kendryte K210 link.lds文件分析

源码

```
/*
 * Copyright (c) 2006-2018, RT-Thread Development Team
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Change Logs:
 * Date           Author       Notes
 */

INCLUDE "link_stacksize.lds"

/*
 * The OUTPUT_ARCH command specifies the machine architecture where the
 * argument is one of the names used in the Kendryte library.
 */
OUTPUT_ARCH( "riscv" )

MEMORY
{
   /* 6M SRAM */
   SRAM : ORIGIN = 0x80000000, LENGTH = 0x600000
}

ENTRY(_start)
SECTIONS
{
    . = 0x80000000 ;

    /* __STACKSIZE__ = 4096; */

    .start :
    {
        *(.start);
    } > SRAM

    . = ALIGN(8);

    .text :
    {
        *(.text)                        /* remaining code */
        *(.text.*)                      /* remaining code */
        *(.rodata)                      /* read-only data (constants) */
        *(.rodata*)
        *(.glue_7)
        *(.glue_7t)
        *(.gnu.linkonce.t*)

        . = ALIGN(8);
```

```
    PROVIDE(__ctors_start__ = .);
    /* old GCC version uses .ctors */
    KEEP(*(SORT(.ctors.*)))
    KEEP(*(.ctors))
    /* new GCC version uses .init_array */
    KEEP (*(SORT(.init_array.*)))
    KEEP (*(.init_array))
    PROVIDE(__ctors_end__ = .);


    /* section information for finsh shell */
    . = ALIGN(8);
    __fsymtab_start = .;
    KEEP(*(FSymTab))
    __fsymtab_end = .;
    . = ALIGN(8);
    __vsymtab_start = .;
    KEEP(*(VSymTab))
    __vsymtab_end = .;
    . = ALIGN(8);

    /* section information for initial. */
    . = ALIGN(8);
    __rt_init_start = .;
    KEEP(*(SORT(.rti_fn*)))
    __rt_init_end = .;
    . = ALIGN(8);


    __spi_func_start = .;
    KEEP(*(.spi_call))
    __spi_func_end = .;

    . = ALIGN(8);

    __rt_utest_tc_tab_start = .;
    KEEP(*(UtestTcTab))
    __rt_utest_tc_tab_end = .;

    . = ALIGN(8);
    _etext = .;
} > SRAM

.eh_frame_hdr :
{
     *(.eh_frame_hdr)
     *(.eh_frame_entry)
} > SRAM
.eh_frame : ONLY_IF_RO { KEEP (*(.eh_frame)) } > SRAM

. = ALIGN(8);

.data :
{
```

```
        *(.data)
        *(.data.*)

        *(.data1)
        *(.data1.*)

        . = ALIGN(8);
        PROVIDE( __global_pointer$ = . + 0x800 );

        *(.sdata)
        *(.sdata.*)

        PROVIDE(__dtors_start__ = .);
        KEEP(*(SORT(.dtors.*)))
        KEEP(*(.dtors))
        PROVIDE(__dtors_end__ = .);

    } > SRAM

    /* stack for dual core */
    .stack :
    {
        . = ALIGN(64);
        __stack_start__ = .;

        . += __STACKSIZE__;
        __stack_cpu0 = .;

        . += __STACKSIZE__;
        __stack_cpu1 = .;
    } > SRAM

    .sbss :
    {
    __bss_start = .;
        *(.sbss)
        *(.sbss.*)
        *(.dynsbss)
        *(.scommon)
    } > SRAM

    .bss :
    {
        *(.bss)
        *(.bss.*)
        *(.dynbss)
        *(COMMON)
    __bss_end = .;
    } > SRAM

    _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) }
    /* DWARF 1.1 and DWARF 2 */
    .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) }
}
```

Part1

```
INCLUDE "link_stacksize.lds"

/*
 * The OUTPUT_ARCH command specifies the machine architecture where the
 * argument is one of the names used in the Kendryte library.
 */
OUTPUT_ARCH( "riscv" )
```

> 关键字：INCLUDE：导入另一个链接文件。

其中，`INCLUDE "link_stacksize.lds"`是另一个链接脚本，打开改文件只有一句，用来定义`stacksize`

```
/* bsp/k210/link_stacksize.lds */
__STACKSIZE__ = 4096;
```

符号`__STACKSIZE__`是指中断栈的栈空间，改参数通过`scons --menuconfig`来配置的，在`bsp/k210/Kconfig`中可以找到如下语句：

```
/* bsp/k210/Kconfig */
config __STACKSIZE__
    int "stack size for interrupt"
    default 4096
```

那这个配置又是如何被写到`link_stacksize.lds`中的呢？在`bsp/k210/SConstruct`中可以看到：

```
# bsp/k210/SConstruct
stack_size = 4096

stack_lds = open('link_stacksize.lds', 'w')
if GetDepend('__STACKSIZE__'): stack_size = GetDepend('__STACKSIZE__')
stack_lds.write('__STACKSIZE__ = %d;' % stack_size)
stack_lds.close()
```

在使用`scons`编译的时候，以上语句就会被执行，`__STACKSIZE__`的值就会被写入`link_stacksize.lds`文件中。

> 关键字：OUTPUT_ARCH:指定芯片架构。

`OUTPUT_ARCH( "riscv" )`用于指定芯片架构，K210是RISCV架构芯片。

Part2

```
MEMORY
{
   /* 6M SRAM */
   SRAM : ORIGIN = 0x80000000, LENGTH = 0x600000
}
```

MEMORY关键字用于描述一个MCU ROM和RAM的内存地址分布（Memory Map），MEMORY中所做的内存描述主要用于SECTIONS中LMA和VMA的定义。K210给用户使用的存储器是6MB的SRAM，起始地址为0x80000000。

> LMA和VMA：每个output section都有一个LMA和一个VMA，LMA是其存储地址，而VMA是其运行时地址，例如将全局变量g_Data所在数据段.data的LMA设为0x80000020（属于ROM地址），VMA设为0xD0004000（属于RAM地址），那么g_Data的值将存储在ROM中的0x80000020处，而程序运行时，用到g_Data的程序会到RAM中的0xD0004000处寻找它。需要注意的是LMA和VMA可以相同也可以不同。

下面我们首先来介绍MEMORY的语法，MEMORY的语法格式如下：

```
MEMORY
{
    <name> [(<attr>)] : ORIGIN = <origin>, LENGTH = <len>
    ...
}
```

> 关键字：MEMORY：声明一个或多个内存区域，其属性指定该区域是否可以写入、读取或执行。

其中`<name>`是所要定义的内存区域的名字，`<origin>`是其起始地址，`<len>`为内存区域的大小。另外，`<attr>`是可选的，并不重要，具体用法可参考GNU Linker的语法说明。

Part3

```
ENTRY(_start)
SECTIONS
{
```

> 关键字：ENTRY：命令用于设置入口点，也就是第一条指令所有的位置。

这里也就是`_start`标示的位置，也就是整个芯片上电后的运行首地址，然后才能一步步引导到内核中。`_start`位于`libcpu/risc-v/k210/startup_gcc.S`中：

```
/* libcpu/risc-v/k210/startup_gcc.S */
.global _start
  .section ".start", "ax"
_start:
  j 1f
  .word 0xdeadbeef
  .align 3
  .global g_wake_up
  g_wake_up:
      .dword 1
      .dword 0
1:
  csrw mideleg, 0
  csrw medeleg, 0
```

> 关键字：SECTIONS：官方的解释是“The SECTIONS command tells the linker how to map input sections into output sections, and how to place the output sections in memory.”，实际上可以把它看成一个描述符，所有的工作要在它的内部完成。
>
> 用于定义output section（输出段）的相应input section（输入段）、LMA和VMA，是整个连接脚本中最为重要的部分。注：output section是实际存储在内存中的“段”，而input section是其构成成员，如.data为数据段，由所有全局变量构成（默认情况下）；.text为代码段，由所有函数构成（默认情况下）...

> SECTIONS的语法如下：
>
> SECTIONS
> {
> <sections-command>
> <sections-command>
> ...
> }
>
> 其中主要的部分是<sections-command>，而SECTIONS{ }属于框架。<sections-command>的语法如下：
>
> ```
> <section> [<address>] [(<type>)] : [AT(<lma>)]
> {
>  <output-section-command>
>  <var{output-section-command>
>  ...
> } [><region>] [AT><lma region>] [:<phdr> :<phdr> ...] [=<fillexp>]
> ```

我们从使用的角度来讲解其语法：（假设有一个全局变量myData，我们用#pragma section命令将其定义为.myData段（input section））

（1）我们首先可以**定义output section**的名字，随便什么都可以，比如.my_data；

（2）然后我们可以定义其**构成成员**，*(.myData)；

（3）接下来我们就要指定.my_data的LMA和VMA了，有4种方法：

a) [

] + [AT()] ;

b) [

] + [AT>] ;

c) [>] + [AT>] ;

d) [>] + [AT()].

但是要注意这些用法的不同：[

] 和 [AT()]必须指定具体的地址，而 [>] 和 [AT>]只需指定内存空间，具体地址紧接着上一个output section的末尾地址。

经过以上步骤，我们得出如下section定义：（这里只列出2种）

```
SECTIONS
{
    .my_data ( 0xD0004000 ) : AT ( 0x80000020 )
    {
        *(.myData)
    }    ...
}
```

```
SECTIONS
{
    .my_data :
    {
        *(.myData)
    } > ram AT> rom    ...
}
```

以上为了说明SECTION的语法，使用了全局变量这种LMA和VMA不同的例子。而对于代码段.text这种LMA与VMA相同的情况，由于默认情况下LMA=VMA，因此可以只定义VMA而不必指明LMA，例如：

```
.text :
{
    *(.text)
    *(.text.*)
    . = ALIGN(4);
} > pfls0
```

7.最后，我们有必要提及“.”这个符号（不是.text、.data中的"."，而是如上例中.=ALIGN(4);中的"."），以下介绍来自于HighTec编译器手册

19.2.17 The Location Counter

The special linker variable *dot* '.' always contains the current output location counter. Since the . always refers to a location in an output section, it may only appear in an expression within a SECTIONS command. The . symbol may appear anywhere that an ordinary symbol is allowed in an expression.

Assigning a value to . will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may never be moved backwards.

```
SECTIONS
{
  output :
    {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . += 1000;
      file3(.text)
    } = 0x12345678;
}
```

In the previous example, the .text section from `file1` is located at the beginning of the output section output. It is followed by a 1000 byte gap. Then the .text section from `file2` appears, also with a 1000 byte gap following before the .text section from `file3`. The notation = 0 x12345678 specifies what data to write in the gaps (see subsubsection 19.2.9.5 on page 266).

> ! . actually refers to the byte offset from the start of the current containing object. Normally this is the SECTIONS statement, who's start address is 0, hence . can be used as an absolute address. If . is used inside a section description however, it refers to the byte offset from the start of that section, not an absolute address.

Thus in a script like this:

```
SECTIONS
{
    . = 0x100
    .text: {
      *(.text)
      . = 0x200


    }
    . = 0x500
    .data: {
      *(.data)
      . += 0x600
    }
}
```

The .text section will be assigned a starting address of 0x100 and a size of exactly 0x200 bytes, even if there is not enough data in the .text input sections to fill this area. (If there is too much data, an error will be produced because this would be an attempt to move . backwards). The .data section will start at 0x500 and it will have an extra 0x600 bytes worth of space after the end of the values from the .data input sections and before the end of the .data output section itself.

注意：input section 和 output section的概念非常重要。

Part4

```
SECTIONS
{
```

```
    . = 0x80000000 ;

    /* __STACKSIZE__ = 4096; */

    .start :
    {
        *(.start);
    } > SRAM

    . = ALIGN(8);

    ...
    ...
}
```

在链接脚本中`.`表示位置计数器，`. = 0x80000000 ;`里表示输出段从0x80000000开始，注意`;`符号，同时当前栈空间为`__STACKSIZE__`。

紧接着，将所有input section中的`.start`段放入output section的`.start`段。`.start`段是一个自定义段，我们在`libcpu/risc-v/k210/startup_gcc.S`中可以看到它的定义：

```
/* libcpu/risc-v/k210/startup_gcc.S */
.global _start
  .section ".start", "ax"
_start:
  j 1f
```

要注意的是`>SRAM`，该代码表示将该output section（`.start`）的VMA放入到SRAM中（此时，VMA==LMA）。

注意：位置计数器会随着输出文件中放入内容而自增，`ALIGN(8)`用于将位置计数器按8字节对齐。

Part5

```
SECTIONS
{
    ...
    ...

    .text :
    {
        *(.text)                        /* remaining code */
        *(.text.*)                      /* remaining code */
        *(.rodata)                      /* read-only data (constants) */
        *(.rodata*)
        *(.glue_7)
        *(.glue_7t)
        *(.gnu.linkonce.t*)

        . = ALIGN(8);
```

```
        PROVIDE(__ctors_start__ = .);
        /* old GCC version uses .ctors */
        KEEP(*(SORT(.ctors.*)))
        KEEP(*(.ctors))
        /* new GCC version uses .init_array */
        KEEP (*(SORT(.init_array.*)))
        KEEP (*(.init_array))
        PROVIDE(__ctors_end__ = .);


        /* section information for finsh shell */
        . = ALIGN(8);
        __fsymtab_start = .;
        KEEP(*(FSymTab))
        __fsymtab_end = .;

        . = ALIGN(8);
        __vsymtab_start = .;
        KEEP(*(VSymTab))
        __vsymtab_end = .;
        . = ALIGN(8);

        /* section information for initial. */
        . = ALIGN(8);
        __rt_init_start = .;
        KEEP(*(SORT(.rti_fn*)))
        __rt_init_end = .;

        . = ALIGN(8);


        __spi_func_start = .;
        KEEP(*(.spi_call))
        __spi_func_end = .;

        . = ALIGN(8);

        __rt_utest_tc_tab_start = .;
        KEEP(*(UtestTcTab))
        __rt_utest_tc_tab_end = .;

        . = ALIGN(8);
        _etext = .;
    } > SRAM

    ...
    ...
}
```

`.text`为代码段，由所有函数构成（默认情况下），首先放置所有输入文件中的`.text`、`.text.*`、`.rodata`、`.rodata*`段。输入文件的`.text`或`.text.*`段中通常存放的是程序代码，`.rodata`或`.rodata*`中通常存放的是只读数据，比如常量或者字符串常量等。

`.glue_7`和`.glue_7t`从查到的资料看是用于ARM和Thumb模式互相调用的，在RISC-V下似乎没有用：

> Stub sections generated by the linker, to glue together ARM and Thumb code. .glue_7 is used for ARM code calling Thumb code, and .glue_7t is used for Thumb code calling ARM code. Apparently always generated by the linker, for some architectures, so better leave them here.

`.gnu_linkonce.t*`与链接器的*vague linkage*有关，在网络上找到如下一段话：

> Those are the names of input section names. The names beginning with .gnu.linkonce are used by gcc for vague linking. See the docs in the gcc manual for why vague linking exists. When the GNU linker sees two input sections with the same name, and the name starts with ".gnu.linkonce.", the linker will only keep one copy and discard the other.

具体可以查看gcc手册里的Vague Linkage章节或者参考这篇文章http://50.116.14.174/blog/post

然后再次按照8字节对齐一下。

在使用C++特性后，即开启`#define RT_USING_CPLUSPLUS`后，会有大量报错。从编译后的错误提示，可以明确，这里少几个符号链接：`ctors_start__`、`ctors_end__`和`__dtors_start`、`dtors_end`。 这几个链接符号分别为 C++ 全局构造代码段和数据段分配起始地址和结束地址，这样，全局构造函数和数据在系统初始化的时候，就会被链接到这里分配的段地址中。

> 关键字：PROVIDE：用于定义这类符号：在目标文件内被引用，但没有在任何目标文件内被定义的符号。

`PROVIDE`为在任何链接目标中没有定义但是被引用的一个符号,而在链接脚本定义一个符号。`PROVIDE (symbol = expression)`。提供定义`_exfun`的例子:

```
SECTIONS
{
  .text :
    {
      *(.text)
      _exfun = .;
      PROVIDE(_exfun = .);
    }
}
```

如果程序定义了' _exfun '(带有前导下划线)，链接器将给出重复定义错误。另一方面，如果程序定义了' exfun'(没有前导下划线)，链接器会默认使用程序中的定义。如果程序引用了' exfun '但没有定义它，链接器将使用链接器脚本中的定义。

目标文件内引用了`__ctors_start__`符号，确没有定义它时，`__ctors_start__`符号对应的地址被定义为.text 的output section之后的第一个字节的地址。

然后针对不同版本GCC，进行了不同的配置。

> 关键字：KEEP：将指示链接器保留指定的节，即使其中没有引用任何符号。

当在链接时执行垃圾收集时，这一点就变得很重要，在链接命令行内使用了选项 -gc-sections 后，链接器可能将某些它认为没用的 section 过滤掉，此时就有必要强制让链接器保留一些特定的 section，可用 KEEP() 关键字达此目的。说的通俗易懂就是：防止被优化。
该语句常见于针对ARM体系结构的链接器脚本中，用于将中断向量表放置在偏移量0x00000000处。如果没有这个指令，代码中可能不会显式引用的表将被删除。

> 关键字：SORT：对满足字符串模式的所有名字进行递增排序，如 SORT(.text*)。

再一次8字节对齐内存操作。

`FSymTab`和`VSymTab`

如链接脚本中注释，`FSymTab`和`VsymTab`是finsh shell使用的。在使用`MSH_CMD_EXPORT`和`FINSH_VAR_EXPORT`将函数或变量导出到finsh shell时，就会相应地在`FSymTab`段和`VsymTab`段中放置内容。
`__fsymtab_start`、`__fsymtab_end`、`__vsymtab_start`、`__vsymtab_end`在finsh shell初始化时用到：

```
/* components/finsh/shell.c */
int finsh_system_init(void)
{
    rt_err_t result = RT_EOK;
    rt_thread_t tid;

#ifdef FINSH_USING_SYMTAB
#if defined(__CC_ARM) || defined(__CLANG_ARM)          /* ARM C Compiler */
    extern const int FSymTab$$Base;
    extern const int FSymTab$$Limit;
    extern const int VSymTab$$Base;
    extern const int VSymTab$$Limit;
    finsh_system_function_init(&FSymTab$$Base, &FSymTab$$Limit);
#ifndef FINSH_USING_MSH_ONLY
    finsh_system_var_init(&VSymTab$$Base, &VSymTab$$Limit);
#endif
#elif defined (__ICCARM__) || defined(__ICCRX__)       /* for IAR Compiler */
    finsh_system_function_init(__section_begin("FSymTab"),
                               __section_end("FSymTab"));
    finsh_system_var_init(__section_begin("VSymTab"),
                          __section_end("VSymTab"));
#elif defined (__GNUC__) || defined(__TI_COMPILER_VERSION__)
    /* GNU GCC Compiler and TI CCS */
    extern const int __fsymtab_start;
    extern const int __fsymtab_end;
    extern const int __vsymtab_start;
    extern const int __vsymtab_end;
    finsh_system_function_init(&__fsymtab_start, &__fsymtab_end);
    finsh_system_var_init(&__vsymtab_start, &__vsymtab_end);
#elif defined(__ADSPBLACKFIN__) /* for VisualDSP++ Compiler */
    finsh_system_function_init(&__fsymtab_start, &__fsymtab_end);
    finsh_system_var_init(&__vsymtab_start, &__vsymtab_end);
#elif defined(_MSC_VER)
    unsigned int *ptr_begin, *ptr_end;
    ...
}
```

又一次8字节对齐内存操作。

定义`__rt_init_start`和`__rt_init_end`内存位置，用于界定自动初始化起始地址和结束地址，好像没有用到。

.rti_fn*

在RT-Thread中如果想要某个初始化函数被自动执行，可以使用如下操作将其导出`INIT_BOARD_EXPORT`、`INIT_PREV_EXPORT`、`INIT_COMPONENT_EXPORT`、`INIT_ENV_EXPORT`、`INIT_APP_EXPORT`，这些宏实现了不同层次的自动初始化机制。而这些宏实际执行的操作就是将被导出的函数放入`.rti_fn.level`段中，其中level对应不同宏的层级。详细原理可以参考RT-Thread 自动初始化详解。

又一次8字节对齐内存操作。

UtestTcTab

`UtestTcTab`是RT-Thread的utest测试框架用到的段，具体参考RT-Thread的官方说明RT-Thread Utest测试。

又一次8字节对齐内存操作。

spi_call

不清楚干了啥.....

又一次8字节对齐内存操作。

_etext

`_etext`用于标示输出文件中代码段的结束，不过代码中并没有使用到。

又一次`>SRAM`，该代码表示将该output section (`.start`) 的VMA放入到SRAM中（此时，VMA==LMA）。

Part6

```
SECTIONS
{
    ...
    ...

    .eh_frame_hdr :
    {
         *(.eh_frame_hdr)
         *(.eh_frame_entry)
    } > SRAM
    .eh_frame : ONLY_IF_RO { KEEP (*(.eh_frame)) } > SRAM

    . = ALIGN(8);

    ...
    ...
```

```
}
```

当使用支持异常的语言比如C++时，处理异常时需要提供额外的信息来描述call frames。这些信息会包含在`.eh_frame`和`.eh_frame_hdr`段中。体参考https://refspecs.linuxfoundation.org

`ONLY_IF_RO`用来说明输出段只有在所有对应输入段都是只读时才会被创建。

> You can specify that an output section should only be created if all of its input sections are read-only or all of its input sections are read-write by using the keyword ONLY_IF_RO and ONLY_IF_RW respectively.

> 关键字：PROVIDE_HIDDEN：`PROVIDE`仅在程序没有定义这个符号时，才去定义这个符号。`HIDDEN`说明该符号对外不可见。

Part7

```
SECTIONS
{
    ...
    ...

    .data :
    {
        *(.data)
        *(.data.*)

        *(.data1)
        *(.data1.*)

        . = ALIGN(8);
        PROVIDE( __global_pointer$ = . + 0x800 );

        *(.sdata)
        *(.sdata.*)

        PROVIDE(__dtors_start__ = .);
        KEEP(*(SORT(.dtors.*)))
        KEEP(*(.dtors))
        PROVIDE(__dtors_end__ = .);

    } > SRAM

    ...
    ...
}
```

`.data`为数据段，主要包含已初始化的数据。

.sdata

关于`.sdata`段：

> .sdata is the section for initialized data, which is accessed using GP relative addressing. .sdata sections are accessed using shorter addresses. Thus code size (and may be execution time) will decrease if often used data is placed in .sdata sections instead of .data sections. For functionality there is no difference between data placement in .data or .sdata sections.

__global_pointer$

关于为什么要使用`__global_pointer$`参考https://www.sifive.com/blog/all-aboard-part-3-linker-relaxation-in-riscv-toolchain，RTT-Thread下K210的启动代码中并没有用到。

dtors_start 和 dtors_end

构造函数和析构函数会被分别放入`.ctors`和`.dtors`中。参考
https://gcc.gnu.org/onlinedocs/gccint/Initialization.html
注：可见使用C++时，需要将上述的`.eh_frame_hdr`、`eh_frame`、`.preinit_array`、`.init_array`、`fini_array`、`.ctors`、`.dtors`段找描述都写入链接脚本。

所以在使用C++特性时，需要和上面.text段中的.ctors一样处理。

Part8

```
SECTIONS
{
    ...
    ...

    /* stack for dual core */
    .stack :
    {
        . = ALIGN(64);
        __stack_start__ = .;

        . += __STACKSIZE__;
        __stack_cpu0 = .;

        . += __STACKSIZE__;
        __stack_cpu1 = .;
    } > SRAM

    ...
    ...
}
```

`.stack`为栈段，定义两个core的栈分布。

其中，`__stack_start__`有被`libcpu`中的`interrupt_gcc.S`和`startup_gcc.S`使用，比如下列代码：

```
  la   sp, __stack_start__
  addi t1, a0, 1
  li   t2, __STACKSIZE__
  mul  t1, t1, t2
  add  sp, sp, t1 /* sp = (cpuid + 1) * __STACKSIZE__ + __stack_start__ */
```

Part9

```
SECTIONS
{
    ...
    ...

   .sbss :
   {
    __bss_start = .;
        *(.sbss)
        *(.sbss.*)
        *(.dynsbss)
        *(.scommon)
    } > SRAM

    .bss :
    {
        *(.bss)
        *(.bss.*)
        *(.dynbss)
        *(COMMON)
    __bss_end = .;
    } > SRAM

    ...
    ...
}
```

`.bss`段，该段初始化全部为0，对应系统中下列代码：

```
/* src/board.c */
void init_bss(void)
{
    unsigned int *dst;

    dst = &__bss_start;
    while (dst < &__bss_end)
    {
        *dst++ = 0;
    }
}
```

注意：`.sbss`和`.bss`中都包含未初始化的数据。关于两者的区别，有点像`.sdata`和`.data`的区别：

> The difference between .bss and .sbss is dependent on the architecture.
> Some architectures have fast addressing modes that can be used to address only a limited amount of memory, and the data in .sbss is addressed using these addressing modes. For example, the MIPS uses GP-relative addressing for .sbss data. By default, data whose size is 8 bytes or fewer is placed in .sbss. Anything larger than 8 bytes is placed in .bss.
> Some architectures do not use .sbss.

另一个解释：.sbss是小的BSS段，用于存放"近"数据，即使用短指针（near）寻址的数据。有利于小的对象组合到单个可以直接寻址的区域。《程序员的自我修养 - - 链接、装载与库》一书的3.3.4节说："以前用过的一些名字如.sdata、.tdesc、sbss、lit4、lit8、reginfo、gptab、liblist、.confict。可以不用理会这些段，它们已经被遗弃了。"话虽然这么说，不过实际使用中还是会碰到。原文链接：https://blog.csdn.net/adaptiver/article/details/6749112

另外，`__bss_end`也是堆的开始：

```
/* bsp/k210/driver/board.h */
#define RT_HW_HEAP_BEGIN    (void*)&__bss_end
#define RT_HW_HEAP_END      (void*)(0x80000000 + 6 * 1024 * 1024)
```

Part10

```
SECTIONS
{
    ...
    ...

    _end = .;

    ...
    ...
}
```

到这里程序用到的段就完事了。

Part11

```
SECTIONS
{
    ...
    ...

    /* 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) }
    /* DWARF 1.1 and DWARF 2 */
    .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) }

    ...
    ...
}
```

调试用到的段，好像RT-Thread的K210不能调试。

参考

[1]. 配置RT-Thread支持C++特性

[2]. RT-Thread下Kendryte K210的链接脚本注解

[3]. 链接脚本（Linker Script）用法解析（一） 关键字SECTIONS与MEMORY

[4]. 链接脚本.lds（详细）总结附实例快速掌握

[5]. 内核链接脚本分析

[6]. 链接脚本的作用及格式