FAQ0050

常见问题解答

将函数加载到Flash或SRAM指定地址的方法

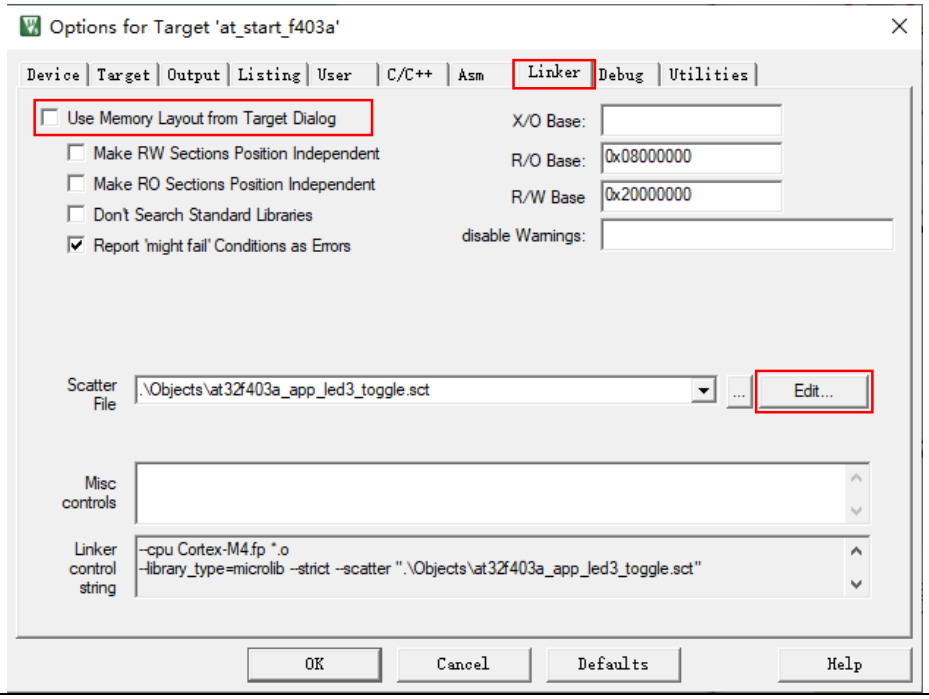
Questions：AT32 部分型号有零等待闪存和非零等待闪存，程序在零等待闪存执行速度比在非零等待闪存执行速度快，如果有函数对执行速度有要求，可以将该函数加载到零等待区执行。当零等待闪存使用完后，如果还有函数对执行速度有要求，可以将该函数加载到 SRAM 执行，前提是 SRAM 还有足够的空间存放该函数代码。

Answer：

有两种将某个函数加载到 Flash 或 SRAM 指定地址执行的方法，以 AT32F403A 为例：

1. 方法一：修改分散加载描述文件

在 Keil 中：修改分散加载描述文件*.sct(scatter file)，在工程选项的 linker 页面中，选择*.sct 文件，编辑*.sct。



如果是要放在Flash零等待区指定地址，添加*.o(ZWROMCODE)，自定义一个叫做ZWROMCODE的section。LR_IROM1 为零等待区，LR_IROM2 为非零等待区，设置如下：

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00020000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00020000  {  ; load address = execution address
```

```
    *.o (RESET, +First)
    *(InRoot$$Sections)
    .ANY (+RO)
    *.o(ZWROMCODE)
    }
    RW_IRAM1 0x20000000 0x00038000   {   ; RW data
       .ANY (+RW +ZI)
    }
}
LR_IROM2 0x08020000 0x000E0000   {      ; load region size_region
    ER_IROM2 0x08020000 0x000E0000   {   ; load address = execution address
       .ANY (+RO)
    }
}
```

也可以根据需要单独分配一块零等待区 LR_IROM2，添加*.o(ZWROMCODE)，自定义一个叫做 ZWROMCODE 的 section。LR_IROM1 和 LR_IROM2 为零等待区，LR_IROM3 为非零等待区，设置如下:

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00010000   {      ; load region size_region
    ER_IROM1 0x08000000 0x00010000   {   ; load address = execution address
       *.o (RESET, +First)
       *(InRoot$$Sections)
       .ANY (+RO)
    }
    RW_IRAM1 0x20000000 0x00038000   {   ; RW data
       .ANY (+RW +ZI)
    }
}

LR_IROM2 0x08010000 0x00010000   {      ; load region size_region
    ER_IROM2 0x08010000 0x00010000   {   ; load address = execution address ZW
       *.o(ZWROMCODE)
    }
}

LR_IROM3 0x08020000 0x000E0000   {      ; load region size_region
```

```
  ER_IROM3 0x08020000 0x000E0000  {  ; load address = execution address NZW
    .ANY (+RO)
  }
}
```

如果是要放在 SRAM 指定地址，根据需要分配 RW_IRAM2 大小，添加*.o(RAMCODE)，自定义一个叫做 RAMCODE 的 section。设置如下，以 SRAM 扩大到 224KB 为例，RW_IRAM1 大小 196KB，RW_IRAM2 大小 28KB。

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x000C0000  {    ; load region size_region
  ER_IROM1 0x08000000 0x000C0000  {  ; load address = execution address
    *.o (RESET, +First)
    *(InRoot$$Sections)
    .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x00031000  {  ; RW data
    .ANY (+RW +ZI)
  }

  RW_IRAM2 0x20031000 0x00007000  {  ; RW data
  *.o(RAMCODE)
  }
}
```

在需要加载到 Flash 或 SRAM 指定地址中的函数前，用__attribute__((section("SECTION_NAME")))声明该函数放在该 section 中，SECTION_NAME 可以自行命名，如下面是使用 RAMCODE。如果有多个函数，那么每个函数前面都需要加__attribute__((section("RAMCODE")))。设置如下：

```
__attribute__((section("RAMCODE")))
void delay(unsigned char num)
{
  while(num)
  {
    num--;
  }
}
__attribute__((section("RAMCODE")))
void test(void)
```

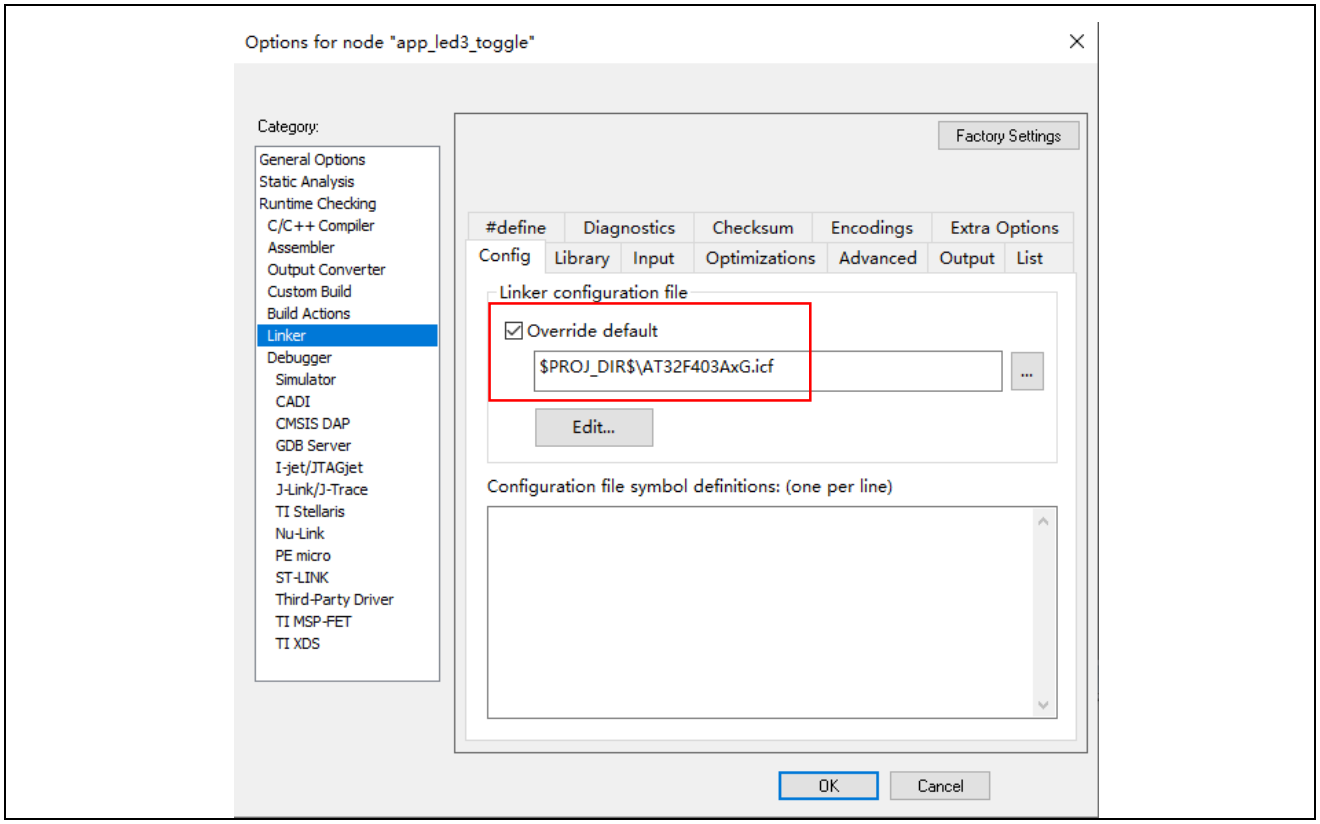
```
{
  for(;;)
  {
    led_gpio_port[led]->odt ^= led_gpio_pin[led];
    delay(10);
  }
}
```

也可以将所有需要加载的函数放在以#pragma arm section code = “RAMCODE” 开头，以#pragma arm section 结尾的中间，声明所有函数放在 RAMCODE section 中。设置如下：

```
#pragma arm section code = “RAMCODE"
void delay(unsigned char num)
{
  while(num)
  {
    num--;
  }
}

void test(void)
{
  for(;;)
  {
    led_gpio_port[led]->odt ^= led_gpio_pin[led];
    delay(10);
  }
}
#pragma arm section
```

在 IAR 中：在安装目录 IAR Systems\Embedded Workbench\arm\config\linker\ArteryTek 下找到相应 MCU 型号的分散加载描述文件*.ICF，复制到 project 路径下，按下图方式启用*.icf 文件。



如果是要放在 Flash 零等待区指定地址，根据需要在零等待区分配一段 region，增加下面红色字体的部分，即可将 my_code.c 内的代码都放入该零等待区内（ZWROM_CODE_region）：

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x080FFFFF;

/* ZMROM CODE area */
define symbol __ICFEDIT_region_ZWROM_CODE_start__ = 0x08010000;
define symbol __ICFEDIT_region_ZWROM_CODE_end__   = 0x0801FFFF;

define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20017FFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x1000;
define symbol __ICFEDIT_size_heap__   = 0x1000;
/**** End of ICF editor section. ###ICF###*/
```

```
define memory mem with size = 4G;

/* Reserved 0x08010000 ~ 0x0801FFFF as ZWROM CODE area */
define region ROM_region = mem:[from __ICFEDIT_region_ROM_start__ to __ICFEDIT_region_ROM_end__]
                         -mem:[from __ICFEDIT_region_ZWROM_CODE_start__ to __ICFEDIT_region_ZWROM_CODE_end__];
define region ZWROM_CODE_region = mem:[from __ICFEDIT_region_ZWROM_CODE_start__ to __ICFEDIT_region_ZWROM_CODE_end__];
define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__ to __ICFEDIT_region_RAM_end__];

define block CSTACK with alignment = 8, size = __ICFEDIT_size_cstack__ { };
define block HEAP with alignment = 8, size = __ICFEDIT_size_heap__ { };

initialize by copy { readwrite };
do not initialize { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region { readonly };

/* Place Code in ZWROM CODE */
place in ZWROM_CODE_region {ro object my_code.o};

place in RAM_region { readwrite,
                      block CSTACK, block HEAP };
```

如果有多个.c 文件需要放入零等待区，place in ZWROM_CODE_region 可以如下书写：

```
/* Place Code in ZWROM CODE */
place in ZWROM_CODE_region {ro object my_code1.o, ro object my_code2.o, ro object my_code2.o};
```

如果是要放在 SRAM 指定地址，以 SRAM 扩大到 224KB 为例，根据需要在零等待区分配一段 region，增加下面红色框线内的部方式分，即可将 my_code.c 内的代码都放入该零等待区内（RAM_CODE_region）

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
```

```
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x080FFFFF;

define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20037FFF;

define symbol __ICFEDIT_region_RAM_CODE_start__ = 0x20037000;
define symbol __ICFEDIT_region_RAM_CODE_end__   = 0x20037FFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x1000;
define symbol __ICFEDIT_size_heap__   = 0x1000;
/**** End of ICF editor section. ###ICF###*/

define memory mem with size = 4G;

define region ROM_region = mem:[from __ICFEDIT_region_ROM_start__ to __ICFEDIT_region_ROM_end__];

/* Reserved 0x20037000 ~ 0x20037FFF as ZWROM CODE area */
define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__ to __ICFEDIT_region_RAM_end__]
                                  -mem:[from __ICFEDIT_region_RAM_CODE_start__ to __ICFEDIT_region_RAM_CODE_end__];
define region RAM_CODE_region = mem:[from __ICFEDIT_region_RAM_start__ to __ICFEDIT_region_RAM_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   { readonly };

/* Place Code in RAM CODE */
place in RAM_CODE_region   {ro object my_code.o};

place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };
```

如果有多个.c 文件需要放入零等待区，place in ZWROM_CODE_region 可以如下书写：

```
/* Place Code in RAM CODE */
place in RAM_CODE_region     {ro object my_code1.o, ro object my_code2.o, ro object my_code2.o};
```

在 AT32 IDE 中：

在代码文件中，将需要放到SRAM的function添加__attribute__((section(".ramfunc")))宣告，将这些 function 全部放到 .ramfunc 这个 section，这个 section 的名称可自行定义，例程这里定义为 .ramfunc。

```
__attribute__((section(".ramfunc")))
void function_in_ram(void)
{
  at32_led_toggle(LED2);
  delay_ms(100);
  at32_led_toggle(LED3);
  delay_ms(100);
  at32_led_toggle(LED4);
  delay_ms(100);
}
```

在 *.ld 文件中，修改以下两个地方

1) 新增_ram_func_size 定义，用来设定需要存放 ram function 的 ram size 大小，例程中定义为 4K；修改 _estack 定义，_estack 是 STACK pointer 的初始值，修改后是要把 stack pointer 初始值设定到 ram function 开始的地址之前，避免 stack 空间跟 ram function 的空间重迭。

```
/* Highest address of the user mode stack */
//_estack = 0x20018000;    /* end of RAM */
_ram_function_size = 4K;
_estack = 0x20018000 - _ram_function_size;    /* end of RAM */
```

```
20
21 /* Entry Point */
22 ENTRY(Reset_Handler)
23
24 /* Highest address of the user mode stack */
25 //_estack = 0x20018000;    /* end of RAM */
26 _ram_function_size = 4K;
27 _estack = 0x20018000 - _ram_function_size;    /* end of RAM */
28
29 /* Generate a link error if heap and stack don't fit into RAM */
30 _Min_Heap_Size = 0x200;      /* required amount of heap  */
31 _Min_Stack_Size = 0x400; /* required amount of stack */
32
33 /* Specify the memory areas */
34 MEMORY
35 {
36 FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 1024K
37 RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 96K
38 }
39
```

2) 添加一段描述，将.ramfuc 这个 section 的起始地址指定到 SRAM 的最后 4K，*(.ramfunc)这个宣告代表所有用__attribute__((section(".ramfunc")))宣告的 function 都会被编排到这个区域.

```
. = ALIGN(4);
. = ORIGIN(RAM) + LENGTH(RAM) - _ram_function_size;
_ramfunc = .;
*(.ramfunc);
```

```
/* Initialized data sections goes into RAM, load LMA copy after code */
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */

  . = ALIGN(4);
  . = ORIGIN(RAM) + LENGTH(RAM) - _ram_function_size;
  _ramfunc = .;
  *(.ramfunc);

  . = ALIGN(4);
  _edata = .;        /* define a global symbol at data end */
} >RAM AT> FLASH
```

照以上的方式修改之后，compiler 编译过后就会把这些 function 的 code 编译备份在 flash 的区域，然后每次上电跑起来时，启动文件 startup_at32xxx.s 会把这些 function 的 code，从 flash 里 copy 到指定的 SRAM 地址。

2. 方法二:只对某些编译器有效比如 Keil，在需要加载到 Flash 或 SRAM 指定地址中的函数前，用__attribute__((section(".ARM.__at_address")))声明该函数放在该 section 中，address 根据需要设置。如果有多个函数，那么每个函数前面都需要加__attribute__((section(".ARM.__at_address")))。

将函数加载到 Flash 指定地址：

```
void Test(void) __attribute__((section(".ARM.__at_0x08001000")));
void Test(void)
{}
```

将函数加载到 SRAM 指定地址

```
void Test(void) __attribute__((section(".ARM.__at_0x20001000")));
void Test(void)
{}
```

类型： MCU 应用

适用型号： AT32 全系列

主功能： FLASH、SRAM

次功能： 无

文档版本历史

日期	版本	变更
2022.2.16	2.0.0	最初版本
2022.11.11	2.0.1	更新几处“非零等待区”的描述错误
2023.1.13	2.0.2	修正部分代码错误
2024.1.8	2.0.3	增加对应AT32 IDE支持的描述
2024.10.29	2.0.4	修正部分代码错误

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。

© 2024 雅特力科技 保留所有权利