

目录

Introduction	2
Background knowledge	3
编译过程	3
代码段，数据段，BSS 段，堆，栈	4
ARM 执行程序步骤	4
Make	5
GCC	6
参数详解	7
常用参数	7
预编译 -E	8
编译-S	10
汇编 -C	10
链接	11
arm-none-eabi 交叉编译工具	12
help	12
arm-none-eabi-gcc	12
arm-none-eabi-ld	12
arm-none-eabi-objdump	13
arm-none-eabi-objcopy	13
arm-none-eabi-size	13
RT700 用到的 gcc command	13
Boot	13
Linkfile 解析	14
C 编译运行	14
Crt0.S	14
_Start.c	15
Pre_main 、 post_main	15
void pre_main()	15
void post_main()	16
Main()	16
CM33 NS boot	17
生成 linkfile	17
编译	20
链接	23
启动流程分析	25
CM33_M boot	31
CM33_M_NS boot	32
HIFI4 boot	32
HIFI1 Boot	33
Zenv Boot	34
Dual core boot	35

Introduction

本文介绍 RT700 testbench boot 流程，包含如下内容：

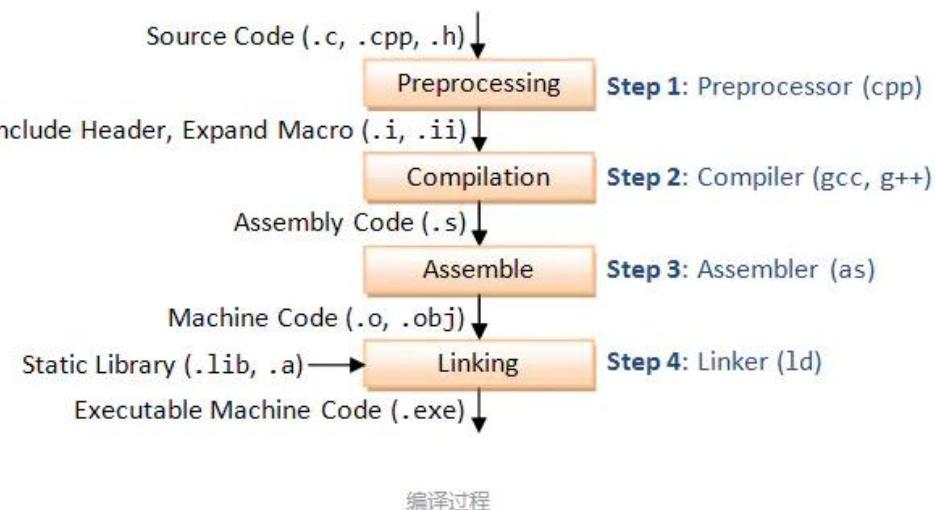
1. 嵌入式 C 程序如何被执行，编译/链接/反汇编知识阐述
2. Testbench 用到的 makefile, 汇编、链接、反汇编 commands
3. 启动汇编 crt0.s 的讲解，从汇编跳转后的第一个 C 程序
4. Testbench 从上电到 test case 执行的详细 flow

Background knowledge

编译过程

CPU 执行程序的本质，是从存储器里面获取机器码。可执行程序从代码到机器码的过程如下：

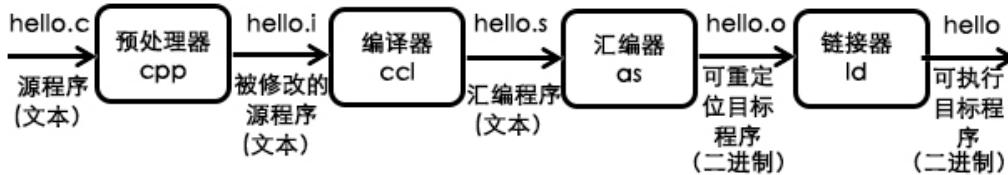
- 预处理(Pre-Processing)
- 编译(Compiling)
- 汇编(Assembling)
- 链接(Linking)



在 Unix/Linux 系统上，从源文件到目标文件的转化是由编译器驱动程序完成的：

```
linux> gcc -o hello hello.c
```

在这里，gcc 编译器驱动程序读取源程序文件 hello.c，并把它翻译成一个可执行目标文件 hello，这个翻译的过程是分为四个阶段完成的，如下图所示，执行这四个阶段的程序（预处理器、编译器、汇编器和链接器）一起构成了编译系统。



预处理阶段。 预处理器（cpp）根据以字符#开头的命令，修改原始C程序。比如hello.c中的第一行的 `#include <stdio.h>` 指令告诉预处理器读取系统文件 `stdio.h` 的内容。并把它直接插入到程序文本中去。结果就得到了另一个C程序，通常是以 “.i” 作为文件扩展名。

编译阶段。 编译器（ccl）将文本文件 `hello.i` 翻译成文本文件 `hello.s`，它包含一个汇编语言程序。汇编语言程序中的每条语句都以一种标准的文本格式确切地描述了一条低级机器语言指令。汇编语言是非常有用的，因为它为不同高级语言的不同编译器提供了通用输出语言。例如：C语言编译器和Fortran编译器产生的输出文件用的都是一样的汇编语言。

汇编阶段。 接下来，汇编器（as）将 `hello.s` 翻译成机器语言指令，把这些指令打包成为一种叫做可重定位目标程序的格式，并将结果保存在目标文件 `hello.o` 中。`hello.o` 文件是一个二进制文件，它的字节编码是机器语言指令而不是字符，如果我们在文本编辑器中打开 `hello.o` 文件，呈现的将是一堆乱码。

链接阶段。 请注意，我们的 `hello` 程序调用了 `printf` 函数，它是标准 C 库中的一个函数，每个 C 编译器都提供，`printf` 函数存在与一个名为 `printf.o` 的单独的预编译目标文件中，而这个文件必须以某种方式并入到我们的 `hello.o` 程序中。链接器（ld）就负责这种并入，结果就得到 `hello` 文件，它是一个可执行目标文件（或者简称为可执行文件）。可执行文件加载到存储器后，由系统负责执行。

代码段，数据段，BSS 段，堆，栈

程序，变量的存储会分成五个段，详解如下。我们写的 code 在进行链接的时候，需要手动配置代码，变量各个段的地址。

<https://www.cnblogs.com/zl1991/p/15039949.html>

ARM 执行程序步骤

以 CM33 为例，ARM 上电后会从 0 地址取 SP，0x4 取 PC。从 PC 指向的地址获取第一条 command。RT700, RTL 做了配置，上电后 vector table 配置到 ROM 0 地址，也就是 0x13000000

表 7.6 上电后的向量表

地址	异常编号	值（32位整数）
0x0000_0000	-	MSP 的初始值
0x0000_0004	1	复位向量（PC 初始值）
0x0000_0008	2	NMI 服务例程的入口地址
0x0000_000C	3	硬 fault 服务例程的入口地址
...	...	其它异常服务例程的入口地址

复位序列

在离开复位状态后，CM3 做的第一件事就是读取下列两个 32 位整数的值：

- 从地址 0x0000,0000 处取出 MSP 的初始值。
- 从地址 0x0000,0004 处取出 PC 的初始值——这个值是复位向量，LSB 必须是 1。然后从这个值所对应的地址处取指。
-

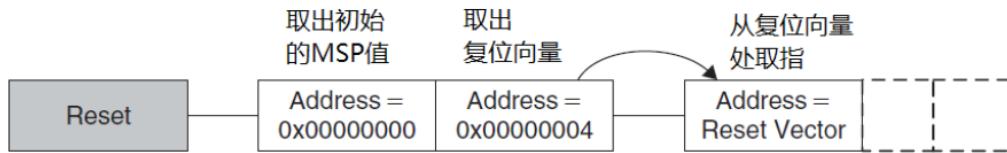


图 3.17 复位序列

Make

Makefile 的本质是为了让整个工程能够按照设定的规则进行编译操作。

RT700 make 过程中产生的 make 文件，bin，hex，反汇编，map 文件都会存放到如下的目录中
testbench/blocks/soc_tb/tool_data/verilog.

RT700 用到的几个 makefile 如下：

run_make_stim.sh

```
Make -f testbench/blocks/soc_tb/testbench/makefile.stimulus -I testbench/blocks/soc_tb/testbench
-C testbench/blocks/soc_tb/vectors/dma/stimulus
SRAY mem map generation
```

testbench/common_blocks/v_ip_makefile/tool_data/compiler/makefile.compile

显示 build elf 需要的文件依赖

Log: Build Directory

testbench/common_blocks/v_ip_makefile/tool_data/compiler/makefile.compile.gxx

C 编译命令

testbench/blocks/soc_tb/testbench/makefile.opts

 定义 C 编译参数

Testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_s/makefile.pre_compile

 定义 code、data section， 默认 stimulus 等，生成 linkfile

GCC

Gcc 是常用的 C 编译器，如下列出常用的一些 command。

<https://www.jianshu.com/p/1bab86143f1c>

参数详解

常用参数

选项名	作用
-c	通知gcc取消连接步骤，即编译源码并在最后生成目标文件
-Dmacro	定义指定的宏，使它能够通过源码中的#define进行检验
-E	不经过编译预处理程序的输出而输送至标准输出
-g3	获得有关调试程序的详细信息，它不能与-o选项联合使用
-Idirectory	在包含文件搜索路径的起点处添加指定目录
-llibrary	提示连接程序在创建最终可执行文件时包含指定的库
-O -O2 -O3	将优化状态打开，该选项不能与-g选项联合使用。当出现多个优化时，以最后一个为准
-O0	关闭所有优化选项
-S	要求编译程序生成来自源代码的汇编程序输出
-v	启动所有警报
.h	预处理文件(标头文件)
-Wall	在发生警报时取消编译操作，即将警报看作是错误
-w	禁止所有的报警
-share	此选项将尽量使用动态库，所以生成文件比较小，但是需要系统由动态库
-shared	产生共享对象文件

-static	使用静态链接。此选项将禁止使用动态库。所以编译出的文件一般都很大，不需要动态连接库
-finline-functions,-fnoinline-functions	启用/关闭内联函数
-g	在编译结果中加入调试信息
-ggdb	加入GDB调试器能识别的格式
-fPIC	使用地址无关代码模式进行编译
-fPIE	使用地址无关代码模式编译可执行文件
-fomit-frame-pointer	禁止使用EBP作为函数帧指针
-fno-builtin	禁止GCC编译器内置函数
-fno-stack-protector	关闭堆栈保护功能
-ffunction-sections	将每个函数编译到独立的代码段
-fdata-sections	将全局/静态变量编译到独立的数据段

预编译 -E

- (1) 预编译

使用 `-E` 参数可以让gcc在预处理结束后停止编译过程：

```
gcc -E hello.c -o hello.i
```

此时若查看hello.i文件中的内容，会发现stdio.h的内容确实都插到文件里去了，而且被预处理的宏定义也都作了相应的处理。

```
1 # 1 "hello.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
7 # 1 "hello.c"
8 # 1 "/usr/include/stdio.h" 1 3 4
9 # 27 "/usr/include/stdio.h" 3 4
10 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
11 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
12 # 1 "/usr/include/features.h" 1 3 4
13 # 424 "/usr/include/features.h" 3 4
14 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
15 ...
```

编译-S

- (2)编译

编译过程通过词法和语法分析，确认所有指令符合语法规则(否则报编译错)，之后翻译成对应的中间码，在linux中被称为RTL(Register Transfer Language)，通常是平台无关的，这个过程也被称为编译前端。编译后端对RTL树进行裁减，优化，得到在目标机上可执行的汇编代码。gcc采用as作为其汇编器，所以汇编码是AT&T格式的，而不是Intel格式，所以在用gcc编译嵌入式汇编时，也要采用AT&T格式。

使用 `-S` 命令

```
gcc -S hello.i -o hello.S
```

gcc默认将.i文件看成是预处理后的C语言源代码，因此上述命令将自动跳过预处理步骤而开始执行编译过程。

也可以使用-x参数让gcc从指定的步骤开始编译为目标代码。

以下为编译后的输出文件hello.s的内容

```
1      .file   "hello.c"
2      .text
3      .section    .rodata
4      .LC0:
5      .string "hello linux"
6      .text
7      .globl  main
```

汇编 -C

汇编得到的结果没有链接，不可执行

- (3)汇编

汇编器是将汇编代码转变成机器可以执行的命令，每一个汇编语句几乎都对应一条机器指令。汇编相对于编译过程比较简单，根据汇编指令和机器指令的对照表——翻译即可。

使用 `-c` 命令

```
gcc -c hello.c -o hello.o
```

链接

也可以使用 `ld` 命令将汇编得到的 `.o` 连接成可执行文件

- (4)链接

```
gcc hello.o -o hello
```

链接器 `ld` 将各个目标文件组装在一起，解决符号依赖，库依赖关系，并生成可执行文件。

`ld -static crt1.o crti.o crtbeginT.o hello.o -start-group -lgcc -lgcc_eh -lc -end-group crtend.o crtst.o` (省略了文件的路径名)。

当然链接的时候还会用到静态链接库，和动态连接库。静态库和动态库都是 `.o` 目标文件的集合。

静态库是在链接过程中将相关代码提取出来加入可执行文件的库(即在链接的时候将函数的代码将从其所在地静态链接库中被拷贝到最终的可执行程序中)，`ar` 只是将一些别的文件集合到一个文件中。可以打包，当然也可以解包。

arm-none-eabi 交叉编译工具

help

```
|awv071589.cn-sha01.nxp.com: arm-none-eabi-ld -help  
Usage: /pkg/arm-gcc-/6.2-2016q4/x86_64-linux/bin/arm-none-eabi-ld [options] file...  
Options:  
-a KEYWORD Shared library control for HP/UX compatibility  
-A ARCH, --architecture ARCH Set architecture  
-b TARGET, --format TARGET Specify target for following input files  
-c FILE, --mri-script FILE Read MRI format linker script  
-d, -dc, -dp Force common symbols to be defined  
-e ADDRESS, --entry ADDRESS Set start address  
-E, --export-dynamic Export all dynamic symbols  
--no-export-dynamic Undo the effect of --export-dynamic  
-EB Link big-endian objects  
-EL Link little-endian objects  
-f SHLIB, --auxiliary SHLIB Auxiliary filter for shared object symbol table  
-F SHLIB, --filter SHLIB Filter for shared object symbol table  
-g Ignored  
-G SIZE, --gpsize SIZE Small data size (if no size, same as --shared)  
-h FILENAME, --soname FILENAME Set internal name of shared library  
-I PROGRAM, --dynamic-linker PROGRAM Set PROGRAM as the dynamic linker to use  
--no-dynamic-linker Produce an executable with no program interpreter header  
-l LIBNAME, --library LIBNAME Search for library LIBNAME  
-L DIRECTORY, --library-path DIRECTORY
```

arm-none-eabi-gcc

arm-none-eabi-gcc [--help](#) 查看帮助信息。

arm-none-eabi-gcc -c a.c 生成a.o文件。

arm-none-eabi-gcc -g -c a.c 生成a.o文件,-g使得如果[反汇编](#)可对应C语言显示。

arm-none-eabi-gcc -c a.c b.c 生成a.o和b.o文件。

arm-none-eabi-gcc -c a.c -march=armv7-a -mcpu=cortex-a8 -mfpu=vfpv3 包含架构信息

arm-none-eabi-ld

arm-none-eabi-ld -T ab.lds a.o b.o -o ab.elf 读链接脚本ab.lds，链接a.o和b.o，生成ab.elf文件。

arm-none-eabi-ld -T ab.lds a.o b.o -o ab.elf -Map ab.map 生成ab.map文件。

arm-none-eabi-objdump

arm-none-eabi-objdump -d -S(可省) a1.o 查看a1.o反汇编可执行段代码

arm-none-eabi-objdump -D -S(可省) a1.o 查看a1.o反汇编所有段代码

arm-none-eabi-objcopy

arm-none-eabi-objcopy -O binary ab.elf ab.bin 生成可在arm平台上运行的bin文件

arm-none-eabi-size

计算文件大小

```
| awv071589.cn-sha01.nxp.com:>arm-none-eabi-size -B dma3_slave_access_sram_cm33_ns_inst0.elf
|   text     data      bss    dec    hex filename
| 20392     2072     2056   24520   5fc8 dma3_slave_access_sram_cm33_ns_inst0.elf
|----- ..
```

RT700 用到的 gcc command

```
| ESC[0;32m[crt0.o] /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/
| arm-none-eabi-as --defsym DEF_LC_NXP_FAB_MODE_DEFAULT=1 --defsym ANATOP_ID=ANATOP_INST0 -mthumb -mcpu=cortex
| ESC[0;32m[dma3_reg_cm33_ns_inst0.elf] _start.o capi.o capi_int.o capi_interrupt_handler.o hard_fault_recover
| arm-none-eabi-ld _start.o capi.o capi_int.o capi_interrupt_handler.o hard_fault_recovery.o secure_portal.o t
| ESC[0;32m[dma3_reg_cm33_ns_inst0.nm] dma3_reg_cm33_ns_inst0.elfESC[0m
| arm-none-eabi-nm --demangle -n --demangle -n dma3_reg_cm33_ns_inst0.elf > dma3_reg_cm33_ns_inst0.nm
| ESC[0;32m[dma3_reg_cm33_ns_inst0.lst] dma3_reg_cm33_ns_inst0.elfESC[0m
| arm-none-eabi-objdump --disassemble-all --disassemble-zeroes --wide --source -hlds dma3_reg_cm33_ns_inst
| ESC[0;32m[dma3_reg_cm33_ns_inst0.srec] dma3_reg_cm33_ns_inst0.elfESC[0m
| arm-none-eabi-objcopy --strip-all -O srec dma3_reg_cm33_ns_inst0.elf dma3_reg_cm33_ns_inst0.srec
| ESC[0;32m[dma3_reg_cm33_ns_inst0.bin] dma3_reg_cm33_ns_inst0.elfESC[0m
| arm-none-eabi-objcopy --strip-all -O binary dma3_reg_cm33_ns_inst0.elf dma3_reg_cm33_ns_inst0.bin
```

Boot

Linkfile 解析

<https://blog.csdn.net/ymj321/article/details/116937630>

V_SS_RT700_SOC_TB_1.10/testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_s/ldscripts/cm33_s.lnk

C 编译运行

testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_s/cm33_s.lnk

```
/* Set the initial value for the stack pointer */
/* Leave 8 bytes at the top for the CAPI */
__SP_INIT      = ADDR(stack) + SIZEOF(stack) - 8;

/* GDB needs to be told where the entry-point is */
__START_ADDR   = ADDR(.startup);
ENTRY(__START_ADDR);
```

Linkfile 指定了入口地址，从_startup 段中获取第一条指令。。。
对应的 crt0.S 中会指定 startup 段。

Crt0.S

V_SS_RT700_SOC_TB_1.9/testbench/blocks/soc_tb/tool_data/compiler/cm33/src/_start.c

ROM offset 0 开始存放 0x400 个中断数据，0x400 处为_startup：
CPU 上电后，第一个运行的命令就是从这里获取。Link 文件中指定了 startup 段为 CPU 获取的第一个指令

```
.section .vectors,"ax"
.globl __vectors
__vectors:
.word __SP_INIT           @ 0    000: reset stack pointer val
.word __startup+1          @ 1    004: reset start address
                           @ The +1 is to indicate Thumb mode

 ifndef MEMORY_at
 else
  .rept 254
   .word __capi_interrupt_handler+1  @ 2-255 The remaining 254 vectors //伪指令 rept 重复执行 254 次。对应
二进制 binary word2-256 都是一样的数值，为_capi_interrupt_handler+1
   .endr                         @ The +1 is to indicate Thumb mode
 endif @ MEMORY_at
.global vector_table_end
vector_table_end:

.section .text.startup,"ax" //linkfile 这段的数据被配置到了 ROM 中
.globl __startup
__startup:
  MOV    r0,#0                  @ Initialize the GPRs
  MOV    r1,#0
  MOV    r2,#0
```

```

MOV    r3,#0
MOV    r4,#0
MOV    r5,#0
MOV    r6,#0
MOV    r7,#0

CPSIE i          @ Unmask interrupts
BL    _start      @ call the C code

```

_Start.c

V_SS_RT700_SOC_TB_1.9/testbench/blocks/soc_tb/tool_data/compiler/cm33/src/_start.c

C 入口函数为_start

第一个 C 函数:

```

void _start()
{ int main_status;

    pre_main();

    main_status = main();

    post_main();

    CAPI_END_SIM(main_status);
}

```

Pre_main 、 post_main

V_SS_RT700_SOC_TB_1.10/testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_s/include/pre_post_main.h

Pre_main() 的作用是在正式进入 main 跑 test case 之前，对 clock, power, secure, syscon 等关键模块做相关配置，目的是为了 test case 能够跑起来

Post_main 的作用是 test case 完成后做一些退出的配置动作。

Main () 跑完后一定要 return 0， CAPI_END_SIM(main_status) 会根据 main 的返回值来判定 test 是否 pass

void pre_main()

```

{
//SHOW32("pre_main", "SHOW32= ", 0x32323232);
INFO("pre_main","In pre main of cm33_s.");
WR32(capi_error_counter, 0x0);//berk will remove it when we have GPR to replace

INFO("pre_main","release_all_peripheral_rst.");
release_all_peripheral_rst();

INFO("pre_main","initialize clock....");
clkctl_initial_config();

```

```

INFO("pre_main","config sense cm33 vector....");

/*to do design will change register from sense to common syscon
#ifndef CM33_M_RESET_VECTOR
W32(REG32(0x40042098), CM33_M_RESET_VECTOR>>7);//system_svtor_i
W32(REG32(0x4004209C), CM33_M_RESET_VECTOR>>7);//system_nsvtor_i
#else
W32(REG32(0x40042098), S_SSRAM_P22_BASE>>7);//system_svtor_i
W32(REG32(0x4004209C), NS_SSRAM_P22_BASE>>7);//system_nsvtor_i
#endif */

TRIGGER_SET(CM33_TBCOMM_BASE);
TRIGGER_SET(CM33_TBCOMM_BASE+1);

#ifndef CM33_NS_RESOURCES_SWITCH
INFO(FILE "secure state switching configuration");
enable_idau();
configure_sau();

if (!tt_get_addr_secure_attr((address_t)&enable_idau))
    ERROR("pre_main", "Function address is Non-secure");

init_nonsec_stack( _SP_INIT_SYMBOL_ ); // initialize NS stack pointer
WR32(NSEC(SYSCTRL_VTOR), _vectors_SYMBOL_ ); // initialize NS vector table base

MB_PUT32(SEC2NS_NSC_VNR_ADDR_MBOX, (address_t)&nsc_func_veneer);
MB_PUT32(SEC2NS_NSC_ADDR_MBOX, (address_t)&nsc_func);

MB_PUT32(NS2SEC_POST_MAIN_HDSK_MBOX, 0xdeaddead);

//sec2ns_switch_blns(0xffffe400 );
#endif
#endif
INFO("pre_main","out pre main of cm33_s.");
return;
}

```

void post_main()

```

{
INFO("post_main","In post main of cm33_s ");

#ifndef CM33_NS_RESOURCES_SWITCH
uint32_t rdata;
MB_GET32(NS2SEC_POST_MAIN_HDSK_MBOX, rdata);
SHOW32("CM33_POST_MAIN","rdata = ",rdata);
FLAG_WAIT(9,1);
INFO("post_main","End post main  ");
//if (rdata>0xbabeface)
//    ERROR("post_main", "NS code did not finish executing");
#endif
SHOW32("post_main", "capi_error_counter= ", capi_error_counter);
}

```

Main()

Test case 中的 main 函数

CM33 NS boot

Testbench V1.20.1.1 为例

分两步，

1. testbench preload CM33 secure 的 code 到 ROM 跟 secure SRAM 中。 Preload non-secure demo 到 NS ram 中。
2. Secure demo pre_main, 配置 non-secure 的 vector, 配置 SAU, IDAU 对 non-secure 空间进行配置
执行 sanity_cm_init.c 中的 main 函数, call Non-secure 的 C code。 _start.c

Code 分布：

CM33 secure demo: 汇编-》ROM, code、data -》SRAM16, nsc-》sram0 末尾

CM33 NS demo: code, data-》SRAM4 NS , vector -》SRAM1 NS

生成 linkfile

Non-secure demo linkfile

```
cat
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/compiler/cm33/cm33_ns/ldscripts/cm33_ns.lnk | m4 -
I/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/compiler/include -P | cpp -P -Dcm33_ns -DCM33_NS_RESOURCES -DCM33_NS -DVMEM=sram_1 -
DXMEM=sram_4 -DUMEM=sram_4 -DSMEM=sram_4 -DDMEM=sram_4 -DSTRAMEM=sram_4 -
I/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/compiler/include >
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/verilog/verilog_FILENOAMES_apc7092.cn-sha01.nxp.com_24999/dma/cm33/cm33_ns/CM33_ns_local.lnk
```

cm33_ns.lnk

生成的 linkfile， CM33_ns_local.lnk 。 SRAM1 放 vector， sram4 放 code

```
File Edit Tools Syntax Buffers Window Help
File Edit Tools Syntax Buffers Window Help
1 MEMORY
2 {
3     sram_0 : org = (0x20000000) , len = (0x8000 )
4     sram_1 : org = ((0x20008000)) , len = (0x8000 )
5     sram_2 : org = (0x20010000) , len = (0x8000 )
6     sram_3 : org = (0x20018000) , len = (0x8000 )
7     sram_4 : org = (0x20020000) , len = (0x10000 )
8     sram_5 : org = (0x20030000) , len = (0x10000 )
9     sram_6 : org = (0x20040000) , len = (0x20000 )
10    sram_7 : org = (0x20060000) , len = (0x20000 )
11    sram_16 : org = (0x20500000) , len = (0x40000 )
12    sram_22 : org = (0x205A0000) , len = (0x10000 )
13    sram_23 : org = (0x205B0000) , len = (0x10000 )
14    boot_rom : org = (0x03000000) , len = (0x40000 )
15    str_mem : org = (0x32000000) , len = ((0x00020000))
16    priv_ram : org = ((0x2000FFFF)-((0x00000800))-((0x00000800)) +1) , len = ((0x00000800))
17 }
18 SECTIONS
19 {
20     strings : { . = ALIGN(8) ; *(.rodata.*str*); . = ALIGN(8); } > str_mem= 0
21     .bootup : { . = ALIGN(8) ; *(.bootup*); . = ALIGN(8); } > boot_rom = 0xff
22     .vectors : { . = ALIGN(8) ; *(.vectors*); . = ALIGN(8); } > sram_1 AT > sram_1 = 0xff
23     .startup : { . = ALIGN(8) ; *(.text.startup*); . = ALIGN(8); } > sram_4 AT > sram_4 = 0xff
24     .rodata : { . = ALIGN(8) ; *(.rodata*); . = ALIGN(8); } > sram_4 AT > sram_4 = 0xff
25     .text : { . = ALIGN(8) ; *(.text*); . = ALIGN(8); } > sram_4 AT > sram_4 = 0xff
26     .data : { . = ALIGN(8) ; *(.data*); . = ALIGN(8); } > sram_4 AT > sram_4 = 0
27     .bss : { . = ALIGN(8) ; *(.bss*); . = ALIGN(8); } > sram_4 AT > sram_4 = 0
28     .sbss : { . = ALIGN(8) ; *(.sbss*); . = ALIGN(8); } > sram_4 AT > sram_4 = 0
29     .priv_mem : { . = ALIGN(4) ; KEEP(*(.priv_mem*)); . = ALIGN(4); } > priv_ram AT > priv_ram = 0
30     stack : { . = . + (((0x00000800))-4); LONG(0); } > sram_4 AT > sram_4 = 0
31 }
32 __SP_INIT = ADDR(stack) + SIZEOF(stack) - 8;
33 __START_ADDR = ADDR(.startup);
34 ENTRY(__START_ADDR);
```

Secure demo linkfile

```
### C-Stimulus      :  
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench(blocks/soc_tb/tool_data  
/compiler/include/sanity cm33 init.c
```

```

### C-RunTime          :
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/compiler/cm33/cm33_s/src/crt0.s

cat
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/compiler/cm33/cm33_s/ldscripts/cm33_s.lnk | m4 -
l/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/compiler/include -P | cpp -P -Dcm33 -DCM33_RESOURCES -DCM33 -DVMEM=boot_rom -
DSTRMEM=str_mem -DXMEM=sram_16 -DUMEM=sram_16 -DSMEM=sram_16 -DDMEM=sram_16 -
DSTREAMEM=sram_16 -
l/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/compiler/include>
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/verilog/verilog_FILERAMES_apc7092.cn-sha01.nxp.com_24999/dma/cm33/cm33_s/CM33_local.lnk

```

SRAM16 放 code, data。 汇编启动代码放置在 ROM 中

```

cm33_s.lnk = (/home/imxrt700_verif1/richard_nxa28190/V_SS_RT...blocks/soc_tb/tool_data/compiler/cm33/cm33_s/ldscripts) - GVIM9
File Edit Tools Syntax Buffers Window Help
49 * .bss      - segment for uninitialized data      (standard GCC)
50 * .sbss     - segment for "small" uninitialized data (standard GCC)
51 * stack     - segment for stack
52 * efuse     - segment for efuse array
53 *   Allocate space for the stack. Doing this provides a check that
54 *   there is sufficient space available in RAM to fit the global
55 *   variables plus the desired amount of stack. If there is not
56 *   enough space, you should get a linker error that looks like:
57 *   "section stack [0400 -> 07ff] overlaps section .data [0000 -> 0407]"
58 */
59
60 SECTIONS
61 {
62     strings      : { . = ALIGN(8) ; *(.rodata.*str*);           . = ALIGN(8); } > STRMEM      AT > STRMEM= 0
63     efuse       : { . = ALIGN(8); KEEP(*(.efuse*));           . = ALIGN(4); } > sram_7 = 0
64     .bootup    : { . = ALIGN(8) ; *(.bootup*);             . = ALIGN(8); } > boot_rom = 0xff
65     .vectors   : { . = ALIGN(8) ; *(.vectors*);            . = ALIGN(8); } > VMEM        AT > VMEM = 0xff
66     .startup   : { . = ALIGN(8) ; *(.text.startup*);        . = ALIGN(8); } > boot_rom = 0xff
67     .rodata    : { . = ALIGN(8) ; *(.rodata*);              . = ALIGN(8); } > XMEM        AT > XMEM = 0xff
68     .text      : { . = ALIGN(8) ; *(.text*);                 . = ALIGN(8); } > XMEM        AT > XMEM = 0xff
69     .data      : { . = ALIGN(8) ; *(.data*);                . = ALIGN(8); } > DMEM        AT > DMEM = 0
70     .bss       : { . = ALIGN(8) ; *(.bss*);                 . = ALIGN(8); } > DMEM        AT > DMEM = 0
71     .sbss      : { . = ALIGN(8) ; *(.sbss*);               . = ALIGN(8); } > DMEM        AT > DMEM = 0
72     .nsc_mem   : { . = ALIGN(4) ; KEEP(*(.nsc_mem*));       . = ALIGN(4); } > nsc_ram    AT > nsc_ram = 0
73     .priv_mem  : { . = ALIGN(4) ; KEEP(*(.priv_mem*));       . = ALIGN(4); } > priv_ram   AT > priv_ram = 0
74     stack      : { . = . + (STACK_SIZE-4);                  LONG(0); } > SMEM        AT > SMEM = 0
75 }
76
77
78
79 /* Set the initial value for the stack pointer */
80 /* Leave 8 bytes at the top for the CAPI */
81 __SP_INIT     = ADDR(stack) + SIZEOF(stack) - 8;
82
83 /* GDB needs to be told where the entry-point is */
84 __START_ADDR  = ADDR(.startup);
85 ENTRY(__START_ADDR);
86

```

```

1 MEMORY
2 {
3     alias_sram_0 : org = (0x10000000) , len = (0x8000 )
4     sram_0 : org = (0x30000000) , len = (0x8000 )
5     sram_1 : org = (0x30008000) , len = (0x8000 )
6     sram_2 : org = (0x30010000) , len = (0x8000 )
7     sram_3 : org = (0x30018000) , len = (0x8000 )
8     alias_sram_4 : org = (0x10020000) , len = (0x10000 )
9     sram_4 : org = (0x30020000) , len = (0x10000 )
10    sram_5 : org = (0x30030000) , len = (0x10000 )
11    sram_6 : org = (0x30040000) , len = (0x20000 )
12    sram_7 : org = (0x30060000) , len = (0x20000 )
13    sram_16 : org = (0x30500000) , len = (0x40000 )
14    boot_rom : org = (0x13000000) , len = (0x40000 )
15    str_mem : org = (0x31000000) , len = ((0x00020000))
16    str_ram : org = ((0x20080000)) , len = ((0x00010000))
17    nsc_ram : org = ((0x20007FFF)-((0x00000800))+1) , len = ((0x00000800))
18    priv_ram : org = ((0x30017FFF)-((0x00000800))-((0x00000800)) +1) , len = ((0x00000800))
19 }
20 SECTIONS
21 {
22     strings : { . = ALIGN(8) ; *(.rodata.*str*); . = ALIGN(8); } > str_mem AT > str_mem= 0
23     efuse : { KEEP(*(.efuse*)); . = ALIGN(4); } > sram_7 = 0
24     .bootup : { . = ALIGN(8) ; *(.bootup*); . = ALIGN(8); } > boot_rom = 0xff
25     .vectors : { . = ALIGN(8) ; *(.vectors*); . = ALIGN(8); } > boot_rom AT > boot_rom = 0xff
26     .startup : { . = ALIGN(8) ; *(.text.startup*); . = ALIGN(8); } > boot_rom AT > boot_rom = 0xff
27     .rodata : { . = ALIGN(8) ; *(.rodata*); . = ALIGN(8); } > sram_16 AT > sram_16 = 0xff
28     .text : { . = ALIGN(8) ; *(.text*); . = ALIGN(8); } > sram_16 AT > sram_16 = 0xff
29     .data : { . = ALIGN(8) ; *(.data*); . = ALIGN(8); } > sram_16 AT > sram_16 = 0
30     .bss : { . = ALIGN(8) ; *(.bss*); . = ALIGN(8); } > sram_16 AT > sram_16 = 0
31     .sbss : { . = ALIGN(8) ; *(.sbss*); . = ALIGN(8); } > sram_16 AT > sram_16 = 0
32     nsc_mem : { . = ALIGN(4) ; KEEP(*(.nsc_mem*)); . = ALIGN(4); } > nsc_ram AT > nsc_ram = 0
33     .priv_mem : { . = ALIGN(4) ; KEEP(*(.priv_mem*)); . = ALIGN(4); } > priv_ram AT > priv_ram = 0
34     stack : { . = . + ((0x00000800))-4); LONG(0); } > sram_16 AT > sram_16 = 0
35 }
36 __SP_INIT = ADDR(stack) + SIZEOF(stack) - 8;
37 __START_ADDR = ADDR(.startup);
38 ENTRY(__START_ADDR);

```

ENTRY(__START_ADDR) //告诉链接器程序的启动地址，

编译

Non-secure demo

编译.c

```

arm-none-eabi-g++ -D DEF_LC_NXP_FAB_MODE_DEFAULT=1 -D ANATOP_ID=ANATOP_INST0 -D
SRC=S_SSRAM_P10_BASE -D DST=S_SSRAM_P10_BASE -D SRC_NS=NS_SSRAM_P10_BASE -D
DST_NS=NS_SSRAM_P10_BASE -D LEN=0x80000 -D ADDR_START=0      -DCM33_NS_RESOURCES=1 -
Dcm33_ns -DCM33_NS_RESOURCES -DCM33_NS -DVMEM=sram_1 -DXMEM=sram_4 -DUMEM=sram_4
-DSMEM=sram_4 -DDMEM=sram_4 -DSTRAMEM=sram_4          -I
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
/compiler/include/ -DCM33 -DCM33_NS_RESOURCES=1 -DCPRINT_REGS_BASE=0x2001FFE0 -
DCPRINT_CMD_OFFSET=0x00 -DCPRINT_ARG_OFFSET=0x8 -DCPRINT_RSP_OFFSET=0x4 -Dcm33_ns -
DCM33_NS_RESOURCES -DCM33_NS   -DVMEM=sram_1 -DXMEM=sram_4 -DUMEM=sram_4 -
DSMEM=sram_4 -DDMEM=sram_4 -DSTRAMEM=sram_4 -DDMA_ID=INST0 -D CAPI_INCLUDE_C=0 -D
CAPI_INCLUDE_STARTUP=0 -D STARTUP_STIM=pre_main -D SHUTDOWN_STIM=post_main -D
CORE_NUM=0 -D PH_BASE=0x37FFFFFF -D CPRINT_REGS_BASE=0x37FFFFFF -D
TARGET_MEM_START=0x34000000 -D TARGET_MEM_SIZE=0x00070000 -D TARGET_MEM_WIDTH=64 - -
MD -c

```

```
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data  
/compiler/cm33/src/_start.c -o _start.o
```

编译.s

```
arm-none-eabi-as --defsym DEF_LC_NXP_FAB_MODE_DEFAULT=1 --defsym  
ANATOP_ID=ANATOP_INST0 --defsym SRC=S_SSRAM_P10_BASE --defsym DST=S_SSRAM_P10_BASE --  
defsym SRC_NS=NS_SSRAM_P10_BASE --defsym DST_NS=NS_SSRAM_P10_BASE --defsym LEN=0x80000  
--defsym ADDR_START=0 -mthumb -mcpu=cortex-m33 --defsym CM33_NS_RESOURCES=1 --defsym  
cm33_ns=1 --defsym CM33_NS_RESOURCES=1 --defsym CM33_NS=1 --defsym cm33_ns=1 --defsym  
CM33_NS=1 --defsym VMEM=sram_1 --defsym XMEM=sram_4 --defsym UMEM=sram_4 --defsym  
SMEM=sram_4 --defsym DMEM=sram_4 --defsym STRAMEM=sram_4 --defsym DMA_ID=INST0 --  
defsym CAPI_INCLUDE_C=0 --defsym CAPI_INCLUDE_STARTUP=0 --defsym STARTUP_STIM=pre_main --  
defsym SHUTDOWN_STIM=post_main --defsym CORE_NUM=0 --defsym PH_BASE=0x37FFFFF0 --defsym  
CPRINT_REGS_BASE=0x37FFFFF0 --defsym TARGET_MEM_START=0x34000000 --defsym  
TARGET_MEM_SIZE=0x00070000 --defsym TARGET_MEM_WIDTH=64  
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data  
/compiler/cm33/cm33_ns/src/crt0.s -o crt0.o
```

Secure demo

编译.c

```
arm-none-eabi-g++ -D DEF_LC_NXP_FAB_MODE_DEFAULT=1 -D ANATOP_ID=ANATOP_INST0 -D  
SRC=S_SSRAM_P10_BASE -D DST=S_SSRAM_P10_BASE -D SRC_NS=NS_SSRAM_P10_BASE -D  
DST_NS=NS_SSRAM_P10_BASE -D LEN=0x80000 -D ADDR_START=0 -DCM33_RESOURCES=1 -DNONE  
-DNONE -DNONE -Dcm33_s -DCM33_RESOURCES -DCM33 -DVMEM=boot_rom -DXMEM=sram_16 -  
DSTRMEM=str_mem -DUMEM=sram_16 -DSMEM=sram_16 -DDMEM=sram_16 -DSTRAMEM=sram_16 -D  
CM33_NS_RESOURCES_SWITCH=1 -I  
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data  
/compiler/include/ -I DTEST_NAME=dma3_slave_access_sram_cm33_ns_inst0 -DVECTOR_SET=dma -  
DMAIN_TEST=main -DDEF_LC_NXP_FAB_MODE_DEFAULT=1 -DANATOP_ID=ANATOP_INST0 -  
DSRC=S_SSRAM_P10_BASE -DDST=S_SSRAM_P10_BASE -DSRC_NS=NS_SSRAM_P10_BASE -  
DDST_NS=NS_SSRAM_P10_BASE -DLEN=0x80000 -DADDR_START=0 -fdata-sections -fno-exceptions -  
std=c99 -Wall -Werror=declaration-after-statement -Wno-unused-function -ffunction-sections -falign-  
functions=4 -fno-zero-initialized-in-bss -O1 -  
I/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/vectors/d  
ma/stimulus -DACTIVE_CORES=0x3 -DACTIVE_CBFMS=0x0 -Wfatal-errors -Werror=implicit-function-  
declaration -Wmain -Werror=main -D ASM=__asm__ -fno-zero-initialized-in-bss -mthumb -  
mcpu=cortex-m33 -nostdlib -ffunction-sections -g -mthumb -march=armv8-m.main -mfpu=fpv5-sp-  
d16 -mfloat-abi=hard -fpermissive -Wno-narrowing -DCM33_RESOURCES=1 -
```

```

DCPRINT_REGS_BASE=0x3001FFE0 -DCPRINT_CMD_OFFSET=0x00 -DCPRINT_ARG_OFFSET=0x8 -
DCPRINT_RSP_OFFSET=0x4 -DADDR_START_SYMBOL_=0x00000000 -
DANATOP_ID_SYMBOL_=0x00000000 -DCAPI_INCLUDE_C_SYMBOL_=0x00000000 -
DCAPI_INCLUDE_STARTUP_SYMBOL_=0x00000000 -DCORE_NUM_SYMBOL_=0x00000000 -
DDMA_ID_SYMBOL_=0x00000000 -DDMEM_SYMBOL_=0x00000000 -DDST_SYMBOL_=0x00000000 -
-DDST_NS_SYMBOL_=0x00000000 -DSHUTDOWN_STIM_SYMBOL_=0x00000000 -
DSMEM_SYMBOL_=0x00000000 -DSRC_SYMBOL_=0x00000000 -DSRC_NS_SYMBOL_=0x00000000 -
DSTARTUP_STIM_SYMBOL_=0x00000000 -DSTREAMEM_SYMBOL_=0x00000000 -
DUMEM_SYMBOL_=0x00000000 -DVMEM_SYMBOL_=0x00000000 -DXMEM_SYMBOL_=0x00000000 -
-DCM33_NS_SYMBOL_=0x00000001 -DCM33_NS_SYMBOL_=0x00000001 -
DCM33_NS_RESOURCES_SYMBOL_=0x00000001 -DCM33_NS_RESOURCES_SYMBOL_=0x00000001 -
DDEF_LC_NXP_FAB_MODE_DEFAULT_SYMBOL_=0x00000001 -Dcm33_ns_SYMBOL_=0x00000001 -
Dcm33_ns_SYMBOL_=0x00000001 -DTARGET_MEM_MSB_SYMBOL_=0x0000003f -
DTARGET_MEM_WIDTH_SYMBOL_=0x00000040 -DTARGET_MEM_WIDTH_SYMBOL_=0x00000040 -
DTARGET_MEM_STACK_SIZE_SYMBOL_=0x00000800 -DTARGET_MEM_SIZE_SYMBOL_=0x00070000 -
DTARGET_MEM_SIZE_SYMBOL_=0x00070000 -DLEN_SYMBOL_=0x00080000 -
D_vectors_SYMBOL_=0x20008000 -Dvector_table_end_SYMBOL_=0x20008400 -
D_START_ADDR_SYMBOL_=0x20020000 -D_startup_SYMBOL_=0x20020000 -
D_end_SYMBOL_=0x20020016 -D_start_SYMBOL_=0x20020020 -
DCAPI_END_SIM_SYMBOL_=0x20020038 -DUNIMPLEMENTED_ISR_SYMBOL_=0x20020058 -
DPROCESS_INTERRUPT_SYMBOL_=0x20020098 -Dcore_enable_irq_SYMBOL_=0x200200c0 -
D_capi_interrupt_handler_SYMBOL_=0x200200f0 -D_hard_fault_recovery_SYMBOL_=0x2002013c -
D_chk_prec_data_err_SYMBOL_=0x20020146 -D_chk_instr_type_SYMBOL_=0x2002014c -
D_rtn_addr_SYMBOL_=0x2002016a -D_chk_instr_err_SYMBOL_=0x2002016e -
D_end_isr_SYMBOL_=0x20020194 -Dset_interrupt_secure_state_SYMBOL_=0x200201b4 -
Djump_to_nsc_SYMBOL_=0x200201cc -D_NS_RETURN_ADDR_SYMBOL_=0x200201ea -
Ddma3_int_isr_SYMBOL_=0x200201fc -Ddma3_err_int_isr_SYMBOL_=0x20020320 -
Ddma_populate_request_SYMBOL_=0x200203b4 -Dipc_dma0_assign_SYMBOL_=0x200203c4 -
Dipc_dma0_release_SYMBOL_=0x2002040c -Dipc_dma1_assign_SYMBOL_=0x20020454 -
Dipc_dma1_release_SYMBOL_=0x2002049c -Dipc_dma2_assign_SYMBOL_=0x200204e4 -
Dipc_dma2_release_SYMBOL_=0x2002052c -Dipc_dma3_assign_SYMBOL_=0x20020574 -
Ddma3_pcc_assign_SYMBOL_=0x200205bc -Dipc_dma3_release_SYMBOL_=0x20020614 -
Ddma3_pcc_release_SYMBOL_=0x2002065c -Dipc_dma0_rstb_set_SYMBOL_=0x200206b4 -
Dipc_dma0_rstb_clr_SYMBOL_=0x200206fc -Dipc_dma1_rstb_set_SYMBOL_=0x20020744 -
Dipc_dma1_rstb_clr_SYMBOL_=0x2002078c -Dipc_dma2_rstb_set_SYMBOL_=0x200207d4 -
Dipc_dma2_rstb_clr_SYMBOL_=0x2002081c -Dipc_dma3_rstb_set_SYMBOL_=0x20020864 -
Ddma3_reset_negate_SYMBOL_=0x200208ac -Dipc_dma3_rstb_clr_SYMBOL_=0x20020904 -
Ddma3_reset_assert_SYMBOL_=0x2002094c -Ddma3_setup_int_SYMBOL_=0x200209a4 -
Ddma3_check_int_SYMBOL_=0x200209e4 -Ddma3_setup_err_int_SYMBOL_=0x20020a38 -
Ddma3_check_err_int_SYMBOL_=0x20020a78 -Ddma3_enable_err_int_SYMBOL_=0x20020ad0 -
Ddma3_validate_channel_SYMBOL_=0x20020b1c -Ddma3_setup_complex_SYMBOL_=0x20020b68 -
Ddma3_setup_hw_SYMBOL_=0x200212b4 -Ddma3_start_sw_SYMBOL_=0x20021304 -
Ddma3_prepare_source_data_SYMBOL_=0x20021350 -
Ddma3_check_destination_data_SYMBOL_=0x20021368 -Dsoc_new_request_SYMBOL_=0x20021bbc -

```

```

Dsoc_allocate_ip_SYMBOL_=0x20021c28 -Dpre_main_SYMBOL_=0x20021c8c -
Dpost_main_SYMBOL_=0x20021d18 -Dmain_SYMBOL_=0x20021d68 -
Dns_return_addr_SYMBOL_=0x20022270 -Dinit_ip_request_SYMBOL_=0x20022274 -
DtrSize_SYMBOL_=0x2002238c -DINTERRUPT_CONTEXT_TABLE_SYMBOL_=0x200223a0 -
DINTERRUPT_JUMP_TABLE_SYMBOL_=0x200227a0 -Dns2sec_comm_isr_msg_SYMBOL_=0x20022ba0 -
Dnsc_addr_SYMBOL_=0x20022ba4 -Dnsc_veneer_addr_SYMBOL_=0x20022ba8 -
D_SP_INIT_SYMBOL_=0x20023470 -DTARGET_MEM_START_SYMBOL_=0x34000000 -
DTARGET_MEM_START_SYMBOL_=0x34000000 -DTARGET_MEM_END_SYMBOL_=0x3406ffff -
DCPRINT_REGS_BASE_SYMBOL_=0x37fffff0 -DPH_BASE_SYMBOL_=0x37fffff0 -DNONE -DNONE -
DNONE -Dcm33_s -DCM33_RESOURCES -DCM33 -DVMEM=boot_rom -DXMEM=sram_16 -
DSTRMEM=str_mem -DUMEM=sram_16 -DSMEM=sram_16 -DDMEM=sram_16 -DSTRAMEM=sram_16 -
DCM33_NS_RESOURCES_SWITCH=1 -D CAPI_INCLUDE_C=0 -D CAPI_INCLUDE_STARTUP=0 -D
STARTUP_STIM=pre_main -D SHUTDOWN_STIM=post_main -D CORE_NUM=1 -D PH_BASE=0x3001FFE0
-D CPRINT_REGS_BASE=0x3001FFE0 -D TARGET_MEM_START=0x34070000 -D
TARGET_MEM_SIZE=0x00070000 -D TARGET_MEM_WIDTH=64 -
I/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench(blocks/soc_tb/tool_data
/compiler/cm33/cm33_s/include - -MD -c
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench(blocks/soc_tb/tool_data
/compiler/include/sanity_cm33_init.c -o sanity_cm33_init.o

```

编译.S

```

arm-none-eabi-as --defsym DEF_LC_NXP_FAB_MODE_DEFAULT=1 --defsym
ANATOP_ID=ANATOP_INST0 --defsym SRC=S_SSRAM_P10_BASE --defsym DST=S_SSRAM_P10_BASE --
defsym SRC_NS=NS_SSRAM_P10_BASE --defsym DST_NS=NS_SSRAM_P10_BASE --defsym LEN=0x80000
--defsym ADDR_START=0 -mthumb -mcpu=cortex-m33 --defsym CM33=1 --defsym
CM33_RESOURCES=1 --defsym cm33_s=1 --defsym CM33_RESOURCES=1 --defsym CM33=1 --defsym
cm33_s=1 --defsym CM33=1 --defsym VMEM=boot_rom --defsym XMEM=sram_16 --defsym
STRMEM=str_mem --defsym UMEM=sram_16 --defsym SMEM=sram_16 --defsym DMEM=sram_16 --
defsym STRAMEM=sram_16 --defsym CM33_NS_RESOURCES_SWITCH=1 --defsym CAPI_INCLUDE_C=0 -
-defsym CAPI_INCLUDE_STARTUP=0 --defsym STARTUP_STIM=pre_main --defsym
SHUTDOWN_STIM=post_main --defsym CORE_NUM=1 --defsym PH_BASE=0x3001FFE0 --defsym
CPRINT_REGS_BASE=0x3001FFE0 --defsym TARGET_MEM_START=0x34070000 --defsym
TARGET_MEM_SIZE=0x00070000 --defsym TARGET_MEM_WIDTH=64
I/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench(blocks/soc_tb/tool_data
/compiler/cm33/cm33_s/src/crt0.s -o crt0.o

```

链接

将.o 文件按照 link 文件的要求整合起来，生成 map 跟 elf 文件

Non-secure demo

```
ESC[0;32m[crt0.o] /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/arm-none-eabi-as --defsym DEF_LC_NXP_FAB_MODE_DEFAULT=1 --defsym ANATOP_ID=ANATOP_INST0 --defsym SRC=S_SSRAM  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.elf] _start.o capi.o capi_int.o capi_interrupt_handler.o hard_fault_recovery.o secure_portal.o  
arm-none-eabi-ld _start.o capi.o capi_int.o capi_interrupt_handler.o hard_fault_recovery.o secure_portal.o  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.nm] dma3_slave_access_sram_cm33_ns_inst0.elfESC[0m  
arm-none-eabi-nm --demangle -n --demangle -n dma3_slave_access_sram_cm33_ns_inst0.elf > dma3_slave_access_sram_cm33_ns_inst0.nm  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.lst] dma3_slave_access_sram_cm33_ns_inst0.elfESC[0m  
arm-none-eabi-objdump --disassemble-all --disassemble-zeroes --wide --source -hlds dma3_slave_access_sram_cm33_ns_inst0.lst  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.srec] dma3_slave_access_sram_cm33_ns_inst0.elfESC[0m  
arm-none-eabi-objcopy --strip-all -O srec dma3_slave_access_sram_cm33_ns_inst0.elf dma3_slave_access_sram_cm33_ns_inst0.srec  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.bin] dma3_slave_access_sram_cm33_ns_inst0.elfESC[0m  
arm-none-eabi-objcopy --strip-all -O binary dma3_slave_access_sram_cm33_ns_inst0.elf dma3_slave_access_sram_cm33_ns_inst0.bin  
  
arm-none-eabi-ld _start.o capi.o capi_int.o capi_interrupt_handler.o hard_fault_recovery.o secure_portal.o  
trustzone_api.o dma3_slave_access.o crt0.o -  
L/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_ns/lib -  
L/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/lib --whole-archive --no-whole-archive -nostdlib -gc-sections --gc-sections -nostdlib --entry=__vectors --defsym TARGET_MEM_WIDTH=64 --defsym  
TARGET_MEM_START=0x34000000 --defsym TARGET_MEM_STACK_SIZE=0x800 --defsym  
TARGET_MEM_END=0x3406ffff --defsym TARGET_MEM_MSB=63 --defsym  
TARGET_MEM_SIZE=0x00070000 -T./CM33_ns_local.lnk -Map  
dma3_slave_access_sram_cm33_ns_inst0.map -o dma3_slave_access_sram_cm33_ns_inst0.elf
```

Secure demo

```
ESC[0;32m[crt0.o] /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/arm-none-eabi-as --defsym DEF_LC_NXP_FAB_MODE_DEFAULT=1 --defsym ANATOP_ID=ANATOP_INST0 --defsym SRC=S_SSRAM  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.elf] _start.o capi.o capi_int.o capi_interrupt_handler.o hard_fault_recovery.o secure_portal.o  
arm-none-eabi-ld _start.o capi.o capi_int.o capi_interrupt_handler.o hard_fault_recovery.o secure_portal.o  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.nm] dma3_slave_access_sram_cm33_ns_inst0.elfESC[0m  
arm-none-eabi-nm --demangle -n --demangle -n dma3_slave_access_sram_cm33_ns_inst0.elf > dma3_slave_access_sram_cm33_ns_inst0.nm  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.lst] dma3_slave_access_sram_cm33_ns_inst0.elfESC[0m  
arm-none-eabi-objdump --disassemble-all --disassemble-zeroes --wide --source -hlds dma3_slave_access_sram_cm33_ns_inst0.lst  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.srec] dma3_slave_access_sram_cm33_ns_inst0.elfESC[0m  
arm-none-eabi-objcopy --strip-all -O srec dma3_slave_access_sram_cm33_ns_inst0.elf dma3_slave_access_sram_cm33_ns_inst0.srec  
ESC[0;32m[dma3_slave_access_sram_cm33_ns_inst0.bin] dma3_slave_access_sram_cm33_ns_inst0.elfESC[0m  
arm-none-eabi-objcopy --strip-all -O binary dma3_slave_access_sram_cm33_ns_inst0.elf dma3_slave_access_sram_cm33_ns_inst0.bin  
  
arm-none-eabi-ld _start.o capi.o capi_int.o capi_interrupt_handler.o hard_fault_recovery.o secure_portal.o  
trustzone_api.o sanity_cm33_init.o crt0.o -  
L/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_s/lib -  
L/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data
```

```
a/compiler/cm33/lib --whole-archive    --no-whole-archive -nostdlib -gc-sections --gc-sections -nostdlib --entry=__vectors --defsym TARGET_MEM_WIDTH=64 --defsym  
TARGET_MEM_START=0x34070000 --defsym TARGET_MEM_STACK_SIZE=0x800 --defsym  
TARGET_MEM_END=0x340dffff --defsym TARGET_MEM_MSB=63 --defsym  
TARGET_MEM_SIZE=0x00070000 -T./CM33_local.lnk -Map dma3_slave_access_sram_cm33_ns_inst0.map  
-o dma3_slave_access_sram_cm33_ns_inst0.elf
```

启动流程分析

Loadfile

Secure demo Load into ROM, sram16

```
UVM_INFO @ 1085131.913200 ns --: [testbench.load_memory_cm33] load_memory_string: ====  
File :  
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data  
/verilog/verilog_FILERAMES_apc7092.cn-  
sha01.nxp.com_24999/CM33_dma3_slave_access_sram_cm33_ns_inst0.rom.hex
```

Non-secure demo, load into SRAM1,SRAM4

```
UVM_INFO @ 1085131.913200 ns --: [testbench.load_memory_cm33_ns] load_memory_string: ====  
File :  
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data  
/verilog/verilog_FILERAMES_apc7092.cn-  
sha01.nxp.com_24999/CM33_NS_dma3_slave_access_sram_cm33_ns_inst0.rom.hex
```

Boot into non-secure demo

Secure demo 从 CM33 BOOT 起来后，执行 pre_main 函数，跳转之前 trustzone 会对 ram 进行 non-secure 的配置

Non-secure 配置

1. 配置 non-secure vctor, vector_SYMBOL. Secure demo 在编译的时候定义了这个变量, 为 non-secure demo 的 vector address

```
0000 -DLEN_SYMBOL_=0x00080000 -D_vectors_SYMBOL_=0x20008000 -Dvector_table_end_SYMBOL_=0x20008400 -D_START_ADDR_SYMBOL_=0x20020000 -D_startup,
```

2. Config SAU,idau 配置 NON-SECURE address, NON-SECURE ram

```
122 #ifdef CM33_NS_RESOURCES_SWITCH
123     INFO(FILE, "secure state switching configuration");
124     enable_idau();
125     configure_sau();
126
127     if (!tt_get_addr_secure_attr((address_t)(&enable_idau)))
128         ERROR("pre_main", "Function address is Non-secure");
129
130     init_nonsec_stack( __SP_INIT_SYMBOL__ ); // initialize NS stack pointer
131     WR32(NSEC(SYSCTRL_VTOR), __vectors_SYMBOL__ ); // initialize NS vector table base
132
133     MB_PUT32(SEC2NS_NSC_VNR_ADDR_MBOX, (address_t)&nsc_func_veneer);
134     MB_PUT32(SEC2NS_NSC_ADDR_MBOX, (address_t)&nsc_func);
135
136     MB_PUT32(NS2SEC_POST_MAIN_HDSK_MBOX, 0xdeaddead);
137
138     //sec2ns_switch_blxns(0xffffe400 );
139 #endif
140 INFO("pre_main","out pre main of cm33_s.");
141     return;
142
143 }
```

Boot into non-secure demo

3. 执行 Sanity_cm33_init.c 中的 main 函数。

CM33_NS_RESOURCES_SWITCH#ns_source_switch 存在, 跳转到 non-secure demo 中, 跳转地址
[#start_symbol](#) (NON-SECURE code 的_start.c)

```
#ifdef CM33_NS_RESOURCES_SWITCH
    INFO(FILE, " Jump to CM33 Non Secure start...");
3000207e: 4b06      ldr r3, [pc, #24] ; (30002098 <main+0x1c>)
30002080: 4a06      ldr r2, [pc, #24] ; (3000209c <main+0x20>)
30002082: 601a      str r2, [r3, #0]
30002084: 4a06      ldr r2, [pc, #24] ; (300020a0 <main+0x24>)
30002086: 601a      str r2, [r3, #0]
30002088: 2215      movs r2, #21
3000208a: 3b08      subs r3, #8
3000208c: 701a      strb r2, [r3, #0]
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.11/testbench(blocks/soc_tb/tool_data/compiler/include/sanity_cm
33_init.c:314
    sec2ns_switch_blxns(_start_SYMBOL_);
3000208e: 4805      ldr r0, [pc, #20] ; (300020a4 <main+0x28>)
30002090: f7fe fa64  bl 3000055c <Z19sec2ns_switch_blxns>
/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.11/testbench(blocks/soc_tb/tool_data/compiler/include/sanity_cm
33_init.c:323
    INFO(FILE, " Jump to CM33_M Non Secure start...");
    sec2ns_switch_blxns(_start_SYMBOL_);
#endif
```

```

290 #ifndef CM33_NS_LPCG_ACCESS_MQS
291     INFO("CM33_NS_ACCESS_CCM_REGS", " allow cm33 nonsecure mode access LP register ...");
292     BSET32(REG32(CLK_ENABLE_MQS1_DIRECT+CLK_LPCG_AUTHEN),BIT9);
293     BSET32(REG32(CLK_ENABLE_MQS2_DIRECT+CLK_LPCG_AUTHEN),BIT9);
294 #endif
295 #ifndef CM33_NS_LPCG_ACCESS_AUD_XCVR
296     INFO("CM33_NS_ACCESS_CCM_REGS", " allow cm33 nonsecure mode access LP register ...");
297     BSET32(REG32(CLK_ENABLE_AUD_XCVR_DIRECT+CLK_LPCG_AUTHEN),BIT9);
298 #endif
299
300 #ifndef CM33_NS_GPIO_REGS_ACCESS
301     INFO("CM33_NS_GPIO_REGS_ACCESS", " allow cm33 nonsecure mode access GPIO register ...");
302     WREM32(REG32(GPIO1_REGS_BASE_ADDRESS +0x10),0xFFFFFFFF,0x0000FFFF,0xFFFFFFFF); //pin allow non secure access
303     WREM32(REG32(GPIO2_REGS_BASE_ADDRESS +0x10),0xFFFFFFFF,0x3FFFFFFF,0xFFFFFFFF); //pin allow non secure access
304     WREM32(REG32(GPIO3_REGS_BASE_ADDRESS +0x10),0xFFFFFFFF,0x3FFFFFFF,0xFFFFFFFF); //pin allow non secure access
305     WREM32(REG32(GPIO4_REGS_BASE_ADDRESS +0x10),0xFFFFFFFF,0x3FFFFFFF,0xFFFFFFFF); //pin allow non secure access
306     WREM32(REG32(GPIO1_REGS_BASE_ADDRESS +0x14),0x3,0x3,0xFFFFFFFF); //interrupt allow non secure access
307     WREM32(REG32(GPIO2_REGS_BASE_ADDRESS +0x14),0x3,0x3,0xFFFFFFFF); //interrupt allow non secure access
308     WREM32(REG32(GPIO3_REGS_BASE_ADDRESS +0x14),0x3,0x3,0xFFFFFFFF); //interrupt allow non secure access
309     WREM32(REG32(GPIO4_REGS_BASE_ADDRESS +0x14),0x3,0x3,0xFFFFFFFF); //interrupt allow non secure access
310 #endif
311
312 #ifndef CM33_NS_RESOURCES_SWITCH
313     INFO(__FILE__, " Jump to CM33_Non Secure start...");  

314     sec2ns_switch_blxns(_start_SYMBOL_ );
315 #endif
316
317 #ifndef CM33_M_NS_RESOURCES_SWITCH
318     INFO(__FILE__, " Jump to CM33_M Non Secure start...");  

319     sec2ns_switch_blxns(_start_SYMBOL_ );
320 #endif
321
322     return 0;
323 }

```

CM33_ns_local.lnk

```

/home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/verilog/verilog_FILENAMES_apc7
090.cn-sha01.nxp.com_29149/dma/cm33/cm33_ns
awv071589.cn-sha01.nxp.com:>grep start_SYMBOL -r
core.a ns symbol table.txt:ADDR.START_SYMBOL =0x00000000 ANATOP.ID_SYMBOL =0x00000000 CAPI_INCLUDE.C_SYMBOL =0x00000000 CAPI_INCLUDE.STARTUP_SYMBOL =0x00000000 CORE.NUM_SYMBOL =0x00000000 DMA.ID_SYMBOL =0x00000000 DMEM_SYMBOL =0x00000000 DST.SYM
BOL =0x00000000 DST.NS_SYMBOL =0x00000000 SHUTDOWN.STIM_SYMBOL =0x00000000 SMEM_SYMBOL =0x00000000 SRC_SYMBOL =0x00000000
SRC.NS_SYMBOL =0x00000000 STARTUP.STIM_SYMBOL =0x00000000 STRAMEM_SYMBOL =0x00000000 UMEM_SYMBOL =0x00000000 VMEM_SYMBOL
=0x00000000 XMEM_SYMBOL =0x00000000 CM33_NS_SYMBOL =0x00000001 CM33_NS_SYMBOL =0x00000001 CM33_NS_RESOURCES_SYMBOL =0x00
000001 CM33_NS_RESOURCES_SYMBOL =0x00000001 DEF.LC.NXP.FAB.MODE.DEFAULT_SYMBOL =0x00000001 cm33_ns_SYMBOL =0x00000001 cm33_n
s_SYMBOL =0x00000001 TARGET.MEM.MSB_SYMBOL =0x0000003F TARGET.MEM.WIDTH_SYMBOL =0x00000040 TARGET.MEM.WIDTH_SYMBOL =0x0000
0040 TARGET.MEM.STACK.SIZE_SYMBOL =0x00000800 TARGET.MEM.SIZE_SYMBOL =0x00070000 TARGET.MEM.SIZE_SYMBOL =0x00070000 LEN.SYM
BOL =0x00080000 vectors_SYMBOL =0x20008000 vector.table.end_SYMBOL =0x20008400 _START.ADDR_SYMBOL =0x20020000 _startup_
_SYMBOL =0x20020000 _end_SYMBOL =0x20020016 start_SYMBOL =0x20020020 CAPI.END.SIM_SYMBOL =0x20020038 UNIMPLEMENTED.ISR.S
YMBOL =0x20020058 PROCESS_INTERRUPT_SYMBOL =0x20020098 core.enable_irq_SYMBOL =0x200200c0 capi.interrupt.handler_SYMBOL =0

```

CM33 ns demo 反汇编 lst 文件

```

File Edit Tools Syntax Buffers Window Help
1509 Disassembly of section .startup:
1510
1511 20020000 <__START_ADDR>:
1512 __START_ADDR():
1513 20020000: 2000    movs   r0, #0
1514 20020002: 2100    movs   r1, #0
1515 20020004: 2200    movs   r2, #0
1516 20020006: 2300    movs   r3, #0
1517 20020008: 2400    movs   r4, #0
1518 2002000a: 2500    movs   r5, #0
1519 2002000c: 2600    movs   r6, #0
1520 2002000e: 2700    movs   r7, #0
1521 20020010: b662    cpsie  i
1522 20020012: f000 f805 bl 20020020 <_start>
1523
1524 20020016 <__end>:
1525 __end():
1526 20020016: bf30    wfi
1527 20020018: e7fd    b.n 20020016 <__end>
1528 2002001a: ffff ffff ; <UNDEFINED> instruction: 0xffffffff
1529 2002001e: Address 0x000000002002001e is out of bounds.
1530
1531
1532 Disassembly of section .text:
1533
1534 20020020 <_start>:
1535 _start():
1536 /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench(blocks/soc_tb/tool_data/compiler/cm
33/src/_start.c:22
1537 #ifdef __cplusplus
1538 extern "C" {
1539 #endif
1540
1541 void _start()
1542 { int main_status;
1543 20020020: b510    push   {r4, lr}
1544 /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench(blocks/soc_tb/tool_data/compiler/cm
33/src/_start.c:24
1545
1546 pre_main();

```

信号分析

Vector 重定向

芯片上电后，secure vector 会默认从 0，重定向到 0x13000000.

然后从 0x13000400 处取指令执行

Secure vctor 如下所示

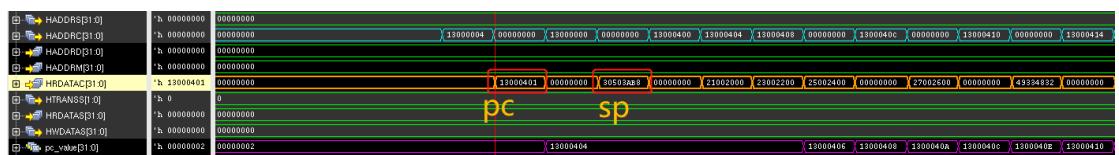
testbench.top.compute_vdd2.cm33_main.INITSVTOR[31:7] = 0x26000



The screenshot shows the Quartus Prime software interface. The top menu bar includes File, Edit, Tools, Project, Scope, and Help. The main window has tabs for Trace Signals, Signal To Trace, and Drivers. The code editor on the right displays Verilog source code for a module named `INITSVTOR`. The code defines various wires and their initial values, including power-on reset configurations and system reset logic. A waveform viewer on the left shows the state of these signals over time.

```
Trace Signals x (i)
Signal To Trace [ch_top compute_vdd2.cm33_main.INITSVTOR[31:7]]
Drivers of: INITSVTOR[31:7]
  • #33_main (INITSVTOR(25'h260000))
    • h0260000 1b 0260000

41 input wire "PPOWERSEL" // Power-on reset (for the entire processor)
42 input wire "PSYSRESET" // Soft reset (non-debug logic only)
43 output wire "nSYSRESETQ" // Request for system reset (nSYSRESET)
44
45 // Static configuration - change on power-on reset only
46 input wire "CFGENDIAN" // D-side endianess config
47 input wire [25:0] "CFGSYSSCALIB" // Secure SysTick calibration
48 input wire [25:0] "CFGNSCALIB" // Non-Secure SysTick calibration
49 input wire "CFGFPFP" // Enable support for hardware floating point (if configured)
50 input wire "CFGDSP" // Enable support for ARNv8-M DSP extensions
51 input wire "CFGSEEXT" // Enable support for ARNv8-M Security extensions (if configured)
52 input wire "MPUNSUSABLE" // Disable support for the non-secure MPU (if configured)
53 input wire "MPDISABLE" // Disable support for the secure MPU (if configured)
54 input wire "SAUDISABLE" // Disable support for the SAU (if configured)
55
56 input wire [7:0] "-h 00" "CFGCODECP" // Enable CDE support for Coprocessors
57
58 // Reset configuration - change on system reset
59 input wire [31:7] "INITSVTOR" // Secure Vector Table offset register out of reset
60 input wire [31:7] "INITNSVTOR" // Non-Secure Vector table offset register out of reset
61 // Controls and hints:
62 input wire "CPUDAT" // Stall processor out of reset
63 output wire [2:0] "CODEHINT" // Prefetch hints
64 input wire "IPUS" // Flush instructions fetched on C-AHB or S-AHB on the previous HREADY cycle
65 output wire "DEBUGS" // Current security state of the processor
66 // C-AHB Interface (AHBAS-AHBBS)
67 output wire "HMASTER" // C-AHB master source. HMASTERC 1'b1 indicates debug
68 output wire [31:0] "ADDRS" // C-AHB address
```



复位序列

在离开复位状态后，CM3 做的第一件事就是读取下列两个 32 位整数的值：

- 从地址 0x0000,0000 处取出 MSP 的初始值。
 - 从地址 0x0000,0004 处取出 PC 的初始值——这个值是复位向量，LSB 必须是 1。然后从这个值所对应的地址处取指。

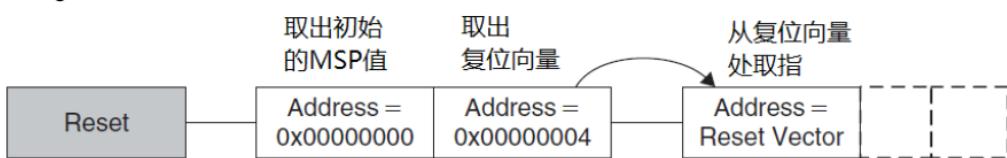


图 3.17 复位序列

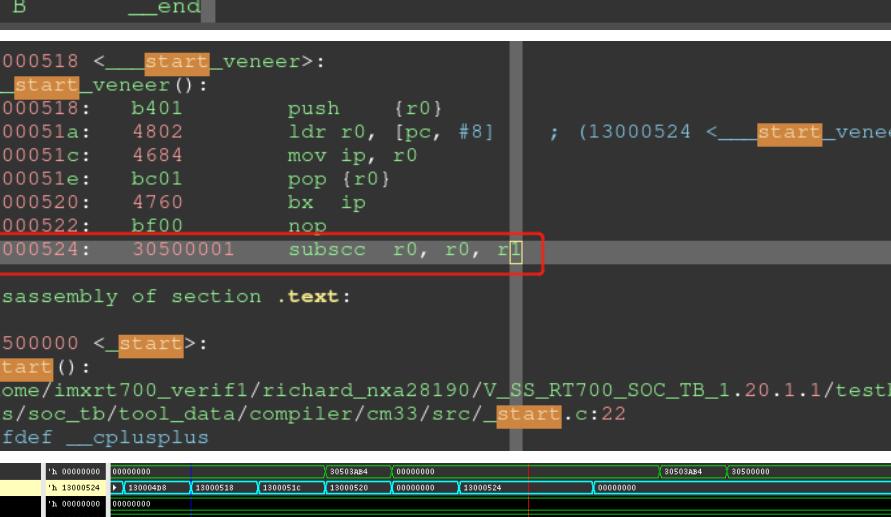
跳转到 c

Crt0.s

```
125          CPSIE    i          @ Unmask interrupts
126          BL      _Start       @ call the C code
127  __end:
128          WFI
129          B      __end        @ pause the core after completion

2447 13000518 <__start_veneer>:
2448 __start_veneer():
2449 13000518: b401      push    {r0}
2450 1300051a: 4802      ldr r0, [pc, #8] ; (13000524 <__start_veneer+0xc>)
2451 1300051c: 4684      mov ip, r0
2452 1300051e: bc01      pop {r0}
2453 13000520: 4760      bx ip
2454 13000522: bf00      nop
2455 13000524: 30500001  subsc r0, r0, r0
2456

2457 Disassembly of section .text:
2458
2459 30500000 <__start>:
2460 __start():
2461 /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/src/_start.c:22
2462 #ifdef __cplusplus


```

[跳转到 NON-SECURE DEMO](#)

跳转到 non-secure demo 的_start.c

Boot into non-secure demo

Secure demo cdoe:

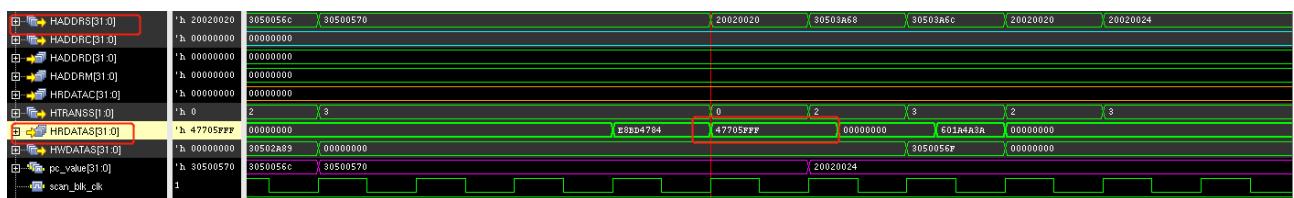
```
3403 void __attribute__((noinline,noclone)) sec2ns_switch_blnxns(address_t addr) {
3404     __asm__ ("PUSH {r0-r12, lr}");
3405     30050564: e92d 5fff! stmdb sp!, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, sl, fp, ip, lr}
3406     /home/limxr700_verif/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_s/src/trustzone_api.c:121
3407     __asm__ ("BIC r0, r0, #1");
3408     30050568: f020 0001 bicw r0, r0, #1
3409     /home/limxr700_verif/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_s/src/trustzone_api.c:122
3410     __asm__ ("BLXNS r0#");
3411     3005056c: 4784 blxns r0
3412     /home/limxr700_verif/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/cm33_s/src/trustzone_api.c:123
3413     __asm__ ("POP {r0-r12, lr}");
3414     3005056e: eabd 5fff! ldmia.w sp!, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, sl, fp, ip, lr}
3415     30050572: 4770 bx lr
```

Non-secure demo code

```

1541 void _start()
1542 { int main_status;
1543 20020020: b510      push    {r4, lr}
1544 /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/src/_start.c:24
1545
1546     pre_main();
1547 20020022: f001 fe33  bl  20021c8c <_Z8pre_mainv>
1548 /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/src/_start.c:26
1549
1550     main_status = main();
1551 20020026: f001 fe9f  bl  20021d68 <main>
1552 2002002a: 4604      mov     r4, r0
1553 /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/src/_start.c:28
1554
1555     post_main();
1556 20020030: f001 fe74  bl  20021d18 <_Z9post_mainv>
1557 /home/imxrt700_verif1/richard_nxa28190/V_SS_RT700_SOC_TB_1.20.1.1/testbench/blocks/soc_tb/tool_data/compiler/cm33/src/_start.c:30
1558
1559     CAPI_END_SIM(main_status);
1560 20020030: 4620      mov     r0, r4
1561 20020032: f000 f801  bl  20020038 <_Z12CAPI_END_SIMi>
1562 20020036: bd10      pop    {r4, pc}
1563
1564 20020038 <_Z12CAPI_END_SIMi>;

```



CM33_M boot

CM33_S : sanity_cm33_m_init.c, cm33_s/src/crt0.s

CM33_M_S: dma3_reg.c, cm33_m_s/src/crt0.s

1. CM33_S demo: vector/crt0.s-> ROM, code/data-> sram16

Pre_main 中配置 CM33_M 的 vector

```

#ifndef CM33_M_BOOT_ENABLE
    INFO("pre_main","config sense cm33 vector....");
#endif CM33_M_RESET_VECTOR
    W32(REG32(SYSCTL_SHA_VDD1_BASE + 0x98), CM33_M_RESET_VECTOR>>7);//system_svto
    W32(REG32(SYSCTL_SHA_VDD1_BASE + 0x9C), CM33_M_RESET_VECTOR>>7);//system_nsvto
#else
    W32(REG32(SYSCTL_SHA_VDD1_BASE + 0x98), S_SSRAM_P28_BASE>>7);//system_svto_i
    W32(REG32(SYSCTL_SHA_VDD1_BASE + 0x9C), S_SSRAM_P28_BASE>>7);//system_nsvto_i
#endif CM33_M_BOOT_ENABLE

```

```
INFO("pre_main","release sense cm33 reset....");//CM33_M demo loading
```

```
BCLR32(REG32(RSTCTL_SHA_VDD1_BASE + 0x0010), BIT31);
```

```
INFO("pre_main","release sense cm33 CPUWAIT....");
```

```
BCLR32(REG32(SYSCTL_SHA_VDD1_BASE + 0xA0), BIT0);
```

```
#endif //CM33_M_BOOT_ENABLE
```

2. Release CM33_M reset, testbench load CM33_M demo, 运行 CM33_M demo

CM33_M demo, vector/code/data->SRAM28

CM33_M_NS boot

CM33_M_NS: dma3_interrupt.c , cm33_m_ns/src/crt0.s, cm33/src/_start.c

CM33_S: sanity_cm33_m_init.c, cm33/cm33_s/src/crt0.s, cm33/src/_start.c

CM33_M_S: sanity_cm33_init.c, cm33_m_s/src/crt0.s, cm33/src/_start.c

1. CM33 vector/crt0.s-> ROM, code/data->SRAM16

程序跑起来后，识别到 CM33_M_BOOT_ENABLE， 配置 CM33_M 的 vector， 然后 release CM33_M reset

2. CM33_M vector/code/data -> SRAM28, CM33_M_NS -> SRAM1/SRAM4

CM33_M demo 识别到 CM33_M_NS_RESOURCES_SWITCH， 配置 SAU/IDAU non-secure 空间，
然后 main 函数中跳转到 CM33_M_NS demo。

跳转地址_start_SYMBOL_， 为 CM33_M_NS demo 的 C 入口地址

HIFI4 boot

HIFI 使用的编译器是 xt-clang

HIFI4: dma3_reg.c, hifi4/src/vectors.S, /hifi4/src/_start.c

CM33_S: cm33_s/src/crt0.s , compiler/include/sanity_hifi4_init.c, cm33/src/_start.c

1. CM33_S vector/crt0.s -> ROM, C code/data->SRAM16

Cm33_s/include/pre_post_main.h 检测到 HIFI4_BOOT_ENABLE。Release HIFI4 reset.

一旦 release HIFI4 reset， testbench 将 HIFI4 DEMO load 到 HIFI4 的 ITCM, DTCM 中。 Demo 开始运行

```
#ifdef HIFI4_BOOT_ENABLE
    INFO("pre_main","release hifi4 reset....");//  

    //W32(REG32(RSTCTL_COM_VDD2_BASE + 0x0024), 0x0000000);  

    BCLR32(REG32(RSTCTL_COM_VDD2_BASE + 0x0024), BIT0);
#endif //HIFI4_BOOT_ENABLE
```

```
MEMORY
{
    sram0_seg : org = 0x20040004, len = 0x3FFC
    dram0_0_seg : org = 0x24000000, len = 0x10000
    iram0_0_seg : org = 0x24020000, len = 0x2E0
    iram0_1_seg : org = 0x240202E0, len = 0x120
    iram0_2_seg : org = 0x24020400, len = 0x178
    iram0_4_seg : org = 0x2402057C, len = 0x1C
    iram0_5_seg : org = 0x24020598, len = 0x4
    iram0_6_seg : org = 0x2402059C, len = 0x1C
```

```

iram0_7_seg : org = 0x240205B8, len = 0x4
iram0_8_seg : org = 0x240205BC, len = 0x1C
iram0_10_seg : org = 0x240205DC, len = 0x1C
iram0_12_seg : org = 0x240205FC, len = 0x1C
iram0_13_seg : org = 0x24020618, len = 0x4
iram0_14_seg : org = 0x2402061C, len = 0x1C
iram0_16_seg : org = 0x2402063C, len = 0x1C
iram0_17_seg : org = 0x24020658, len = 0xF9A8
string_0_seg : org = 0x36000000, len = 0x20000
}

```

```

UVM_INFO @ 1559524.603122 ns --: [testbench.load_memory_hifi4] Loading DTCM .....
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform:HIFI4 Loading
into mem from address 24000000 to 2400ffff
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address=9008000 over range
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address >= 24000f94 Load default value into mem...
UVM_INFO @ 1559524.603122 ns --: [testbench.load_memory_hifi4] Loading ITCM .....
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform:HIFI4 Loading
into mem from address 24020000 to 2402ffff
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address=d800000 over range
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address >= 24020628 Load default value into mem...
UVM_INFO @ 1559524.603122 ns --: [testbench.load_memory_hifi4] Loading Shared SRAM6 .....
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform:HIFI4 Loading
into mem from address 20040000 to 2005ffff
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address=9000000 over range
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address >= 20043f1c Load default value into mem...
UVM_INFO @ 1559524.603122 ns --: [testbench.load_memory_hifi4] Loading Shared SRAM7 .....
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform:HIFI4 Loading
into mem from address 20060000 to 2007ffff
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address=9000000 over range
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address >= 0xxxxxxxx Load default value into mem...
UVM_INFO @ 1559524.603122 ns --: [testbench.load_memory_hifi4] Loading string .....
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform:HIFI4 Loading
into mem from address 36000000 to 3607ffff
UVM_INFO @ 1559524.603122 ns --: [testbench.memory_map_load_gnu_hex] platform: HIFI4
address >= 36010634 Load default value into mem...

```

HIFI1 Boot

HIFI1 的 vector table 是 RTL 设置好的，为 HIFI1 ITCM 的 start address

```

1244 |
1245 |     wire [31:0] reset_vector;
1246 |     assign reset_vector = StatVectorSel_W ? 32'h580660 : 32'h580000;
1247 |

```

HIFI1: hifi/hifi1/src/vectors.S, dma/stimulus/dma3_soft_trigger.c

CM33: sanity_hifi1_init.c, cm33_s/src/crt0.s

CM33_M: sanity_hifi1_init.c, cm33_m_s/src/crt0.s

- CM33 vector-> ROM, data/code->SRAM16
上电后，ROM 启动，执行 CM33 的 code，识别到 CM33_M_BOOT_ENABLE，对 CM33_M 进行 vector 的配置，然后 release CM33_M reset
- Testbench load CM33 demo， code/data -> SRAM28
CM33_M 启动后，CM33_M_S/include/pre_post_main.h 识别到 HIFI1_BOOT_ENABLE，release HIFI1 reset

```
#ifdef HIFI1_BOOT_ENABLE
    INFO("pre_main","release hifi1 reset....");
    W32(REG32(RSTCTL_SEN_VDD1_BASE + 0x0010), 0x00000000);
    W32(REG32(SYSCtrl_SEN_VDD1_BASE + 0x0300), 0x00000000);
#endif //HIFI1_BOOT_ENABLE
```

- Testbench load HIFI1 demo， code->HIFI1 ITCM ,data->SRAM23
运行 HIFI1 demo

Zenv Boot

ZENV 使用的编译器 riscv32-unknown-elf-g++

Zenv: dma3_reg.c zenv/src/zv0335_crt0.s
CM33: cm33_s/src/crt0.s sanity_zenv_init.c

- CM33 VECTOR->ROM, code/data->SRAM16,
程序识别到 ZENV_BOOT_ENABLE， 配置 zenv 的 vector， release zenv 的 reset


```
#ifdef ZENV_BOOT_ENABLE
    INFO("pre_main","config EZH vector....");
    #ifdef ZENV_RESET_VECTOR
    INFO("pre_main","config EZH reset vector to ZENV_RESET_VECTOR");
    W32(REG32(SYSCtrl_MED_VDD2_BASE + 0xD4), ZENV_RESET_VECTOR);           // MTVEC
    W32(REG32(SYSCtrl_MED_VDD2_BASE + 0xD8), ZENV_RESET_VECTOR+0x400);       // STVEC
    W32(REG32(SYSCtrl_MED_VDD2_BASE + 0xD0), (ZENV_RESET_VECTOR+0x800)>>2); // RSTBASE
    #else
    W32(REG32(SYSCtrl_MED_VDD2_BASE + 0xD4), 0x24100000);           // MTVEC
    W32(REG32(SYSCtrl_MED_VDD2_BASE + 0xD8), 0x24100400);           // STVEC
    W32(REG32(SYSCtrl_MED_VDD2_BASE + 0xD0), 0x24100800>>2); // RSTBASE
    #endif
    INFO("pre_main","release ezhv reset....");
    BCLR32(REG32(RSTCTL_MED_VDD2_BASE + 0x0010), BIT5);
#endif //ZENV_BOOT_ENABLE
```
- Testbench load ZENV demo,运行 zenv demo

Vector、code -> ZENV_ITCM , data->SRAM17

Dual core boot

以 CM33&CM33_M 为例:

1. VPLAN 中填入 CM33&CM33_M，生成的 arg 文件中就会指定两者的 stimulus 为相同的 main
2. CM33 跑的最快，可以在 CM33_M 的 main 跑起来之前，通过 CM33 做一些配置，实现 CM33_M 对相关功能的访问

如果两个 core 需要互相控制 code 执行，可以通过 trigger_get/trigger_set 的方式进行