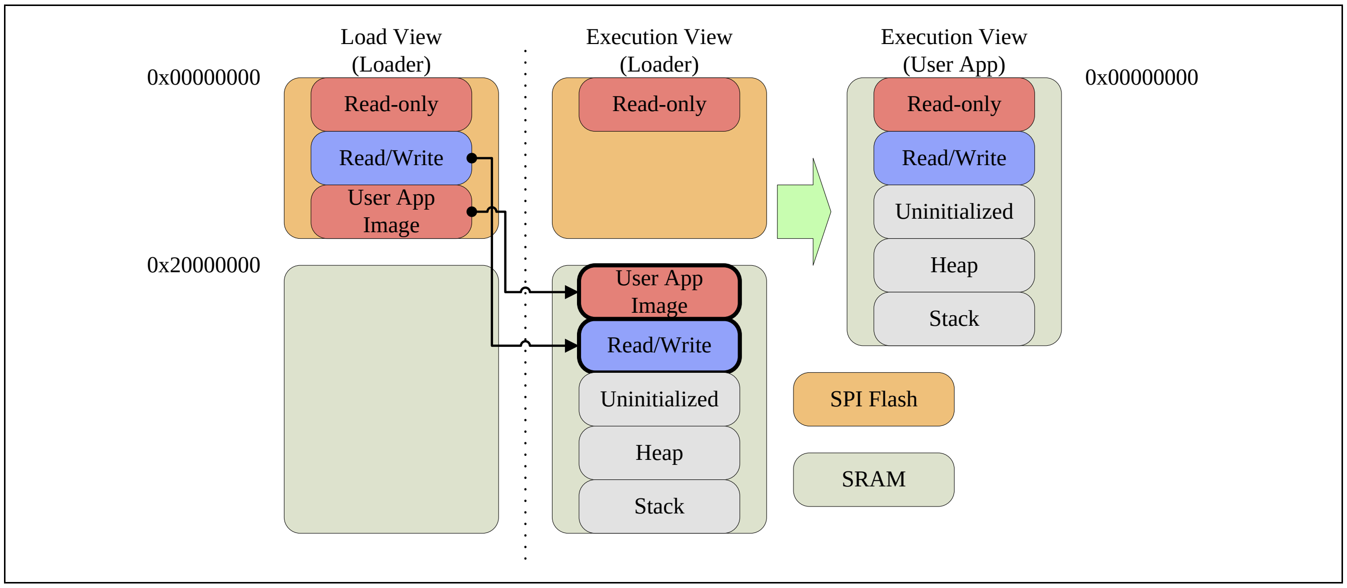


Figure 2-6 Loader Memory Model

2.5.2 Improvement after applying this model

This model needs a longer initialization time for the loader to copy the user application image to SRAM. After the user application starts, speed will be the best because code/data runs on SRAM.

2.5.3 Sample in BSP

The `BootTemplate/FullOnSRAM` and `BootTemplate/Loader` samples together give a complete example of this model.

2.5.3.1 Loader

The loader program is responsible for the following subtasks:

1. Load the user application image from SPI Flash to SRAM.

```
#if defined ( __CC_ARM )
extern uint32_t Load$$ER_RAMIMG$$RO$$Base[];
extern uint32_t Load$$ER_RAMIMG$$RO$$Length[];
extern uint32_t Image$$ER_RAMIMG$$RO$$Base[];
memcpy((void *) Image$$ER_RAMIMG$$RO$$Base, Load$$ER_RAMIMG$$RO$$Base, (unsigned long)
Load$$ER_RAMIMG$$RO$$Length);
#endif
```

2. Remap SRAM [6] from 0x20000000 to 0x00000000. This setting will not become effective until the step immediately following this step.

```
SYS->LVMPADDR = (uint32_t) g_au8RamImg;
SYS->LVMPLEN = 128; // Map 128 KB
```

3. Reset CPU, and SRAM remapping to 0x00000000 becomes effective at the same time.

```
SYS->IPRST0 |= SYS_IPRST0_CPURST_Msk;
```

2.5.3.2 Generate user application image

To make the user application image built into the loader program, follow the steps below:

1. Create the user application image using the fromelf tool attached with Keil MDK-ARM.

fromelf --bin FullOnSRAM.axf --output FullOnSRAM.bin

2. Convert the user application image file FullOnSRAM.bin to a C-style char array FullOnSRAM.dat. User may get related tool from the web site: http://sourceforge.net/projects/bin2header/
3. Compile the C-style char array FullOnSRAM.dat directly into the loader program.

```
#if defined ( __CC_ARM )
static __align(32) const uint8_t g_au8RamImg[] __attribute__((section("ramimg")));
static __align(32) const uint8_t g_au8RamImg[] = {
#   include "FullOnSRAM.dat"
};
#elif defined (__ICCARM__)
#pragma data_alignment=32
static const uint8_t g_au8RamImg[] @ "ramimg";
static const uint8_t g_au8RamImg[] = {
#   include "FullOnSRAM.dat"
};
#endif
```

[6] The unit is KB. Change the mapping range based on real memory model of user application.

2.6 Overlay (Tasks Located at the Same Address of SRAM)

This model is applied when the whole program is too large to fit into 128 KB SRAM but it has some pieces of code/data which can execute independently, just like stages in a game. An overlay is one of multiple pieces of code/data that can be loaded to a pre-determined memory region (called overlay region) on demand at runtime. Initially, each overlay is stored on SPI Flash, and during run-time, an overlay can be copied to a pre-determined address on SRAM for execution there when required. This can later be replaced by another overlay when required. Only one overlay can occupy that overlay region at any time.

2.6.1 Memory Address Space

Based on the Typical model, overlay is moved to SRAM on demand for faster execution. (refer to Figure 2-7 Overlay Memory Model).

- Original read-only section is split into two parts: non-overlay and multiple overlays.
  - Non-overlay part is located on SPI Flash.
  - An overlay is placed on SPI Flash at the start and then copied to SRAM at run-time when required.
- Read/Write section is placed on SPI Flash at the start and then copied to SRAM at run-time for being writable.
- Uninitialized section and reserved regions (heap and stack) are located on SRAM.

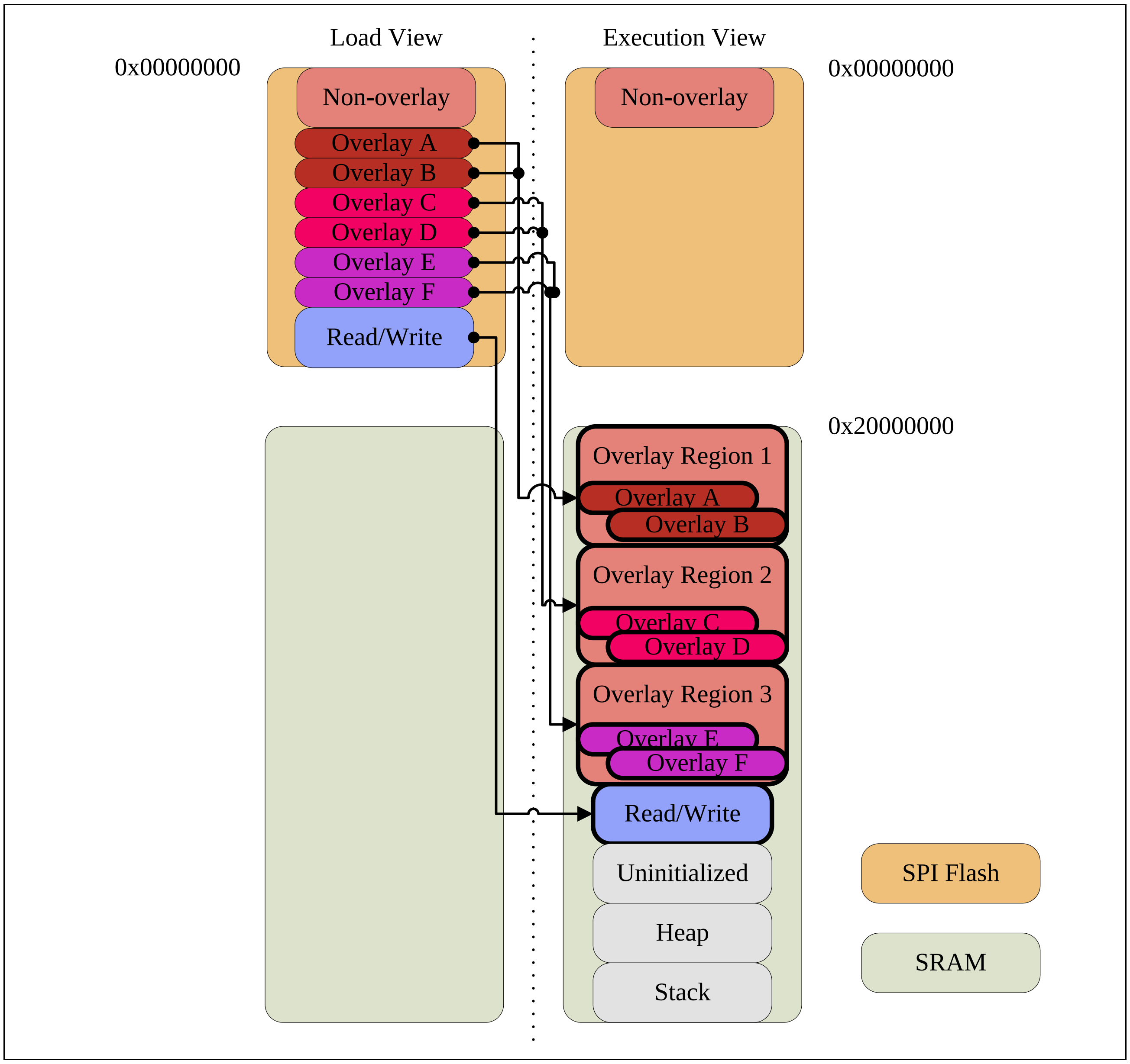


Figure 2-7 Overlay Memory Model

2.6.2 Improvement after applying this model

Improvement of this model depends on how a program is divided into overlays. To run an overlay, it is necessary to load it into SRAM first. This is an unavoidable cost. User must be careful to design his overlay structure to avoid unnecessary load overlay operations. If a program can behave like a game which has multiple stages, it is inherently suitable for this model.

2.6.3 Sample in BSP

The `BootTemplate/Overlay` sample gives a complete example of this model. The following lists some notes of this sample:

- Define an overlay table in both usrprog_ovly_tab.c and the linker script file. The overlay table consists of overlays and overlay regions. In this sample, the overlay table is defined as below:
  overlay a/b   overlay region 1
  overlay c/d/e   overlay region 2
  overlay f   overlay region 3
- Load/exec addresses of an overlay are determined through the linker script file. These addresses can be acquired through linker-generated symbols. See DEFINE_OVERLAY in ovlymgr.h for how to access them.
- The function `load_overlay()` is responsible for loading overlay to overlay region through SPIM DMA Read mode or `memcpy().` All overlay manager code (ovlymgr.c) must be located in SRAM for running SPIM DMA Read mode.
- User must be responsible for calling `load_overlay()` to load overlay before its execution.
- Overlaid program cannot be source-level debugged.
- This sample just demonstrates how to overlay code. Overlay data is not supported.

2.6.3.1 Keil scatter-loading description file

The following is an example of Keil scatter-loading description file for such model. Portions related to overlay are highlighted.

```
LR_ROM      0x00000000
{
     ER_STARTUP +0
    {
        startup_nuc505Series.o(RESET, +First)    ; vector table
        *(InRoot$$Sections)                      ; library init
        ; If neither (+ input_section_attr) nor (input_section_pattern) is specified, the
default is +RO.
        startup_nuc505Series.o                   ; startup
        system_nuc505Series.o(i.SystemInit)
     }

    ER_RO       +0
    {
           *(+RO)
     }

    ; Relocate vector table in SRAM for fast interrupt handling.
    ER_VECTOR2  0x20000000  EMPTY   0x00000400
    {
    }

    ; Critical code located in SRAM. Loaded by ARM C library at startup.
    ER_FASTCODE_INIT +0
    {
        clk.o(+RO)          ; CLK_SetCoreClock() may take a long time if it is run on SPI
Flash.
        ovlymgr.o(+RO)    ; Overlay manager itself must locate in SRAM because it will be
responsible for loading code
                            ; through SPIM DMA Read.
    }

    ER_RW       +0
      {
            *(+RW)
      }

    ER_ZI       +0
    {
        *(+ZI)
    }

     ; Define overlay table:
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ER_OVERLAY_A +0  OVERLAY NOCOMPRESS
    {
        *(overlay_a)
    }
    ER_OVERLAY_B +0  OVERLAY NOCOMPRESS
    {
        *(overlay_b)
    }
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ; Serve to separate overlay regions.
    ER_SEPARATOR_1  +0  EMPTY   0   {}

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ER_OVERLAY_C +0  OVERLAY NOCOMPRESS
    {
        *(overlay_c)
    }
    ER_OVERLAY_D +0  OVERLAY NOCOMPRESS
    {
        *(overlay_d)
    }
    ER_OVERLAY_E +0  OVERLAY NOCOMPRESS
    {
        *(overlay_e)
    }
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ; Serve to seperate overlay regions.
    ER_SEPARATOR_2  +0  EMPTY   0   {}

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ER_OVERLAY_F +0  OVERLAY NOCOMPRESS
    {
        *(overlay_f)
    }
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ; Serve to mark end of used SRAM.
    ER_INDICATOR_END  +0  EMPTY   0 {}
}
```

2.6.3.2 IAR linker configuration file

The following is an example of IAR linker configuration file for such model. Portions related to overlay are highlighted.

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__   = 0x00000000;
define symbol __ICFEDIT_region_ROM_end__     = 0x001FFFFF;
define symbol __ICFEDIT_region_RAM_start__   = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__     = 0x2001FFFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__   = 0x400;
define symbol __ICFEDIT_size_heap__     = 0x800;
/**** End of ICF editor section. ###ICF###*/


define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to
__ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to
__ICFEDIT_region_RAM_end__];


define block CSTACK            with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP              with alignment = 8, size = __ICFEDIT_size_heap__     { };
define block VECTOR2           with alignment = 8, size = 0x400                     { };


/* Overlay manager (ovlymgr.c) must locate in SRAM because it is responsible for loading
overlay through SPIM Read mode. */
initialize by copy      { readwrite, readonly object clk.o, readonly object ovlymgr.o };
/* Overlays are initially stored in SPI Flash and then copied to SRAM for execution at
run-time when required. */
initialize manually     {
    section overlay_a, section overlay_b, section overlay_c, section overlay_d,
    section overlay_e, section overlay_f
};
//initialize by copy with packing = none { section __DLIB_PERTHREAD }; // Required in a
multi-threaded application
do not initialize       { section .noinit };


/* Define overlay table:
  overlays a/b    overlay region 1
  overlays c/d/e    overlay region 2
  overlay f    overlay region 3 */
define overlay OVERLAY_1        { section overlay_a };
define overlay OVERLAY_1        { section overlay_b };
define overlay OVERLAY_2        { section overlay_c };
define overlay OVERLAY_2        { section overlay_d };
define overlay OVERLAY_2        { section overlay_e };
define overlay OVERLAY_3        { section overlay_f };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region             { readonly };
place at start of RAM_region    { block VECTOR2 };
place in RAM_region             { readwrite, block CSTACK, block HEAP };
place in RAM_region             { overlay OVERLAY_1, overlay OVERLAY_2, overlay 
OVERLAY_3 };
```

Revision History

Date	Revision	Description
2015.10.12	1.00	1. Initially issued.

Important Notice