应用笔记

如何修改链接脚本文件进行数据定位

前言

在编译一个程序时，最后是将每个输出文件链接在一起，最后一步就是运行“.ld”文件内容。每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，生成目标文件。

本应用笔记将以 KF32L530MNS 为例进行介绍如何修改链接脚本。

本应用笔记使用的 KF32 IDE 与 KF32Lxxx 外设固件库及代码例程可以从 ChipON 官方网站 www.chipon-ic.com 下载。

Github：https://github.com/ChipON-FAE-AE
Gitee：https://gitee.com/Cucoo/BSP

目录

1. 使用自定义的.ld 文件

在编译过程, 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 Linker Release->通用设定->芯片脚本文件。是如下图 1：

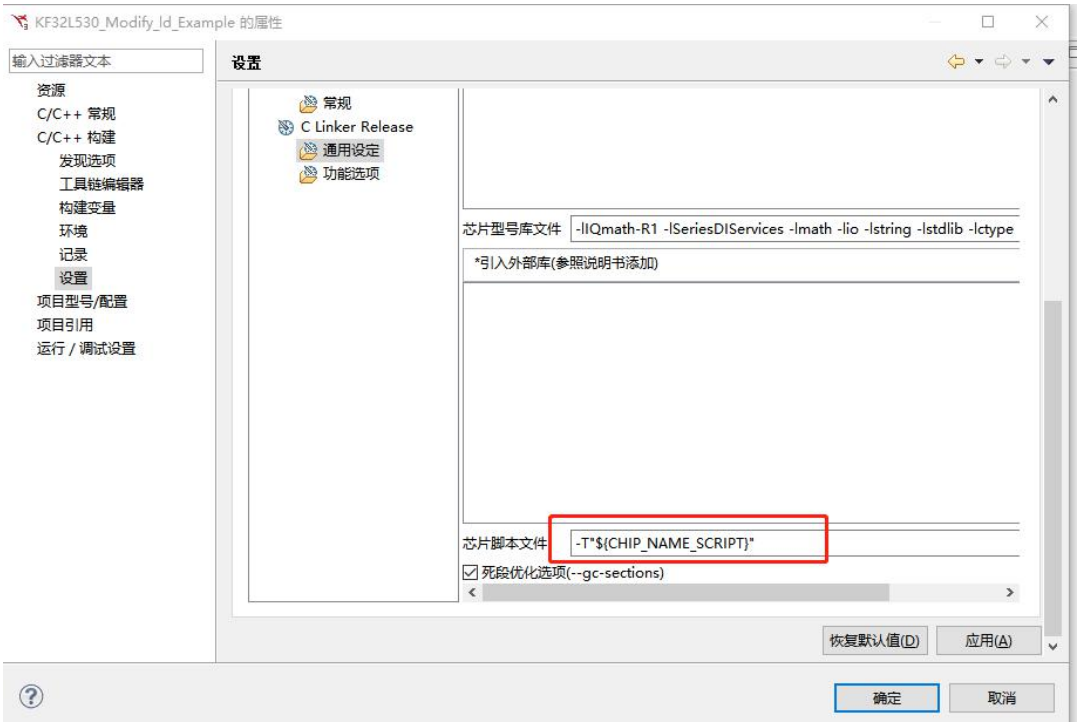

图 1：默认芯片脚本路径

默认的.ld 文件在安装目录下，地址路径为：**ChipONIDE/KungFu32/ChiponCC32/scripting/ccr1_issue_v0/xxx.ld**。xxx 为芯片型号，此处为 KF32L530MNS.ld。

将该路径的对应的 ld 文件复制到工程路径下，更改路径位置为：**-T"${ProjDirPath}/${CHIP_NAME}.ld"**。更改路径的意思为该工程目录下的芯片型号。

2. 内存地址说明

KF32L530MNS 的内存拥有 512K 的 Flash 和 128 的 RAM。

通过 KF32L 系列用户手册。可以看到 Flash 的起始地址为 0x0000 0000；RAM 的起始地址为 0x1000 0000。链接文件中可以对程序中 Flash 和 RAM 的起始地址及长度进行规定。

```
MEMORY
{
  flash       : ORIGIN =  0x00000000, LENGTH = 0x00080000
  ram         : ORIGIN =  0x10000000, LENGTH = 0x00020000
  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
}
```

图 2：默认起始地址与长度

3. 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。可以定义在代码的起始段，如定义在 0x0000 0200 处。

```
.text :
{
  . = 0x0000;
  KEEP (*vector.o(.text*))   /* chip interrupt vector ,writed in file named vector.s or vector
  . = (. + 3) & (-4);
  __vec_end__ = .;
  . = 0x200;            /* 向量表占用512字节 */
  KEEP (*(.ConstData*)) /* 在flash定义位置入口 */
  *(.text*)             /* 自动分配函数空间 */
  *(.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 */
  . = (. + 3) & (-4);
  *(.usertext*)          /* reserved for custom */
  *(.userrdata*)
  *(.userrodata*)
  . = (. + 3) & (-4);
  __text_end__ = .;
              /*record: global variable value of initialization */
} > flash
```

图 3：定义段名为”ConstData“在 0x200 处

调用方法为：使用 section 关键字，修改编译后，可以通过.map 文件或者 HEX 文件查看定义的位置。如下图 4：

```
__attribute__((section(".ConstData")))
uint8_t Test[] = "12345678";
```

```
0000020x:   3231   3433   3635   3837   0000   0000   4503   3800   12345678.....E.8
```

图 4：定义变量在”ConstData“段，HEX 中读取变量位置

如何修改链接脚本文件进行数据定位

若定义在代码的结束段定义，需要保证不覆盖代码，定义方式如下图 5：

```
.text :
{
  . = 0x0000;
  KEEP (*vector.o(.text*))   /* chip interrupt vector ,writed in file named vector.s or vecto
  . = (. + 3) & (-4);
  __vec_end__ = .;
  . = 0x200;            /* 向量表占用512字节 */
  *(.text*)              /* 自动分配函数空间 */
  *(.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 */
  . = (. + 3) & (-4);
  *(.usertext*)          /* reserved for custom */
  *(.userrdata*)
  *(.userrodata*)
  . = 0x4000;
  KEEP (*(.ConstData*)) /* 在flash定义位置入口 */
  . = (. + 3) & (-4);
  __text_end__ = .;
             /*record: global variable value of initialization */
} > flash
```

```
0000400x:   3231   3433   3635   3837   0000   0000   FFFF   FFFF   12345678.....
```

图 5：定义”ConstData“段，放置代码结束段，并从 HEX 读出

4. 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。在**”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 段中定义需要用户段”UserRAMData“，偏移地址为 0x1000。如下图 6：

```
.data :
{
  . = (. + 3) & (-4);
__data_start__ = .;
  KEEP (*(.ramvector*))       /* default server for vector writed in ram */
  . = 0x1000 ;
  . = (. + 3) & (-4);
  *(.UserRAMData*)
  . = (. + 3) & (-4);
  *(.lpdata*)                 /* Reset but Keep Space  */
/*  . = 0x4000 ; */                  /* Max length limit is 16K byte */
  . = (. + 3) & (-4);
  *(.indata*)                 /* server space for ram function */
  *(.inrdata*)
  *(.inrodata*)
  . = (. + 3) & (-4);
  *(.data*)           /* global variable with  initialization */
  . = (. + 3) & (-4);
__data_end__ = .;
  . = (. + 3) & (-4);
```

图 6：定义"UserRAMData"段在 0x1000 处

调用方法为：使用 section 关键字，修改编译后，可以通道.map 文件查看定义的位置。如下图 7：

```
__attribute__((section(".UserRAMData")))
volatile uint32_t Test2;
```

```
*(.UserRAMData*)
.UserRAMData$Test2
                0x10001000        0x4 ./main.o
                0x10001000                Test2
                0x10001004                . = ((. + 0x3) & 0xfffffffc)
```

图 7：读取.map 文件地址

5. 低功耗模式下保持数据

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，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**"startup"**函数，需要更改启动逻辑，启动后从**"main"**开始运行。更改启动逻辑方式请参考应用手册 ***AN32002 快速唤醒如何降低功耗章节。***

6. 版本历史

文档版本历史

日期	版本	变更
2021 年 2 月 8 日	V1.0	初始版本。