STM32F4 系列内部有 64KB CCM 耦合内存，外加 192KB RAM，内部 256KB。但是，CCM RAM 在 RT-Studio 中使用。CCM 部分特殊，挂在 D-Bus 上，只能通过不能访问，不能使用所有 CCM 访问部分，默认没有 使用这部分 CCM。

在 MDK 工程中新增 CCM 比较方便，可以编辑 sct 文件。不过，与 MDK 不同，RT-Studio 使用 gcc 编译 代码。

本文档近期的项目开发中的 RAM，需要开启 CCM。在本论坛查到的问题/文章中，都不能解决同时在 CCM 中使用未初始化变量和初始化变量 4296 的一些配置方式会导致 bin 文件过大参考。上，提出简单的修改 lds 的方案，分别是：仅用于未初始化变量、仅用于初始化变量、同时支持未初始化和初始化变量使用场景。

简单说明一下，论坛中 lds 代码高亮是使用```c，最终查看了网页源代码才了解。 本文参考链接：

RT-Thread-link.ld 修改使能满足 CCM RAM，但数据出错 RT-Threads 问答社区 - RT-Thread

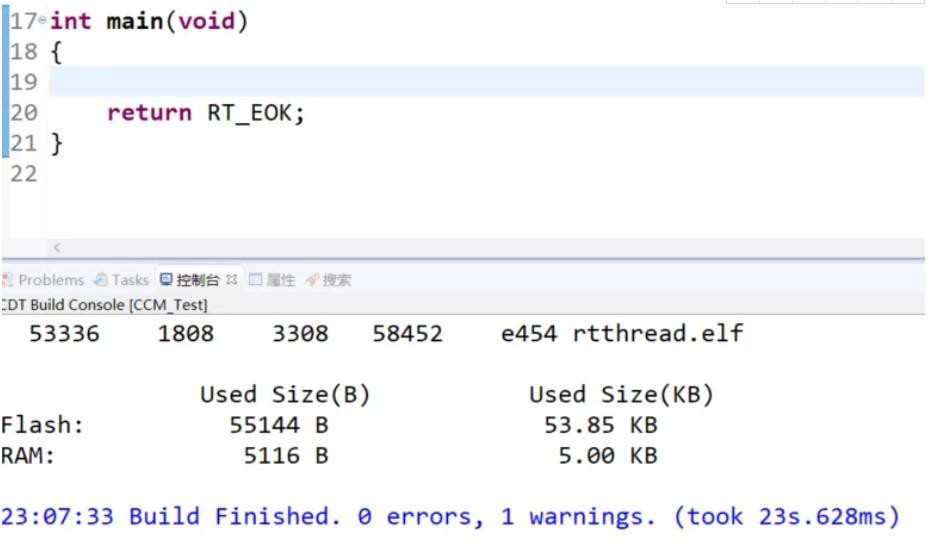
RT-Thread-STM32F4 在 RT-Thread Studio 中使用 CCM SRAMRT-Thread 问答社区 - RT-Thread RT-Thread-rtthread 正常，bin 文件生成 RT-Thread-Thread 问答社区 - RT Studio

1 第一种 CCM 添加方案：未初始化参数

一种 CCM 添加方案开始支持时未初始化设置在 CCM RAM 中不能声明初始值。

1.创建工程，将根据电路实际情况进行修改。





3.打开并查看 lds 文件



4.找到 linkscripts->STM32F429VI 目录下的 lds 文件，使用文本编辑器编辑代码，增加一行 CCM 的描述， 如第 5 行所示。

MEMORY

{

ROM (rx) : ORIGIN = 0x08000000, LENGTH = 2048k /\* 2048K flash \*/ RAM (rw) : ORIGIN = 0x20000000, LENGTH = 192k /\* 192K sram \*/ CCM (rw) : ORIGIN = 0x10000000, LENGTH = 64k /\* 64K ccm sram \*/

}



6.在 lds 文件中添加.ccm section 的描述，位于.data 部分。

.ccm :

{

. = ALIGN(4);

\_sccm = .; /\*create a global symbol at ccmram start \*/

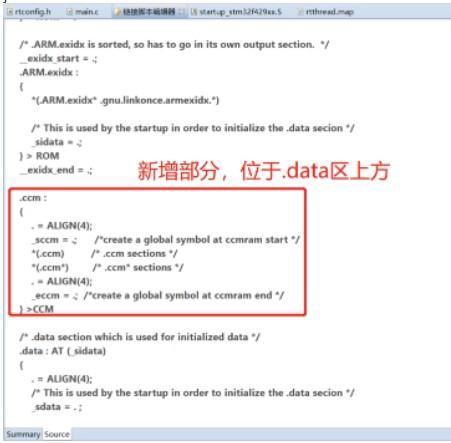
\*(.ccm) /\* .ccm sections \*/

\*(.ccm\*) /\* .ccm\* sections \*/

. = ALIGN(4);

\_eccm = .; /\*create a global symbol at ccmram end \*/

} >CCM



attribute ((section(".ccm"))) rt\_uint8\_t array[4];

attribute ((section(".ccm"))) rt\_uint8\_t \*ptr; int main(void)

{

ptr = array;

for(rt\_uint8\_t i = 0; i < 4; i++)

{

\*ptr = 7 + i; ptr++;

}

rt\_kprintf("ptr = 0x%08x, stores in 0x%08x\n", ptr, &ptr); for(rt\_uint8\_t i = 0; i < 4; i++)

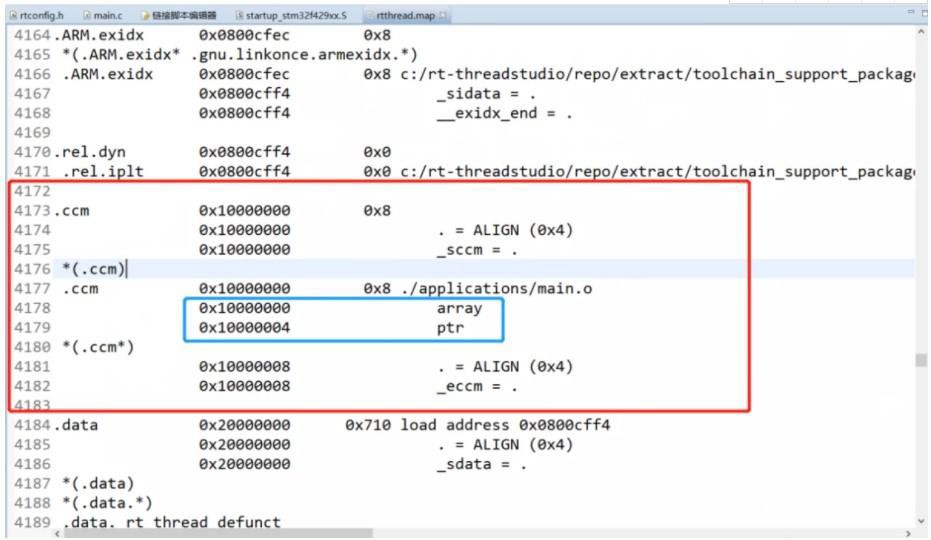
{

rt\_kprintf("array[%d] = %d\n",i,array[i]);

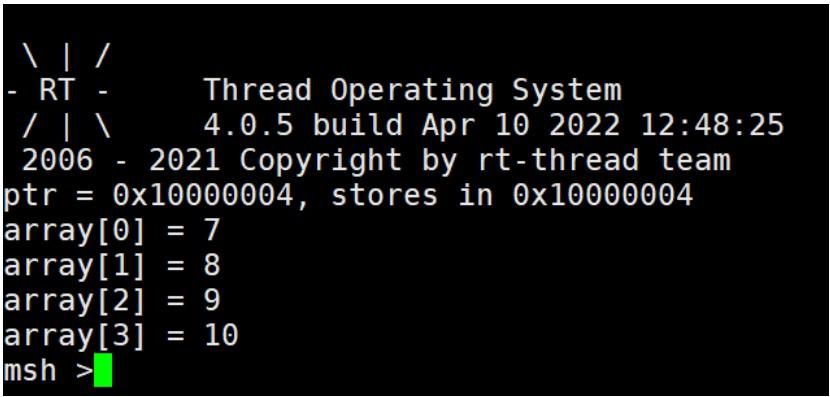
}

return RT\_EOK;

}

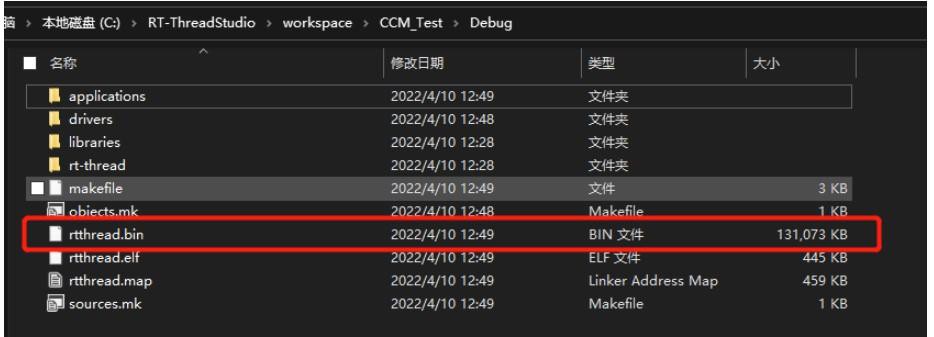


9.下载程序，观察结果。



显然，此时数据被写入到了所有数组中，并且指针指向正常。结果中，ptr=0x10000004 是因为在循环中自 增了 4 次。

11.打开 Debugthread.bin 文件，发现了 131073KB。



12.再次修改.ccm 部分，添加 NoLOAD 关键词后重新编译，bin 文件大小正常。

{

. = ALIGN(4);

\_sccm = .; /\*create a global symbol at ccmram start \*/

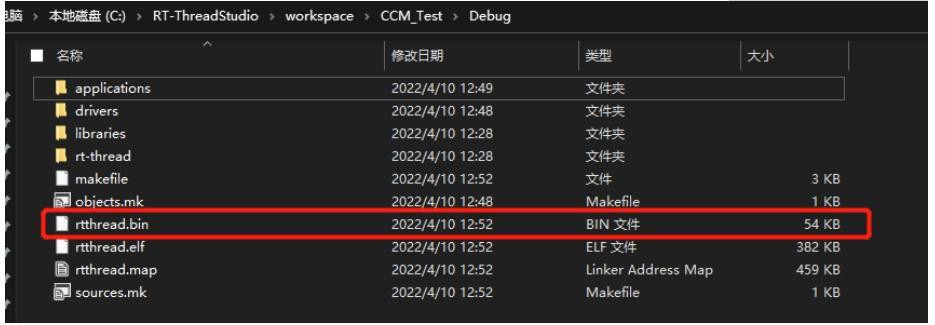
\*(.ccm) /\* .ccm sections \*/

\*(.ccm\*) /\* .ccm\* sections \*/

. = ALIGN(4);

\_eccm = .; /\*create a global symbol at ccmram end \*/

} >CCM



声明.ccm 中的变量赋初值是在正确的情况下，将提供的初始值支持对 0 时为 0 也不行。

2 另外三个 CCM 添加方案：初始化参数 支持添加对 CCM RAM 中的变量赋初始值。 1.修改.ccm 和.data 的 lds 部分

\_siccm = LOADADDR(.ccm);

.ccm :

{

. = ALIGN(4);

\_sccm = .; /\*create a global symbol at ccmram start \*/

\*(.ccm\*) /\* .ccm\* sections \*/

. = ALIGN(4);

\_eccm = .; /\*create a global symbol at ccmram end \*/

} >CCM AT>ROM

/\* .data section which is used for initialized data \*/

\_sidata = LOADADDR(.data);

.data :

{

. = ALIGN(4);

/\* This is used by the startup in order to initialize the .data secion \*/

\_sdata = . ;

\*(.data)

\*(.data.\*)

\*(.gnu.linkonce.d\*)

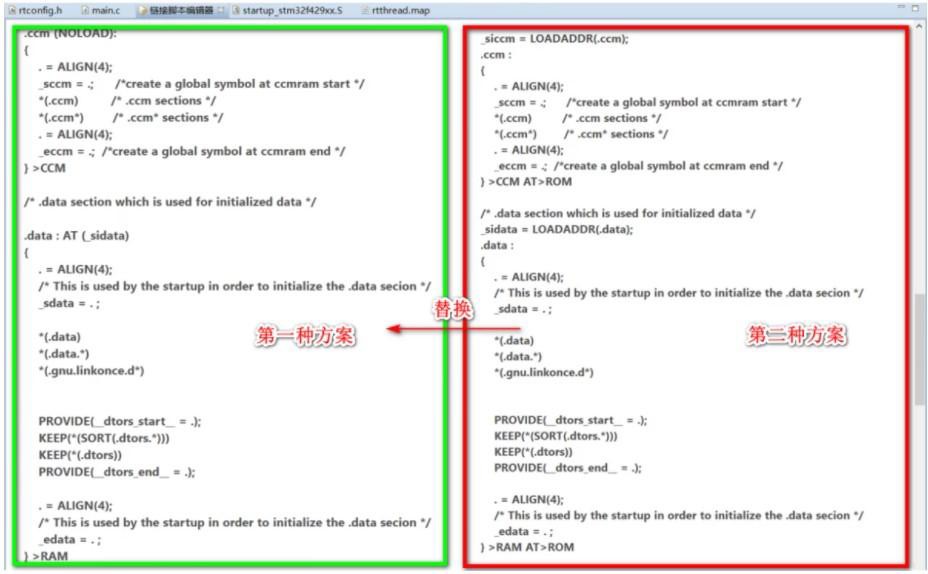
PROVIDE( dtors\_start = .); KEEP(\*(SORT(.dtors.\*))) KEEP(\*(.dtors))

PROVIDE( dtors\_end = .);

. = ALIGN(4);

/\* This is used by the startup in order to initialize the .data secion \*/

\_edata = . ;



2.修改启动代码

启动代码 startup\_stm32f429xx.s 位于

libraries->CMSIS->Device->ST->STM32F4xx->Source->Templates->gcc 下目录，较深。 原始的启动代码只对.data 区进行初始化，我们需要对.ccm 区也现在进行初始化。

步骤 1。在原始启动代码的.word \_ebss 行后添加如下代码，从 lds 中引入\_sciccm、\_sccm 和\_eccm。

/\* start address for the initialization values of the .data ccm section. defined in linker script \*/

.word \_siccm

/\* start address for the .ccm section. defined in linker script \*/

.word \_sccm

/\* end address for the .ccm section. defined in linker script \*/

.word \_eccm

第 2 步。在启动的代码的 bcc CopyDataInit 和 ldr r2, =\_sbss 两行最初插入的代码中，对.ccm 中的进行初 始化，即，从 Flash 中间容器到 CCM RAM。注释，很好理解。

/\* Copy the data CCM segment initializers from flash to SRAM \*/

b LoopCopyCCMInit /\* 调用 LoopCopyCCMInit \*/ CopyCCMInit:

ldr r3, =\_siccm /\* Flash 中存储的 CCM RAM 变量初始值的起始地址 \*/ ldr r3, [r3, r1] /\* r3 <-[r3+r1] \*/

str r3, [r0, r1] /\* [r0 + r1] <- r3 \*/ adds r1, r1, #4 /\* 偏移量加 4\*/

LoopCopyCCMInit:

ldr r0, =\_sccm /\* .ccm section 的起始地址 \*/ ldr r3, =\_eccm /\* .ccm section 的终止地址 \*/ adds r2, r0, r1 /\* r2 <- 指针加偏移量\*/

cmp r2, r3 /\* 指针是否已经指向了终止地址？ \*/ bcc CopyCCMInit

3.修改 main.c，编译、下载、观察运行结果

attribute ((section(".ccm"))) rt\_uint8\_t array[4] = {1,2,3,4};

attribute ((section(".ccm"))) rt\_uint8\_t \*ptr = array; int main(void)

{

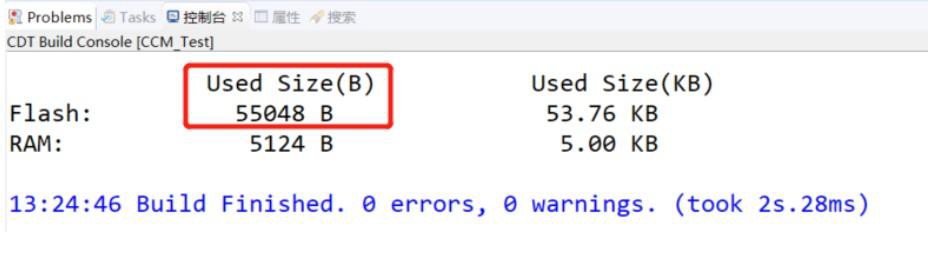
rt\_kprintf("ptr = 0x%08x, stores in 0x%08x\n", ptr, &ptr); for(rt\_uint8\_t i = 0; i < 4; i++)

{

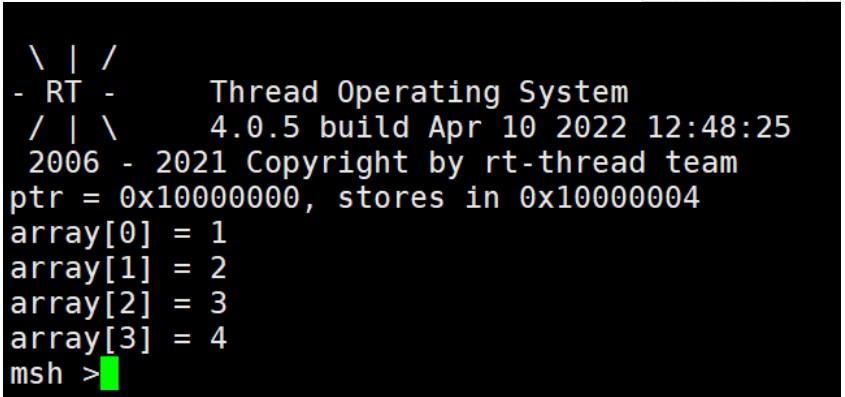
rt\_kprintf("array[%d] = %d\n",i,array[i]);

}

return RT\_EOK;



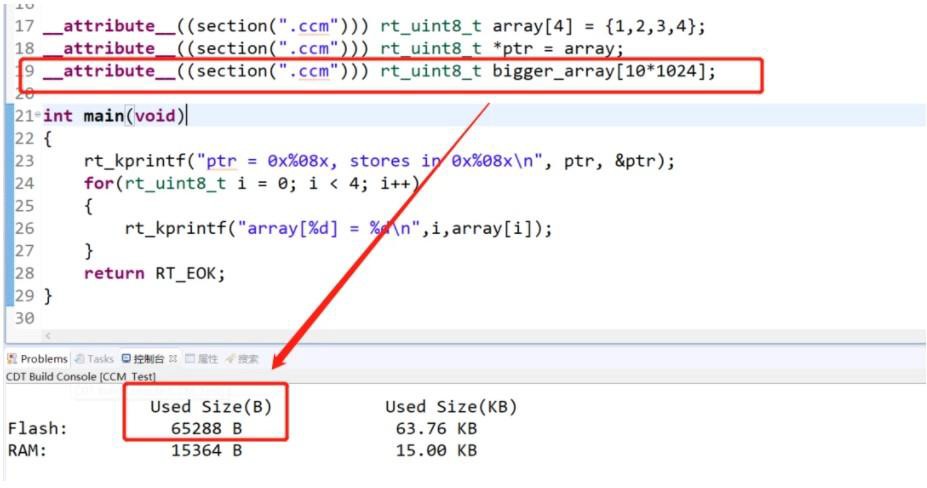
记住 55048 这个数据，方便会。



很明显，此时在值。ccm 段中的变量已经被最终实现了。

4.如果在 main.c 再添加一个大型未初始化的教程，会发生什么？

attribute ((section(".ccm"))) rt\_uint8\_t bigger\_array[1024];



编译后，Flash 容量变成了 65288。明显，65288 - 55048 = 10240。

large\_array 虽然没有被初始化，但它却被分配在了部分，按照我们的启动约束后的代码，我们的 STM32， 要从 Flash 中保存的数据启动到该 10K 后。我们最终是需要将 CCM 配置成既支持.data，也支持.bss 的方 式。

3 第一种 CCM 方案：共存

第一个 CCM 方案，既定支持参数设置在 CCM 中支持未初始化的想法解决方案是在 CCM 中的两个部 分，用于添加数据数据，用于存放。数据。

1. 在这种方式的指导下，我们可以快速在.ccm 部分中再次添加一个.ccmbss 部分。

.ccmbss (NOLOAD):

{

. = ALIGN(4);

\_sccmbss = .; /\*create a global symbol at ccmram start \*/

\*(.ccmbss) /\* .data sections \*/

\*(.ccmbss\*) /\* .data\* sections \*/

. = ALIGN(4);

\_eccmbss = .; /\*create a global symbol at ccmram end \*/

} >CCM

2.修改 main.c

attribute ((section(".ccm"))) rt\_uint8\_t array[4] = {1,2,3,4};

attribute ((section(".ccm"))) rt\_uint8\_t \*ptr = array;

attribute ((section(".ccmbss"))) rt\_uint8\_t bigger\_array[10\*1024]; int main(void)

{

rt\_kprintf("ptr = 0x%08x, stores in 0x%08x\n", ptr, &ptr); for(rt\_uint8\_t i = 0; i < 4; i++)

{

bigger\_array[i] = i; /\* 假装使用了 bigger\_array, 防止编译器优化掉这个数组\*/ rt\_kprintf("array[%d] = %d, bigger\_array[%d] = %d\n",

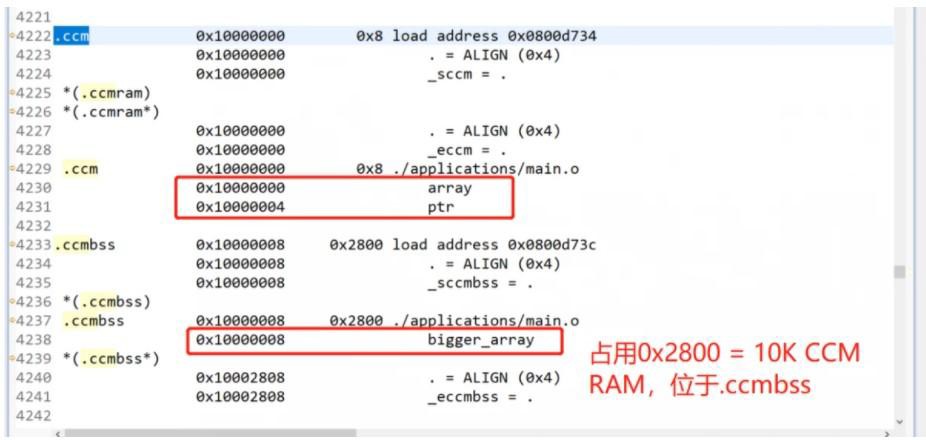
i,array[i], i,bigger\_array[i]);

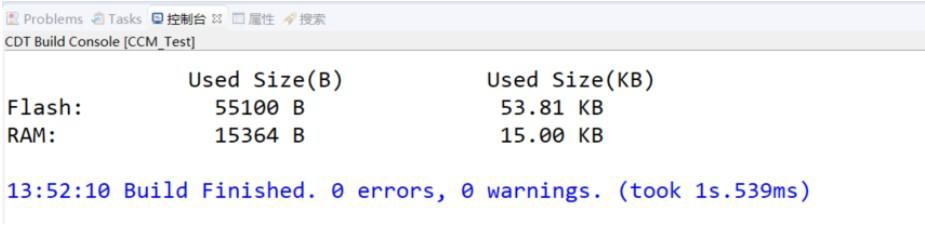
}

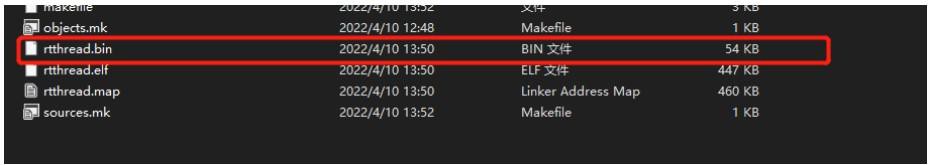
return RT\_EOK;

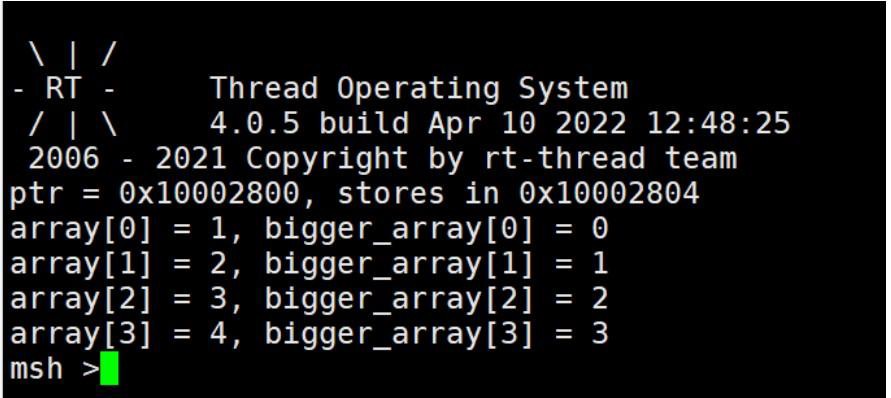
}

3.查看编译文件，新增的 10KFlash 占用也需要查看地图空间，bin。



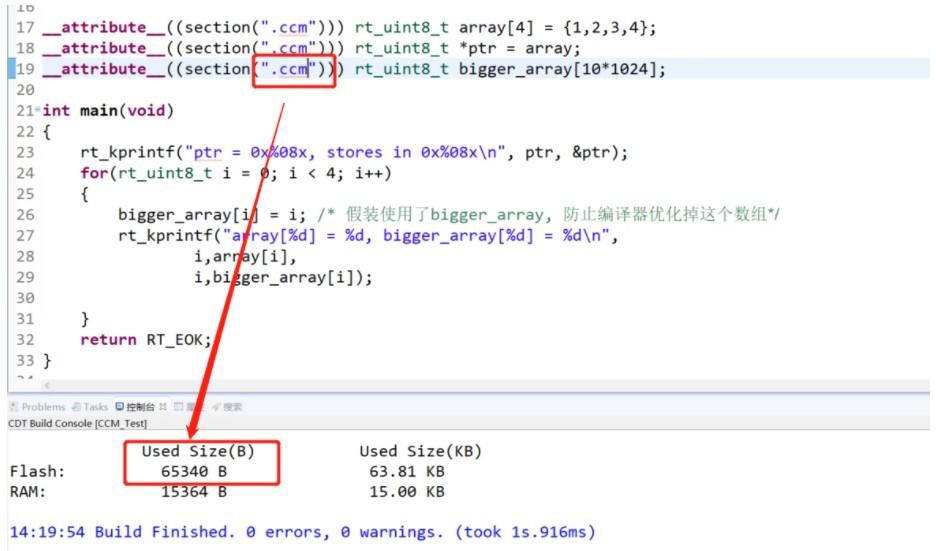






4 再次启动阵列的存放位置，K\_修改器在.ccm，表示，此时 Flash10，初始化时启动位置会。在 1K RAM

初始化位置。



5.最后附上完整的 lds 文件，RTT 快乐！

/\*

\* linker script for STM32F429VI with GNU ld

\*/

/\* Program Entry, set to mark it as "used" and avoid gc \*/

MEMORY

{

ROM (rx) : ORIGIN = 0x08000000, LENGTH = 2048k /\* 2048K flash \*/ RAM (rw) : ORIGIN = 0x20000000, LENGTH = 192k /\* 192K sram \*/ CCM (rw) : ORIGIN = 0x10000000, LENGTH = 64k /\*64K ccm ram\*/

}

ENTRY(Reset\_Handler)

\_system\_stack\_size = 0x400; SECTIONS

{

.text :

{

. = ALIGN(4);

\_stext = .;

KEEP(\*(.isr\_vector)) /\* Startup code \*/

. = ALIGN(4);

\*(.text) /\* remaining code \*/

\*(.text.\*) /\* remaining code \*/

\*(.rodata) /\* read-only data (constants) \*/

\*(.rodata\*)

\*(.glue\_7)

\*(.glue\_7t)

\*(.gnu.linkonce.t\*)

/\* section information for finsh shell \*/

fsymtab\_start = .; KEEP(\*(FSymTab))

fsymtab\_end = .;

. = ALIGN(4);

vsymtab\_start = .; KEEP(\*(VSymTab))

vsymtab\_end = .;

/\* section information for utest \*/

. = ALIGN(4);

rt\_utest\_tc\_tab\_start = .; KEEP(\*(UtestTcTab))

rt\_utest\_tc\_tab\_end = .;

/\* section information for at server \*/

. = ALIGN(4);

rtatcmdtab\_start = .; KEEP(\*(RtAtCmdTab))

rtatcmdtab\_end = .;

. = ALIGN(4);

/\* section information for initial. \*/

. = ALIGN(4);

rt\_init\_start = .; KEEP(\*(SORT(.rti\_fn\*)))

rt\_init\_end = .;

PROVIDE( ctors\_start = .); KEEP (\*(SORT(.init\_array.\*))) KEEP (\*(.init\_array)) PROVIDE( ctors\_end = .);

. = ALIGN(4);

\_etext = .;

} > ROM = 0

/\* .ARM.exidx is sorted, so has to go in its own output section. \*/

exidx\_start = .;

.ARM.exidx :

{

\*(.ARM.exidx\* .gnu.linkonce.armexidx.\*)

/\* This is used by the startup in order to initialize the .data secion \*/

\_sidata = .;

\_siccmram = .;

} > ROM

exidx\_end = .;

/\* .data section which is used for initialized data \*/

\_sidata = LOADADDR(.data);

.data :

{

. = ALIGN(4);

/\* This is used by the startup in order to initialize the .data secion \*/

\*(.data)

\*(.data.\*)

\*(.gnu.linkonce.d\*)

PROVIDE( dtors\_start = .); KEEP(\*(SORT(.dtors.\*))) KEEP(\*(.dtors))

PROVIDE( dtors\_end = .);

. = ALIGN(4);

/\* This is used by the startup in order to initialize the .data secion \*/

\_edata = . ;

} >RAM AT > ROM

\_siccm = LOADADDR(.ccm);

.ccm :

{

. = ALIGN(4);

\_sccm = .; /\*create a global symbol at ccmram start \*/

\*(.ccmram) /\* .data sections \*/

\*(.ccmram\*) /\* .data\* sections \*/

. = ALIGN(4);

\_eccm = .; /\*create a global symbol at ccmram end \*/

} >CCM AT>ROM

.ccmbss (NOLOAD):

{

\_sccmbss = .; /\*create a global symbol at ccmram start \*/

\*(.ccmbss) /\* .data sections \*/

\*(.ccmbss\*) /\* .data\* sections \*/

. = ALIGN(4);

\_eccmbss = .; /\*create a global symbol at ccmram end \*/

} >CCM

.stack :

{

. = ALIGN(4);

\_sstack = .;

. = . + \_system\_stack\_size;

. = ALIGN(4);

\_estack = .;

} >RAM

bss\_start = .;

.bss :

{

. = ALIGN(4);

/\* This is used by the startup in order to initialize the .bss secion \*/

\_sbss = .;

\*(.bss)

\*(.bss.\*)

\*(COMMON)

/\* This is used by the startup in order to initialize the .bss secion \*/

\_ebss = . ;

\*(.bss.init)

} > RAM

bss\_end = .;

\_end = .;

/\* Stabs debugging sections. \*/

.stab 0 : { \*(.stab) }

.stabstr 0 : { \*(.stabstr) }

.stab.excl 0 : { \*(.stab.excl) }

.stab.exclstr 0 : { \*(.stab.exclstr) }

.stab.index 0 : { \*(.stab.index) }

.stab.indexstr 0 : { \*(.stab.indexstr) }

.comment 0 : { \*(.comment) }

/\* DWARF debug sections.

* Symbols in the DWARF debugging sections are relative to the beginning
* of the section so we begin them at 0. \*/

/\* DWARF 1 \*/

.debug 0 : { \*(.debug) }

.line 0 : { \*(.line) }

/\* GNU DWARF 1 extensions \*/

.debug\_srcinfo 0 : { \*(.debug\_srcinfo) }

.debug\_sfnames 0 : { \*(.debug\_sfnames) }

.debug\_aranges 0 : { \*(.debug\_aranges) }

.debug\_pubnames 0 : { \*(.debug\_pubnames) }

/\* DWARF 2 \*/

.debug\_info 0 : { \*(.debug\_info .gnu.linkonce.wi.\*) }

.debug\_abbrev 0 : { \*(.debug\_abbrev) }

.debug\_line 0 : { \*(.debug\_line) }

.debug\_frame 0 : { \*(.debug\_frame) }

.debug\_str 0 : { \*(.debug\_str) }

.debug\_loc 0 : { \*(.debug\_loc) }

.debug\_macinfo 0 : { \*(.debug\_macinfo) }

/\* SGI/MIPS DWARF 2 extensions \*/

.debug\_weaknames 0 : { \*(.debug\_weaknames) }

.debug\_funcnames 0 : { \*(.debug\_funcnames) }

.debug\_typenames 0 : { \*(.debug\_typenames) }

.debug\_varnames 0 : { \*(.debug\_varnames) }

}

4 小结

.bs：领取未初始化（未初始化）或者 0 的属性值和属性类型）的；

.data：已使用初始化的权利和权利的数据。

从第一个解决方案和启动中可以进一步发现，这第一个代码解决方案的.ccmbs 部分并没有对未初始化的参 数进行清 0。官方自带启动代码中，在 LoopF illZerobs 函数中。 bss 区的数据进行清零进一步操作。 如果需要修改对.ccmbss 中的数据进行清零可以参照 LoopF illZerobs，简单的启动代码。如此篇幅， 比较了，这部分就不再是展开了。