

目录

目录	1
1. 绪论	2
2. 使用自定义的.ld 文件	2
3. 内存地址说明	3
4. 一般性定义常量在指定 Flash 地址	4
5. 一般性定义变量及函数在指定 RAM 空间	5
6. 特殊填充的实现	6
典型程序空间拆分	6
段内偏移拆分与差异填充值	6
7. 基于文件与段名的匹配规则实施布局输出	7
文件规则应用	7
库文件规则应用	7
全文件捕获应用	8
8. 特殊应用：低功耗模式下数据保持应用	8
9. 典型应用：bootloader 双程序开发模式应用	9
10. 宏脚本分支(语法不支持，可应用控制)	10
11. 版本历史	10
附录 A: MAP 文件内容的组织	11
附录 B: 基本脚本变量与描述	13
附录 C: 拆分 data 段表述示例脚本	14

1. 绪论

在编译一个项目程序时，将每个输出文件链接在一起，其使用“.ld”脚本描述文件组成输出。即一个链接过程都由链接脚本控制，链接脚本主将定义的 **section** 对与文件内的输出文件读取、合并并生成目标文件。

通过对脚本的调整可以实现特殊应用的程序布局实现，如 bootloader 双应用程序开发，自带部分升级特性的指定函数段地址，或差异化应用的 RAM 变量指定地址。

KF32 输出文件格式为 elf32。Ram 空间设计部分地址空间为低功耗唤醒不丢失数据区域，如前面 16K，具体应以产品规格书为准，即变型使用 startup 和差异化唤醒的 startup2[控制 for 循环跳过. lpdata 段]。

默认情况下程序代码函数段归属为 flash 空间下的 .text，全局初始化常量归属为 flash 空间下 .rdata 段，局部初始化常量归属为 flash 空间下 .rodata 段。全局初始化变量归属为 ram 空间下 .data，全局未初始化变量归属为 ram 空间下 .bss 或 COMM。另外 C++ 类的全局初始化与和初始化列表归属为 flash 空间下的 .init 和 .init_array。

脚本中具有 = 运算中的 . 为语法格式的当前偏移量地址。段配置格式为 file(内容段)，其中 * 为通配符，如 *.text* 对应任意文件的代码 text 起始段。*vector.o(.text*) 对应编译输出对象 vector.o 文件下的代码 text 起始段。

__text_end__、__data_start__、__data_end__、__bss_start__、__bss_end__、__allot_end__、__initial_sp 等为必须的脚本符号(变量)，该符号构成工具实现的部分。其中 __data_start__ 必须为映射到 ram 的 data 起始部分，若需要保留 RAM 的前 N 空间，可以通过修改 MEMORY 下的 ram : ORIGIN = 0x10000000, LENGTH = 0x00004000 实现偏移和保留后的 ram 可用空间长度。

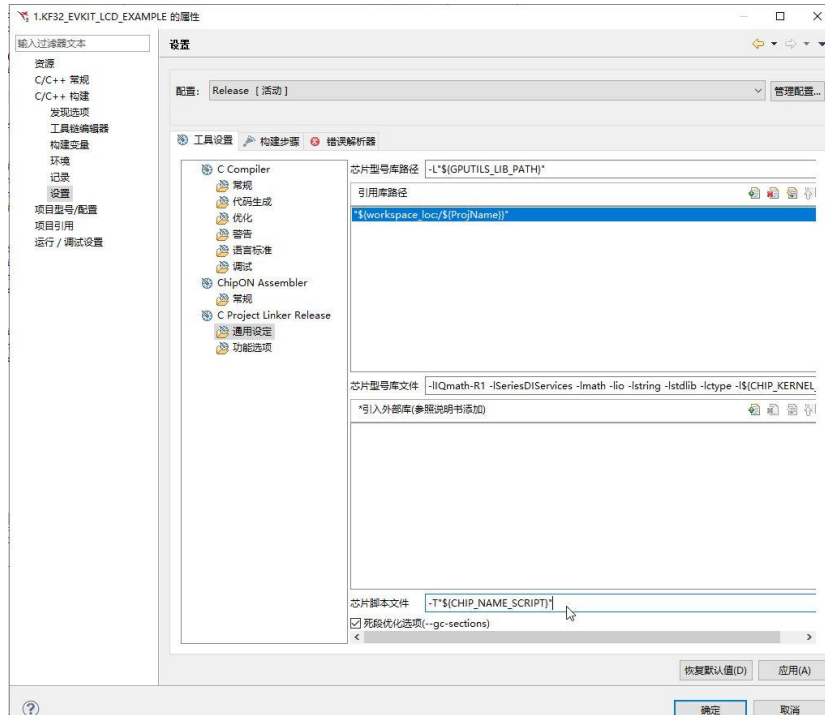
程序文件的生成会将 ram 段的全局初始化部分附加在 flash 段的 __text_end__ 位置后面空间，即代码与全局数据构成完整的程序内容。

脚本变量是一种只有地址没有值的变量，可以做源码中声明与使用。

更多见官网资料：<https://sourceware.org/binutils/docs-2.40/ld.html>。

2. 使用自定义的.ld 文件

编译过程，IDE 默认从安装目录下获取 .ld 文件。默认的路径通过工程属性->C/C++ 构建->C Link 是如图，其中“\${CHIP_NAME_SCRIPT}”为工具变量：



默认的.ld 文件在安装目录下的：ChipONIDE/KungFu32/ChiponCC32/[选择工具链，如 **ccrl_issue**]/scripting /[项目型号如 **KF32L530MNS**].ld。

将该路径的对应的项目型号相关的 ld 文件复制到工程路径下，更改芯片脚本文件为：

-T\"{ProjDirPath}/{CHIP_NAME}.ld”，或-T\"../[脚本文件名如 KF32L530MNS].ld”。

其中\${X}为 IDE 变量引用格式，如字面意思 ProjDirPath 的项目所在路径和 CHIP_NAME 的项目使用型号。

3. 内存地址说明

如 KF32L530MNS 的内存拥有 512K 的 Flash 和 128 的 RAM。Flash 的起始地址为 0x0000 0000；RAM 的起始地址为 0x1000 0000。链接文件中根据型号资源配置 Flash 和 RAM 的起始地址及长度。脚本下内存配置如下：

MEMORY

```
{
  flash      : ORIGIN      = 0x00000000, LENGTH = 0x00010000
  ram        : ORIGIN      = 0x10000000, LENGTH = 0x00004000
  da_mem     : ORIGIN      = 0x0C001C00, LENGTH = 0x00000400
  mode_mem   : ORIGIN      = 0x0C001800, LENGTH = 0x00000004
  pro_mem    : ORIGIN      = 0x0C001000, LENGTH = 0x00000004
  ee_mem     : ORIGIN      = 0x7F000000, LENGTH = 0x00001000
  config_mem1 : ORIGIN      = 0x31000000, LENGTH = 0x00000010
  config_mem2 : ORIGIN      = 0x31000010, LENGTH = 0x00000010
}
```

注：内存的起始地址与长度只能是常量，但可以是灵活常量也可以做运算。如 **0x400 1024 1K 64K-8** 均为有效的表达。

编程相关空间为代码 **flash** 和变量 **ram**，其他空间为辅助的定义 **hex** 文件内容下地址偏移与大小，如 **da_mem** 的 **flash date** 区域，**mode_mem** 的模式配置区域，**pro_mem** 的加密模式区域，其他保留或扩展。

脚本文件下布局控制实现段配置格式如下，详细见文件与控制描述部分。

SECTIONS

```
{
  .text :
  {
    ...
  } > flash
  .data :
  {
    ...
  } > ram
  ...
}
```

注：在定义 **flash** 或 **ram** 段时，段名结尾必须以*为结束，代码部分不需要。即同名合并原则下，工具给予每个函数或变量段后缀从而确定唯一，链接器默认开启死段优化，即未使用代码的不分配空间，同时意味着不能配置相同前缀的段，如 **A*** 与 **AAA*** 将视为相同。针对资源赋值可以使用综合的数据表达，如 **0x00010000**、**64K**、**2K+192**，在 **SECTIONS** 下的段控制 **. = 0xXXX**(其中“=”前后必须使用空格间隔)的值为基于当前的偏移，即非目标区域的绝对地址，该最终值由链接器基于 **MEMORY** 的起始值计算实现。

若指定地址的分割空间，可以指定填充值，如在 **flash** 填充区域设置填充值 **0x12345678**，则修改对应 **text** 段格式如下：

```
.text :
{
  ...
} > flash =0x12345678
```

4. 一般性定义常量在指定 Flash 地址

在应用中若需要将常量指定 **Flash** 地址，可以在 **text** 段进行段定位。在 **text** 前 512 字节需要预留给向量表存放，不可占用【即芯片启动机制：中断向量表与初始 SP 和程序地址入口】。

```
.text :
{
  . = 0x0000;
  KEEP (*vector.o(.text*)) /* chip interrupt vector ,writed in file named vector.s or
vector.c */
```

当前工具设计向量表使用独立的文件，即 **vector.c** 或 **vector.asm**。因此设计了该文件代码 **obj** 优先存放，并对应的代码 **text** 段存放的实现中断向量表入口。另外保存了对象名字为 **_start** 的与脚本指定入口 **ENTRY(_start)** 一致。【其中 **KEEP** 关键字修饰指明不参与死段优化，若向量表混合到其他文件或同文件下具有多个 **text** 段时，应调整脚本使能一致，如 **KEEP (*(.vector*))** 并向量表内容段使用 **__attribute__((section(".vector")))** 修饰】。

示例定义常量参数代码的起始段，如定义在向量表后的 **0x0000 0200** 处。修改脚本为：

```
  . = 0x0000;
  KEEP (*vector.o(.text*)) /* chip interrupt vector ,writed in file named vector.s or
vector.c */

  . = (. + 3) & (-4); /* align 4 */
  __vec_end__ = .;
  . = 0x200; /* after and befor '=' must space by write */
```

```
KEEP (*.ConstData*) /* define code section function ,must end with '*' */
*(.text*) /* default function code space */
```

代码编写使用 **section** 关键字指定,如

```
__attribute__((section(".ConstData"))) uint8 Test[] = "12345678";
```

编译通过后可以通过.map 文件或者 HEX 文件查看定义的位置是否生效。

若定义在代码的结束段定义,需要保证不覆盖代码,示例代码量小于等于 0x4000,修改脚本实现常量信息存放在 0x4000:

```
*(.userrodata*)
.= 0x4000; /* after and before '=' must space by write */
KEEP (*.ConstData*) /* define code section function ,must end with '*' */
.= (. + 3) & (-4);
__text_end__ = .;

/*record: global variable value of initialization */

} > flash
```

5. 一般性定义变量及函数在指定 RAM 空间

RAM 在上电时内容是随机的,使用前需要将拥有初值的变量赋值。如在"./config/vector.c"文件中规定了单片机从"startup"函数开始运行,"startup"函数的作用是给拥有初值的变量赋值。即在运行 main 之前,将变量初始化完毕。

在应用中若需要将指定 RAM 地址,即在.data 中定义需要用户段".xxxx",并指定其偏移地址。示例定义在 0x10001000 的 UserRAMData 修改脚本如下:

```
*(.indata*) /* server space for ram function */
*(.inrdata*)
*(.inrodata*)

.= 0x1000;
KEEP(*(.UserRAMData*))
*(.UserRAM2Data*)

.= (. + 3) & (-4);
*(.data*) /* global variable with initialization */

.= (. + 3) & (-4);
__data_end__ = .;
```

代码部分追加属性修饰指定段,如 __attribute__((section(".UserRAMData"))) uint8 Testram[] = "12345678";编译通过后可以通过.map 文件或者 反汇编 lst 文件查看定义的位置是否生效。

NOTE: 需要注意的事,编译处理变量默认段为.data、.bss 或 COMMON,并且同属性合并连续存放,__data_end__与__bss_start__应连续,即中间不应插入自定义段。

__data_start__与__data_end__之间的内容作为全局带初始化识别,即若区间指定段的无初始化默认内容为 0.该部分数据内容会附加到 flash 的__text_end__标明空间后面。

NOTE: 设计在__data_end__之后的固定地址段变量应该代码编写为初始化的全局变量(初始化值会丢失),设计在__data_start__与__data_end__之间的指定地址段应该将地址分配在前面,即顺序使用安排,若不连续并需要建立在靠后,建议地址与实际需求结尾地址相

近并被. data 段间隔的 data 空间应能满足默认全局变量空间需求(否则基于__data_start__与__data_end__规则填充 0 占用额外占用 flash 空间存放与增长初始化时间)。

6. 特殊填充的实现

可以处理器的Flash空间拆分为多个连续或不连续空间。推荐使用建立不同的内存对象与对应段的布局，. text仍应作为默认通用段落的保持，其将实现将. data段的初始化数据进行附加实现启动引导的初始化。示例如下：

典型程序空间拆分

```
MEMORY
{
    flash          : ORIGIN    = 0x00000000, LENGTH = 64K
    Flash2         : ORIGIN    = 64K,      LENGTH = 1K
    Flash3         : ORIGIN    = 65K,      LENGTH = 1K
    .....
```

注：不应该定义资源重合的memory段，即地址在芯片端为唯一资源。

```
SECTIONS {

    .text : {
        . = 0x0000;
        KEEP (*vector.o(.text*))
        *(.text* )
        . = ALIGN(4);
        __text_end__ = .;
    } >flash
    .mycode1: {
        KEEP (*(.constBufferA*))
    } >flash1
    .mycode2: {
        KEEP (*(.constBufferB*))
    } >flash2
```

段内偏移拆分与差异填充值

当在同一个段落中，通过 . = Value 的实现偏移布局是，此时将产生段落对齐的间隙。当段落修饰填充时，以配置值作为该段落的填充，默认填充值为 0x00。如

```
        __text_end__ = .;
    } >flash =0xFF
```

可以在该段落内部采用高优先级的填充命令 FILL。如

```
.text : {
    . = 0x0000;
    KEEP (*vector.o(.text*))

    FILL(0x00)
    . = 0x400 ;
    *(.textA* )

    FILL(0xFF)
    . = 0x800 ;
    *(.textB* )
```



```
*(.text* )  
. = ALIGN(4);  
    text_end = .;  
} >flash =0x99
```

注：FILL 优先级高，仅针对出现后的区域生效填充值。

7. 基于文件与段名的匹配规则实施布局输出

每一行段落的典型规则为 **A(B(C))**，其中 **A** 可以输入关键字或标准方法。**B** 对应文件匹配规则，**C** 对应段落匹配规则。如 `KEEP (*vector.o(.text*))`

当 **A** 不存在是的规则格式为 **B(C)**，即文件与段落规则，如 `*(.text*)`。需要注意的是规则从上往下的应用，因此 `*(.Const*)` `*(.Const1*)` 的第二条规则为无效规则。此时可以通过文件规则进一步约束。

文件规则应用

要排除与文件名通配符匹配的文件列表，可以使用 **EXCLUDE_FILE** 来匹配除其列表中指定的文件以外的所有文件。例如：`EXCLUDE_FILE (*crtend.o *otherfile.o) *(.ctors)` 将导致包括除 `crtend.o` 和 `otherfile.o` 名后缀以外的所有文件的所有 `.ctors` 段。**EXCLUDE_FILE** 也可以放在段的列表中，例如：`*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)`。其结果与前面的示例相同。

将 **EXCLUDE_FILE** 与多个段一起使用时，这个排除命令仅仅对紧随其后的段有效，例如：

```
*(EXCLUDE_FILE (*somefile.o) .text .rdata)
```

将导致包含除 `somefile.o` 以外的所有文件的所有 `.text` 段，而包括 `somefile.o` 在内的所有文件的所有 `.rdata` 段都将被包含。要从 `somefile.o` 中排除 `.rdata` 段部分，可以将示例修改为：

```
*(EXCLUDE_FILE (*somefile.o) .text EXCLUDE_FILE  
(*somefile.o) .rdata)
```

或者，将 **EXCLUDE_FILE** 放在段列表之外(在选择输入文件之前)，将导致排除操作对所有段有效。因此，前一示例可以重写为：

```
EXCLUDE_FILE (*somefile.o) *(.text .rdata)
```

库文件规则应用

可以指出特别的关联库名称的文件，命令是[库匹配模板:与文件匹配的模式]，冒号两边不能有空格。

'archive:file' 在库中寻找能够匹配的文件

'archive:' 匹配整个库

':file' 匹配文件但不匹配库

'archive' 和 'file' 中的一个或两个都可以包含 **shell** 通配符。在基于 **DOS** 的文件系统上，链接器会假定一个单字跟着一个冒号是一个特殊的驱动符，因此 `'c:myfile.o'` 是一个文件的特殊使用，而不是关联库 `'c'` 的 `'myfile.o'` 文件。

'archive:file': 可以使用在 **EXCLUDE_FILE** 列表中, 但不能出现在其他链接脚本内部。例如, 你不能使用'archive:file'从 **INPUT** 命令中取出一个库相关的文件。

全文件捕获应用

如果你使用一个文件名而不指出段列表, 则所有的输入文件的段将被放入输出段。通常不会这么做, 但有些场合比较有用, 例如:

data.o

当你使用一个文件名且不是'archive:file'特殊命令, 并且不含任何通配符, 链接器将先查看你是否在命令行上或者在 **INPUT** 命令里指定了该文件。如果没有这么做, 链接器尝试将文件当作输入文件打开, 就像文件出现在了命令行一样。注意与 **INPUT** 命令有区别, 因为链接器不会在库文件路径搜索文件。

8. 特殊应用：低功耗模式下数据保持应用

单片机的 SRAM 分为主要分为两个区域。一部分 LPRAM 在 STANDBY 及 STOP1 的低功耗模式仍然可以保持, 为 SRAM 区域的前 16K 【规格型号下的】, 另一部分为通用 RAM。休眠前, 将 PM CTL0 的 bit19 置 1, 即可保持数据。在 SRAM 的前 16K 空间中规划出 section 段, 将要保持的变量放置于该段, 该变量的数据可以在 STANDBY 及 STOP1 的低功耗模式下保持。

按照正常的启动逻辑, 单片机在复位后会先执行"startup"函数, 执行此函数会将变量进行初始化。在 STANDBY 模式及 STOP1 模式下, 唤醒后代码从头运行。

若需要低功耗下保持数据, 需要避免每次复位处调用默认"startup"函数, 需要更改启动逻辑, 启动后从"main"开始运行【即修改向量表文件替换 startup 为 main】。main 函数判断是否需要初始调用默认 startup 或调用变型的 startup 函数, 如 startup1, 基于 startup 修改内容:

```
while(begin < end)
{
```

```
    *begin++ = *s++;
```

```
}
```

为

```
extern unsigned int __lpdata_start__; // 脚本变量
extern unsigned int __lpdata_end__; // 脚本变量
```

```
while(begin < end)
```

```
{
```

```
    if(begin == ( unsigned int * )(&__lpdata_start__))
```

```
        begin = ( unsigned int * )(&__lpdata_end__);
```

```
    *begin++ = *s++;
```

```
}
```

变量使用 `__attribute__((section(".lpdata")))` 修饰, 默认脚本为

```
KEEP (*(ramvector*)) /* default server for vector writed in ram */
```

```
__lpdata_start__ = (. + 3) & (-4);
```

```
*(.lpdata*) /* Reset but Keep Space */
```

```
/* . = 0x4000 ; */ /* Max length limit is 16K byte */
```

```
__lpdata_end__ = (. + 3) & (-4);
```

```
. = (. + 3) & (-4);
```



```
*(.indata*)          /* server space for ram function */
*(.inrdata*)
*(.inrodata*)

. = (. + 3) & (-4);
*(.data*)            /* global variable with initialization */
```

若低功耗保持数据空间需求明显小于空间，即根据 ram 布局顺序 **lpdata** 段实现在前 16K 空间内容，若数据空间需求量接近或可能接近下，可调整脚本为：

```
*(.lpdata*)          /* Reset but Keep Space */
. = 0x4000 ;          /* Max length limit is 16K byte */
```

即使能__lpdata_end 为 0x4000 的偏移空间占用，**lpdata** 未使用空间将默认填充为 0 的假定强制使用，也可以根据使用量设计如前 1K 空间作为 lpdata 区域的配置值 0x400。

9. 典型应用：bootloader 双程序开发模式应用

复制型号模板 ld 文件到项目中，修改链接脚本为项目下的对应 ld 文件。示例 boot 空间为 0x4000 并 app 起始地址空间为 0x4000[向量表保持在起始地址位置]下的修改脚本内容：

```
MEMORY
{
    flash      : ORIGIN = 0x00004000, LENGTH = 0x00010000-0x00004000
    ram        : ORIGIN = 0x10000000, LENGTH = 0x00004000
```

修改起始地址和长度实现 boot 与 app 的程序空间划分，其他内容无特殊需求下保持不变。

其中boot程序引导进入app时，根据app脚本所在向量表位置实现向量表重映射与代码跳转

```
#define ApplicationAddress 0x00004000
typedef void (*pFunction) (void);
pFunction Jump_To_Application;
volatile uint32_t JumpAddress;
void __set_MSP(uint32_t value){
    asm(" MOV SP,%0 \n": : "r"(value): "sp");
}
void SYSCTL_Vector_Offset_Config (uint32_t VectorOffset){
    SYS_VECTOFF = VectorOffset;
}
{
    asm ("DSI"); // 根据需要自行关闭 boot 中断源外设
    SYSCTL_Vector_Offset_Config(0x4000);
    JumpAddress = *(__IO uint32_t*) (ApplicationAddress + 4);
    Jump_To_Application = (pFunction) JumpAddress;
    __set_MSP(*(__IO uint32_t*) ApplicationAddress);
    Jump_To_Application();
}
```

10. 宏脚本分支(语法不支持，可应用控制)

一般的宏主要作用在编译器阶段，即先启动预处理输出中间文件用于最后的编译。作为链接器脚本的文本语法，其不做该过程，即不支持**#if #else #endif**的语法用于脚本内容的提供。

同上描述，可以建立脚本模板，其可以使用**#include**下的**#define**。但其仅为模板文件，因此可以调用编译器动作，选择仅启动预处理，即实时输出目标脚本文件。

作为基于 **makefile** 的构建系统，其内容具有**-include ../makefile.defs**的语句表达，因此可在构建目录的上一级建立 **makefile.defs** 文件，内容输入示例：

```
../KF32A150MQVuse.ld ../KF32A150MQV.ld _LD_Head_Define.h
```

```
kf32-gcc -E -x c -P ../KF32A150MQV.ld -o ../KF32A150MQVuse.ld
```

解释：将模板文件**../KF32A150MQV.ld**使用**-x**选项声明为**C**文件，使用**-E -P**的仅选择预处理并不输出多余信息(如**#**开头的文件与路径信息)。 **../KF32A150MQVuse.ld**为链接使用的脚本文件，**_LD_Head_Define.h**提供宏，**KF32A150MQV.ld**使用宏分支的脚本模板。

局限性：作为集成的开发环境，工具链的配置往往针对其待编译的目标文件。因此命令行配置的**-D**属性不能有效的转移到 **makefile.defs** 下的使用。直接在 **kf32-gcc -E -x c -P ../KF32A150MQV.ld -o ../KF32A150MQVuse.ld**上扩展选项具有操作隐蔽性，实际可应用性差。推荐使用头文件定义，并头文件作为**c**源码系统服务，从而具有统一的属性，即**C**代码或**c**文件嵌汇编代码使用该宏获取布局的信息，而不建议独立维护脚本并**c**代码引用脚本变量的使用[自行建立模板脚本也需要本文档中脚本语法的相关知识运用]。

11. 版本历史

序号	版本	变更描述	影响范围	日期
1	V1.0	初稿	无	2021-2-18
2	V1.1	修正低功耗保持数据应用介绍的初始化表达错误 修正低功耗保持 ram 变量属性声明段名".lpdata*"为不带*的".lpdata" 添加内存地址说明部分的脚本中名需要以*为原因的介绍	原表达初始化功能异常	2021-6-1
3	V1.2	修正 RAM 定义的描述的自定义段()错误,仅 KEEP 修饰下需要; 增加脚本常量值支持灵活的书写。	原格式书写链接阶段脚本语法错误	2022-3-22
4	V1.3	添加附录的信息, 添加详细完整语法的链接 添加填充与文件名布局的说明 添加宏脚本应用实现 其他文档内容更新描述		2023-12-2

附录 A: MAP 文件内容的组织

#MAP 文件内容的组织逻辑描述与注解(#为注解的起始约定)

Archive member included because of file (symbol) #内容: 哪个库下的那个对象文件被识别, 界定其开始

示 例 内 容 : **H:/Program Files (x86)/ChipON IDE/KungFu32/ChipONCC32/ccr1_issue/lib\libmath.a(s_atan.o)**
示 例 内 容 : **./user/general_dev/general_dev_state.o (atan)**

@@@@@Discarded input sections #内容:内容那些未参数程序执行的丢弃, 所以地址为无值的 0.不会具有芯片地址关联, 界定其开始

#示例内容: **.text\$vehicle_1s_average_speed_init**
示 例 内 容 : **0x00000000**
0x4c ./user/vehicle_base/average_speed_every_1_second.o
解 释 : 相 对 路 径 **user/vehicle_base** 下 文 件 下 **average_speed_every_1_second** 函数编译代码长度 **0x4C** 字节, 未分配无绑定起始地址到芯片独占资源

@@@@@Memory Configuration #内容:
脚本 **ld** 文件下的可使用地址和长度信息, 界定其开始

# 示 例 内 容 : Name	Origin	Length
Attributes		
#示例内容: flash	0x00000000	0x00080000

@@@@@Linker script and memory map #内容:那些参与的文件编译的结果对象文件并随后跟随芯片空间与资源空间的名信息 起始地址信息 长度信息 文件关联信息, 界定其开始

#示例内容: **LOAD ./user/vehicle_base/accident.o**
#示例内容: **.text\$SaveUpdInfo**
示 例 内 容 : **0x00002210**
0x14 ./user/system_manage/sys_fw_update.o
#示例内容: **0x00002210** **SaveUpdInfo**
#示例内容: **.text\$ReadUpdInfo**
示 例 内 容 : **0x00002224**
0x44 ./user/system_manage/sys_fw_update.o
#示例内容: **0x00002224** **ReadUpdInfo**
#解释: 相对路径 **user/system_manage** 下的 **sys_fw_update** 下的函数 **SaveUpdInfo**, 其函数长度 **0x14**,其起始地址 **0x2210**。跟随的 **ReadUpdInfo** 顺序地址为 **0x2224**

#如为何输出文件 **elf** 的选择约定惯例。函数到工具内部归属组织为段名, 一般前缀起始为 **.text**。 变量根据其定义情况下有前缀 **.data .bss .comm** 等。格式后面是原始函数或变量的名引用

#**.stab** 等的起始为调试信息, 是工具调试需要的信息, 可执行输出如 **hex** 仅

是 **flash** 和 **ram** 等的内容的组织

#.fill 为函数或变量具有对齐要求下的插入的空隙。即使能下面的函数或变量的起始地址安装 **n bits** 对齐

#.data\$init\$a 可认知为对象 **a**。源码是具有赋初值的如 **int a=3** 的举例。同理 **.data\$static\$a** 对应 **static int a=3** 的举例。同理 **.bss\$inti\$a** 对应具有初值，但值为 **0** 转移为无初值的识别。

#链接基于脚本按照代码 全局初始化变量 全局未初始化变量的组织函数或变量的芯片资源地址分配。**map** 是分配结果下的一种展示方式。

附录 B：基本脚本变量与描述

序号	变量名	功能说明	已测试 是否有 影响的 变量	备注	工具应用说明
1	__vec_end__	指定向量表结束地址			未涉及
2	__init_class_start	指定类初始化函数空间起始地址			C++必须
3	__init_class_end	指定类初始化函数空间结束地址			C++必须
4	__text_end__	指定 flash 程序段的结尾地址	否		必须的尾地址起始数据表
5	__data_start__	指定 ram 的起始地址	否		必须的数据初值始地址
6	__lpdata_start__	指定低功耗 ram 的起始地址			低功耗 16K 用户级
7	__lpdata_end__	指定低功耗 ram 的结束地址			低功耗 16K 用户级
8	__data_end__	指定 ram 大小偏移的结束地址			必须的数据尾地址知道长度
9	__bss_start__	指定 bss 的起始地址	是		必须的零数据始地址
10	__bss_end__	指定 bss 的结束地址			必须的零地址尾地址知道长度
11	__Heap_Start__	指定堆的起始地址	是		必要的 malloc 库实现资源
12	__Logic_Heap_End__				布局 ram 空间的控制变量
13	__Heap_End__	指定堆的结束地址	是		必要的 malloc 库实现资源
14	__MAX_Stack_LIMITS__	指定使用栈的限制最大空间	是		布局 ram 空间的控制变量
15	__LMA_STACK_LIMITS__		是		布局 ram 空间的控制变量
16	__Logic_Stack_End__	指定使用栈的结束段	是		布局 ram 空间的控制变量
17	__Logic_Stack_Start__	指定使用栈的起始段	是		布局 ram 空间的控制变量
18	__initial_sp	指定栈顶地址	是		联东的脚步 sp 初始值
19	__Heap_length__	使用函数 malloc/free/calloc/realloc	是		布局 ram 空间的控制变量
20					

附录 C: 拆分 data 段表述示例脚本

注: 该脚本示例将 **RAM** 空间拆分为多个段落, 但应该结合 **data** 段与脚本变量名的配合工具保存初始化数据到程序空间。

注: 程序空间也支持按该风格进行拆分, 一个简单的差分可以拆分不同的 **memory**。

注: 拆分段落可以指定关联对应的内存区域, 若同一个内存区域, 必须使用 **AT** 属性控制其起始地址的偏移。

```
/*#####*/
/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-kungfu32-little", "elf32-kungfu32-big", "elf32-kungfu32-little")
OUTPUT_ARCH(kungfu32)
ENTRY(_start)
SEARCH_DIR(".");
/*#####*/
MEMORY{
    flash      :   ORIGIN = 0x00000000,       LENGTH = 512K
    ram        :   ORIGIN = 0x10000000,       LENGTH = 64K

    da_mem     : ORIGIN = 0x0C001C00, LENGTH = 0x00000400
    mode_mem   : ORIGIN = 0x0C001800, LENGTH = 0x00000004
    pro_mem    : ORIGIN = 0x0C001000, LENGTH = 0x00000004
    ee_mem     : ORIGIN = 0x7F000000, LENGTH = 0x00001000
    config_mem1 : ORIGIN = 0x31000000, LENGTH = 0x00000010
    config_mem2 : ORIGIN = 0x31000010, LENGTH = 0x00000010
}
    Heap_length = 0x100;
DEFAULT_STACK_SIZE = 0x100;
/*#####*/
PROVIDE(__initial_sp = ORIGIN(ram) + LENGTH(ram));
/*#####*/
/*
The wildcard patterns are like those used by the Unix shell.
'*' matches any number of characters
'?' matches any single character
'[chars]' matches a single instance of any of the chars; the '-' character may be used to
specify a range of characters, as in '[a-z]' to match any lower case letter
\" quotes the following character

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 not be moved backwards
inside an output section, and may not be moved backwards outside of an output section if
so doing creates areas with overlapping LMAs.

The full description of an output section looks like this:
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] [,]
```

When link-time garbage collection is in use ('--gc-sections'), it is often useful to mark sections

that should not be eliminated. This is accomplished by surrounding an input section's wildcard entry with KEEP(), as in KEEP(*(.init)) or KEEP(SORT_BY_NAME(*)(.ctors)).

*/

```
/*#####*/
SECTIONS {
```

```
    .text : {
/*#####*/
        . = 0x0000;
        KEEP (*vector.o(.text*))
        . = ALIGN(4);
        __vec_end__ = .;
/*#####*/
        *(.text)

        . = ALIGN(4);
        *(.ctors)
        *(.dtors)
        __init_class_start = (. + 3) & (-4);
        KEEP (*(.init_array)) /* class init function list space */
        __init_class_end = .;
        *(.init)
        *(.fini)
/*#####*/
        *(.rdata) /* global const variable space */
        *(.rodata) /* auto const variable space */

        . = ALIGN(4);
        __text_end__ = .;
        /*record: global variable value of initialization */

    } >flash
/*#####*/
    PROVIDE (__etext = __text_end__);
    PROVIDE (_etext = __text_end__);
    PROVIDE (etext = __text_end__);
/*#####*/
}
```

```
/*#####*/
SECTIONS {
```

```
/*#####*/
    .data : AT ( LOADADDR(.text) - ADDR(.text) + __text_end__ ) {
        __data_start__ = .;
        FILL(0x00000000) /* more enable */
/*#####*/
        KEEP (*(.ramvector)) /* default server for vector written in ram */
        . = ALIGN(4);
/*#####*/
        HIDDEN ( __lpdata_start__ = . );
        *(.lpdata) /* Reset but Keep Space */
        /* . = 0x4000 ; */ /* here length limit is 16K byte ,if all config ,delete comment
                                symbol*/

        . = ALIGN(4);
        HIDDEN ( __lpdata_end__ = . );
/*#####*/
        . = ALIGN(4);
        *(.indata) /* server space for ram function */
        *(.inrdata)
        *(.inrodata)
/*#####*/
        . = ALIGN(4);
        *(.data) /* global variable with initialization */
        . = ALIGN(4);
        __data_end__ = .;
/*#####*/
    } > ram
```

```
/*+++++*/
PROVIDE( __data_start2__ = ADDR(.data));
PROVIDE( __data_source__ = LOADADDR(.data));
PROVIDE( __data_end2__ = ADDR(.data) + SIZEOF(.data) );
PROVIDE( __data_source_end__ = LOADADDR(.data) + SIZEOF(.data) );
/*+++++*/
.bss (ADDR(.data) + SIZEOF(.data)) (NOLOAD) : {
    __bss_start__ = .; /* global variable with no initialization */
    *(.bss*)
    *(comm*)
    . = ALIGN(4);
    __bss_end__ = .;
} >ram
/*+++++*/
PROVIDE( __bss_start2__ = ADDR(.bss) );
PROVIDE( __bss_end2__ = ADDR(.bss) + SIZEOF(.bss));
/*+++++*/
PROVIDE( __allot_end2__ = ADDR(.bss) + SIZEOF(.bss) );
PROVIDE( __Heap_Start2__ = ADDR(.bss) + SIZEOF(.bss) );
PROVIDE( __heap_start__ = ADDR(.bss) + SIZEOF(.bss) );

.bss.heap (ADDR(.bss) + SIZEOF(.bss)) (NOLOAD) : {
    __allot_end__ = . ;
    __Heap_Start__ = . ;
    KEEP( *(.heap_mem*) )
    . = ALIGN(4);
    __Logic_Heap_End__ = .;
    . = DEFINED(Heap_Mem) ? ALIGN(4) : ( __Logic_Heap_End__ + __Heap_length__ );
    . = ALIGN(4);
    __Heap_End__ = . ;
    __MAX_Stack_LIMITS__ = .;
} >ram
PROVIDE( __heap_end__ = ADDR(.bss.heap) + SIZEOF(.bss.heap) );
/*+++++*/
.bss.stack (ADDR(.bss.heap) + SIZEOF(.bss.heap)) (NOLOAD) : {
    __Logic_Stack_End__ = . ;
    KEEP( *(.stack_mem*) )
    HIDDEN( __Logic_Stack_Start__ = . );
    . += ( __Logic_Stack_Start__ - __Logic_Stack_End__ ) ? (0) : ( __Logic_Stack_End__ +
DEFAULT_STACK_SIZE );
    . = ALIGN(4);
} >ram
/*+++++*/
}

/*+++++*/

SECTIONS {
/*+++++*/
.flashdata :
{
    KEEP( *(.flashdata*) ) /* Flash Data */
    . = 0x0400; /* length limit */
} > da_mem
/*+++++*/
.debugormode :
{
    KEEP( *(.modeconfig*) ) /* Mode */
} > mode_mem

.protectmode :
{
    KEEP( *(.protectconfig*) ) /* Protect */
} > pro_mem
/*+++++*/
.eeprom :
{

```

```
KEEP (*.eeprom*)) /* EEPROM */
. = 0x1000; /* length limit */
} > ee_mem
/*#####*/
.config1 :
{
    KEEP (*.config1*)) /* reserved config 1 */
} > config_mem1

.config2 :
{
    KEEP (*.config2*)) /* reserved config 2 */
} > config_mem2
/*#####*/
.bss._statch_sizes 0 : { *(.stack_sizes) }
.bss._version_info 0 : { *(.version_info) }
/*#####*/
.gnu.attributes 0 : { KEEP (*.gnu.attributes) }
/DISCARD/ : { *(.note.GNU-stack) *(.gnu_debuglink) *(.gnu.lto_) }
/DISCARD/ : { *(.note.note.*) }
/*#####*/
/* 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_line.*.debug_line_end) }
.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) }
/*-----*/
/* DWARF 3 */
.debug_pubtypes 0 : { *(.debug_pubtypes) }
.debug_ranges 0 : { *(.debug_ranges) }
/*-----*/
/* DWARF 4 */
.debug_types 0 : { *(.debug_types) }
/*-----*/
```

```
/* DWARF 5. */
.debug_addr 0 : { *(.debug_addr) }
.debug_line_str 0 : { *(.debug_line_str) }
.debug_loclists 0 : { *(.debug_loclists) }
.debug_macro 0 : { *(.debug_macro) }
.debug_names 0 : { *(.debug_names) }
.debug_rnglists 0 : { *(.debug_rnglists) }
.debug_str_offsets 0 : { *(.debug_str_offsets) }
.debug_sup 0 : { *(.debug_sup) }

/*#####*/
}

/*#####*/
HIDDEN (__Max_Address__ = LOADADDR(.data) + SIZEOF(.data) );
HIDDEN (__Limit_Address__ = /* ORIGIN(flash) */ LOADADDR(.text) + LENGTH(flash) );

ASSERT( __Max_Address__ <= __Limit_Address__ , "Error: text and data can not fill in
flash");
/*#####*/
```