# AN14893

## S32K3xx Linker File and Startup Code

Rev. 1.0 — 28 December 2025 Application note

**Document information**

| Information | Content |
|---|---|
| Keywords | Linker file, startup code |
| Abstract | The linker script is a text file which has the file extension of .ld. It explains how different sections of the object files should be merged to create an executable file. It also includes the code and data memory address and size information. |

# 1 Introduction

The linker file is written using the GNU linker command language and most of the startup code is using assembly language. There are some situations that the programmer needs to modify the code of these areas. This application note can help them to better understand the GNU linker command language that showed up in the S32K3 linker file, how the memory of S32K3 is allocated by the linker file, how S32K3 is initialized after SBAF initialization and what is done before jumping to the main function. Some typical examples are also provided to show the modification for the linker file and startup code.

***Note:*** *The code mentioned in this AN is based on RTD 2.0.1.*

The linker script is a text file which has the file extension of .ld. It explains how different sections of the object files should be merged to create an executable file. It also includes the code and data memory address and size information.

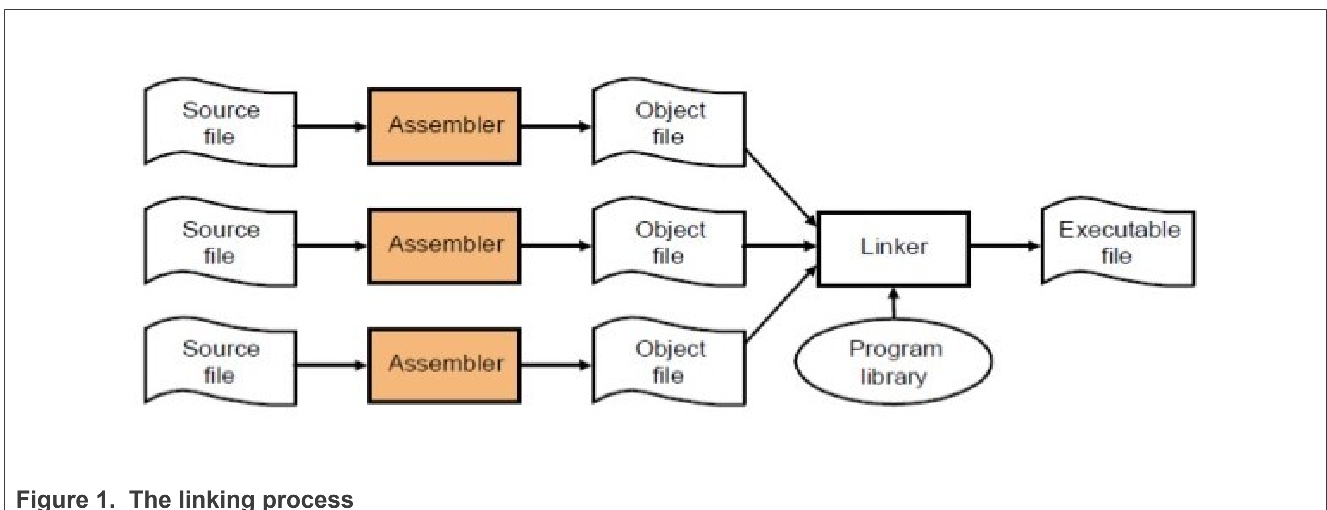


**Figure 1. The linking process**

# 2 ENTRY command

This command is used to set the “Entry point address” information in the header of the executable .elf file. It’s the first code to execute after the microcontroller reset.

For S32K3, we can see the code **`ENTRY`**`(Reset_Handler)` in the linker file. It indicates the function `Reset_Handler()` is the first code to execute for the application cortex M7 core after the sBAF execution.

# 3 MEMORY command

This command allows you to describe the different memories present in the target and their start address and size information.

The typical syntax is:



© 2026 NXP B.V. All rights reserved.

```
MEMORY
{
     name1 (attribute) : ORIGIN = origin1, LENGTH = len1
    name2 (attribute) : ORIGIN = origin2, LENGTH = len2
    …
}
```

**Figure 2. Syntax: MEMORY command**

The linker will calculate the total code and data memory consumed so far and will report an error if the data, code or stack areas cannot fit into available size.

The name of the memory region can be any legal name compliant with the C variable naming rule, it will be later referenced by other parts of the linker script.

The attribute can be “**R**” (Read-only), “**W**” (read/write), “**X**” (Executable), “**A**” (Allocatable), “**I/L**” (Initialized) or their combinations. It can also be left open.

## 4 SECTIONS command

The SECTIONS command tells the linker how to map input sections into output sections,

and how to place the output sections in memory.

There are 4 kinds of typical section types.

text – code

rodata – read-only data

data – read-write initialized data

bss – read-write zero initialized data

There are 2 concepts VMA and LMA. VMA means Virtual Memory Address or runtime address, LMA means Load Memory Address or ROM address. For the global variables which have the initial value(data section), these initial values are stored in the flash and are loaded to the RAM by the startup code. Thus the ROM address is LMA, the RAM address is VMA. The bss section does not have the LMA.

The typical syntax is:


All information provided in this document is subject to legal disclaimers.

© 2026 NXP B.V. All rights reserved.

```
SECTIONS
{
    .flash :
    {
        …
        *(.text)
        *(.rodata)
        …
    } > int_flash
    .sram_data : AT(__sram_data_rom) /*LMA is __sram_data_rom */
    {
        *(.data)
    } > int_sram /*VMA is in int_sram */
    .sram_bss (NOLOAD) : /*There is no LMA for bss section*/
    {
        *(.bss)
    } > int_sram /*VMA is in int_sram */
}
```

**Figure 3. Syntax: SECTIONS command**

## 5 Wildcard character (*)

The wildcard character (*) simply tells the linker to merge the section of all the input file.

Code examples:

```
*(.text)   /*merge the .text section of all the input file*/
*(.rodata) /*merge the .rodata section of all the input file*/
*(.data)   /*merge the .data section of all the input file*/
```

**Figure 4. Code example: Wildcard character (*)**

## 6 Location counter (.)

This is a special linker symbol denoted by a dot (.). The linker automatically updates this symbol with location(address) information. It can be used to track and define the boundaries of various sections. This location counter can be set to any specific value as well.

Code examples:



© 2026 NXP B.V. All rights reserved.

```
__text_end = .; /*the symbol __text_end equals to current address*/
. = ALIGN(4);   /*increment the current address to the next 4-bytes aligned address*/
. += 0x100;     /*increment the current address by 0x100 */
. = 0x20401000; /*locate the current address to 0x20401000*/
```

Figure 5. Code example: Location counter (.)

# 7 ALIGN command

The ALIGN command simply returns the address of the next aligned boundary if the current address is not aligned. The aligned value must be the power of 2.

```
. = ALIGN(4096); /*increment the current address to the next 4096-bytes aligned address*/
. = ALIGN(10);   /*wrong. 10 is not power of 2.*/
```

Figure 6. Code example 1

From the linker file and map file of S32K3, we can see after storing the .boot_header which is 0x2c bytes long from the flash address 0x0040_0000, since the current address is 0x0040_002c which is not 4096 bytes aligned, the location counter increments to the next 4096 bytes aligned boundary, which is 0x0040_1000. And the symbol _text_start which indicates the start address of the code section start here.

```
    .flash :
    {
      KEEP(*(.boot_header))
        . = ALIGN(4096);

        __text_start = .;


.flash          0x00400000    0x12148
 *(.boot_header)
 .boot_header   0x00400000       0x2c ./Project_Settings/Startup_Code/startup_cm7.o
                0x00401000                . = ALIGN (0x1000)
 *fill*         0x0040002c      0xfd4
                0x00401000                __text_start = .
```

Figure 7. Code example 2

# 8 AT command

The AT command indicates the LMA of the section.

Code example:

```
.sram_data : AT(__sram_data_rom) /*the LMA of .sram_data section is __sram_data_rom */
{
    …
    *(.data)
    …
} > int_sram /* the VMA of .sram_data section is in int_sram */
```

Figure 8. Code example: AT command



© 2026 NXP B.V. All rights reserved.

## 9 KEEP command

When link-time garbage collection is in use ('--gc-sections') as link option, it is often useful to mark sections that should not be eliminated. This is accomplished by surrounding an input section's wildcard entry with KEEP().

Code examples:

```
KEEP(*(.boot_header)) /*keep the .boot_header section not be eliminated by the linker*/
```

Figure 9. Code example: KEEP command

## 10 Linker script symbol

A symbol is the name of an address. It's similar to the variable declaration in the 'C' application. Most of the symbols are to record the boundaries of the sections and are referred by the startup code for memory initialization. And of course they can be used in the 'C' application.

Code examples:

```
__INT_SRAM_START = ORIGIN(int_sram);/*symbol __INT_SRAM_START equals the origin address of memory region int_sram*/
__INT_ITCM_END = ORIGIN(int_itcm) + LENGTH(int_itcm);/*symbol __INT_ITCM_END equals the origin address of memory region int_itcm plus its length*/
__sram_bss_start = .;/*symbol __sram_bss_start equals current address*/
__BSS_SRAM_SIZE = __sram_bss_end - __sram_bss_start; /*symbol __BSS_SRAM_SIZE equals the end address of .sram_bss section minus the start address of .sram_bss section*/
```

Figure 10. Code example: Linker script symbol

## 11 Explanation of the S32K3 linker file

Below shows a full (`.sram_data)` section definition from the S32K3 linker file <linker_flash_s32k344.ld>.



© 2026 NXP B.V. All rights reserved.

```
    .flash :
    {
      …
      *(.text) /*merge all .text section of all input files*/
      …
    } > int_flash /*the .flash section is located in the memory region int_flash */

    . = ALIGN(4); /*increment the location counter to a 4-bytes aligned address*/
    __text_end = .; /*the address of the the symbol __text_end which indicates the end of
                      the occupation of .flash section in the memory region int_flash is
                      current address.*/
    __sram_data_rom = __text_end; /*The address of the symbol __sram_data_rom which
                                   indicate the start address of the copy for the .sram_data
                                   section in flash is equal to the symbol __text_end */

    .sram_data : AT(__sram_data_rom) /*the start of the LMA of .sram_data section is in
__sram_data_rom */
    {
        . = ALIGN(4); /*increment the location counter to a 4-bytes aligned address*/
        __sram_data_begin__ = .; /*the VMA of symbol __sram_data_begin__ which indicates
                                   the start address of the .sram_data section is current
                                   address */
        . = ALIGN(4);
        *(.ramcode)    /*merge all .ramcode section of all input files*/
        . = ALIGN(4);
        *(.data)  /*merge all .data section of all input files*/
        *(.data*) /*merge all the section which has the prefix .data of all input files*/
        . = ALIGN(4);
        *(.mcal_data) /*merge all .mcal_data section of all input files*/
        . = ALIGN(4);
        __sram_data_end__ = .; /*the VMA of symbol __sram_data_end__ which indicates the
                                end address of .sram_data section is current address */
    } > int_sram /*the .sram_data section is located in the memory region int_sram */

    __sram_data_rom_end = __sram_data_rom + (__sram_data_end__ - __sram_data_begin__);
    /*calculate the end of the LMA of .sram_data section.It will be used for ROM copy*/
    .sram_bss (NOLOAD) : /*no data need to load from flash for the .sram_bss section */
    {
      …
      *(.bss) /*merge all .bss section of all input files*/
      …
    } > int_sram /*the .sram_bss section is located in the memory region int_sram */
```

Figure 11. Code example: S32K3 linker file

## 12 Startup Code Overview

After SBAF initialization, the HSE core(ARM Cortex M0+) goes to sleep, the application core(ARM Cortex M7) takes over. Like other ARM Cortex-M core MCUs, it starts executing from the function Reset_Handler(), it jumps to the function main() where the engineer can start writing the application code. The below chart shows the whole procedure of the startup code, which is written by the assembly code mainly.



© 2026 NXP B.V. All rights reserved.

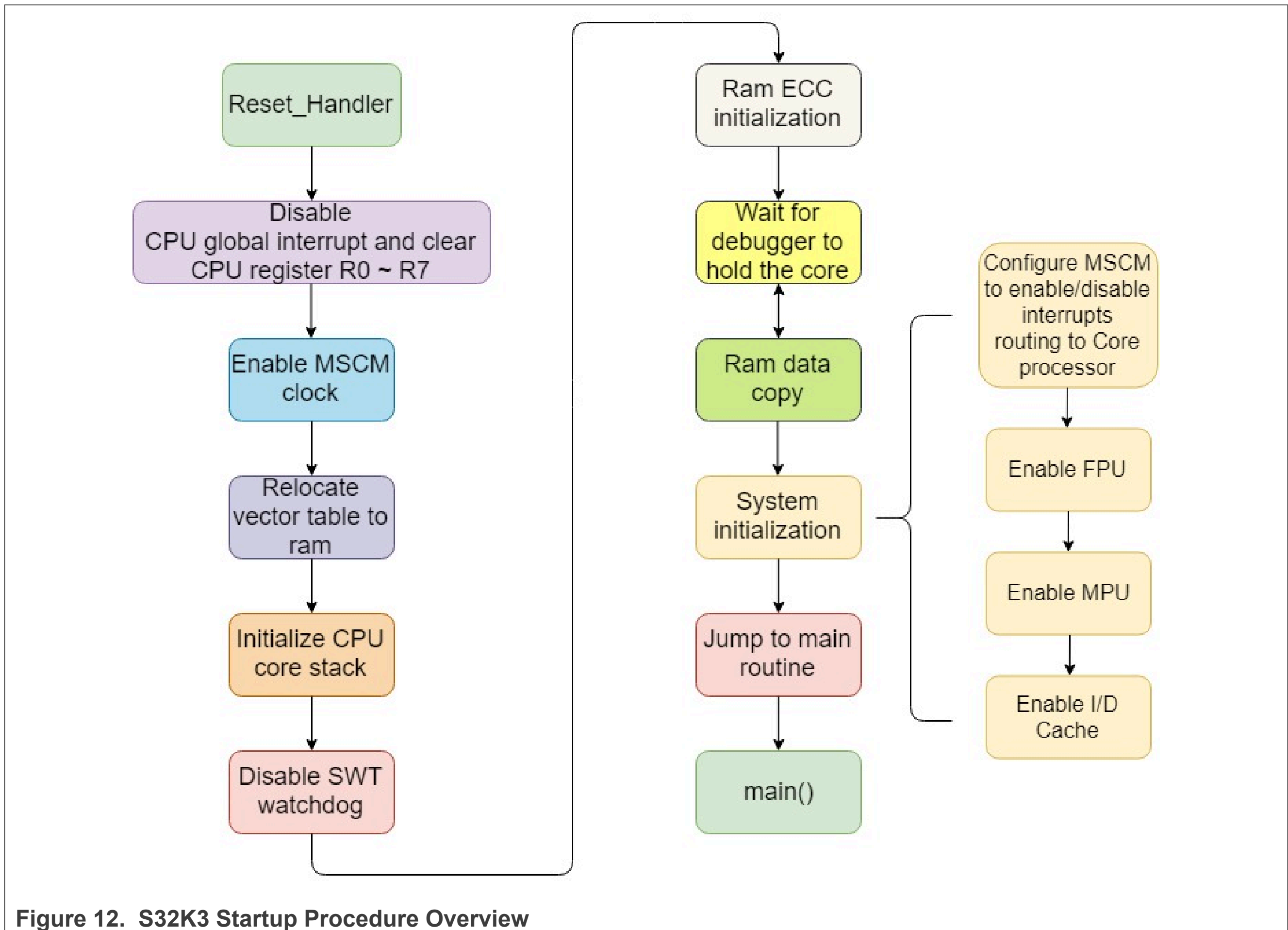


Figure 12. S32K3 Startup Procedure Overview

## 12.1 Boot header

The boot header is defined in the file startup_cm7.s. It is also called Image Vector Table (IVT). It contains the information of boot configuration, core start address, Life Cycle configuration,.etc. For more details please check the boot chapter of the S32K3 Reference Manual. It can only be stored in the 5 fixed addresses (0x0040_0000, 0x0050_0000, 0x0060_0000, 0x0070_0000, 0x1000_0000) and be parsed by the sBAF. The default S32K3 linker file store the boot header in the address 0x0040_0000.



© 2026 NXP B.V. All rights reserved.

| Address offset | Size in bytes | Content | Details |
|---|---|---|---|
| 00h | 4 | IVT marker | Magic number that marks the starting of the IVT location. Its value must be 5AA5_5AA5h. |
| 04h | 4 | Boot configuration word | Configuration word that allows you to select the various configuration options available to boot the chip. See Boot configuration word for details. |
| 08h | 4 | Reserved | — |
| 0Ch | 4 | Cortex-M7_0 application core start address | Boot address of the Cortex-M7_0 application core in the code flash memory area. It must honor core VTOR alignment restrictions. SBAF uses this field when the BOOT_SEQ field in Boot configuration word is 0. |
| 10h | 4 | Reserved | — |
| 14h | 4 | Cortex-M7_1 application core start address | Boot address of the Cortex-M7_1 application core in flash memory. It must honor core VTOR alignment restrictions. SBAF uses this field only when the BOOT_SEQ field in Boot configuration word is 0.<br>This field is ignored if lockstep configuration is enabled. |
| 18h | 8 | Reserved | — |
| 20h | 4 | Reserved | — |
| 24h | 4 | LC configuration address | Configuration word address that allows you to advance the chip's LC to the next stage. See Address LC configuration word for details. |
| 28h | 4 | Reserved | — |
| 2Ch | 4 | Reserved | — |
| 30h | 192 | Reserved | — |
| F0h | 16 | Reserved | — |

**Figure 13. Boot header definition details**

```
.section ".boot_header","ax"
  .long SBAF_BOOT_MARKER /* IVT marker */
  .long (CM7_0_ENABLE << CM7_0_ENABLE_SHIFT) | (CM7_1_ENABLE << CM7_1_ENABLE_SHIFT) /* Boot
configuration word */
  .long 0 /* Reserved */
  .long CM7_0_VTOR_ADDR /* CM7_0 Start address */
  .long 0 /* Reserved */
  .long CM7_1_VTOR_ADDR /* CM7_1 Start address */
  .long 0 /* Reserved */
  .long 0 /* Reserved */
  .long XRDC_CONFIG_ADDR /* XRDC configuration pointer */
  .long LF_CONFIG_ADDR /* Lifecycle configuration pointer */
  .long 0 /* Reserved */
```

**Figure 14. Code example: Boot header**

## 12.2 Disable global interrupt and clear CPU register

S32K3 disables the global interrupt and clears the CPU register R0 – R7 at the beginning of the Reset_Handler().

Code:



© 2026 NXP B.V. All rights reserved.

```
cpsid i
 mov   r0, #0
 mov   r1, #0
 mov   r2, #0
 mov   r3, #0
 mov   r4, #0
 mov   r5, #0
 mov   r6, #0
 mov   r7, #0
```

**Figure 15. Code example: disable global interrupt and clear CPU register**

## 12.3 Enable MSCM clock

This step enables the clock of MSCM module which contains CPU configuration, On-chip memory control, interrupt router control,.etc by writing 1 to Register PRTN1_COFB0_CLKEN[REQ24]. The enablement requires the MODE_UPD register configuration and following with a valid key combination.

```
#ifndef NO_MSCM_CLOCK_INIT
InitMSCMClock:
  /* If the MSCM clock is enabled, skip this sequence */
  ldr r0, =MCME_PRTN1_COFB0_STAT
  ldr r1, [r0]
  ldr r2, =MCME_MSCM_REQ
  and r1, r1, r2
  cmp r1, 0
  bne SetVTOR

  /* Enable clock in PRTN1 */
  ldr r0, =MCME_PRTN1_COFB0_CLKEN
  ldr r1, [r0]
  ldr r2, =MCME_MSCM_REQ
  orr r1, r2
  str r1, [r0]

  /* Set PUPD field */
  ldr r0, =MCME_PRTN1_PUPD
  ldr r1, [r0]
  ldr r2, =1
  orr r1, r2
  str r1, [r0]

  /* Trigger update */
  ldr r0, =MCME_CTL_KEY
  ldr r1, =MCME_KEY
  str r1, [r0]
  ldr r1, =MCME_INV_KEY
  str r1, [r0]
#endif
/* Check MSCM clock in PRTN1 */
WaitForClock:
  ldr r0, =MCME_PRTN1_COFB0_STAT
  ldr r1, [r0]
  ldr r2, =MCME_MSCM_REQ
  and r1, r1, r2
  cmp r1, 0

  beq WaitForClock
```

**Figure 16. Code example: Enable MSCM clock code**

 © 2026 NXP B.V. All rights reserved.

## 12.4 Relocate vector table to RAM

This vector table(`intc_vector`) is defined in the file Vector_Table.s. It is stored in the flash primarily and copied to the SRAM. This step sets the base address of the vector table RAM location to the VTOR(Vector Table Offset Register) thus the core can jump to the relevant ISR according to different vector numbers. There are 2 benefits of relocating the vector table to the RAM instead of flash, the first one is when the core fetches the ISR once an interrupt occurs, accessing RAM will be quicker than accessing the flash. The other benefit is the user may need to modify the ISR names in their user code since the vector table is in the RAM now instead of flash.

Code:

```
SetVTOR:
/* relocate vector table to RAM */
ldr  r0, =VTOR_REG
ldr  r1, =__RAM_INTERRUPT_START

str  r1,[r0]
```

**Figure 17. Code example: Relocate vector table to RAM**

## 12.5 Initialize CPU core stack

This step initializes the stack pointer MSP.

Code:

```
/*GetCoreID*/
ldr  r0, =0x40260004
ldr  r1,[r0]

ldr  r0, =MAIN_CORE
cmp  r1,r0
beq     SetCore0Stack
b SetCore1Stack

SetCore0Stack:
  /* set up stack; r13 SP*/
  ldr  r0, =__Stack_start_c0
  msr MSP, r0
  b DisableSWT0

SetCore1Stack:
  /* set up stack; r13 SP*/
  ldr  r0, =__Stack_start_c1

  msr MSP, r0
```

**Figure 18. Example code block: Initialize CPU core stack**

## 12.6 Disable SWT watchdog

The software watchdog timer is enabled by default. Need to be disabled during the startup process.

Code:

 © 2026 NXP B.V. All rights reserved.

```
DisableSWT0:
  ldr  r0, =0x40270010
  ldr  r1, =0xC520
  str  r1, [r0]
  ldr  r1, =0xD928
  str  r1, [r0]
  ldr  r0, =0x40270000
  ldr  r1, =0xFF000040
  str  r1, [r0]
  b    RamInit

DisableSWT1:
  ldr  r0, =0x4046C010
  ldr  r1, =0xC520
  str  r1, [r0]
  ldr  r1, =0xD928
  str  r1, [r0]
  ldr  r0, =0x4046C000
  ldr  r1, =0xFF000040
  str  r1, [r0]

  b    RamInit
```

**Figure 19. Disable SWT watchdog**

## 12.7 RAM ECC initialization

After the chip's power on reset and before using the RAM, the user must initialize it to avoid ECC errors. The program size must be 2 words(64 bits) aligned. The same procedure is done for SRAM, DTCM and ITCM. This is the most time-consuming step for the whole startup procedure.

Code:

```
RamInit:
/* Initialize SRAM ECC */
ldr r0, =__RAM_INIT
    cmp  r0, 0
/* Skip if __SRAM_INIT is not set */
    beq SRAM_LOOP_END
            ldr r1, =__INT_SRAM_START
            ldr r2, =__INT_SRAM_END
            subs    r2, r1
            subs    r2, #1
            ble SRAM_LOOP_END
            movs    r0, 0
            movs    r3, 0
            SRAM_LOOP:
            stm r1!, {r0,r3}
            subs r2, 8
```



© 2026 NXP B.V. All rights reserved.

```
bge SRAM_LOOP
```

```
SRAM_LOOP_END:
            DTCM_Init:
```

```
/* Initialize DTCM ECC */
```

```
    ldr  r0, =__DTCM_INIT
            cmp  r0, 0
```

```
/* Skip if __DTCM_INIT is not set */
```

```
    beq DTCM_LOOP_END
```

```
/* Enable TCM */
```

```
    LDR r1, =CM7_DTCMCR
            LDR r0, [r1]
```

```
LDR r2, =0x1
```

```
    ORR r0, r2
```

```
STR r0, [r1]
```

```
    ldr r1, =__INT_DTCM_START
            ldr r2, =__INT_DTCM_END
            subs    r2, r1
            subs    r2, #1
            ble DTCM_LOOP_END
            movs    r0, 0
            movs    r3, 0
            DTCM_LOOP:
            stm r1!, {r0,r3}
            subs r2, #8
```

```
bge DTCM_LOOP
```

```
DTCM_LOOP_END:
            ITCM_Init:
```

```
/* Initialize ITCM ECC */
```

```
    ldr  r0, =__ITCM_INIT
            cmp  r0, 0
```

```
/* Skip if __TCM_INIT is not set */
```

```
    beq ITCM_LOOP_END
```

```
/* Enable TCM */
```

```
    LDR r1, =CM7_ITCMCR
            LDR r0, [r1]
```



© 2026 NXP B.V. All rights reserved.

```
LDR r2, =0x1
     ORR r0, r2
STR r0, [r1]
     ldr r1, =__INT_ITCM_START
             ldr r2, =__INT_ITCM_END
             subs    r2, r1
             subs    r2, #1
             ble ITCM_LOOP_END
             movs    r0, 0
             movs    r3, 0
             ITCM_LOOP:
             stm r1!, {r0,r3}
             subs r2, #8
             bge ITCM_LOOP
             ITCM_LOOP_END:
```

## 12.8 Wait for debugger to hold the core

In some situations the debugger wants to hold the core, the debugger can write a fixed value 0x5A5A5A5A to the variable RESET_CATCH_CORE defined in the file ../Project_Settings/Startup_Code/system.c.

```
DebuggerHeldCoreLoop:
  ldr  r0, =RESET_CATCH_CORE
  ldr  r0, [r0]
  ldr  r1, =0x5A5A5A5A
  cmp  r0, r1
  beq  DebuggerHeldCoreLoop
```

Figure 20. Example code: Debugger held core loop

## 12.9 RAM data copy

This step copies the data which is in the RAM but stored in flash primarily to its RAM location. It includes

1. Copy the vector table from ROM to RAM.
2. Copy initialized data(.data) from ROM to RAM.
3. Copy code that should reside in RAM from ROM.
4. Clear the zero-initialized data(.bss) section.



© 2026 NXP B.V. All rights reserved.

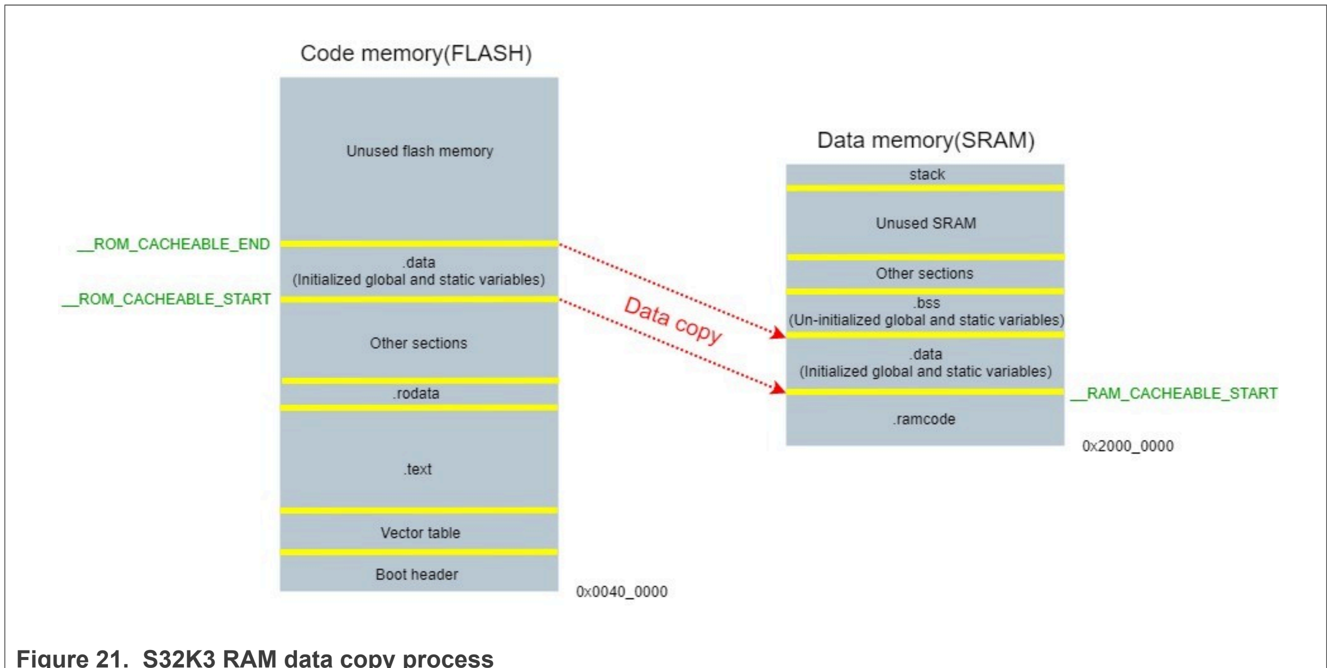


**Figure 21. S32K3 RAM data copy process**

There is a copy table defined in the startup_cmy.s file. There are 4 sections defined by default. Each section has 3 symbols: RAM_START, ROM_START, ROM_END. The addresses of the symbols are obtained in the linker file and are referred in the function `init_data_bss()` to do the copy work.

```
76 .section ".init_table", "a"
77   .long 4
78   .long __RAM_CACHEABLE_START
79   .long __ROM_CACHEABLE_START
80   .long __ROM_CACHEABLE_END
81   .long __RAM_NO_CACHEABLE_START
82   .long __ROM_NO_CACHEABLE_START
83   .long __ROM_NO_CACHEABLE_END
84   .long __RAM_SHAREABLE_START
85   .long __ROM_SHAREABLE_START
86   .long __ROM_SHAREABLE_END
87   .long __RAM_INTERRUPT_START
88   .long __ROM_INTERRUPT_START
89   .long __ROM_INTERRUPT_END
```

**Figure 22. Copy table with four sections**

For the variables in the bss section, due to no initialized value is stored in the flash, thus each section has only 2 symbols, RAM_START and RAM_END. The function `init_data_bss()` initializes these sections by writing zero.


© 2026 NXP B.V. All rights reserved.

```
90 .section ".zero_table", "a"
91   .long 3
92   .long __BSS_SRAM_SH_START
93   .long __BSS_SRAM_SH_END
94   .long __BSS_SRAM_NC_START
95   .long __BSS_SRAM_NC_END
96   .long __BSS_SRAM_START
97   .long __BSS_SRAM_END
```

**Figure 23. Copy table with three sections**

Code:

```
_INIT_DATA_BSS:
  bl init_data_bss
```

**Figure 24. init_data_bss function**

The implementation of the function `init_data_bss()` is in the file ../Project_Settings/Startup_Code/startup.c.

## 12.10 System initialization

This step does some system initialization. It includes

1. Configure MSCM to enable/disable interrupts routing to Core processor.
2. Enable FPU.
3. Enable MPU.
4. Enable I/D Cache.

These functions can be enabled/disabled by adding/removing the MACROs defined in Project/Property/C/C++ Build/Settings/Standard S32DS C Complier/Preprocessor/Defined symbols.

These MACROs includes

I_CACHE_ENABLE,

D_CACHE_ENABLE,

ENABLE_FPU,

MPU_ENABLE.

To enable FPU, the user also needs to make sure the option Project/Property/C/C++ Build/Settings/Target Processor/Float ABI is FP instructions(hard). It tells the compiler to compile the floating point calculation code by dedicated FPU instructions instead of ARM eabi library.

```
__SYSTEM_INIT:
  bl SystemInit
```

**Figure 25. Code example: SystemInit() function**

The implementation of the function `SystemInit`() is in the file ../Project_Settings/Startup_Code /system.c.



© 2026 NXP B.V. All rights reserved.

### 12.11 Jump to main routine

This is the last step of the startup code. It enables the CPU global interrupt, switches to user mode if it's enabled by Autosar, and jumps to the main function finally.

Code:

```
_MAIN:
  cpsie i
  bl startup_go_to_user_mode
  bl main
```

**Figure 26. Code example: Main routine**

## 13 Examples

### 13.1 Relocating code in RAM

Sometimes it is required to relocate the function to RAM for faster execution or other reason. In the S32K3 default linker file, there is a `.ramcode` section defined already, simply use the keyword **`__attribute__`** to relocate the function into this section. The same keyword must be added to the declaration of the function too.

Code Example:

```
void __attribute__ ((section(".ramcode"))) LED_function()
{
uint32_t i,j;
    Siul2_Dio_Ip_TogglePins(LED_GREEN_PORT,1<<LED_GREEN_PIN);
for(i=0;i<4000;i++)
for(j=0;j<4000;j++)
;
}
```

We can see from the map file the LED_function() is at the RAM address 0x2040_80dc.

```
.sram_data      0x20408000      0x1b8 load address 0x00412150
                0x20408000                . = ALIGN (0x4)
                0x20408000                __sram_data_begin__ = .
                0x20408000                . = ALIGN (0x4)
 *(.ramcode)
 .ramcode       0x20408000       0xdc ./RTD/src/Clock_Ip_Specific.o
.ramcode 0x204080dc 0x44 ./src/main.o
0x204080dc LED_function
```

### 13.2 Relocating code in ITCM

ITCM(Instruction Tightly Coupled Memory) is zero wait memory, the time of CPU accessing ITCM will be faster than accessing flash or SRAM. To use the ITCM memory, below steps are needed.



© 2026 NXP B.V. All rights reserved.

1. Define the ITCM section in the linker file. Since the default linker file has defined the ITCM memory region `int_itcm`, we can start with the section definition.

```
__itcm_rom = __shareable_data_rom_end; /*obtain the LMA of .itcm section,
 the .itcm section is put after .shareable_bss section*/
.itcm : AT(__itcm_rom)
{
    . = ALIGN(4);
    __itcm_start = .;
    *(.itcm_code)
    . = ALIGN(4);
    __itcm_end = .;
} > int_itcm
__itcm_rom_end = __itcm_rom + (__itcm_end - __itcm_start);
```

The default linker file does not include the ARM EABI GNU reserved section placement, user should add them and place them into Flash to avoid **TCM**(itcm/dtcm) or data section placement overlaps.

```
    .flash :
    {
```

...

```
    } > int_flash
    ARM.exidx :
{
```

```
*(.ARM.exidx*)
*(.gnu.linkonce.armexidx.*)
*(.glue*)
*(.vfp11*)
*(.v4*)
*(.iplt*)
*(.rel*)
```

```
} > int_flash
```

Without adding the `ARM.exidx` section, you will see the below linking error.

1. Obtain the addresses for data copy from flash to ITCM in the linker file.

```
__RAM_ITCM_START    = __itcm_start;
__ROM_ITCM_START    = __itcm_rom;
__ROM_ITCM_END      = __itcm_rom_end;
```

The 3 new symbols(`__RAM_ITCM_START`, `__ROM_ITCM_START`, `__ROM_ITCM_END`) are defined in the init_table, change the total elements from 4 to 5.

```
.section ".init_table", "a"
```

```
.long 5
```

 © 2026 NXP B.V. All rights reserved.

```
.long __RAM_CACHEABLE_START
.long __ROM_CACHEABLE_START
.long __ROM_CACHEABLE_END
.long __RAM_NO_CACHEABLE_START
.long __ROM_NO_CACHEABLE_START
.long __ROM_NO_CACHEABLE_END
.long __RAM_SHAREABLE_START
.long __ROM_SHAREABLE_START
.long __ROM_SHAREABLE_END
.long __RAM_INTERRUPT_START
.long __ROM_INTERRUPT_START
.long __ROM_INTERRUPT_END
.long __RAM_ITCM_START
.long __ROM_ITCM_START
```

```
.long __ROM_ITCM_END
```

Finally, add `__attribute__ ((section(".itcm_code")))` to the function which you wish to be relocated in the ITCM.

```
/*@brief: This is the ISR of SIUL IRQ1, source 08-15*/
void __attribute__ ((section(".itcm_code"))) SIUL_1_Handler()
{
```

```
/* Clear pending interrupt flag */
```

```
    (Siul2_Icu_Ip_pBase[0])->DISR0 = 0xFFFFFFFF;/*Clear all the interrupt
 pending bit*/
```

```
Siul2_Dio_Ip_TogglePins(LED_RED_PORT,1<<LED_RED_PIN);
printf("ISR occurred %d times.\n",++g_counterInIsrStandbyRam);
__asm("isb ");
```

```
}
```

Now we can see from the map file the function `SIUL_1_Handler()` is located in the ITCM address 0x0000_0000.

```
.itcm           0x00000000       0x44 load address 0x0041230c
                0x00000000                . = ALIGN (0x4)
                0x00000000                __itcm_start = .
 *(.itcm_code)
```

```
.itcm_code 0x00000000 0x44 ./src/main.o
```


All information provided in this document is subject to legal disclaimers.

© 2026 NXP B.V. All rights reserved.

**0x00000000 SIUL_1_Handler**

```
                 0x00000044                 . = ALIGN (0x4)
                 0x00000044                 __itcm_end = .
                 0x00412350                 __itcm_rom_end = (__itcm_rom +
  (__itcm_end - __itcm_start))
```

## 13.3 Relocating data in DTCM

DTCM(Data Tightly Coupled Memory) is similar to ITCM but is dedicated for data. Below shows the steps needed for using it.

1. Define the DTCM section in the linker file. Since there are 2 kinds of data types .data(with initialized value) and .bss(without initialized value). We created 2 sections .dtcm_data and .dtcm_bss in DTCM.

```
    __dtcm_rom = __itcm_rom_end; /*obtain the LMA of .dtcm_data section,
  the .dtcm_data section is put after .itcm section*/
    .dtcm_data : AT(__dtcm_rom)
    {
        . = ALIGN(4);
        __dtcm_data_start = .;
        *(.dtcm_data)
        . = ALIGN(4);
        __dtcm_data_end = .;
    } > int_dtcm
    __dtcm_rom_end = __dtcm_rom + (__dtcm_data_end - __dtcm_data_start);
    .dtcm_bss (NOLOAD) :
    {
        . = ALIGN(4);
        __dtcm_bss_start = .;
        *(.dtcm_bss)
        . = ALIGN(4);
        __dtcm_bss_end = .;
    } > int_dtcm
```

1. Obtain the boundary addresses for data initialization in the linker file.

```
    __RAM_DTCM_START   = __dtcm_data_start;
    __ROM_DTCM_START   = __dtcm_rom;
    __ROM_DTCM_END     = __dtcm_rom_end;
    __BSS_DTCM_START         = __dtcm_bss_start;
    __BSS_DTCM_END           = __dtcm_bss_end;
    __BSS_DTCM_SIZE          = __dtcm_bss_end - __dtcm_bss_start;
```

The 3 new symbols (`__RAM_DTCM_START`, `__ROM_DTCM_START`, `__ROM_DTCM_END`) are defined in the init_table, change the total elements from 5 to 6 since we have created the ITCM section. The 2 symbols(`__BSS_DTCM_START`, `__BSS_DTCM_END`) are defined in the zero_table, change the total elements from 3 to 4.

```
 .section ".init_table", "a"
```

```
.long 6
.long __RAM_CACHEABLE_START
.long __ROM_CACHEABLE_START
.long __ROM_CACHEABLE_END
```



© 2026 NXP B.V. All rights reserved.

```
.long __RAM_NO_CACHEABLE_START
.long __ROM_NO_CACHEABLE_START
.long __ROM_NO_CACHEABLE_END
.long __RAM_SHAREABLE_START
.long __ROM_SHAREABLE_START
.long __ROM_SHAREABLE_END
.long __RAM_INTERRUPT_START
.long __ROM_INTERRUPT_START
.long __ROM_INTERRUPT_END
.long __RAM_ITCM_START
.long __ROM_ITCM_START
.long __ROM_ITCM_END
```

```
   .long __RAM_DTCM_START
   .long __ROM_DTCM_START
 .long __ROM_DTCM_END
 .section ".zero_table", "a"
```

```
.long 4
.long __BSS_SRAM_SH_START
.long __BSS_SRAM_SH_END
.long __BSS_SRAM_NC_START
.long __BSS_SRAM_NC_END
.long __BSS_SRAM_START
.long __BSS_SRAM_END
```

```
   .long __BSS_DTCM_START
   .long __BSS_DTCM_END
```

Last, use key word **`__attribute__`** to relocate the data into the DTCM.

```
uint32_t __attribute__ ((section(".dtcm_bss"))) g_counterDtcmBss;
uint32_t __attribute__ ((section(".dtcm_data"))) g_counterDtcmData = 20;
```

Now we can see from the map file the 2 variables are in the DTCM .bss section and .data section respectively.

```
.dtcm_data       0x20000408        0x4 load address 0x00412758
                 0x20000408                . = ALIGN (0x4)
                 0x20000408                __dtcm_data_start = .
 *(.dtcm_data)
 .dtcm_data      0x20000408        0x4 ./src/main.o
```

**`0x20000408 g_counterDtcmData`**

```
                 0x2000040c                . = ALIGN (0x4)
                 0x2000040c                __dtcm_data_end = .
```



© 2026 NXP B.V. All rights reserved.

```
                0x0041275c                __dtcm_rom_end = (__dtcm_rom +
 (__dtcm_data_end - __dtcm_data_start))
.dtcm_bss       0x2000040c        0x4 load address 0x0041275c
                0x2000040c                . = ALIGN (0x4)
                0x2000040c                __dtcm_bss_start = .
 *(.dtcm_bss)
 .dtcm_bss      0x2000040c        0x4 ./src/main.o
```

**0x2000040c g_counterDtcmBss**

```
                0x20000410                . = ALIGN (0x4)
                0x20000410                __dtcm_bss_end = .
```

## 13.4 Relocating data in standby RAM

Only the data in the standby RAM can be retained during standby mode while most of the other RAM will be powered off. There is 32KB standby RAM for S32K344. Below shows the steps to enable it.

1. The standby RAM is at the beginning of SRAM, the default linker uses it as normal SRAM, we need to create a separate memory region for it.

The default SRAM memory region definition is below:

```
int_sram : ORIGIN = 0x20400000, LENGTH = 0x0002DF00    /* 183.9K */
```

The memory region definition after extracting the standby RAM.

```
int_standbysram : ORIGIN = 0x20400000, LENGTH = 0X00008000/* standby ram 32KB*/
int_sram : ORIGIN = 0x20400000 + 0x8000, LENGTH = 0x0002DF00 - 0x8000/* 183.9K
 -0x8000*/
```

1. Define the standby RAM section. To make the code easier we created a .bss section only. If user wishes to create a .data section for standby RAM, please refer to how it's done in the DTCM.

```
    .sram_standby (NOLOAD):
    {
```

```
. += ALIGN(4);
__standby_bss_start = .;
*(.sram_standby_bss)
__standby_bss_end = .;
```

```
} > int_standbysram
```

1. Define 2 symbols(`__BSS_STANDBY_SRAM_START`, `__BSS_STANDBY_SRAM_END`) in the zero table for data initialization and increase the total elements from 4 to 5.

```
.section ".zero_table", "a"
```

```
.long 5
.long __BSS_SRAM_SH_START
.long __BSS_SRAM_SH_END
```



© 2026 NXP B.V. All rights reserved.

```
.long __BSS_SRAM_NC_START
.long __BSS_SRAM_NC_END
.long __BSS_SRAM_START
.long __BSS_SRAM_END
.long __BSS_DTCM_START
.long __BSS_DTCM_END
.long __BSS_STANDBY_SRAM_START
```

```
.long __BSS_STANDBY_SRAM_END
```

And obtain the boundary addresses of the .bss section for clearing zero table.

```
    __BSS_STANDBY_SRAM_START = __standby_bss_start;
__BSS_STANDBY_SRAM_END = __standby_bss_end;
```

Obtain the boundary addresses for the whole standby RAM for ECC data initialization.

```
    __STANDBY_SRAM_START       = ORIGIN(int_standbysram);
__STANDBY_SRAM_SIZE        = LENGTH(int_standbysram);
```

1. We have known that all the RAM needs to be ECC initialized before using. But for the data in standby RAM, we need to skip it in case of standby exit to retain the data. This is the standby RAM ECC initialization function being added in the file startup.c.

```
static void standbyRamEccInit(void)
{
```

```
register uint32_t Cnt;
register long long *pDest;
if (IP_MC_RGM->DES & MC_RGM_DES_F_POR_MASK)/*init the standbyRAM only when it's
power on Reset*/
{
/* Standby SRAM ECC init */
Cnt = (uint32)(&__STANDBY_SRAM_SIZE) / 8;
pDest = ( long long *)(&__STANDBY_SRAM_START);
while (Cnt--)
{
*pDest = ( long long)0;
pDest++;
}
IP_MC_RGM->DES = MC_RGM_DES_F_POR_MASK; /* Write 1 to clear F_POR */
}
```

```
}
```



© 2026 NXP B.V. All rights reserved.

In the data copy function **init_data_bss()**, we need to do the same skip for standby exit. Since we only defined the .bss section in the standby RAM, we skip the zeroTable only. If user defined .data section, the same skip is needed for the initTable. And we put the **standbyRamEccInit()** function before the data copy process.

```
extern uint32_t __STANDBY_SRAM_START;
extern uint32_t __STANDBY_SRAM_SIZE;
extern uint32_t __BSS_STANDBY_SRAM_START;
void init_data_bss(void);
static void standbyRamEccInit(void);
void init_data_bss(void)
{
```

```
const Sys_CopyLayoutType * copy_layout;
const Sys_ZeroLayoutType * zero_layout;
const uint8 * rom;
uint8 * ram;
uint32 len = 0U;
uint32 size = 0U;
uint32 i = 0U;
uint32 j = 0U;
const uint32 * initTable_Ptr = (uint32 *)__INIT_TABLE;
const uint32 * zeroTable_Ptr = (uint32*)__ZERO_TABLE;
/*Add the standbyRam ECC init here. If user want to define the .data segment in the standbyRam, need to skip the data copy when it is recovered from the standby mode.*/
standbyRamEccInit();
/* Copy initialized table */
```

```
    len = *initTable_Ptr;
    initTable_Ptr++;
    copy_layout = (const Sys_CopyLayoutType *)initTable_Ptr;
```

```
for(i = 0; i < len; i++)
```

```
    {
        rom = copy_layout[i].rom_start;
        ram = copy_layout[i].ram_start;
        size = (uint32)copy_layout[i].rom_end -
  (uint32)copy_layout[i].rom_start;
```

```
for(j = 0UL; j < size; j++)
```

```
        {
            ram[j] = rom[j];
        }
    }
```

```
/* Clear zero table */
```



© 2026 NXP B.V. All rights reserved.

```
len = *zeroTable_Ptr;
     zeroTable_Ptr++;
zero_layout = (const Sys_ZeroLayoutType *)zeroTable_Ptr;
for(i = 0; i < len; i++)
     {
/*bypass the data initialization for standbyRAM if it's not power on reset.*/
if((IP_MC_RGM->DES__BSS_STANDBY_SRAM_START != zero_layout[i].ram_start))
{
ram = zero_layout[i].ram_start;
size = (uint32)zero_layout[i].ram_end - (uint32)zero_layout[i].ram_start;
for(j = 0UL; j < size; j++)
{
ram[j] = 0U;
}
}
     }
 }
```

Finally,use key word **`__attribute__`** to put the data into the standby RAM.

```
uint32_t __attribute__ ((section(".sram_standby_bss"))) g_counterStandbyRam;
uint32_t __attribute__ ((section(".sram_standby_bss")))
 g_counterInIsrStandbyRam;
```

In the map file we can see the 2 variables are in the standby RAM.

```
.sram_standby   0x20400000        0x8
                0x00000000                . = (. + ALIGN (0x4))
                0x20400000                __standby_bss_start = .
 *(.sram_standby_bss)
 .sram_standby_bss
                0x20400000        0x8 ./src/main.o
                0x20400000                g_counterStandbyRam
                0x20400004                g_counterInIsrStandbyRam
                0x20400008                __standby_bss_end = .
```

***Note:*** *The D-cache need to be disabled to make the standby RAM work, or you can set this memory region as non-cacheable in the MPU settings.*

## 13.5 Relocating data in data flash

There are 128KB data flash in S32K344 and is not defined by the default linker file. We know from the HSE RM that the HSE firmware used the last 40KB, so we have 88KB that can be used by the application core. For using it,


All information provided in this document is subject to legal disclaimers.

© 2026 NXP B.V. All rights reserved.

1. we need to create the memory region as always.

```
int_data_flash : ORIGIN = 0x10000000, LENGTH = 0x16000 /* 128-40 =88KB data
 flash,HSE used the last 40KB*/
```

2.Define the .data_flash section

```
    .data_flash :
    {
        . = ALIGN(4);
        *(.DFlash)
    } > int_data_flash
```

3.Use the key word **`__attribute__`** to relocate the variable in this section.

```
uint8_t __attribute__ ((section(".DFlash"))) g_msg[] = "Hello,S32K3";
```

Now we can see from the map file the variable is in the data flash.

```
.data_flash      0x10000000         0xc
                 0x10000000                 . = ALIGN (0x4)
 *(.DFlash)
 .DFlash         0x10000000         0xc ./src/main.o
```

**`0x10000000 g_msg`**

## 13.6 Linking a binary file

There are some situations we need to link an existing binary file to the current project. Such as the firmware of the HSE core, the firmware of other MCU. Below shows the step.

1. Specify the memory region for storing the binary file.

```
sja1110_BINARY (R)      : ORIGIN = 0x00500000, LENGTH = 0x100000 /
```

1. Specify the file format of the binary file outside of the MEMORY region and SECTIONS region.

```
TARGET(binary) /* specify the file format of binary file */
INPUT (..\SJA1110_bin_file\flash_image.bin)/*file path can be either absolute or
 relative*/
OUTPUT_FORMAT(default) /* restore the out file format */
```

1. Define the section.

```
.sja1110_bin :
 {
```

```
. = ALIGN (0x4);
__sja1110_bin_start__ = .;
..\SJA1110_bin_file\flash_image.bin (.data)
. = ALIGN (0x4);
```



© 2026 NXP B.V. All rights reserved.

```
__sja1110_bin_end__ = .;
```

```
 } > sja1110_BINARY
__sja1110_BIN_START = ORIGIN(sja1110_BINARY);
__sja1110_BIN_SIZE = __sja1110_bin_end__ - __sja1110_bin_start__;
```

Though the map file we can see a binary file of length 0xa6860 is stored at the flash address 0x0050_0000.

```
TARGET(binary)
LOAD ..\SJA1110_bin_file\flash_image.bin
.sja1110_bin     0x00500000    0xa6860
                 0x00500000                . = ALIGN (0x4)
                 0x00500000                __sja1110_bin_start__ = .
 ..\SJA1110_bin_file\flash_image.bin(.data)
 .data           0x00500000    0xa6860 ..\SJA1110_bin_file\flash_image.bin
```

```
0x00500000 _binary____SJA1110_bin_file_flash_image_bin_start
```

```
                 0x005a6860
 _binary____SJA1110_bin_file_flash_image_bin_end
                 0x005a6860                . = ALIGN (0x4)
                 0x005a6860                __sja1110_bin_end__ = .
                 0x00500000                __sja1110_BIN_START = ORIGIN
 (sja1110_BINARY)
```

```
0x000a6860 __sja1110_BIN_SIZE = (__sja1110_bin_end__ - __sja1110_bin_start__)
```

## 13.7 Relocating the stack in DTCM

For better MCU performance, sometimes the stack can be relocated to DTCM from SRAM.

1. The default DTCM and stack memory region are defined as below.

```
int_dtcm : ORIGIN = 0x20000000, LENGTH = 0x00020000    /* 64K */
int_sram_stack_c0 : ORIGIN = 0x2042E000, LENGTH = 0x00001000    /* 4KB  */
```

We need to redefine the new stack region from the DTCM.

```
int_dtcm       : ORIGIN = 0x20000000, LENGTH = 0x00020000 - 0x1000    /* 64K -
 0x1000*/
int_stack_dtcm : ORIGIN = 0x20020000-0x1000, LENGTH = 0x1000/*Set last 4KB DTCM
 as stack*/
```

1. Change the symbols from the default SRAM address to the DTCM address. The `__Stack_start_c0` will be assigned to the MSP in the startup code.

Old:

```
__Stack_end_c0   = ORIGIN(int_sram_stack_c0);
__Stack_start_c0 = ORIGIN(int_sram_stack_c0) + LENGTH(int_sram_stack_c0);
```

New:

```
__Stack_end_c0   = ORIGIN(int_stack_dtcm);
__Stack_start_c0 = ORIGIN(int_stack_dtcm) + LENGTH(int_stack_dtcm);
```

 © 2026 NXP B.V. All rights reserved.

Last, we need to update the boundary for the DTCM end address `__INT_DTCM_END` which is used for ECC initialization.

Old:

```
__INT_DTCM_START = ORIGIN(int_dtcm);
__INT_DTCM_END   = ORIGIN(int_dtcm) + LENGTH(int_dtcm);
```

New:

```
__INT_DTCM_START = ORIGIN(int_dtcm);
__INT_DTCM_END   = ORIGIN(int_dtcm) + LENGTH(int_dtcm) + LENGTH(int_stack_dtcm);
```

Now we can see from the map file the stack start address is the end address of the DTCM.

```
0x2001f000           __Stack_end_c0 = ORIGIN (int_stack_dtcm)
0x20020000           __Stack_start_c0 = (ORIGIN (int_stack_dtcm) + LENGTH
 (int_stack_dtcm))
```

## 13.8 Relocating the vector table in DTCM

For quicker ISR response, the vector table can also be relocated to the DTCM instead of SRAM. Simply move the vector table section from the SRAM to the DTCM. Note that the start address of the vector table must be 4096 bytes aligned.

Old:

```
    .non_cacheable_data : AT(__non_cacheable_data_rom)
    {
        . = ALIGN(4);
```

```
__non_cacheable_data_start__ = .;
```

```
/*the vector table start*/
        . = ALIGN(4096);
        __interrupts_ram_start = .;
        . += __interrupts_rom_end - __interrupts_rom_start;
        . = ALIGN(4);
        __interrupts_ram_end = .;
/*the vector table end*/
        *(.mcal_data_no_cacheable)
        . = ALIGN(4);
        *(.mcal_const_no_cacheable)
        . = ALIGN(4);
        HSE_LOOP_ADDR = .;
```

```
LONG(0x0);
```

```
        __non_cacheable_data_end__ = .;
} > int_sram_no_cacheable
```

New:

```
    .dtcm_data : AT(__dtcm_rom)
    {
        . = ALIGN(4);
```

 © 2026 NXP B.V. All rights reserved.

```
        __dtcm_data_start = .;
```

/*the vector table start*/

```
        . = ALIGN(4096);
        __interrupts_ram_start = .;
        . += __interrupts_rom_end - __interrupts_rom_start;
        . = ALIGN(4);
        __interrupts_ram_end = .;
```

/*the vector table end*/

. = **ALIGN**(4);

```
        *(.dtcm_data)
        . = ALIGN(4);
        __dtcm_data_end = .;
 } > int_dtcm
```

We can see from the address the vector table are in the DTCM.

```
.dtcm_data       0x20000000      0x40c load address 0x00412378
                 0x20000000                . = ALIGN (0x4)
                 0x20000000                __dtcm_data_start = .
                 0x20000000                . = ALIGN (0x1000)
```

**0x20000000 __interrupts_ram_start = .**

```
                 0x20000408                . = (. + (__interrupts_rom_end -
  __interrupts_rom_start))
 *fill*          0x20000000      0x408
                 0x20000408                . = ALIGN (0x4)
```

**0x20000408 __interrupts_ram_end = .**

## 13.9 #pragma GCC section

When we have a large amount of data or functions to be relocated to a specific section, we don't want to add **`__attribute__`** to the variables one by one, the "**`#pragma`** GCC section" can be used to relocate multiple variables/functions in one section at one time.

The typical format is below:

```
#pragma
Variables/functions definition
#pragma
```

This example shows relocating multiple variables with initial values in the DTCM.

```
#pragma GCC section data ".dtcm_data"
uint32_t g_number1 = 100;
uint32_t g_number2 = 200;
uint32_t g_number3 = 300;
uint32_t g_number4 = 400;
uint32_t g_number5 = 500;
uint32_t g_number6 = 600;
```



© 2026 NXP B.V. All rights reserved.

```
#pragma GCC section data "default"
```

We can see from the map file these variables are in the DTCM.

```
*(.dtcm_data)
 .dtcm_data     0x20000408       0x18 ./src/User.o
                0x20000408                g_number1
                0x2000040c                g_number2
                0x20000410                g_number3
                0x20000414                g_number4
                0x20000418                g_number5
                0x2000041c                g_number6
```

This examples shows relocating multiple functions in the SRAM.

```
#pragma GCC section text ".ramcode"
/*@brief: This function toggles the blue LED every 1 second.*/
void LED_blue_function()
{
```

```
uint32_t i,j;
```

```
    Siul2_Dio_Ip_TogglePins(LED_BLUE_PORT,1<<LED_BLUE_PIN);
```

```
for(i=0;i<4000;i++)

for(j=0;j<4000;j++)

;
```

```
}
/*@brief: This function toggles the red LED every 1 second.*/
void LED_red_function()
{
```

```
uint32_t i,j;
```

```
    Siul2_Dio_Ip_TogglePins(LED_RED_PORT,1<<LED_RED_PIN);
```

```
for(i=0;i<4000;i++)

for(j=0;j<4000;j++)

;
```

```
}
void delay_function()
{
```

```
uint32_t i,j;

for(i=0;i<4000;i++)

for(j=0;j<4000;j++)

;
```

```
}
```


All information provided in this document is subject to legal disclaimers.

© 2026 NXP B.V. All rights reserved.

```
#pragma GCC section text "default"
```

The same **#pragma** directive need to be added to the function declaration.

```
#pragma GCC section text ".ramcode"
void LED_blue_function(void);
void LED_red_function(void);
void delay_function(void);
#pragma GCC section text "default"
```

We can see from the map file these functions are in the SRAM.

```
.ramcode        0x204080dc       0xbc ./src/User.o
                0x204080dc                LED_blue_function
                0x20408120                LED_red_function
```

**0x20408164 delay_function**

## 14 Others

For DMA transmitting, the user must use the variables from the non-cacheable section instead of the cacheable section, otherwise, the DMA may not get the correct data in the end address.

One memory region can contain several sections, but one section can't be distributed to several non-contiguous memory regions.

There is tiny difference between the GNU linker script and other platforms such as IAR and GreenHills, but the whole startup procedure is the same.

There is no functional restriction that code and data are placed in either the DTCM or ITCM. But best performance is achieved if code is placed in ITCM and data in DTCM.

For more knowledge of the GNU linker script, please refer to the file <ld.pdf> in the S32DS installation location C:\NXP\S32DS.3.4\S32DS\build_tools\gcc_v10.2\gcc-10.2-arm32-eabi\arm-none-eabi\share\docs\pdf\ld.pdf

## 15 Note about the source code in the document

The example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2026 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN

CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 16 Revision history

**Table 1. Revision history**

| Document ID | Revision date | Description |
|---|---|---|
| AN14893 v. 1.0 | 28 December 2025 | Initial release |

# Legal information

## Definitions

**Draft** — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

## Disclaimers

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

**Right to make changes** — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Terms and conditions of commercial sale** — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Suitability for use in non-automotive qualified products** — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

**HTML publications** — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

**Translations** — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Security** — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

**NXP B.V.** — NXP B.V. is not an operating company and it does not distribute or sell products.

## Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

**NXP** — wordmark and logo are trademarks of NXP B.V.



© 2026 NXP B.V. All rights reserved.

## Tables

Tab. 1. Revision history ............................................. 32

## Figures

Fig. 1. The linking process .......................................... 2
Fig. 2. Syntax: MEMORY command ........................... 3
Fig. 3. Syntax: SECTIONS command .......................... 4
Fig. 4. Code example: Wildcard character (*) ............. 4
Fig. 5. Code example: Location counter (.) .................. 5
Fig. 6. Code example 1 ............................................... 5
Fig. 7. Code example 2 ............................................... 5
Fig. 8. Code example: AT command ........................... 5
Fig. 9. Code example: KEEP command ....................... 6
Fig. 10. Code example: Linker script symbol ................ 6
Fig. 11. Code example: S32K3 linker file ....................... 7
Fig. 12. S32K3 Startup Procedure Overview ................. 8
Fig. 13. Boot header definition details ............................ 9
Fig. 14. Code example: Boot header ............................ 9
Fig. 15. Code example: disable global interrupt and clear CPU register .......................................... 10
Fig. 16. Code example: Enable MSCM clock code ...... 10
Fig. 17. Code example: Relocate vector table to RAM ............................................................... 11
Fig. 18. Example code block: Initialize CPU core stack .............................................................. 11
Fig. 19. Disable SWT watchdog .................................. 12
Fig. 20. Example code: Debugger held core loop ........ 14
Fig. 21. S32K3 RAM data copy process ...................... 15
Fig. 22. Copy table with four sections .......................... 15
Fig. 23. Copy table with three sections ........................ 16
Fig. 24. init_data_bss function ..................................... 16
Fig. 25. Code example: SystemInit() function ............... 16
Fig. 26. Code example: Main routine ........................... 17



© 2026 NXP B.V. All rights reserved.

## Contents

| | | |
|---|---|---|
| **1** | **Introduction** | **2** |
| **2** | **ENTRY command** | **2** |
| **3** | **MEMORY command** | **2** |
| **4** | **SECTIONS command** | **3** |
| **5** | **Wildcard character (*)** | **4** |
| **6** | **Location counter (.)** | **4** |
| **7** | **ALIGN command** | **5** |
| **8** | **AT command** | **5** |
| **9** | **KEEP command** | **6** |
| **10** | **Linker script symbol** | **6** |
| **11** | **Explanation of the S32K3 linker file** | **6** |
| **12** | **Startup Code Overview** | **7** |
| 12.1 | Boot header | 8 |
| 12.2 | Disable global interrupt and clear CPU register | 9 |
| 12.3 | Enable MSCM clock | 10 |
| 12.4 | Relocate vector table to RAM | 11 |
| 12.5 | Initialize CPU core stack | 11 |
| 12.6 | Disable SWT watchdog | 11 |
| 12.7 | RAM ECC initialization | 12 |
| 12.8 | Wait for debugger to hold the core | 14 |
| 12.9 | RAM data copy | 14 |
| 12.10 | System initialization | 16 |
| 12.11 | Jump to main routine | 17 |
| **13** | **Examples** | **17** |
| 13.1 | Relocating code in RAM | 17 |
| 13.2 | Relocating code in ITCM | 17 |
| 13.3 | Relocating data in DTCM | 20 |
| 13.4 | Relocating data in standby RAM | 22 |
| 13.5 | Relocating data in data flash | 25 |
| 13.6 | Linking a binary file | 26 |
| 13.7 | Relocating the stack in DTCM | 27 |
| 13.8 | Relocating the vector table in DTCM | 28 |
| 13.9 | #pragma GCC section | 29 |
| **14** | **Others** | **31** |
| **15** | **Note about the source code in the document** | **31** |
| **16** | **Revision history** | **32** |
| | **Legal information** | **33** |

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.

**© 2026 NXP B.V.** **All rights reserved.**