N32H7xx series software TCM user guide

Introduction

This document is designed for users to quickly understand the basic functions and usage of the TCM peripherals in N32H7xx series microcontrollers (MCUs), in order to guide users to easily use this product.

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

Content

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

1 TCM SRAM Memory Characteristics

CM SRAM memory is constructed with SRAM and includes the following types: TCM (ITCM, D0TCM, D1TCM) used as the tightly-coupled memory for the M7 core, and AXI SRAM (AXISRAM2/3) used as general-purpose on-chip SRAM. TCM memory can also be accessed via DMA through MDMA over the AHBS interface of the Cortex-M7 core.

Users can configure the sizes of ITCM and DTCM in 128KB granularity, with the remaining memory space automatically allocated to the two AXI SRAM2/3. Note that DTCM is divided into two parts: D0TCM is allocated to the lower 32-bit section of the 64-bit DTCM data bus, while D1TCM is allocated to the upper 32-bit section.

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

2 TCM SRAM Memory Mapping

Table 2-1 shows the memory mapping of code and RAM spaces. Memory regions are allocated accordingly based on the TCM SRAM configuration. ITCM and DTCM can be allocated up to 1MB in total according to the configuration. If no space is allocated to ITCM and DTCM, the entire 1MB of SRAM will be assigned to AXI SRAM2 (512KB) and AXI SRAM3 (512KB) equally.

Note: The addresses of AXI SRAM 1/2/3 are always contiguous. Any allocation for ITCM/DTCM will be deducted starting from the end address of AXI SRAM3, refer to Table 3-1 *for details.*

Table 2-1 Memory Mapping Table of TCM SRAM and Related Modules

Region	Start Address	End Address	Size	Cortex M7	Cortex M4
Peripheral (Register access)	0x5111_8000	0x5111_BFFF	16KB	OTPC	
	0x5110_4000	0x5110_4FFF	4KB	ECCMON1	
	0x5004_B000	0x5004_BFFF	4KB	TCMSRAMC	
RAM (WBWA)	0x240A_0000	0x2411_FFFF	512KB	AXI SRAM 3	
	0x2402_0000	0x2409_FFFF	512KB	AXI SRAM 2	
	0x2400_0000	0x2409_FFFF	128KB	AXI SRAM 1	
	0x2000_0000	0x200F_FFFF	1MB	DTCM	Reserved
Code (WT)	0x0000_0000	0x000F_FFFF	1MB	ITCM	INIT VTOR REMAP

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

3 TCM Controller

The TCM controller converts the TCM (64-bit ITCM bus or two 32-bit buses: D0TCM and D1TCM) interface to an SRAM interface. The TCM interface operates synchronously with the Cortex-M7 core at the same frequency. The TCM controller is responsible for allocating SRAM banks to ITCM, D0TCM, D1TCM, AXI SRAM2 and AXI SRAM3, and ensuring no bank conflicts occur between them. Based on the ITCMS and DTCMS settings, it calculates the size of each memory type and allocates the corresponding SRAM banks accordingly.

Any out-of-bounds address access to TCM memory, including incorrect size settings, will return an error response. In such cases, the memory content remains unchanged and the read data is undefined.

This controller supports the ECC function with Single-Error Correction and Double-Error Detection (SECDED), and features an error injection function. The enable signal for the ECC function is derived from the ECC monitor. If the ECC function of the corresponding interface is enabled, the TCM controller will output the ECC error type, address, data and the corresponding ECC code to the ECC monitor when an ECC error occurs.

Note: For more details on ECC processing, refer to section 2.4 ECCMON in EN_UM_N32H7xx Series User Manual.pdf.

3.1 TCM SRAM Size Configuration

The TCM SRAM memory size can be defined at system startup via OTP configuration. If the configuration is correct, the status bit (CSA) of TCMC_CTR will be set. Users must check this status bit before accessing the TCM SRAM memory. When CSA = L, any operation on TCM and AXI SRAM2/3 will result in unpredictable behavior.

3.1.1 OTP Configuration

Details of the size configuration for OTP using the TCM_SZ_CFG_OTP setting are shown below.

Table 3-1 OTP Configuration Table(For N32H76x and N32H78x series)

No.	TCM_SZ_CFG_OTP [5:0]	ITCM Size(KB)	DTCM Size(KB)	AXISRAM Size(KB)
1	6'h00	1024	0	128
2	6'h01	896	128	128
3	6'h02	768	256	128
4	6'h03	640	384	128
5	6'h04	512	512	128
6	6'h05	384	640	128
7	6'h06	256	768	128

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

8	6'h07	128	896	128
9	6'h08	0	1024	128
10	6'h09	896	0	256
11	6'h0A	768	128	256
12	6'h0B	640	256	256
13	6'h0C	512	384	256
14	6'h0D	384	512	256
15	6'h0E	256	640	256
16	6'h0F	128	768	256
17	6'h10	0	896	256
18	6'h11	768	0	384
19	6'h12	640	128	384
20	6'h13	512	256	384
21	6'h14	384	384	384
22	6'h15	256	512	384
23	6'h16	128	640	384
24	6'h17	0	768	384
25	6'h18	640	0	512
26	6'h19	512	128	512
27	6'h1A	384	256	512
28	6'h1B	256	384	512
29	6'h1C	128	512	512
30	6'h1D	0	640	512
31	6'h1E	512	0	640
32	6'h1F	384	128	640

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

33	6'h20	256	256	640
34	6'h21	128	384	640
35	6'h22	0	512	640
36	6'h23	384	0	768
37	6'h24	256	128	768
38	6'h25	128	256	768
39	6'h26	0	384	768
40	6'h27	256	0	896
41	6'h28	128	128	896
42	6'h29	0	256	896
43	6'h2A	128	0	1024
44	6'h2B	0	128	1024
45	6'h2c~3F	0	0	1152

Table 3-2 OTP Configuration Table(For N32H73x series)

No.	TCM_SZ_CFG_OTP [5:0]	ITCM Size(KB)	DTCM Size(KB)	AXISRAM Size(KB)
1	6'h1E	512	0	256
2	6'h1F	384	128	256
3	6'h20	256	256	256
4	6'h21	128	384	256
5	6'h22	0	512	256
6	6'h23	384	0	384
7	6'h24	256	128	384
8	6'h25	128	256	384
9	6'h26	0	384	384

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

10	6'h27	256	0	512
11	6'h28	128	128	512
12	6'h29	0	256	512
13	6'h2A	128	0	640
14	6'h2B	0	128	640
15	6'h2c~3F	0	0	768

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

4 Scatter Loading File

A scatter loading file is a configuration file used in embedded system development to describe the program memory layout, and is widely applied especially in the development of ARM architecture microcontrollers. By defining the specific locations and sizes of code, data, stack and heap in the memory, it helps developers optimize memory usage and ensures that the program can be correctly allocated to the specified memory regions during compilation and linking.

4.1 Map File

It is recommended to understand the map file first before learning about the scatter loading file. A map file is a detailed report file generated by the linker that describes the program memory layout.

4.1.1 Image Component Sizes

The Image Component Sizes section provides detailed size information for each component of the program image. Among them, Code (inc. data), RO Data, RW Data, ZI Data and Debug are common fields, which respectively represent the memory distribution of different parts of the program.

Figure 4-1 Map_Image Component Sizes 1

```
==============================================================================

Image component sizes

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

       232         18          0          4          0       1727   log.o
       618        304          0          4          0       4074   main.o
       132         22          0          0          0       4583   misc.o
       912         26          0          0          0      11536   n32h76x_78x_eccmon.o
       240          0          0          0          0      23618   n32h76x_78x_gpio.o
       272          4          0          0          0       5061   n32h76x_78x_it.o
      1616        140          0          0          0     116149   n32h76x_78x_rcc.o
       408         52          0          0          0      12354   n32h76x_78x_usart.o
        36          8       1000          0       8192       1016   startup_n32h76x.o
       116         18          0          0          0     876017   system_n32h76x_78x.o

    ----------------------------------------------------------------------
      4592        592       1032          8       8192    1056135   Object Totals
         0          0         32          0          0          0   (incl. Generated)
        10          0          0          0          0          0   (incl. Padding)

    ----------------------------------------------------------------------
```

Code (inc. data): Represents the size of the program's code segment, including the actual executable code of the program and the constant data embedded in the code (such as immediate values, jump tables, etc.), which is typically stored in Flash.

RO Data: Stands for Read-Only Data, referring to the constant data defined in the program, as well as global variables and string constants modified by const. This data will not be modified during program operation and can therefore be stored in Flash.

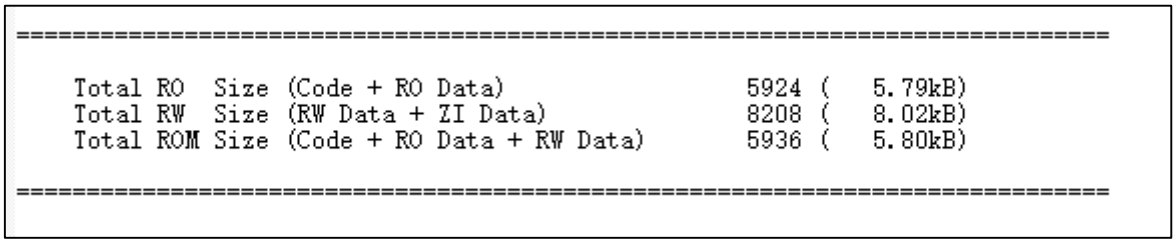
RW Data: Stands for Read-Write Data, including initialized global variables and static variables. This data is loaded into RAM when the program starts and can be modified during program operation. The initial values of RW Data are stored in Flash and copied to RAM at program startup.

ZI Data: Stands for Zero-Initialized Data, including uninitialized global variables and static variables. These variables are initialized to zero when the program starts, occupy no Flash space, and are only allocated in RAM.

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

Debug: Represents the space occupied by debugging information, which usually includes debug-related data such as debug symbols, line number information, and variable names. It does not affect program operation, is only used during the debugging and development phase, is not loaded onto the target device, and exists only in debug files.

Figure 4-2 Map_Image Component Sizes 2



English Translation

At the end of the Image Component Sizes section, the total memory footprint of the program in Flash (ROM) and RAM is presented. Total RO Size, Total RW Size, and Total ROM Size are key memory footprint statistics.

Total RO Size (Code + RO Data): Total size of the Read-Only section, calculated as Total RO Size = Code + RO Data, which is stored in Flash.

Total RW Size (RW Data + ZI Data): Total size of the Read-Write section, calculated as Total RW Size = RW Data + ZI Data, which is stored in RAM.

Total ROM Size (Code + RO Data + RW Data): Total size occupied by the program in Flash, calculated as Total ROM Size = Code + RO Data + RW Data, which is stored in Flash. If a bin file is generated after project compilation, the size of this bin file is equal to Total ROM Size. The initial values of RW Data are stored in Flash and copied to RAM during program startup, so RW occupies space in both FLASH and RAM.

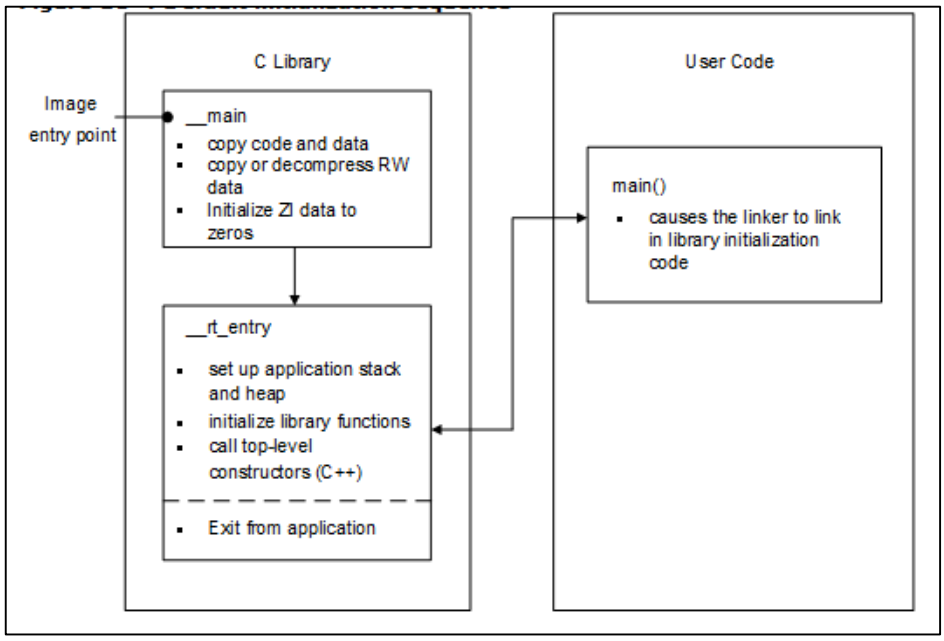
Who copies RW Data from Flash to RAM, and when?

Answer: This is implemented by the __main() function in the startup file.

__main and __rt_entry are auto-generated functions by the compiler, serving as key components in the program startup process. Their primary responsibility is to initialize the C runtime environment and ensure the program can correctly jump from the startup code to the user-defined main() function.

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

Figure 4-3 Default Initialization Flow



Main Functions of __main include:

- **Initialize RW Data**: Based on the definitions in the Scatter Loading File, locate the start address and size of RW Data in Flash, and copy the initialized global and static variables from Flash to RAM.
- **Zero out ZI Data**: Clear the RAM region where uninitialized global and static variables are located.

Main Functions of __rt_entry include:

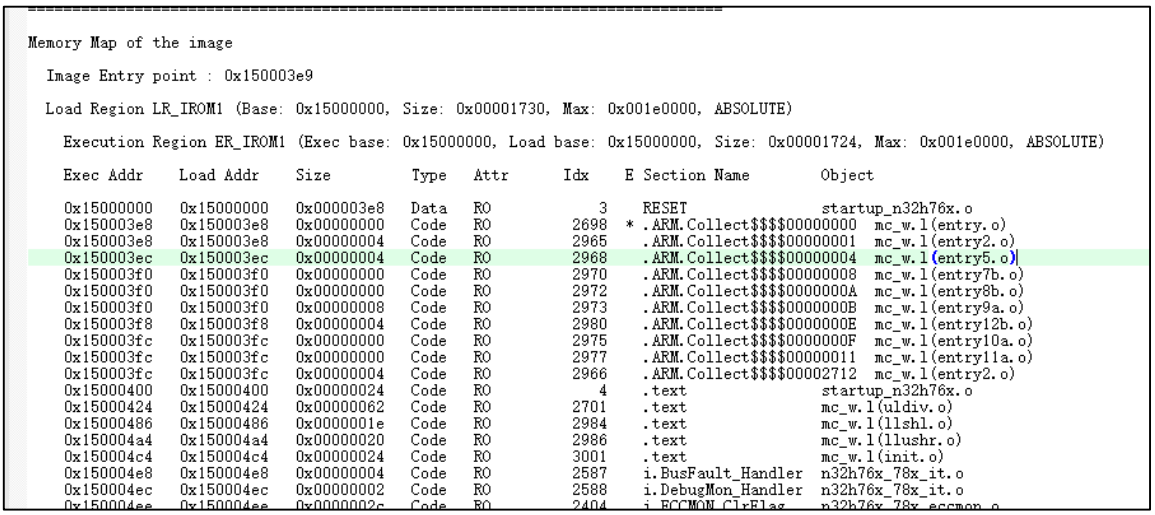
- **Initialize the heap and stack**: Set the initial states of the Heap and Stack.
- **Jump to the main() function**: After completing the initialization, jump to the user's main() function.

4.1.2 Memory Map of the Image

The Memory Map of the Image is a key section in the map file, which details the memory layout and allocation of the program image. By analyzing the Memory Map of the Image, developers can obtain the specific locations and sizes of various program components (such as code, data, heap and stack) in the memory, thereby optimizing memory usage and ensuring the correct operation of the program.

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

Figure 4-4 Memory Map of the Image



```
Memory Map of the image

  Image Entry point : 0x150003e9

  Load Region LR_IROM1 (Base: 0x15000000, Size: 0x00001730, Max: 0x001e0000, ABSOLUTE)

    Execution Region ER_IROM1 (Exec base: 0x15000000, Load base: 0x15000000, Size: 0x00001724, Max: 0x001e0000, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x15000000   0x15000000   0x000003e8   Data   RO            3    RESET               startup_n32h76x.o
    0x150003e8   0x150003e8   0x00000000   Code   RO         2698  * .ARM.Collect$$$$00000000  mc_w.l(entry.o)
    0x150003e8   0x150003e8   0x00000004   Code   RO         2965    .ARM.Collect$$$$00000001  mc_w.l(entry2.o)
    0x150003ec   0x150003ec   0x00000004   Code   RO         2968    .ARM.Collect$$$$00000004  mc_w.l(entry5.o)
    0x150003f0   0x150003f0   0x00000000   Code   RO         2970    .ARM.Collect$$$$00000008  mc_w.l(entry7b.o)
    0x150003f0   0x150003f0   0x00000000   Code   RO         2972    .ARM.Collect$$$$0000000A  mc_w.l(entry8b.o)
    0x150003f0   0x150003f0   0x00000008   Code   RO         2973    .ARM.Collect$$$$0000000B  mc_w.l(entry9a.o)
    0x150003f8   0x150003f8   0x00000004   Code   RO         2980    .ARM.Collect$$$$0000000E  mc_w.l(entry12b.o)
    0x150003fc   0x150003fc   0x00000000   Code   RO         2975    .ARM.Collect$$$$0000000F  mc_w.l(entry10a.o)
    0x150003fc   0x150003fc   0x00000000   Code   RO         2977    .ARM.Collect$$$$00000011  mc_w.l(entry11a.o)
    0x150003fc   0x150003fc   0x00000004   Code   RO         2966    .ARM.Collect$$$$00002712  mc_w.l(entry2.o)
    0x15000400   0x15000400   0x00000024   Code   RO            4    .text               startup_n32h76x.o
    0x15000424   0x15000424   0x00000062   Code   RO         2701    .text               mc_w.l(uldiv.o)
    0x15000486   0x15000486   0x0000001e   Code   RO         2984    .text               mc_w.l(llshl.o)
    0x150004a4   0x150004a4   0x00000020   Code   RO         2986    .text               mc_w.l(llushr.o)
    0x150004c4   0x150004c4   0x00000024   Code   RO         3001    .text               mc_w.l(init.o)
    0x150004e8   0x150004e8   0x00000004   Code   RO         2587    i.BusFault_Handler  n32h76x_78x_it.o
    0x150004ec   0x150004ec   0x00000002   Code   RO         2588    i.DebugMon_Handler  n32h76x_78x_it.o
    0x150004ee   0x150004ee   0x0000002c   Code   RO         2404    i.ECCMON_ClrFlag    n32h76x_78x_eccmon.o
```

Load Region (LR_IROM1):Represents the memory layout of the program during loading. It clearly specifies the base address, used size, maximum available size and attributes of the load region.

Execution Region (ER_IROM1):Represents the memory layout of the program during execution, including the base address, used size, maximum available size and attributes of the execution region.

When the execution region and the load region share the same base address, it means the code and data are located at the same address during both loading and execution. If their base addresses differ, the __main function is required to copy the code and data from the load address to the execution address.

4.2 Functions of the Scatter Loading File

The primary function of a Scatter Loading File is to provide the linker with detailed memory layout information, ensuring the program is correctly allocated to the specified memory regions during compilation and linking. Specifically, a Scatter Loading File can:

Define the base addresses and sizes of memory regions: By specifying the base addresses and sizes of memory regions, it ensures different parts of the program are correctly loaded to the designated memory regions.

Specify the distribution of code segments, data segments, stack and heap segments across different memory regions: It enables the allocation of code, data, stack and heap to separate memory regions for optimized memory usage.

Support the configuration of multiple memory regions: Adapted for complex embedded systems, it allows the configuration of multiple memory regions to ensure the rational allocation of system resources.

Optimize memory usage: Through rational configuration of the Scatter Loading File, developers can precisely control the program's memory layout and avoid memory waste.

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

4.3 Structure of the Scatter Loading File

A scatter loading file typically consists of multiple sections, each describing the configuration of memory regions and load regions. The following is the structure of a typical scatter loading file:

```
LR_ROM1 0x15000000 0x1E0000  {    ; Load region definition
  ER_ROM1 0x15000000 0x1E0000  {  ; Execution region definition
       *.o (RESET, +First)           ; Reset vector table
       *(InRoot$$Sections)           ; Initialization code of standard library
        .ANY (+RO)                   ; Read-only data (code and constants)
  }
  RW_RAM1 0x24000000 0x00020000  {  ; Read-write data region
       .ANY (+RW +ZI)                ; Read-write data and zero-initialized data
  }
}
```

4.3.1 Load Region

A load region defines the memory layout of a program during loading. Typically, a load region corresponds to Flash or other non-volatile memory. The start address and size of a load region are specified according to hardware configuration.

```
load_region_name  (base_address  | ("+" offset)) [attribute_list] [max_size]
{
      execution_region_description+
}
```

load_region_name： The name of the load region, which can be defined arbitrarily by the user.

The start address of a load region can be specified in one of the following two formats:

base_address: Indicates the start address of objects in this load region during linking; the address must be word-aligned.

+offset: Indicates the start address of objects in this load region during linking is at an offset of offset bytes after the end address of the previous load region. If this load region is the first one, its start address is offset, and the value of offset must be divisible by 4.

attribute_list: Specifies the attributes of the load region content, including the following; the default attribute of a load region is ABSOLUTE.

- ABSOLUTE: Absolute address;
- PI: Position-independent;
- RELOC: Relocatable;
- OVERLAY: Overlay;

NOCOMPRESS: Not compressible.

max_size: Specifies the maximum size of the load region. If the actual size of the load region exceeds this value, the

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

linker will report an error; the default value is 0xFFFFFFFF.

execution_region_description: Represents an execution region; the "+" sign after it indicates that one or more execution regions can be included. For details about execution regions, refer to the subsequent section.

For more details about load regions, refer to: Keil help/Arm Compiler 5 User's Guides/Linker User Guide /Scatter File Syntax/Load region descriptions

4.3.2 Execution Region

An execution region defines the memory layout of a program during runtime. Execution regions can be distributed across different memory blocks, such as internal RAM, external RAM, etc. Each execution region can contain multiple code segments and data segments.

```
exec_region_name (base_address | "+" offset) [attribute_list] [max_size | length]
{
    input_section_description*
}
```

exec_region_name: The name of the execution region, which can be defined arbitrarily by the user.

The start address of an execution region can be specified in one of the following two formats:

base_address: Indicates the start address of objects in this execution region during linking; the address must be word-aligned.

+offset: Indicates the start address of objects in this execution region during linking is at an offset of offset bytes after the end address of the previous execution region, and the value of offset must be divisible by 4.

attribute_list: Specifies the attributes of the execution region content:

- ABSOLUTE: Absolute address;
- PI: Position-independent;
- RELOC: Relocatable;
- OVERLAY: Overlay;

FIXED: Fixed address. Both the load address and execution address are specified by the base address specifier, which must be an absolute base address or have an offset of 0.

ALIGN alignment: alignment=4, 8, 16… increases the alignment constraint of the execution region from 4 to alignment. The value of alignment must be a positive power of 2. If the execution region has a base_address, it must be aligned to alignment. If the execution region has an offset, the linker aligns the calculated region base address to an alignment boundary;

- EMPTY: Reserves a blank memory block of a specified length in the execution region, typically used for the heap or stack.
- ZEROPAD: Zero-initialized segments are written to the ELF file as zero-padded blocks, so no zero-filling is required at runtime;

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

- PADVALUE: Defines the value for any padding. If PADVALUE is specified, a value must be assigned to it;
- NOCOMPRESS: Not compressible;
- UNINIT: Uninitialized data.

max_size: Specifies the maximum size of the execution region. If the actual size of the execution region exceeds this value, the linker will report an error; the default value is 0xFFFFFFFF.

length: If the specified length is a negative value, base_address is used as the region end address. It is usually used with EMPTY to represent a stack that decreases in size in memory.

input_section_description: Specifies the content of input sections.

For more details about execution regions, refer to: Keil help/Arm Compiler 5 User's Guides/Linker User Guide/ Scatter File Syntax/Execution region descriptions

4.3.3 Input Section

A section is the smallest unit in a scatter loading file, used to describe the specific allocation of code or data.

```
module_select_pattern [ "(" input_section_selector ( "," input_section_selector )* ")" ]
( " + " input_section_attr| input_section_pattern| input_symbol_pattern| section_properties)
```

module_select_pattern: Wildcards "*" and "?" are supported. The "*" symbol matches zero or more characters, and the ? symbol matches any single character. All characters are case-insensitive during matching.

An input section matches the module selector pattern when module_select_pattern matches one of the following:

- The name containing the section and the object file;
- The library member name (without the leading path name);
- The full name of the library (including the path name). If the name contains spaces, wildcards can be used to simplify the search. For example, use *libname.lib to match C:\lib dir\libname.lib.

input_section_attr: Attribute selector. A + sign precedes each input_section_attr. Selectors are case-insensitive. The following selectors are recognized:

- RO-CODE;
- RO-DATA;
- RO, which selects both RO-CODE and RO-DATA;
- RW-DATA;
- RW-CODE;
- RW, which selects both RW-CODE and RW-DATA;
- ZI;
- ENTRY: the section containing the ENTRY point.
- The following synonyms are recognized:
- CODE for RO-CODE;
- CONST for RO-DATA;

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

- TEXT for RO;
- DATA for RW;
- BSS for ZI.
- The following pseudo-attributes are recognized:
- FIRST;
- LAST.

Use FIRST and LAST to mark the first and last sections—order of placement is critical. For example, a specific input section must be in the first section and a checksum must be in the last section.

One or more .ANY patterns can be used to fill an execution region with arbitrary allocation. In most cases, using a single .ANY is equivalent to using the * module selector.

For more details about input sections, refer to: Keil help/Arm Compiler 5 User's Guides/Linker User Guide/ Scatter File Syntax/Input section descriptions.

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

4.4 Scatter Loading Application Examples

4.4.1 General Scatter Loading Configuration

Assume only the Flash and AXI SRAM of the N32H7xx are used:

Flash base address: 0x15000000, size: 0x1E0000 bytes;

RAM base address: 0x24000000, size: 0x20000 bytes.

The scatter loading file configuration is as follows:

```
LR_ROM1 0x15000000 0x1E0000  {    ; Load region definition
  ER_ROM1 0x15000000 0x1E0000  {  ; Execution region definition
    *.o (RESET, +First)             ; Reset vector table
    *(InRoot$$Sections)              ; Standard library initialization code
    .ANY (+RO)                  ; Read-only data (code and constants)
  }
  RW_RAM1 0x24000000 0x20000  {   ; Read-write data region
    .ANY (+RW +ZI)                ; Read-write data and zero-initialized data
  }
}
```


NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

4.4.2 Scatter Loading File Configuration for Multiple RAM Blocks

Assume the Flash, AXI SRAM and AHB SRAM of the N32H7xx need to be used:

Flash base address: 0x15000000, size: 0x1E0000 bytes;

AXI SRAM base address: 0x24000000, size: 0x20000 bytes;

AHB SRAM base address: 0x30000000, size: 0x20000 bytes.

The scatter loading file configuration is as follows:

```
LR_ROM1 0x15000000 0x1E0000  {    ; Load region definition
  ER_ROM1 0x15000000 0x1E0000  {  ; Execution region definition
     *.o (RESET, +First)              ; Reset vector table
     *(InRoot$$Sections)               ; Standard library initialization code
     .ANY (+RO)                  ; Read-only data (code and constants)
  }
  RW_RAM1 0x24000000 0x20000  {   ; Read-write data region 1
     .ANY (+RW +ZI)                ; Read-write data and zero-initialized data
  }
    RW_RAM2 0x30000000 0x20000  {   ; Read-write data region 2
     .ANY (+RW +ZI)                ; Read-write data and zero-initialized data
  }
}
```


NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

4.4.3 Assigning a Specific Function/Variable/Array to SRAM for Execution

Based on the scenario in section 5.2, if you need to assign a function/variable/array to AXI SRAM, there are two methods:

- **__attribute__((setion("mySectionCode")))**

First, define the section in the scatter loading file. The scatter loading file configuration is as follows:

```
LR_ROM1 0x15000000 0x1E0000  {    ; Load region definition
  ER_ROM1 0x15000000 0x1E0000  {  ; Execution region definition
    *.o (RESET, +First)             ; Reset vector table
    *(InRoot$$Sections)              ; Standard library initialization code
    .ANY (+RO)                   ; Read-only data (code and constants)
  }
  RW_RAM1 0x24000000 0x20000  {   ; Read-write data region 1
       .ANY (+RW +ZI)               ; Read-write data and zero-initialized data
       *(mySectionCode)                ; mySectionCode section definition
       *(mySectionData)                ; mySectionData section definition
  }
    RW_RAM2 0x30000000 0x20000  {   ; Read-write data region 2
    .ANY (+RW +ZI)               ; Read-write data and zero-initialized data
  }
}
```

To assign a function to mySectionCode in C language, the code is as follows:

```
__attribute__((section("mySectionCode "))) void myFunction(void) { }
```

To assign a variable to mySectionData, the code is as follows:

```
int myVariable __attribute__((section("mySectionData"))) = 42;
```

To assign an array to mySectionData, the code is as follows:

```
uint8_t myTable[256] __attribute__((section("mySectionData "))) = {};
```

Note: When using Keil Compiler Version 6, variables/functions cannot share the same section name, whereas this is allowed in Compiler Version 5.

Method 2: __attribute__((section(".ARM.__at_0x24000000")))

There is no need to define sections in the scatter loading file; you can directly assign functions/variables/arrays to specific addresses. The C code implementation is as follows:

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

```
int myVariable __attribute__((section(".ARM.__at_0x24004000"))) = 100;

uint8_t myTable[256] __attribute__((section(".ARM.__at_0x24008000")))={};

__attribute__(( section(".ARM.__at_0x15008000"))) void myFunction(void) {     }
```


NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

4.4.4 Scatter Loading File Configuration for Executing Partial Program Copied to ITCM

Assume the Flash, AXI SRAM and ITCM of the N32H7xx need to be used. Since ITCM features a faster execution speed than Flash, it is desired that the user code be executed in ITCM.

Flash base address: 0x15000000, size: 0x1E0000 bytes;AXI SRAM base address: 0x24000000, size: 0x20000 bytes;ITCM base address: 0x00000000, size: 0x20000 bytes.

The programs that must be stored in Flash are as follows:

Reset Vector Table: The first part executed during system startup.

Standard library initialization code: Contains __main() and __rt_entry(). Since the copy of the program from Flash to ITCM is executed by __main(), the code for __main() and all code executed before __main() must only run in Flash.

startup_n32h76x.o file: Both __main() and systemInit() are called in Reset_Handler, which is part of the startup file.

system_n32h7xx.o file: systemInit() is also executed before entering __main().

The scatter loading file configuration is as follows:

```
LR_ROM1 0x15000000 0x1E0000  {    ; Load region definition
  ER_ROM1 0x15000000 0x1E0000  {  ; Execution region definition
    *.o (RESET, +First)             ; Reset Vector Table
    *(InRoot$$Sections)             ; Standard library initialization code, including __main(), __rt_entry()
       startup_n32h76x.o (+RO)          ; Startup file read-only data (code and constants)
       system_n32h7xx.o (+RO)        ; System file read-only data (code and constants)
  }
  ER_ITCM 0x00000000 0x20000  {   ; ITCM execution region
    .ANY (+RO)                ; Read-only data (code and constants)
  }
       RW_RAM1 0x24000000 0x20000  {   ; Read-write data region 1
       .ANY (+RW +ZI)               ; Read-write data and zero-initialized data
  }
}
```

Note: *The stack cannot be placed in DTCM.*

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

5 History versions

Version	Date	Notes
V1.1.0	2026.01.02	1,Initial version

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg

6 Disclaimer

This document is the exclusive property of NSING TECHNOLOGIES PTE. LTD. (Hereinafter referred to as NSING). This document, and the product of NSING described herein (Hereinafter referred to as the Product) are owned by NSING under the laws and treaties of Republic of Singapore and other applicable jurisdictions worldwide. The intellectual properties of the product belong to NSING Technologies Inc. and NSING Technologies Inc. does not grant any third party any license under its patents, copyrights, trademarks, or other intellectual property rights. Names and brands of third party may be mentioned or referred thereto (if any) for identification purposes only. NSING reserves the right to make changes, corrections. enhancements, modifications, and improvements to this document at any time without notice. Please contact NSING and obtain the latest version of this document before placing orders. Although NSING has attempted to provide accurate and reliable information, NSING assumes no responsibility for the accuracy and reliability of this document. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. In no event shall NSING be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising in any way out of the use of this document or the Product. NSING Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, Insecure Usage'. Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, all types of safety devices, and other applications intended to supporter sustain life. All Insecure Usage shall be made at user's risk. User shall indemnify NSING and hold NSING harmless from and against all claims, costs, damages, and other liabilities, arising from or related to any customer's Insecure Usage Any express or implied warranty with regard to this document or the Product, including, but not limited to. The warranties of merchantability, fitness for a particular purpose and non-infringement are disclaimed to the fullest extent permitted by law. Unless otherwise explicitly permitted by NSING, anyone may not use, duplicate, modify, transcribe or otherwise distribute this document for any purposes, in whole or in part.

NSING Technologies Pte. Ltd.
Address: 20 Science Park, #03-15/16
Teleteck Park, East Wing, Singapore 117674
Email: sales@nsing.com.sg