



AN_使用 TCM 或 SDRAM 运行代码的方法

目录

1. 概述.....	3
2. 开发工具.....	3
2.1 软件	3
2.2 硬件	3
2.3 资源介绍	3
3. KEIL 配置说明	4
3.1 分散加载配置说明.....	4
3.2 系统初始化说明	6
4. IAR 配置说明	7
4.1 分散加载配置说明.....	7
5. VSCODE+GCC 配置说明.....	8
5.1 启动文件部分	8
5.2 .LD 文件部分	10
6. 中断向量表拷贝	12
7. 版本历史.....	13
8. 声明.....	14

1.概述

本文以 N32H76x 系列 MCU 为例，介绍了在 Windows 环境下基于 Keil 编辑器，让代码加速运行的方法。

本文提供两种加速方法，一种是将代码放到内部 TCMSRAM 中运行，一种是将代码放到外部 SDRAM 中运行。

2.开发工具

2.1 软件

- 1) 编辑器 KEIL MDK-ARM V5.34
- 2) VSCode+GCC

2.2 硬件

- 1) 核心板 N32H787XIB7-HMI V1.0

2.3 资源介绍

➤ TCMSRAM

TCMSRAM 存储器由 SRAM 存储器构建，包括以下类型：作为 Cortex-M7 紧密耦合存储器使用的 TCM（ITCM、D0TCM、D1TCM），以及作为芯片上通用 SRAM 使用的 AXISRAM（AXISRAM2/3）。

用户可以以 128KB 为粒度配置 ITCM、DTCM 的大小，其余的内存空间将自动分配给两个 AXI SRAM2/3。

➤ SDRAM

MCU 支持 SDRAM 控制器，可访问外部 SDRAM 设备，最大支持访问 4x16Mx32bit (256 MB) 的 SDRAM，SDRAM1 默认访问地址范围为 0xC0000000~0xCFFFFFFF，此时可正常读写数据；当基地址重映射到 0x60000000 时，可以运行代码。

本应用笔记 SDRAM 以 W9825G6KH 型号为例，容量为 4Mx16Bitx4Bank(32 MB)，需要连接到 SDRAM1 控制器。

3. KEIL 配置说明

3.1 分散加载配置说明

➤ Keil 环境：下面以 512K ITCM + 512K DTCM 的配置为例进行说明

```

1  ; *****
2  ; *** Scatter-Loading Description File generated by uVision ***
3  ; *****
4
5  LR_IROM1 0x15000000 0x00100000 { ; load region size_region
6  ER_IROM1 0x15000000 0x00100000 { ; load address = execution address
7      *.o (RESET, +First)
8      *(InRoot$$Sections)
9      startup_n32h76x.o (+RO)
10     system_n32h76x_78x.o (+RO)
11     n32h76x_78x_it.o (+RO)
12 }
13
14 ;ITCM
15 RW_ITCM 0x00000400 0x00010000 { ; 512K ITCM offset 0x400 for VECTOR TABLE
16     .ANY (+RO)
17     .ANY (+XO)
18 }
19
20 ;AXI SRAM
21 RW_AXI_SRAM 0x24000000 0x00020000 { ; 128K AXI-SRAM stack
22     .ANY (+RW +ZI)
23     *(STACK)
24 }
25
26 ;DTCM
27 RW_DTCM 0x20000000 0x00080000 { ; 512K DTCM
28     .ANY (+RW +ZI)
29 }
30
31 ;AHB-SRAM
32 RW_AHB_SRAM 0x30000000 0x00020000 { ; 128K AHB-SRAM
33     system_n32h76x_78x.o (+RW +ZI)
34     *(STACK)
35 }
36
37
38

```

➤ Keil 环境：下面以 512K SDRAM 的配置为例进行说明

```

1  ; *****
2  ; *** Scatter-Loading Description File generated by uVision ***
3  ; *****
4
5  LR_IROM1 0x15000000 0x00100000 { ; load region size_region
6  ER_IROM1 0x15000000 0x00100000 { ; load address = execution address
7      *.o (RESET, +First)
8      *(InRoot$$Sections)
9      startup_n32h76x.o (+RO)
10     system_n32h76x_78x.o (+RO)
11     n32h76x_78x_it.o (+RO)
12 }
13
14 ;SDRAM
15 RW_SDRAM 0x60000400 0x00010000 { ; 512K SDRAM offset 0x400 for VECTOR TABLE
16 .ANY (+RO)
17 .ANY (+XO)
18 }
19
20 ;AXI SRAM
21 RW_AXI_SRAM 0x24000000 0x00020000 { ; 128K AXI-SRAM stack
22 ; .ANY (+RW +ZI)
23 *(STACK)
24 }
25
26 ;DTCM
27 RW_DTCM 0x20000000 0x00080000 { ; 512K DTCM
28 .ANY (+RW +ZI)
29 }
30
31 }
32
33
  
```

LR_IROM1 代表定义加载区域为 0x15000000，大小为 1M；该区域表示所有代码从该区域加载：

1. ER_IROM1 定义执行区域为 0x15000000，大小为 1M；其中复位向量表，标准库的初始化代码，以及 startup_n32h76x.o, system_n32h76x_78x.o, n32h76x_78x_it.o 代码在该区域执行；
2. RW_ITCM 定义了另外一块执行区域为 ITCM，地址为 0x00000400，大小为 512K；
RW_SDRAM 定义了另外一块执行区域为 SDRAM，地址为 0x60000400，大小为 512K；
表示除了 startup_n32h76x.o, system_n32h76x_78x.o, n32h76x_78x_it.o 的以外的所有代码都放在该区域运行，大小用户可配置；
3. RW_AXI_SRAM 定义了读写数据区域为 AXI SRAM，地址 0x24000000，大小为 128K；（用户可配置），表示栈区定义到该区域；
4. RW_DTCM 定义了另外一块读写数据区域为 DTCM，地址 0x20000000，大小为 512K；（用户可配置），表示工程中所有全局变量都定义到该区域；

注意：

1. ITCM, DTCM 的可以在代码中进行自由配置；
2. 未配置好 TCM 之前，不要将栈区定义到 DTCM 区域；
3. 该分散加载是以 512K ITCM + 512K DTCM 或 SDRAM 为例说明的，客户可以根据自己的实际需求去调整，关于 TCM 配置请参考《CN_UG_N32H7xx_TCM_User_Guide.pdf》；

3.2 系统初始化说明

所使用 SDRAM 加速，则 SDRAM 控制器需要先 system_n32h76x_78x.c（在 0x15000000 地址段）中完成初始化，再跳转到 main 函数（在 0x60000000 地址段），然后在 SDRAM 中运行所有其他代码。

system_n32h76x_78x.c 初始化内容为（详细见源码）：

1. 配置系统时钟为 600M，系统时钟源为 HSI-PLL1A
2. 配置 SDRAM 时钟为 133M，时钟源为 HSE-PLL3B(665M)，5 分频
3. 配置 SDRAM 设备的所有 GPIO
4. 初始化 SDRAM 控制器的基地址、mask 地址、时序、基本配置内容、使能重映射

```

main.c  system_n32h76x_78x.c
367  /* FPU settings -----
368  #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
369      SCB->CPACR |= ((3UL << (10*2))|(3UL << (11*2))); /* set CI
370  #endif
371
372  /*SEVONPEND enabled so that an interrupt coming from the CI
373      detectable by the CPU after a WFI/WFE instruction.*/
374  SCB->SCR |= SCB_SCR_SEVONPEND_Msk;
375
376  #ifdef CORE_CM7
377      RCC->SYSBUSDIV1=0x01001100;
378
379      __DBG_RCC_DELAY_US(1);
380      /* Configure APB1 APB2 APB5 APB6 clock is 150M */
381      RCC->SYSBUSDIV2=0x04040404;
382
383      /* configure PLL1 source is HSI, frequency is 600M */
384      /* get the register value */
385      temp_value1 = RCC->PLL1CTRL1;
386      temp_value2 = RCC->PLL1CTRL2;
387
388      /* Clear BWAJ[11:0] bits */
389      temp_value1 &= RCC_PLL_BWAJ_MASK;
390      /* Clear CLKF[25:0] and CLKR[5:0] bits */
391      temp_value2 &= RCC_PLL_CLKR_CLKF_MASK;
392
393      /* Set BWAJ[11:0] bits - 600M */
394      temp_value1 |= 0x3;
395      /* Set CLKF[25:0] and CLKR[5:0] bits */
396      temp_value2 |= 0x25800;
397
398      /* Store the new value */
399      RCC->PLL1CTRL1 = temp_value1;
400      RCC->PLL1CTRL2 = temp_value2;
401
402      /* Enable PLL module power */
403      RCC->PLL1CTRL1 |= RCC_PLL_LDO_ENABLE;
404      __DBG_RCC_DELAY_US(10);
405
406      /* Enable power to analog circuitry in PLL */
407      RCC->PLL1CTRL1 &= (~RCC_PLL_POWER_DOWN);
408
409      /* Select Clock Source */
410      temp_value1 = RCC->PLL1CTRL1;
411      temp_value1 &= RCC_PLL_SRC_MASK;

```

4.IAR 配置说明

4.1 分散加载配置说明

```

34  define block ITCM_CODE      { readwrite code };
35  define block ITCM_DATA      { readwrite data }
36      except {
37          section .bss,
38          section .data
39      };
40
41  do not initialize { section .noinit };
42  initialize by copy { readwrite };
43  initialize by copy with packing = none { readonly }
44      except {
45          section .intvec,
46          ro object system_n32h76x_78x.o,
47          ro object startup_n32h76x_EWARM.o,
48          ro object n32h76x_78x_it.o,
49          ro object *.a
50      };
51
52  if (isdefinedsymbol(__USE_DLIB_PERTHREAD))
53  {
54      // Required in a multi-threaded application
55      initialize by copy with packing = none { section __DLIB_PERTHREAD };
56  }
57
58  place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };
59
60  place in EITCM_region { block ITCM_CODE, block ITCM_DATA};
61  place in IROM_region { readonly };
62  place in IRAM_region { readwrite, block CSTACK, block PROC_STACK, block HEAP };
63
64

```

IROM_region 代表定义加载区域为 0x15000000，大小为 1M；该区域表示所有代码从该区域加载；

IRAM_region 代表读写数据区域：

EITCM_region 代表读写执行区域：

1. 定义除了复位向量表，标准库的初始化代码，以及 startup_n32h76x_EWARM.o，system_n32h76x_78x.o，n32h76x_78x_it.o 代码不会拷贝到 TCM 去执行，其他的用户代码都会默认拷贝到 ram 中执行；
2. 将步骤 1 中的拷贝的代码（属性为 readwrite code 和 readwrite data）放在 ITCM，地址为 0x00000400，大小为 1M；

5.VSCode+GCC 配置说明

1. GCC 环境只演示了如何将代码存储在 FLASH, 上电搬运到 SRAM 中运行的方式, 不包含 SDRAM 部分。
2. 本文主要讲解 VSCode+GCC 下将存储在 FLASH 代码搬运到 SRAM 运行, 关于 VSCode+GCC 环境的配置和讲解, 具体请查看《AN_N32H76x_GCC Development Environment Application Note_V1.0.0》和《AN_N32H78x_GCC Development Environment Application Note_V1.0.0》

5.1 启动文件部分

需要在启动文件中添加部分代码, 用于拷贝 FLASH 的数据/代码到指定的 SRAM 位置。

1. 将.ld 文件中各段文件的段名在启动文件中进行定义

```
/* start address for the initialization values of the .data section.
defined in linker script */
.word _sidata
/* start address for the .data section. defined in linker script */
.word _sdata
/* end address for the .data section. defined in linker script */
.word _edata
/* start address for the .bss section. defined in linker script */
.word _sbss
/* end address for the .bss section. defined in linker script */
.word _ebss
/*some param defined in linker script*/
.word _siram_code
.word _s_ram_code
.word _e_ram_code
.word _fp_flash_rodata
.word _fp_s_rodata
.word _fp_e_rodata
.word __exidx_start
.word __exidx_end
.word _fp_exidx
.word __preinit_array_start
.word __preinit_array_end
.word _sipreinit_array
.word __init_array_start
.word __init_array_end
.word _siinit_array
.word __fini_array_start
.word __fini_array_end
.word _sifini_array
```

2. 定义了一个拷贝各段数据到 SRAM 部分的函数宏定义


```

/* 宏定义：统一复制过程 */
.macro COPY_SECTION section_start, section_end, load_address
    ldr r0, =\section_start    /* RAM目标地址 */
    ldr r2, =\load_address    /* Flash源地址 */
    ldr r1, =\section_end     /* RAM结束地址 */
    subs r3, r1, r0           /* 计算长度 */
    ble 1f                    /* 如果长度为0则跳过 */
/* 复制循环（每次4字节） */
0: ldr r4, [r2], #4
   str r4, [r0], #4
   subs r3, r3, #4
   bgt 0b
1:
.endm

```

3. 将执行数据拷贝的操作放到 SystemInit 之后，因为 ITCM/DTCM 的初始化是在 SystemInit 函数中进行的，没有进行初始化前无法对这个区域进行访问

```

Reset_Handler:
    ldr r0, _estack
    mov sp, r0    /* set stack pointer */

/* Zero fill the bss segment. */
    ldr r2, _sbss
    ldr r4, _ebss
    movs r3, #0
    b LoopFillZerobss

FillZerobss:
    str r3, [r2]
    adds r2, r2, #4

LoopFillZerobss:
    cmp r2, r4
    bcc FillZerobss

/* Call the clock system initialization function.*/
    bl SystemInit

/* 复制所有需要初始化的段 */
    COPY_SECTION _s_ram_code, _e_ram_code, _siram_code    /* 代码段 */
/* COPY_SECTION _fp_s_rodata, _fp_e_rodata, _fp_flash_rodata    只读数据 */
    COPY_SECTION _sdata, _edata, _sidata    /* 数据段 */
/* COPY_SECTION __exidx_start, __exidx_end, _fp_exidx    ARM异常索引 */
/* COPY_SECTION __preinit_array_start, __preinit_array_end, _sipreinit_array */
    COPY_SECTION __init_array_start, __init_array_end, _siinit_array
    COPY_SECTION __fini_array_start, __fini_array_end, _sifini_array
    COPY_SECTION _sirq_vector_ram, _eirq_vector_ram, _sirqvector    中断向量表 */
/* Call static constructors */
    bl __libc_init_array
/* Call the application's entry point.*/
    bl main

LoopForever:
    b LoopForever

```

5.2 .ld 文件部分

.ld 文件用来做各个代码/数据位置的定义，重点关注的如下：

1. 定义各块区域

_estack 定义栈顶的位置，_Min_Heap_Size/_Min_Stack_Size 定义堆栈的大小；

此例程将 SRAM 进行了三段划分，分为 AXI_SRAM(128KB)，ITCM_SRAM(1MB)，AHB_SRAM(352KB)，因为本例程是将执行代码放到 ITCM_SRAM 执行，所以 ITCM_SRAM 的前 1K 预留出来用于存储中断向量表，当放到其他区域执行时同理。

FLASH 用于存储各类数据/代码。

```

51  /* Highest address of the user mode stack */
52  _estack = 0x30058000; /* end of RAM */
53  /* Generate a link error if heap and stack don't fit into RAM */
54  _Min_Heap_Size = 0x2000; /* required amount of heap */
55  _Min_Stack_Size = 0x1000; /* required amount of stack */
56
57  /* Specify the memory areas */
58  MEMORY
59  {
60      AXI_SRAM (xrw): ORIGIN = 0x24000000, LENGTH = 0x00020000 /* 128KB */
61      ITCM_SRAM (xrw): ORIGIN = 0x00004000, LENGTH = 0x000FFC00 /* 1024KB - 1KB */
62      AHB_SRAM (xrw): ORIGIN = 0x30000000, LENGTH = 0x58000 /* 352KB */
63      FLASH (rx) : ORIGIN = 0x15000000, LENGTH = 0x1E0000 /* 2*1024K - 128K */
64  }

```

2. 定义各代码的存储和执行位置

中断向量表默认在 FLASH 存储及执行，当代码都搬运到 SRAM 指定位置后，跳转到 mian 函数立刻执行将向量表搬运到 SRAM，具体见章节 6。

```

69  /* The startup code goes first into FLASH */
70  .isr_vector :
71  {
72      . = ALIGN(4);
73      KEEP(*(.isr_vector)) /* Startup code */
74      . = ALIGN(4);
75  } >FLASH

```

系统初始化函数和启动文件放到 FLASH 执行，用于时钟配置和代码/数据搬运。

其他所有的执行 Code 放到 ITCM_SRAM 区域，data 放到 AHB_SRAM。

看下图“>FLASH”表示存放在 FLASH 且在 FLASH 执行，“>ITCM_SRAM AT> FLASH”表示存放在 FLASH，在 ITCM_SRAM 区域执行，其他同理。

在第 95 行“.text”段表示所有执行代码都在此区域，存放在 FLASH，在 ITCM_SRAM 区域执行；但第 84 行“.text_flash”优先于“.text”段执行，所以“startup_n32h76x_gcc”和“system_n32h76x_78x”文件的代码会存放到 FLASH 并在 FLASH 执行，剩余代码在 ITCM_SRAM 执行。

```

77  .init :
78  {
79      . = ALIGN(4);
80      KEEP(*(.init))
81      . = ALIGN(4);
82  } >FLASH
83
84  .text_flash :
85  {
86      . = ALIGN(4);
87      build/startup_n32h76x_gcc.o(.text*)
88      build/startup_n32h76x_gcc.o(.rodata*)
89      build/system_n32h76x_78x.o(.text*)
90      build/system_n32h76x_78x.o(.rodata*)
91      . = ALIGN(4);
92  } >FLASH
93
94  /* The program code and other data goes into ram */
95  .text :
96  {
97      . = ALIGN(4);
98      _s_ram_code = .;    /* section start address*/
99      *(.text)            /* .text sections (code) */
100     *(.text*)           /* .text* sections (code) */
101     *(.glue_7)          /* glue arm to thumb code */
102     *(.glue_7t)         /* glue thumb to arm code */
103     *(.eh_frame)
104
105     KEEP (*(.init))
106     KEEP (*(.fini))
107
108     . = ALIGN(4);
109     _e_ram_code = .;    /* define a global symbols at end of code */
110 } >ITCM_SRAM AT > FLASH
111
112 _siram_code = LOADADDR(.text);

```

6. 中断向量表拷贝

由于向量表在 FLASH，为了代码加速，代码初始化需要将 FLASH 的向量表拷贝到加速区域中，函数如下所示：

➤ TCM_SRAM

```
#define      ITCM_BASE      (0x00000000UL)
void CopyVectTableToITCM(void)
{
    volatile uint32_t* pSrcVect = (uint32_t*)(FLASH_BASE);
    volatile uint32_t* pDestVect = (uint32_t*)(ITCM_BASE);
    uint32_t numVECT = 0x400 / 4;
    for(uint32_t i = 0; i < numVECT; i++)
    {
        pDestVect[i] = pSrcVect[i];
    }
    SCB->VTOR = ITCM_BASE;
    __ISB();
    __DSB();
}
```

➤ SDRAM

```
#define      SDRAM_REMAP_BASE      (0x60000000UL)
void CopyVectTableToSDRAM(void)
{
    volatile uint32_t* pSrcVect = (uint32_t*)(FLASH_BASE);
    volatile uint32_t* pDestVect = (uint32_t*)(SDRAM_REMAP_BASE);
    uint32_t numVECT = 0x400 / 4;
    for(uint32_t i = 0; i < numVECT; i++)
    {
        pDestVect[i] = pSrcVect[i];
    }
    SCB->VTOR = SDRAM_REMAP_BASE;
    __ISB();
    __DSB();
}
```

7.版本历史

日期	版本	修改记录
2025.04.29	V1.0.0	初始版本
2025.06.25	V1.1.0	1. 增加 VSCode+GCC 环境例程 2. 增加 IAR 的例程配置说明 3. 文档部分结构优化

8. 声明

国民技术股份有限公司（下称“国民技术”）对此文档拥有专属产权。依据中华人民共和国的法律、条约以及世界其他法域相适用的管辖，此文档及其中描述的国民技术产品（下称“产品”）为公司所有。

国民技术在此并未授予专利权、著作权、商标权或其他任何知识产权许可。所提到或引用的第三方名称或品牌（如有）仅用作区别之目的。

国民技术保留随时变更、订正、增强、修改和改良此文档的权利，恕不另行通知。请使用人在下单购买前联系国民技术获取此文档的最新版本。

国民技术竭力提供准确可信的资讯，但即便如此，并不推定国民技术对此文档准确性和可靠性承担责任。

使用此文档信息以及生成产品时，使用者应当进行合理的设计、编程并测试其功能性和安全性，国民技术不对任何因使用此文档或本产品而产生的任何直接、间接、意外、特殊、惩罚性或衍生性损害结果承担责任。

国民技术对于产品在系统或设备中的应用效果没有任何故意或保证，如有任何应用在其发生操作不当或故障情况下，有可能致使人员伤亡、人身伤害或严重财产损失，则此类应用被视为“不安全使用”。

不安全使用包括但不限于：外科手术设备、原子能控制仪器、飞机或宇宙飞船仪器、所有类型的安全装置以及其他旨在支持或维持生命的应用。

所有不安全使用的风险应由使用人承担，同时使用人应使国民技术免于因为这类不安全使用而导致被诉、支付费用、发生损害或承担责任时的赔偿。

对于此文档和产品的任何明示、默示之保证，包括但不限于适销性、特定用途适用性和不侵权的保证责任，国民技术可在法律允许范围内进行免责。

未经明确许可，任何人不得以任何理由对此文档的全部或部分进行使用、复制、修改、抄录和传播。