STM32CubeIDE 实用技巧之 ld 链接文件

关键字：STM32CubeIDE，链接文件

前言

STM32CubeIDE 是 ST 推出的免费集成编译环境，基于 Eclipse 开源框架，集成了 GCC、GDB 等免费的编译器、链接器，支持 STM32 全系列芯片，可以创建 C/C++工程，支持调试、波形实时仿真、一键下载等。

在实际项目中，有时候需要对内存进行细分时，比如指定变量/函数/文件到特殊地址等等，KEIL 可以通过“*.sct”文件来实现；IAR 可以通过“*.icf”文件来实现；对于 STM32CubeIDE，可以通过“*.ld”链接文件来实现。

本文将介绍 GCC 的“*.ld”链接文件的常见用法，供大家参考使用。

基本概念

“*.ld”链接文件组合了许多对象和归档文件，重新定位它们的数据并绑定符号引用。通常，编译程序的最后一步是运行“*.ld”链接文件。

通俗来讲，链接文件可以描述输入文件中的段，将其映射到输出文件中，并指定输出文件中的内存分配。

以下就是链接文件涉及到的相关概念：

内存（Memory）

语法：

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

注释：这里的“attr”只能由以下特性组成：

‘R’	Read-only section
‘W’ --	Read/write section
‘X’ --	Executable section
‘A’ --	Allocatable section
‘I’ --	Initialized section
‘L’ --	Same as ‘I’
‘!’ --	Invert the sense of any of the attributes that follow

示例：

```
/* Memories definition */
MEMORY
{
      RAM    (xrw)    : ORIGIN = 0x20000300,   LENGTH = 36K
      FLASH    (rx)   : ORIGIN = 0x08000000,   LENGTH = 128K
}
```

注释：

“xrw”表示“RAM”区是可读、可写和可执行的，且 RAM 的起始地址为“0x20000000”，长度为 36K。

“rx”表示“FLASH”区是可读和可执行的，FLASH 的起始地址为“0x08000000”，长度为 128K。

段（Section）

Section 有 loadable(可加载) 和 allocatable(可分配)两种类型。不可加载也不可分配的内存段，通常包含某些调试信息。

loadable(可加载)是指：程序运行时，该段内容应该被加载到内存中。

allocatable(可分配)是指：该段的内容应该被预留出，但不应该加载任何别的内容（某些情况下，这些内存必须归零）。

“可加载”和“可分配”的 section 都有两个地址：“VMA”和“LMA”。

VMA（the virtual memory address）：这是运行输出文件时，该 section 的地址。VMA 是可选项，可以不设置。

LMA（load memory address）：这是加载 section 时的地址。

在大多数情况下，这两个地址是相同的。当然也可以不相等，比如下面的例子就是 LMA 和 VMA 不同的案例：数据段被加载到 ROM 中，然后在程序启动时复制到 RAM 中（通常用于初始化全局变量）。此时 ROM 地址就是 LMA，RAM 地址就是 VMA。

语法：

```
SECTIONS
{
        section [address] [(type)] :
        {
            [AT(lma)]
            [ALIGN(section_align) | ALIGN_WITH_INPUT]
            [SUBALIGN(subsection_align)]
            [constraint]
            {
              output-section-command
              output-section-command
              ...
        } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp] [,]
        ...
}
```

注释：大多数的段仅使用了上述的一部分属性。

示例：

```
/* Sections */
SECTIONS
{
  /* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* Initialized data sections into "RAM" Ram type memory */
  .data :
  {
```

```
    . = ALIGN(4);
    _sdata = .;         /* create a global symbol at data start */
    *(.data)            /* .data sections */
    *(.data*)           /* .data* sections */

    . = ALIGN(4);
    _edata = .;         /* define a global symbol at data end */

  } >RAM AT> FLASH
}
```

注释：上述示例中“.isr_vector”的 LMA 与 VMA 是相等的。“.data”因为有“>RAM AT> FLASH”的修饰，表示.data 段的 VMA 为 RAM，LMA 为 FLASH。即.data 段的内容会放在 FLASH 中，但是运行时，会加载到 RAM 中。

链接脚本

完整的“*.ld”链接文件通常会包含入口点、memory 以及 section 的内容。

入口点

语法：ENTRY(symbol)

用途：程序中要执行的第一个指令，也称为入口点。

示例：

```
/* Entry Point */
ENTRY(Reset_Handler)
```

注释：在 STM32 的工程中，默认的入口点是“Reset_Handler”函数。

常用命令

1. ASSERT

语法：ASSERT(exp, message)

确保 exp 是非零值，如果为零，将以错误码的形式退出链接文件，并输出 message。

用途：在必要的位置添加断言，可以清晰的定位问题。

示例：

```
/* The usage of ASSERT */
.test :
{
    ASSERT ((_estack > (_Min_Stack_Size + _Min_Heap_Size)),"Error: There is an ERR
occurred");
}
```

注释：当示例中的“_estack”大于“_Min_Stack_Size + _Min_Heap_Size”时，会出现如下的信息。

```
c:\st\stm32cubeide_1.4.0\stm32cubeide\plugins\com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.7
-2018-q2-update.win32_1.4.0.202007081208\tools\arm-none-eabi\bin\ld.exe: Error: There is an ERR occurred
collect2.exe: error: ld returned 1 exit status
make: *** [makefile:50: L053_KS_Example_Sec.elf] Error 1
"make -j8 all" terminated with exit code 2. Build might be incomplete.
```

2. PROVIDE

语法：PROVIDE(symbol = expression)

用途：在某些情况下，链接器脚本只需要定义一个被引用的符号，并且该符号不是由链接中包含的任何对象定义的。

示例：

```
    PROVIDE (TEST_Symbol = .);
```

3. HIDDEN

语法：HIDDEN(symbol = expression)
用途：对于 ELF 目标端口，符号将被隐藏且不被导出。
示例：

```
      HIDDEN (TEST_Symbol = .);
```

4. PROVIDE_HIDDEN

语法：PROVIDE_HIDDEN (symbol = expression)
用途：和 PROVIDE 用法相同，是 PROVIDE 和 HIDDEN 的结合体。
示例：

```
      PROVIDE_HIDDEN (__preinit_array_start = .);
```

5. KEEP

用途：当链接器使用('--gc-sections') 进行垃圾回收时，KEEP()可以使得被标记段的内容不被清除。
示例：

```
/* The startup code into "FLASH" Rom type memory */
.isr_vector :
{
     . = ALIGN(4);
     KEEP(*(.isr_vector)) /* Startup code */
     . = ALIGN(4);
} >FLASH
```

简单脚本示例

首先我们来看一个简单的脚本示例：

```
SECTIONS
{
      . = 0x10000;
      .text : { *(.text) }
      . = 0x8000000;
      .data : { *(.data) }
      .bss : { *(.bss) }
}
```

注释：这里指定了 code、已初始化的数据以及未初始化的数据的内存分布。对于这个示例，程序会在地址 0x10000 处装载代码，并且数据会从地址 0x8000000 开始。特殊符号“.”，是位置计数器（location counter），按输出段的大小递增，设置位置计数器可以改变输出段的地址，在“SECTIONS”命令的开头，位置计数器的值为“0”。位置计数器可以进行算数运算。

高阶使用

当一个工程需要对内存进行特殊调整和配置的时候，这就要求我们对 ld 链接文件有较深的理解才可以灵活运用。
这里列举常用高阶操作：

位置计数器

在 section 的描述中，位置计数器“.”可以进行算术运算，由此产生空隙来满足特定需求。

```
.SE_CallGate_Fun :
{
    . = ALIGN(4);
```

```
    . = . + 0x04;
    *(.SE_CallGate_Fun)
    . = ALIGN(4);
} >CG_FLASH
```

注释：例如这里的“. = . + 0x04;”就实现了 SE_CallGate_Fun 段的空间中，插入一段 0x04 的空隙。

指定“变量”的输出地址

可以定义如下的 memory，然后将“变量”存放于该 memory，就能控制“变量”的输出地址。

```
/* Memories definition */
MEMORY
{
     FW_RAM (xrw)    : ORIGIN = 0x20000000,   LENGTH = 0x300 /* 0x20000000 ~
0x200002FF */
     RAM    (xrw)    : ORIGIN = 0x20000300,   LENGTH = 35K
     FLASH    (rx)   : ORIGIN = 0x08000000,   LENGTH = 128K
}
```

同时在 c 文件中，在定义“变量”时，添加如下对应的属性

```
__attribute__((section(".FW_RAM"))) uint8_t key[8] = {0,1,2,3,4,5,6,7 };
```

注释：变量将位于“0x20000000 ~ 0x200002FF”区域（如果仅仅只有 key 数组位于该区域，将从 0x20000000 开始存放，如果有多个变量存储于该区域，将按照编译的顺序，从 0x20000000 依次存放）。

指定“函数”的输出地址

可以定义如下的 memory 和 section，然后将“函数”存放于该 section，就能控制“函数”的输出地址。

```
/* Memories definition */
MEMORY
{
  FLASH    (rx)    : ORIGIN = 0x08000000,   LENGTH = 0x300 /* 0x08000000 ~ 0x080002FF
*/
  CG_FLASH (rx)    : ORIGIN = 0x08000300,   LENGTH = 0x134 /* 0x08000300 ~ 0x08000433
*/
  RAM    (xrw)     : ORIGIN = 0x20000300,   LENGTH = 0x900 /* 0x20000300 ~ 0x20001FFF
*/
}

/* Sections */
SECTIONS
{
      …
      .SE_CallGate_Fun :
      {
          . = ALIGN(4);
          . = . + 0x4;
          *(.SE_CallGate_Fun)
          . = ALIGN(4);
      } >CG_FLASH
      …
}
```

同时在 c 文件中，在“函数”的实现部分，添加如下对应的属性：

```
__attribute__((section(".SE_CallGate_Fun"))) uint32_t call_gate(Callgate_Func_Type_t
ftype, void *param)
```

注释：函数“call_gate”将存放于 0x08000304 处（留意此处的位置计数器将产生 0x04 的内存间隙）。

指定“文件”的输出地址

可以定义如下的 memory 和 section，然后将指定的文件存放于该 section，就能控制“文件”的输出地址。

```
/* Memories definition */
MEMORY
{
  FLASH    (rx)    : ORIGIN = 0x08000000,   LENGTH = 0x300 /* 0x08000000 ~ 0x080002FF
*/
  FW_FLASH (rx)    : ORIGIN = 0x08000434,   LENGTH = 0x2BCC/* 0x08000434 ~ 0x08003000
*/
  RAM    (xrw)     : ORIGIN = 0x20000300,   LENGTH = 0x900 /* 0x20000300 ~ 0x20001FFF
*/
}

/* Sections */
SECTIONS
{
       …
       .main_section :
       {
            . = ALIGN(4);
           Core/Src/main.o(.text*)
            . = ALIGN(4);
       } >FLASH
       …
}
```

注释：示例中将 main.o 指定到 FLASH 区域中；更改 FLASH 的地址或者 main_section 的 LMA，就可以实现将特定文件指定到特定内存区域。

总结

STM32CubeIDE 链接文件（*.ld）的详细说明文档位于“Help”-> “Read STM32CubeIDE Documentation”-> “C/C++ Linker”（ld.pdf）。该文档对 ld 文件的格式和内容进行了详细的说明，需要时可以查阅。
实际项目中，通过综合利用上述的技巧，可以实现对内存的准确分配，来满足不同的应用场景。

重要通知 – 请仔细阅读

意法半导体公司及其子公司 （“ST”）保留随时对 ST 产品和 / 或本文档进行变更的权利，恕不另行通知。买方在订货之前应获取关于 ST 产品的最新信息。 ST 产品的销售依照订单确认时的相关 ST 销售条款。

买方自行负责对 ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的 ST 产品如有不同于此处提供的信息的规定，将导致 ST 针对该产品授予的任何保证失效。

ST 和 ST 徽标是 ST 的商标。若需 ST 商标的更多信息，请参考 *www.st.com/trademarks*。所有其他产品或服务名称均为其 各自所有者的财产。

本文档是 ST 中国本地团队的技术性文章，旨在交流与分享，并期望借此给予客户产品应用上足够的帮助或提醒。若文中内容存有局限或与 ST 官网资料不一致，请以实际应用验证结果和 ST 官网最新发布的内容为准。您拥有完全自主权是否采纳本文档（包括代码，电路图等）信息，我们也不承担因使用或采纳本文档内容而导致的任何风险。

本文档中的信息取代本文档所有早期版本中提供的信息。