2012

三天入门 Cortex-M4
—Kinetis 系列

野火 Kinetics 开发板教程

最适合初学者入门 Kinetis 的教程

freescale

作者：野火

野火嵌入式开发工作室

2012-3-14

野火 Kinetics 开发板教程

前言

随着技术的发展，单片机型号越来越繁多，入门的难度也逐渐加多，学习周期自然不断延长。为了让初学者快速入门，减少学习时间，尤其是为了那些参加智能车比赛而没时间学习深入研究 Kinetis 单片机朋友，我们特意写了 Kinetis 开发板的教程。力求大大减少初学者的学习时间。

野火 Kinetis 开发教程，主要有 **IAR 的使用教程**、**Kinetis 启动流程讲解**、**野火 K60 库的调用** 三个部分组成。我们不再详细讲解寄存器，而且推荐你们直接调用我们的函数库。野火 K60 函数库，函数内部会自动计算频率，设置分频，直接调用，减少你们的后顾之忧，可以加快你们的开发速度。

目前，单片机型号如此之多，而产品的开发所允许给我们的时间越来越少，我们完全没有必要深入研究寄存器设置，就算你能把寄存器背得滚瓜烂熟，过段时间不去接触，还是没法记住的。

现在的单片机开发，工程师往往都是利用官方的固件库来进行开发，而不再是靠自己重新建立自己的函数库进行开发。例如 ST 公司推出的 ST 库，让你可以完全不需要考虑底层开发而直接开发自己的产品。飞思卡尔公司，在这方便确实让人感到失望，这也是野火嵌入式工作室要建立自己的野火 Kinetis 库的原因。

野火 K60 库的函数接口，尽量追求简洁明了、通俗易懂，力求初学者见其名就会用。目前提供了 21 个入门实验：野火 Kinetis 核心板实验例程 ，包含了目前智能车比赛最常用的几个模块。

想快速上手 K60 单片机吗？野火 Kinetis K60 库，是你最好的选择！！！

我们的口号是：**三天入门 Kinetis！**

我们喊出这样的口号并不是吹的，已经有两位大三参加智能车的朋友（同一个队）在使用我们的教程与 k60 库三天时间内就能把 xs128 上的摄像头 0v7725 驱动程序和舵机控制程序移植到 Kinetis 上跑起来；一位大二的师弟，不到一天的时间就从入门到搞定摄像头 ov7725 和液晶 LCD 的移植……你会是下一位吗？

只要我们敢拼，一切皆有可能！！！*Nothing is impossible*！！！

由于个人能力及时间所限，出错之处，在所难免，欢迎各位指出错误及提出建议：minimcu@foxmail.com

——野火嵌入式开发工作室

联系我们

Technical support

目录

Kinetis 的启动分析（初学者大概浏览一下即可）

初步入门：初始化函数启动执行顺序

上电复位 → ① 中断向量表 → ② Reset_Handler → ③ start → ④ common_startup → ⑤ sysinit → ⑥ main

中断向量表在 vectors.c 里定义，在 vectors.h 文件里，前两个元素分别映射到 __BOOT_STACK_ADDRESS 和 Reset_Handler（官方自带例程为__startup），第一个为起始 SP 栈指针、第二个为复位中断服务函数。

上电复位，系统会查询中断向量表，把第一个元素的值赋给 SP 寄存器，把第二个值赋给 PC 寄存器，即上电后马上运行 Reset_Handler 函数。

上电复位后第一个运行的函数，在 crt0.s 里用汇编实现。

```
1.          SECTION .noinit : CODE
2.          EXPORT  Reset Handler
3.      Reset_Handler ;初始化寄存器
4.          MOV     r0,#0
5.          MOV     r1,#0
6.          MOV     r2,#0
7.          MOV     r3,#0
8.          MOV     r4,#0
9.          MOV     r5,#0
10.         MOV     r6,#0
11.         MOV     r7,#0
12.         MOV     r8,#0
13.         MOV     r9,#0
14.         MOV     r10,#0
15.         MOV     r11,#0
16.         MOV     r12,#0
17.         CPSIE   i ; 关中断
18.             import start
19.             BL      start
20.     ;调用 C 语言 编写的 start 函数
21.     __done
22.             B       __done
23.             END
```

在 start.c 文件里定义，精简后的代码：

```
void start(void)
{
    wdog_disable();//关闭看门狗
    common_startup();
      //复制数据段、中断向量表到 ram
    sysinit(); //时钟初始化
    main();     /** main 函数 **/
    while(1);   //避免停止运行
}
```

就是初始化系统：关闭看门狗、复制数据到内存、配置时钟，然后进入 main 函数

复制中断向量表、初始化数据段、以__ramfunc 声明的子函数到 RAM 区(对 flash 操作的函数必须这样)

初始化时钟和初始化串口，精简版代码：

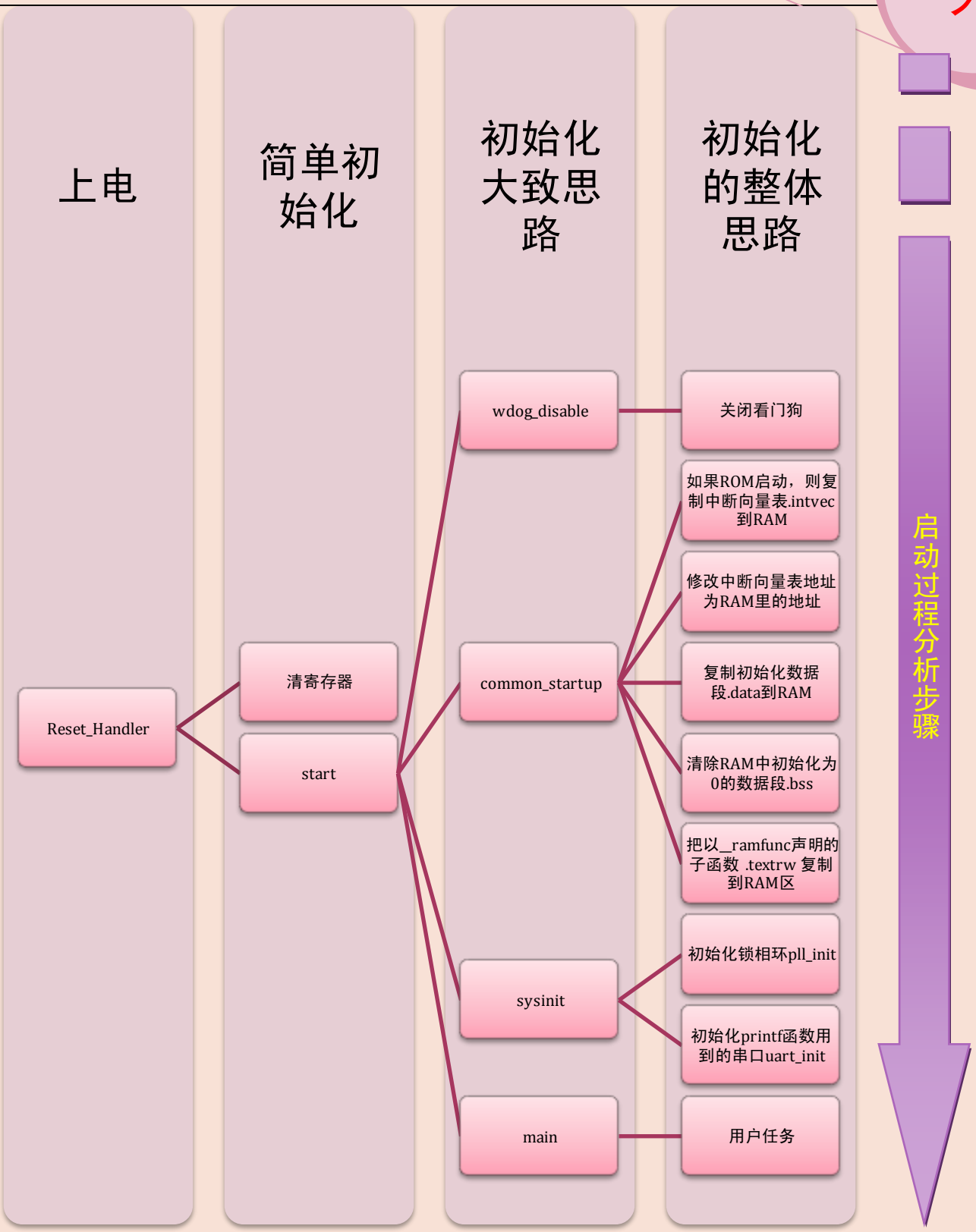
```
void sysinit (void)
{
    core_clk_mhz = pll_init(MCG_CLK_MHZ);
    trace_clk_init();  //使能跟踪时钟，用于调试
    fb_clk_init();     //FlexBus 时钟初始化
    uart_init(FIRE_PORT, FIRE_BAUD);
              //初始化 printf 函数所用到的串口
}
```

这里就是我们的执行函数，我们在这里编写自己的应用文件。

pll_init

pll_init 这个函数是用于锁相环超频的，在 mcg.c 里定义。MCG 模块一共有 9 中工作模式，按功耗高低顺序：PEE、FEE、FEI、PBE、FBE、FBI、BLPE、BLPI、stop。

高频率意味着高功率，为了获取更高的频率，我们这里 pll_init 就是把模式进入 PEE。

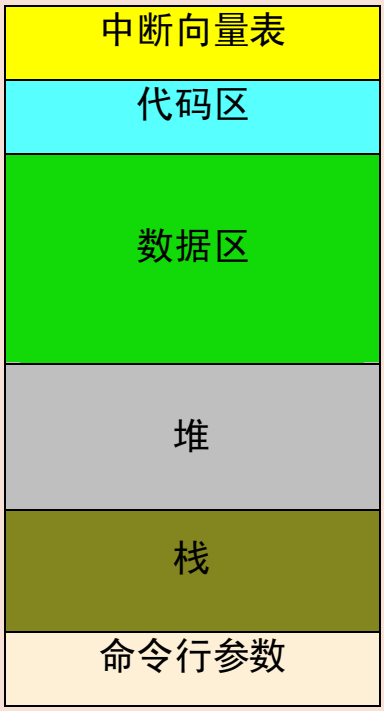


注:common_startup 函数并没有复制 常量数据 .rodata 、代码.text 。如果是 RAM 启动，代码会直接编译进去 RAM，掉电就会丢失数据。如果是 ROM 启动，就会复制中断向量表到 RAM，设置中断向量表地址为 RAM 的地址，以加快中断响应速度。

逐步提高：ROM、RAM 启动工作原理、ICF 文件讲解

对于初学者而言，对单片机的内存分配往往最让人头疼，很多人学了单片机几年都不知道单片机内部的内存使用情况是如何分配的。要了解 ROM、RAM 启动，首先需要对 链接器 Linker 如何分配内存有一定的了解。

通常，对于栈生长方向向下的单片机，其内存一般模型是：



中断向量表段 .intvec

函数代码段 .text

未初始化变量段 .bss

常量段 .rodata

已初始化全局变量和静态变量段 .data

动态分配数据

程序运行中由程序员调用 malloc 等函数来申请。

局部变量

在函数内部定义的非静态的变量。

```
1.  int a = 0;                      //全局初始化区，.data 段
2.  static int b=20;                //全局初始化区，.data 段
3.  char *p1;                       //全局未初始化区 .bss 段
4.  const int A = 10;               //.rodata 段   } 注意区别！！
5.  volatile const int B = 10;      //.data 段
6.  main()                          //代码区 .text
7.  {
8.      int b;                      //栈
9.      char *p2;                   //栈
10.     static int c = 0;           //全局（静态）初始化区 .data 段
11.     char s[] = "123456";        //栈                  } 注意区别！！
12.     char *p3 = "123456";        //123456\0 在常量区，p3 在栈上。
13.     p1 = (char*) malloc(10);    //分配得来的 10 和 20 个字节的区域就在堆区
14.     p2 = (char*) malloc(20);
15.     strcpy(p1, "123456");       //123456\0 在常量区
16.                                 //编译器可能会将它与 p3 所指向的"123456"优化成一个地方
17. }
```

IAR 里可以设置 Linker 合并相同的段，即优化成一个地方。看后面 IAR 设置工程选项中 Linker 的设置教程

注：此表格及代码内容参考网上资料

参考原文出处：http://blog.chinaunix.net/uid-15473693-id-388637.html

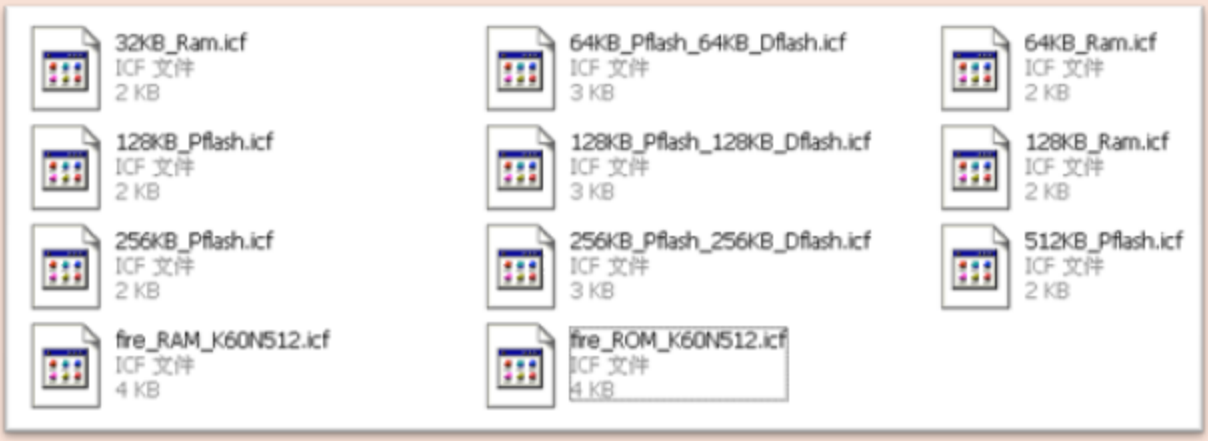
那编译到时候，编译器是如何为这些变量数据分配地址的呢？

其实，这就是链接器 Linker 在发挥它的作用，它会根据配置文件，来为这些变量数据分配合适的地址，这样我们就可以不需要考虑这些内存分布就能写出可运行的代码。

编译器对代码进行编译，一般分为四个步骤：

通常情况下，链接器 Linker 的配置文件都是由官方提供，一般情况下，我们不需要更改这些。但出于学习的目的，我们非常有必要去研究一下这些配置文件。

在 fire_Kinetis\build\config files 文件夹下，你可以看到有很多的 linker 配置文件：



这里的文件是用来分配数据在内存中的位置，配置 ROM、RAM 启动，linker 根据这些文件来为 Kinetics 分配 4G 的虚拟寻址空间地址。如果把代码部分编译进去 RAM，那就是 RAM 启动；如果把代码数据编译进去 ROM，那就是 ROM 启动（flash 启动）。

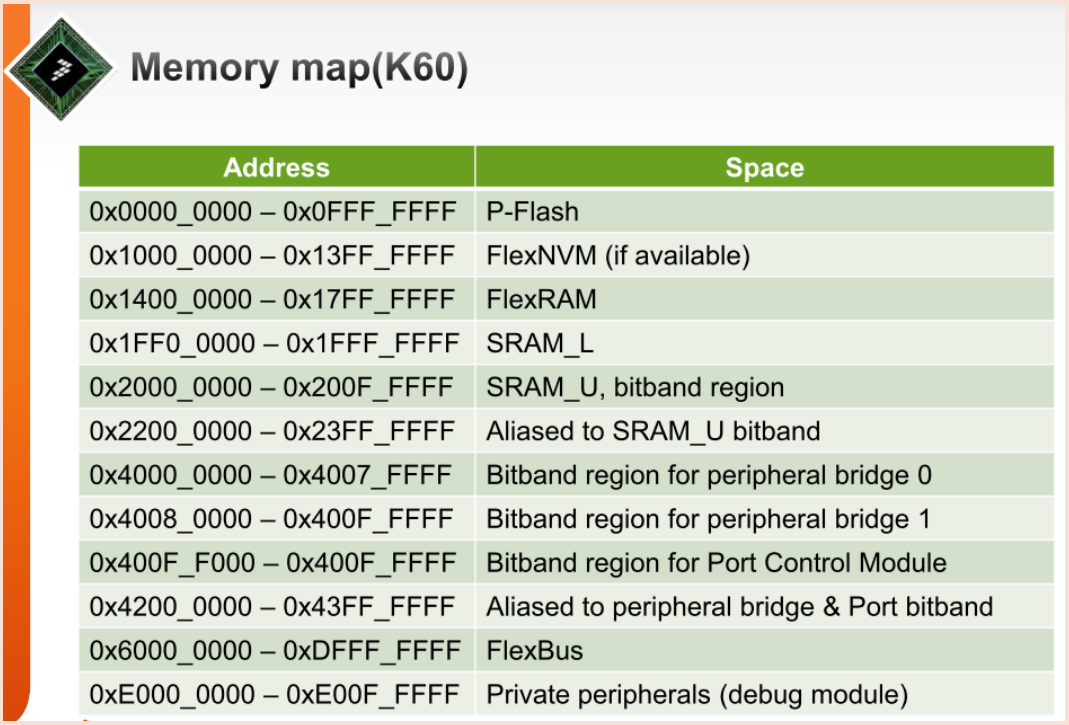
不同的型号，flash 内存大小不一样，所以配置 Linker 文件也会不一样，以 K60 为例：

Device	Program flash (KB)	Block 0 (P-Flash) address range[1]	FlexNVM (KB)	Block 1 (FlexNVM/ P-Flash) address range[1]	FlexRAM (KB)	FlexRAM address range
MK60DN256ZVLQ10	256	0x0000_0000 – 0x0001_FFFF	—	0x0002_0000 – 0x0003_FFFF	—	N/A
MK60DX256ZVLQ10	256	0x0000_0000 – 0x0003_FFFF	256	0x1000_0000 – 0x1003_FFFF	4	0x1400_0000 – 0x1400_0FFF
MK60DN512ZVLQ10	512	0x0000_0000 – 0x0003_FFFF	—	0x0004_0000 – 0x0007_FFFF	—	N/A
MK60DN256ZVMD10	256	0x0000_0000 – 0x0001_FFFF	—	0x0002_0000 – 0x0003_FFFF	—	N/A
MK60DX256ZVMD10	256	0x0000_0000 – 0x0003_FFFF	256	0x1000_0000 – 0x1003_FFFF	4	0x1400_0000 – 0x1400_0FFF
MK60DN512ZVMD10	512	0x0000_0000 – 0x0003_FFFF	—	0x0004_0000 – 0x0007_FFFF	—	N/A

Device	SRAM (KB)
MK60DN256ZVLQ10	64
MK60DX256ZVLQ10	64
MK60DN512ZVLQ10	128
MK60DN256ZVMD10	64
MK60DX256ZVMD10	64
MK60DN512ZVMD10	128

野火 Kinetis 核心板自带为 MK60DN512，512KB 的 Program flash 大小，128KB 的 SRAM 大小，没有 FlexNVM 和 FlexRAM。

K60 的 4G 虚拟寻址空间就是按照内存空间的映射图来进行配置：



Memory map(K60)

Address	Space
0x0000_0000 – 0x0FFF_FFFF	P-Flash
0x1000_0000 – 0x13FF_FFFF	FlexNVM (if available)
0x1400_0000 – 0x17FF_FFFF	FlexRAM
0x1FF0_0000 – 0x1FFF_FFFF	SRAM_L
0x2000_0000 – 0x200F_FFFF	SRAM_U, bitband region
0x2200_0000 – 0x23FF_FFFF	Aliased to SRAM_U bitband
0x4000_0000 – 0x4007_FFFF	Bitband region for peripheral bridge 0
0x4008_0000 – 0x400F_FFFF	Bitband region for peripheral bridge 1
0x400F_F000 – 0x400F_FFFF	Bitband region for Port Control Module
0x4200_0000 – 0x43FF_FFFF	Aliased to peripheral bridge & Port bitband
0x6000_0000 – 0xDFFF_FFFF	FlexBus
0xE000_0000 – 0xE00F_FFFF	Private peripherals (debug module)

如果要 RAM 启动，我们要把代码编译进 SRAM_L，定义中断向量表的位置。

可以打开 fire_RAM_K60N512.icf 这个文件（用记事本打开就行），这个配置文件是参考 128KB_Ram.icf 来重新修改和添加了注释。

官方的配置文件写得真的思路太乱了，如果你仅仅用 ICF 编辑软件来配置官方的配置文件，是非常容易出错的！你必须要用 ICF 配置好后，还要打开配置文件来修改。

野火已经对配置文件进行修改，在里面添加了很多注释：

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x1fff0000;
                                        //RAM 启动要设置为 RAM 起始地址，ROM 启动也要设为 ROM 的起始地址
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x0;              //ROM 地址
define symbol   ICFEDIT region ROM end     = 0x00040000;
define symbol __ICFEDIT_region_RAM_start__ = 0x1fff0000;       //RAM 地址
define symbol __ICFEDIT_region_RAM_end__   = 0x20000000;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x1000;                //堆大小
define symbol __ICFEDIT_size_heap__   = 0x800;                 //栈大小
/**** End of ICF editor section. ###ICF###*/

/**** 上边是由 ICF 编辑，下面是由我们手动配置 ****/


define symbol __region_RAM2_start__ = 0x20000000;//SRAM 是分成两块的，RAM2 即 SRAM_U ， RAM 为 SRAM_L
define symbol __region_RAM2_end__ = 0x20000000  + __ICFEDIT_region_RAM_end__
                                                - __ICFEDIT_region_RAM_start__;


define exported symbol __VECTOR_TABLE = __ICFEDIT_intvec_start__;
                          //代码编译进 ROM ,则 0x00000000 ; RAM，则 __ICFEDIT_region_RAM_start__

define exported symbol __VECTOR_RAM = __ICFEDIT_region_RAM_start__ ;
                //前面的 RAM 留给 RAM User Vector Table
                //如果是 ROM 启动，则 common_startup 函数会把 __VECTOR_TABLE 的数据复制到  __VECTOR_RAM


define exported symbol __BOOT_STACK_ADDRESS = __region_RAM2_end__ - 8;
                                        //0x2000FFF8;启动栈地址，中断向量表的第一个元素就是指向这里


/* 决定代码编译的地址 */
define exported symbol   code start    =   ICFEDIT intvec start   + 0x410;
              //代码编译进 ROM ,则  __ICFEDIT_region_ROM_start__ + 0x410 ,
              //RAM，则 __ICFEDIT_region_RAM_start__  + 0x410 , +0x410 ,是前面的留给 Vector Table

define memory mem with size = 4G;            //4G 的虚拟寻址空间

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

define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__
                                  to   __ICFEDIT_region_RAM_end__]
                           | mem:[from __region_RAM2_start__ to __region_RAM2_end__];
```

中断向量表地址。Flash 和 SRAM 启动的中断向量表地址是不一样的

Kinetis 的 SRAM 是分成两块的。这里是设置第二块的地址范围

中断向量表地址

可以看出栈的生长方向为向下

这里就是代码的编译地址。加上 0x410，是前面的留给中断向量表

设置 ROM 和 RAM 的可用内存地址范围

```
define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__    { };  //堆，8 字节对齐
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__      { };  //栈，8 字节对齐

//手动初始化，在 common_startup 函数 里完成
initialize manually { readwrite };          // 未初始化数据 .bss
initialize manually { section .data};       // 已初始化数据
initialize manually { section .textrw };    // __ramfunc 声明的子函数

do not initialize  { section .noinit };     // 复位中断向量服务函数

define block CodeRelocate { section .textrw_init };
define block CodeRelocateRam { section .textrw };
            //CodeRelocateRam 把代码复制到 RAM 中(对 flash 操作的函数必须这样)


place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };
     //vectors.c 中设置 #pragma location = ".intvec" ，告诉编译器，这个是中断向量表，编译进去 .intvec



place at address mem:__code_start__ { readonly section .noinit };
            //在 crt0.s 中设置了复位中断函数为 SECTION .noinit : CODE  ，即把代码编译进去 .noinit



place in RAM_region    { readonly, block CodeRelocate };
                                  //把代码编译进去 RAM (调试用，掉电丢失)，非调试，则设为 ROM_region

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

设置堆栈的大小

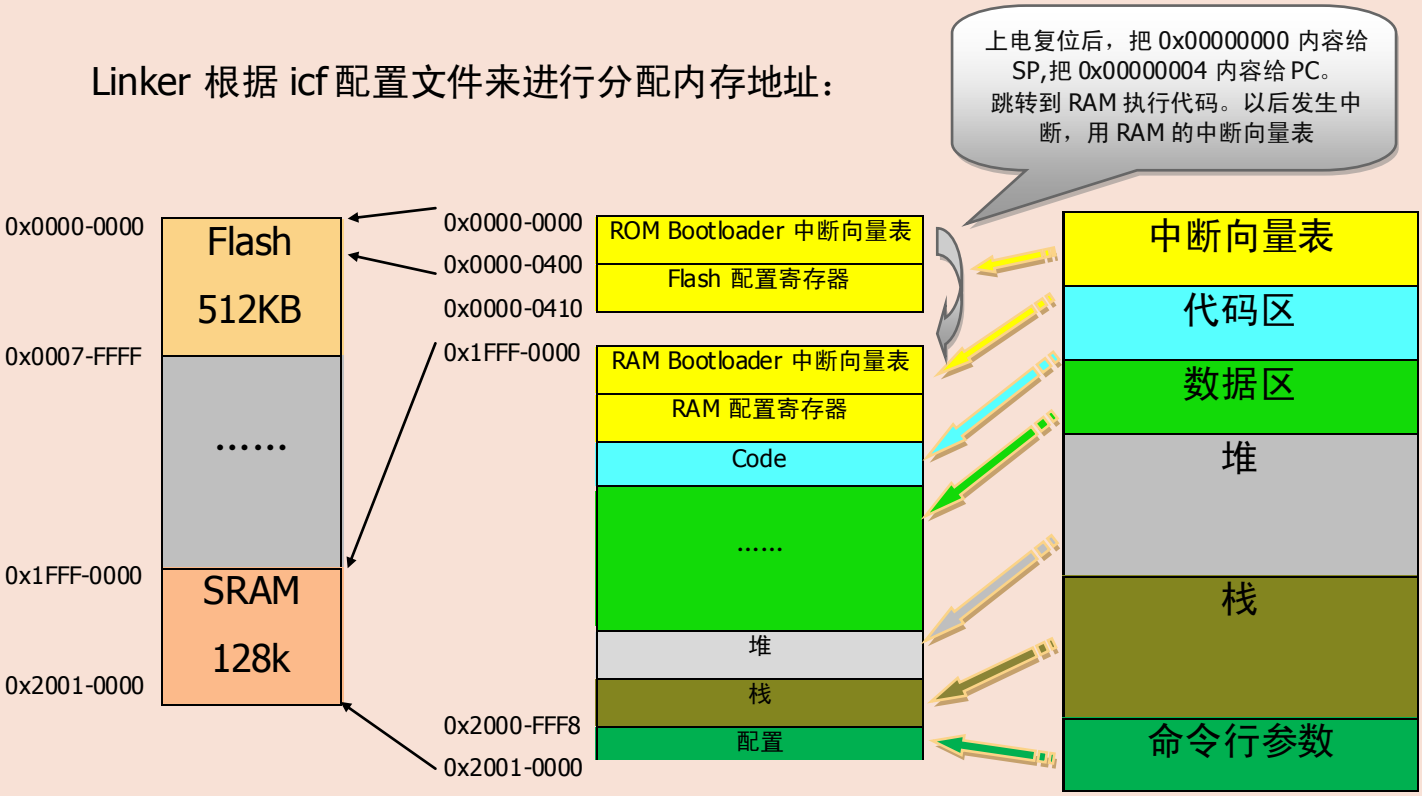
取消自动初始化，改为由 common_startup 函数来初始化

设置中断向量表位置放在 __ICFEDIT_intvec_start__ 地址上

设置 .noinit 段放在代码开头，即复位中断服务函数，把复位函数编译在代码开头

RAM 启动，会把只读数据、代码都编译进去 RAM

Linker 根据 icf 配置文件来进行分配内存地址：



如果是 ROM 启动（flash 启动）：

```
/*###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__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x0;
define symbol __ICFEDIT_region_ROM_end__   = 0x00040000;
                      //0x00040000:P-flashk 256k  D-flash 256k   0x00080000:P-flashk 512k
define symbol __ICFEDIT_region_RAM_start__ = 0x1fff0000;
                                             //前面的 0x410 RAM 留给 RAM User Vector Table 。
define symbol __ICFEDIT_region_RAM_end__   = 0x20000000;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x2000;
define symbol __ICFEDIT_size_heap__   = 0x2000;
/**** End of ICF editor section. ###ICF###*/


/**** 上边是由 ICF 编辑，下面是由我们手动配置 ****/

define symbol __region_RAM2_start__ = 0x20000000;
                                                 //SRAM 是分成两块的，RAM2 即 SRAM_U，RAM 为 SRAM_L
define symbol __region_RAM2_end__ = 0x20000000 + __ICFEDIT_region_RAM_end__
                                                  - __ICFEDIT_region_RAM_start__;


define exported symbol __VECTOR_TABLE = __ICFEDIT_intvec_start__;
                        //代码编译进 ROM ,则 0x00000000 ; RAM，则 __ICFEDIT_region_RAM_start__
define exported symbol __VECTOR_RAM = __ICFEDIT_region_RAM_start__;
                        //前面的 RAM 留给 RAM User Vector Table,即这里的设置。所以减 0x410
                        //common_startup 函数就是把 __VECTOR_TABLE 的数据复制到 __VECTOR_RAM

define exported symbol __BOOT_STACK_ADDRESS = __region_RAM2_end__ - 8;
                                                        //0x2000FFF8;  启动栈地址

/* 决定代码编译的地址 */
define exported symbol __code_start__ = __ICFEDIT_intvec_start__ + 0x410;
                                                    //+0x410 ,是前面的留给 Vector Table

define memory mem with size = 4G;                   //4G 的虚拟寻址空间

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

define region RAM_region   =
          mem:[from __ICFEDIT_region_RAM_start__ + 0x410  to __ICFEDIT_region_RAM_end__]
        | mem:[from __region_RAM2_start__ to __region_RAM2_end__];


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

//手动初始化，在 common_startup 函数 里完成
initialize manually { readwrite };          // 未初始化数据 .bss
initialize manually { section .data};       // 已初始化数据
initialize manually { section .textrw };    // __ramfunc 声明的子函数

do not initialize  { section .noinit };     // 复位中断向量服务函数

define block CodeRelocate { section .textrw_init };
define block CodeRelocateRam { section .textrw };
                  // 把 CodeRelocate 代码复制到 RAM 中的 CodeRelocateRam (对 flash 操作的函数必须这样)

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };
    //vectors.c 中设置#pragma location = ".intvec"，告诉编译器，这个是中断向量表，编译进去 .intvec

place at address mem:__code_start__ { readonly section .noinit };
```

中断向量表地址。Flash 和 SRAM 启动的中断向量表地址是不一样的

设置 P-flash 和 D-flash 的分区大小

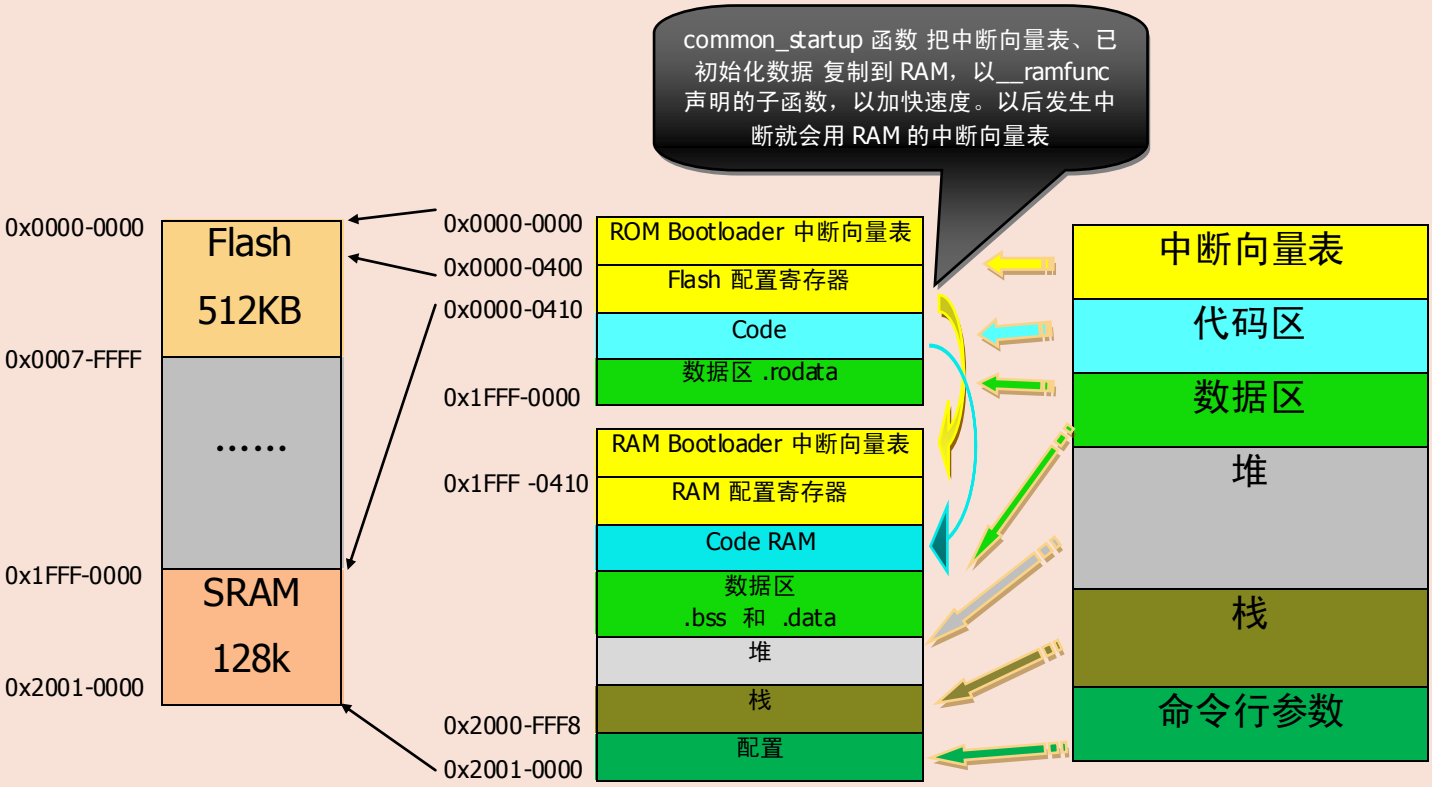
设置堆栈大小

堆栈大小

```
68.                                    //在 crt0.s 中设置了 Reset_Handler 为 SECTION .noinit : CODE
69.                                    //即把 Reset_Handler 编译进__code_start__
70.
71.  place in ROM_region    { readonly, block CodeRelocate };
72.                                    //把代码编译进去 ROM（调试用），非调试，则设为 ROM_region
73.
74.  place in RAM_region    { readwrite, block CodeRelocateRam,
75.                          block CSTACK, block HEAP };
```

ROM 启动，就要把只读数据和代码编译进 ROM

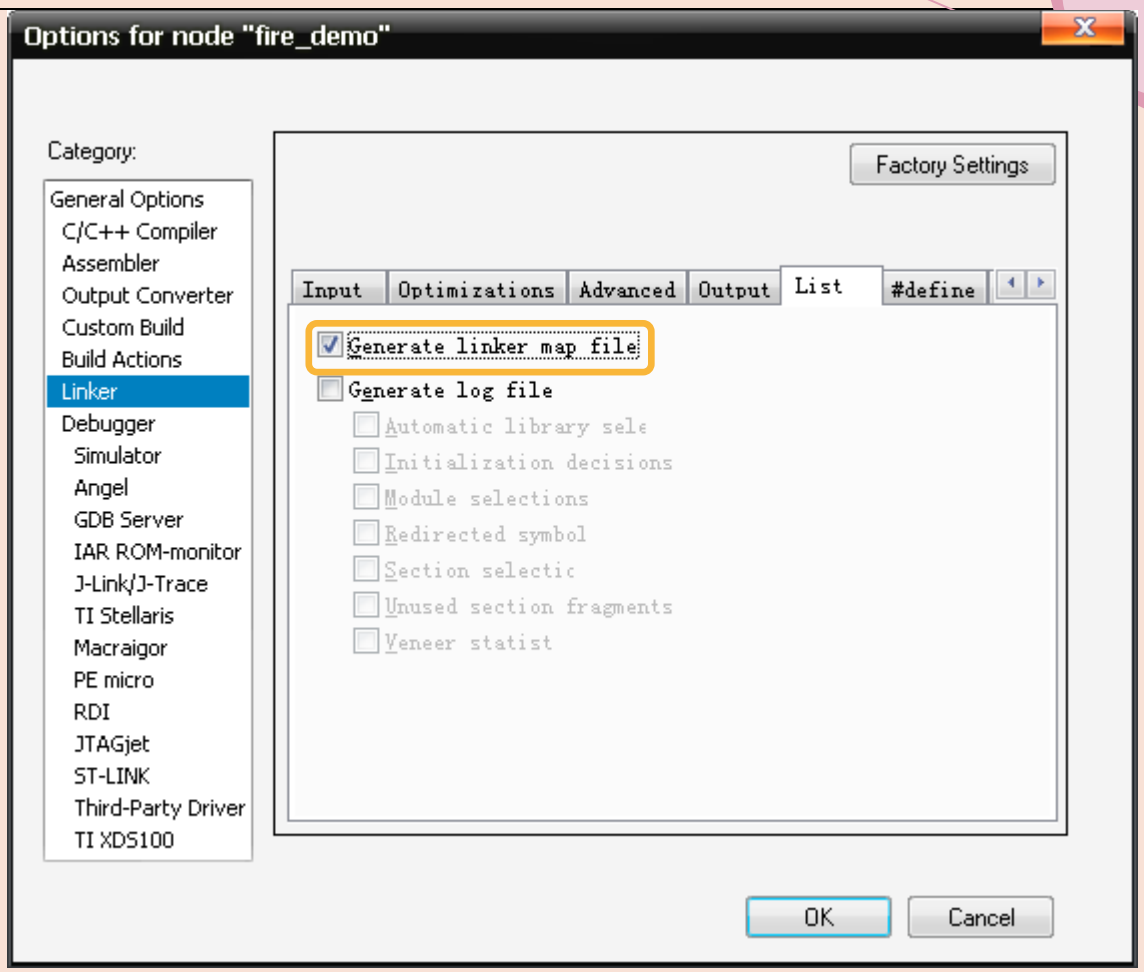
Linker 根据 icf 配置文件来进行分配内存地址：



前面的内容，不知道大家是否都看懂呢？熟悉单片机的内存分配，是每一个电子软件工程师所必备的。如果对单片机的内存分配都不了解，那就算会写单片机的驱动程序，也难以保证程序的质量。

那在 IAR 里，怎么知道编译生成的文件的内存图？

在工程选项里，设置生成 map file 就行：



设置好，编译后，在 fire_Kinetis\build\xxx_demo\xxx 模式\List 文件夹下就会有 .map 文件。

这里的内容，就是 ICF 配置文件底部的宏替换内容

```
"A1":  place at 0x00000000 { ro section .intvec };
"A2":  place at 0x00000410 { ro section .noinit };
"P1":  place in [from 0x00000000 to 0x00040000] { ro, block CodeRelocate };
"P2":  place in [from 0x1fff0410 to 0x20010000] {
          rw, block CodeRelocateRam, block CSTACK, block HEAP };
```

```
  Section              Kind        Address    Size  Object
  -------              ----        -------    ----  ------
"A1":                                        0x410
  .intvec              const    0x00000000   0x410  vectors.o [1]
                              - 0x00000410   0x410
"A2":                                         0x3c
  .noinit              ro code  0x00000410    0x3c  crt0.o [1]
                              - 0x0000044c    0x3c
"P1":                                       0x19a4
  .text                ro code  0x0000044c   0x55c  printf.o [1]
  .text                ro code  0x000009a8    0x20  stdlib.o [1]
  .text                ro code  0x000009c8   0x520  start.o [1]
```

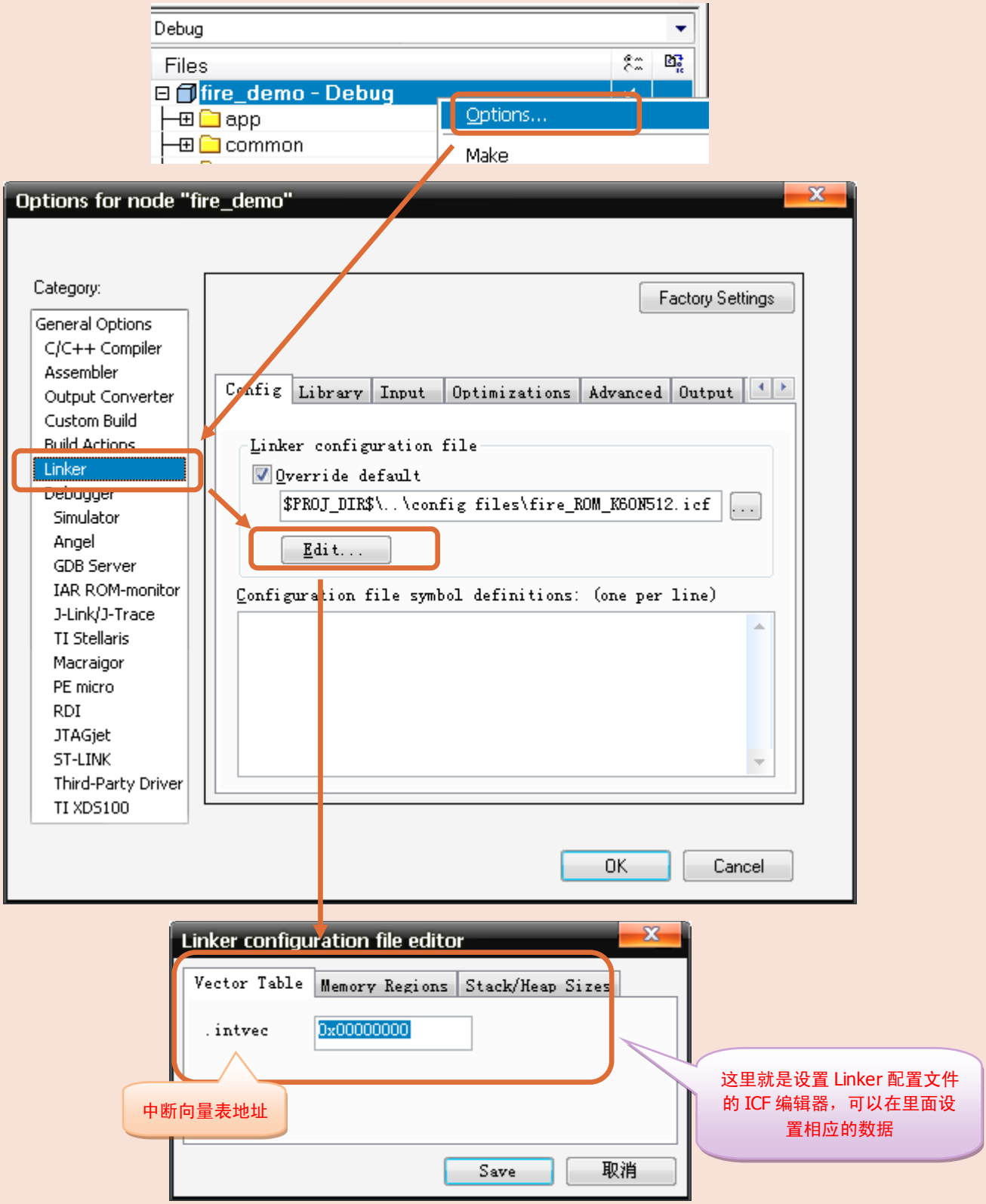
中断向量表

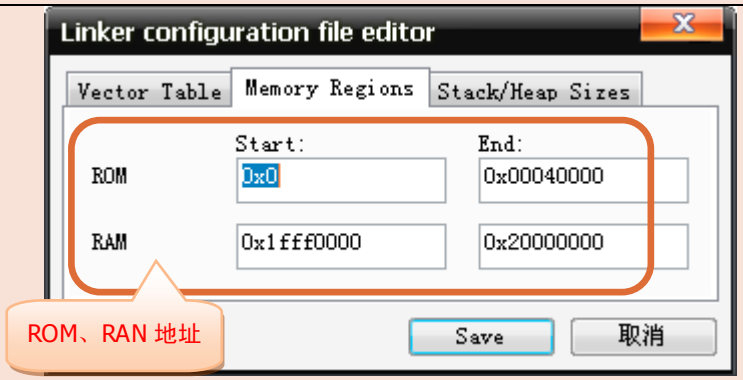
复位中断服务函数

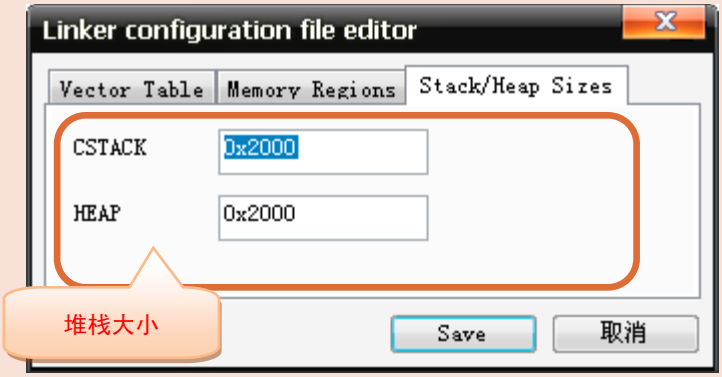
程序代码

那什么是 ICF 编辑器呢？

呵呵，在工程名里右键——Options——Linker




Linker configuration file editor
Vector Table
Memory Regions
Stack/Heap Sizes
Start:
End:
ROM
0x0
0x00040000
RAM
0x1fff0000
0x20000000
Save
取消
ROM、RAN 地址


Linker configuration file editor
Vector Table
Memory Regions
Stack/Heap Sizes
CSTACK
0x2000
HEAP
0x2000
Save
取消
堆栈大小

IAR 的使用

安装 IAR

野火推荐的是直接在野火提供的 K60 库模版上添加你们自己的代码，而不是自己重建工程，但学会自己建立工程是必须的，这里给大家详解建立 IAR 工程的步骤。

一、　下载安装软件

目前最新的 IAR for ARM 为 v6.30，支持更多的 Kinetics 系列芯片，因此推荐大家更新，避免因为版本太低而出现不兼容，甚至出现异常错误的情况。

下载地址：CD-EWARM-6301-3142.7z

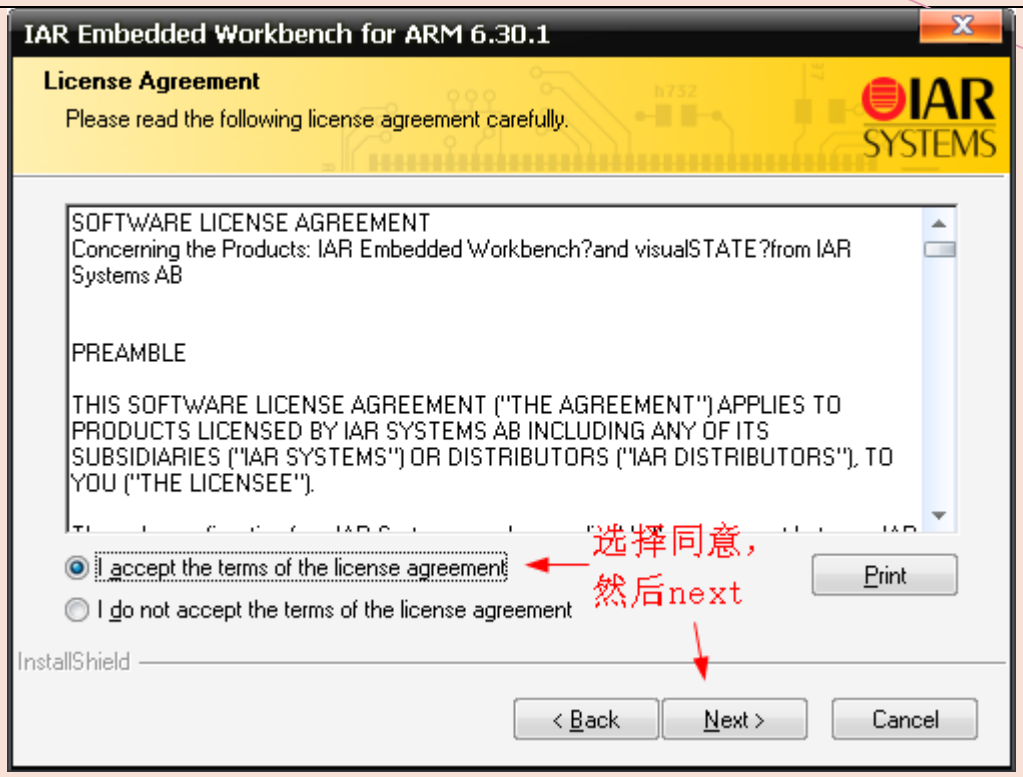
二、　安装 IAR 详细过程

1. 下载后解压文件，打开目录，运行安装文件：

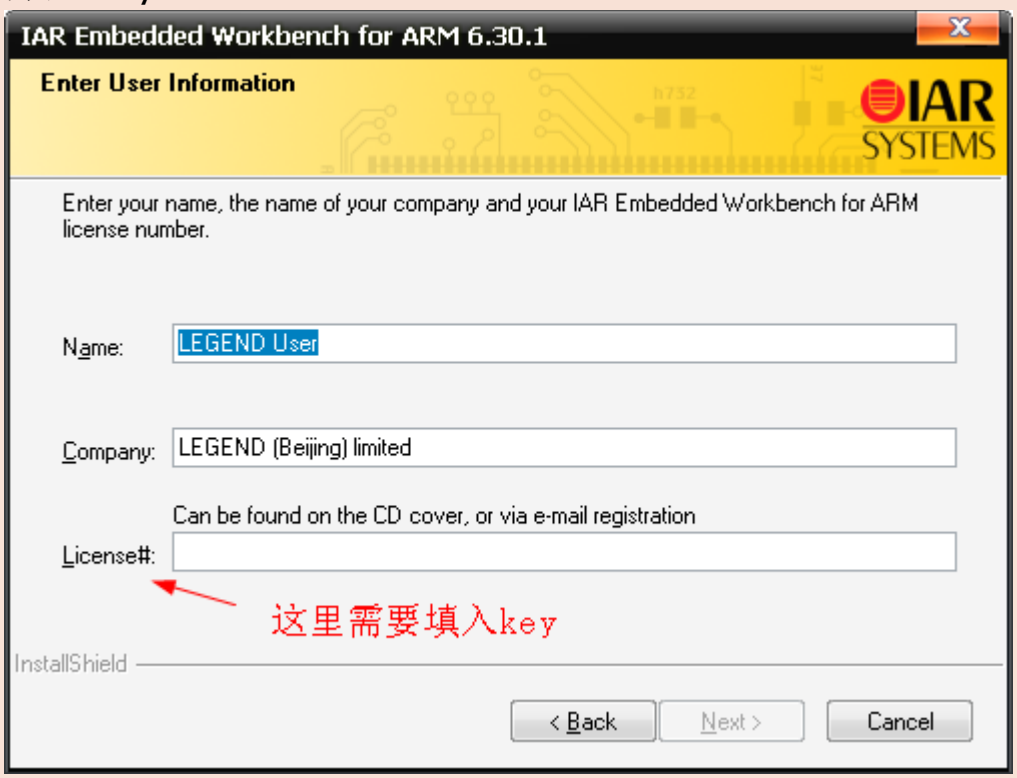
2. 选择安装 IAR Embedded Workbench

3. next

4. 选择同意，然后点击 next

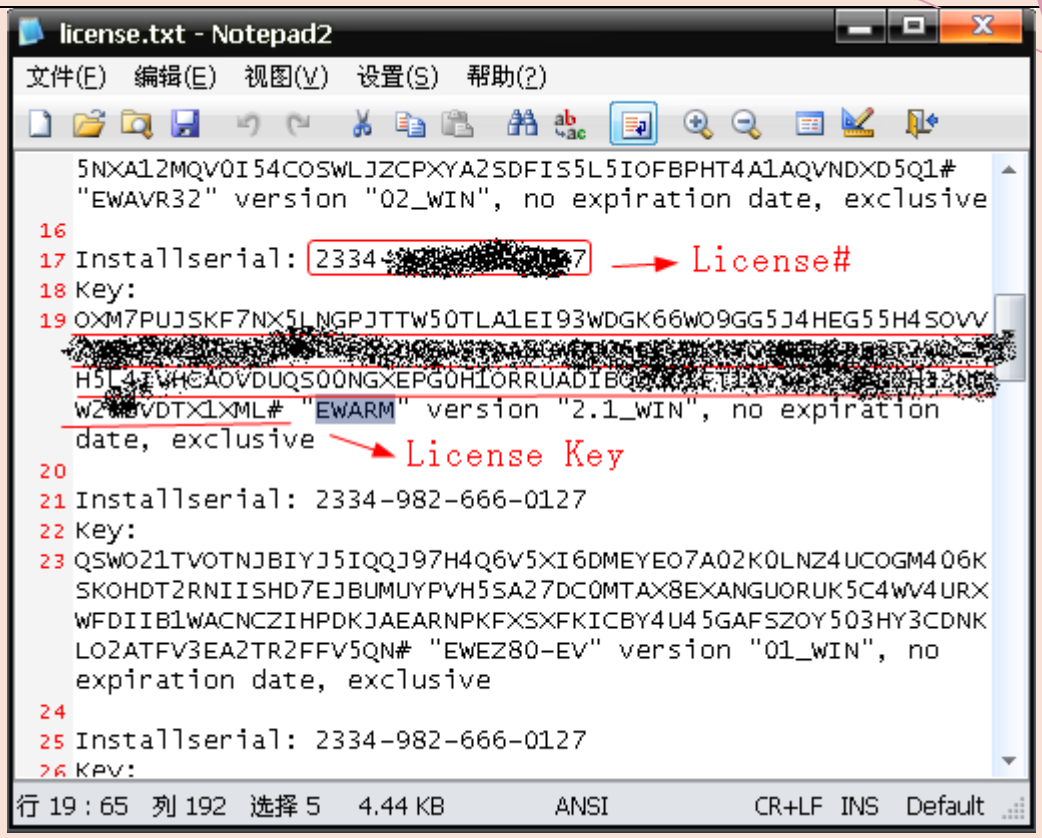


5. 需要填入 key

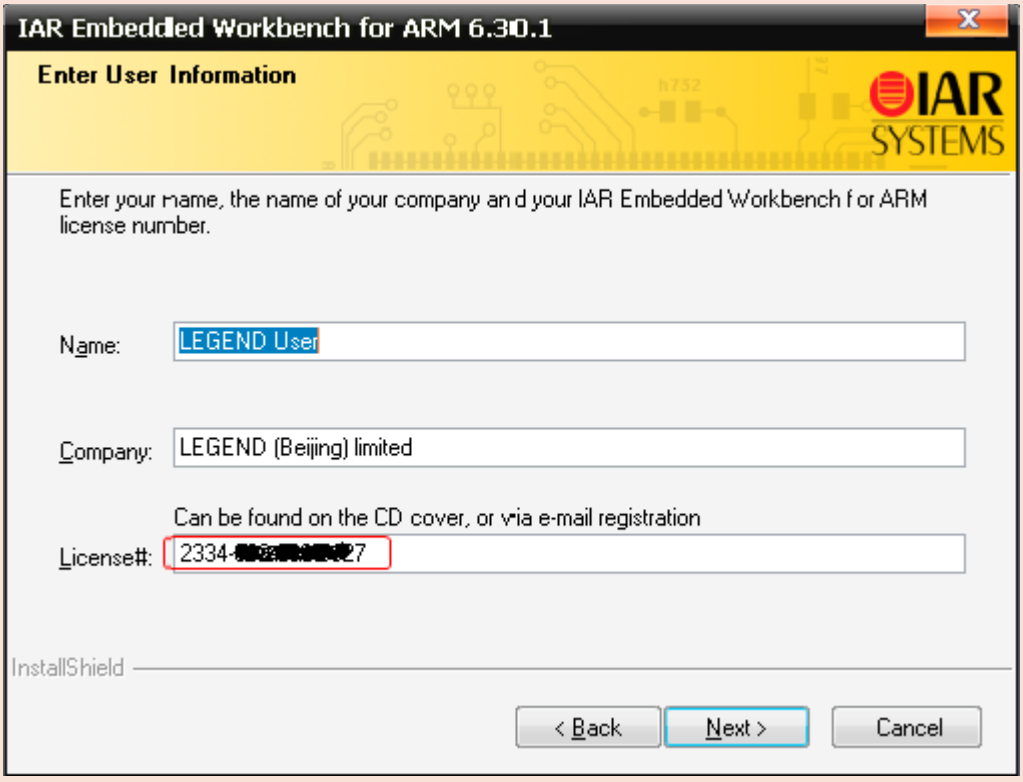


6. 打开软件目录，找到注册机文件，并运行它。

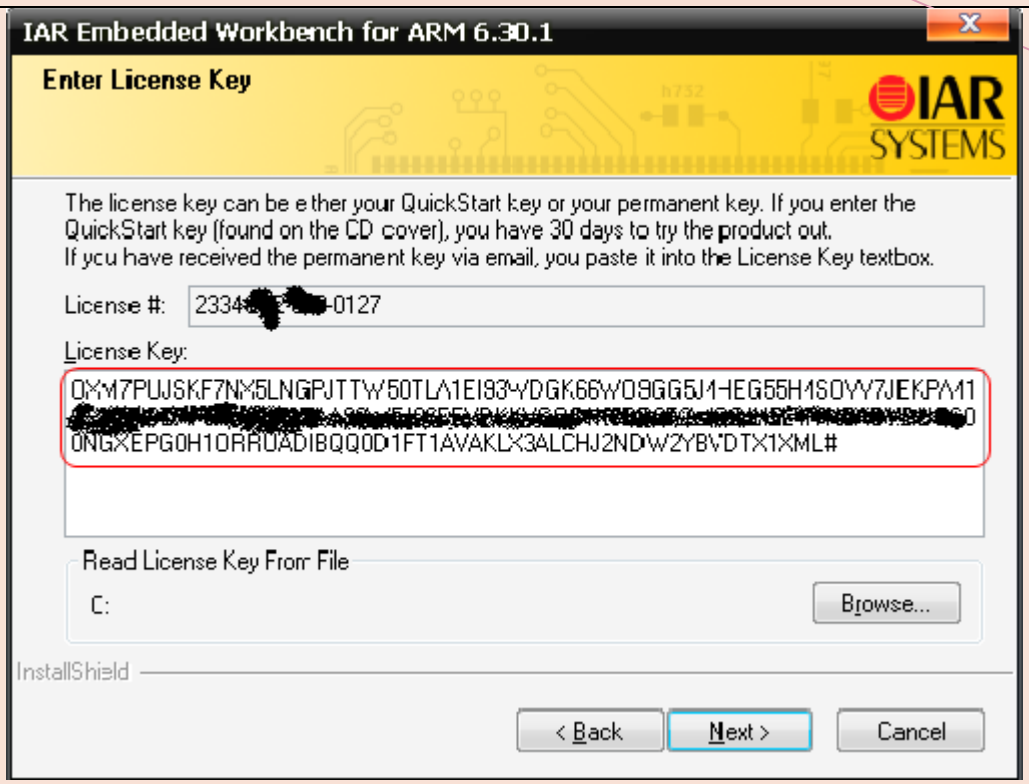
按下任意键，会自动打开 license.txt 文件，并提示：搜索"EWARM"，找到所在行的 key。



key 上一行的 Installserial 即为我们这里所需要的：

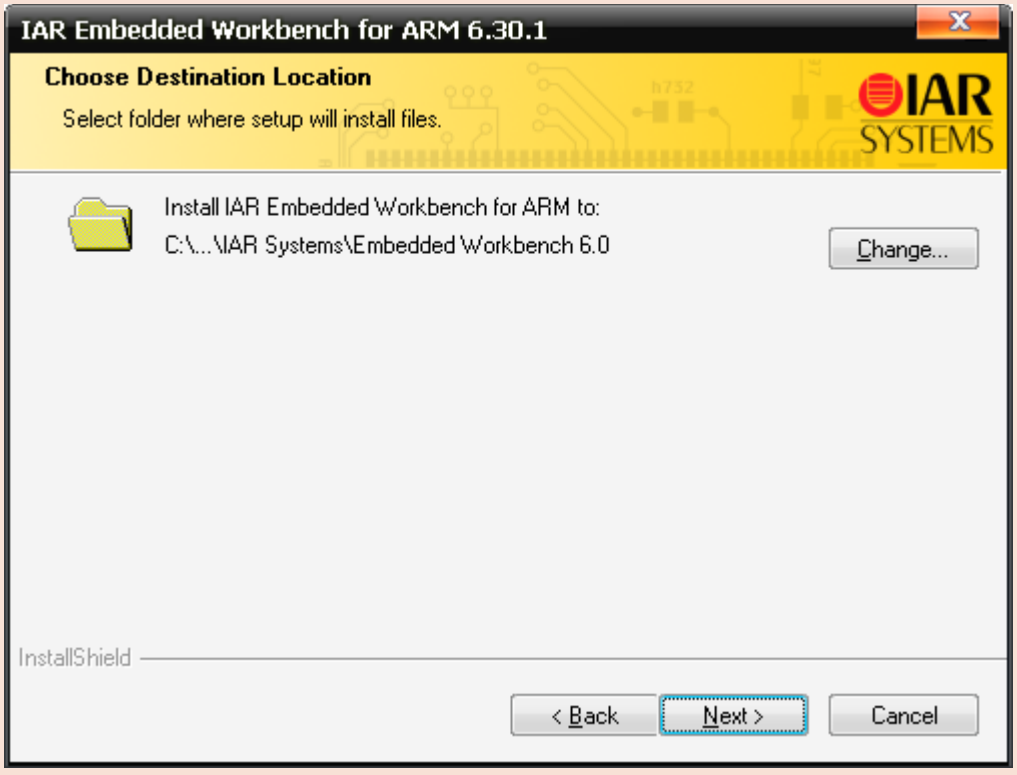


7. Next 后，填入 key 所在行的 License Key:

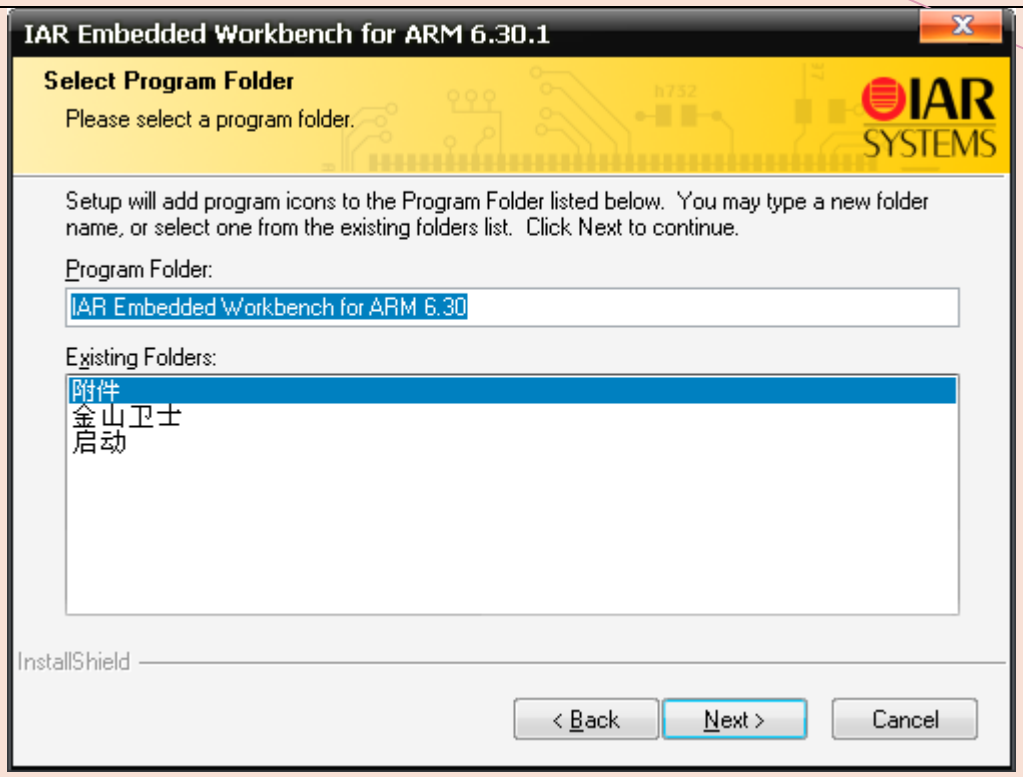


包括#号哦！！！

8. Next 后，根据自己情况选择安装路径……

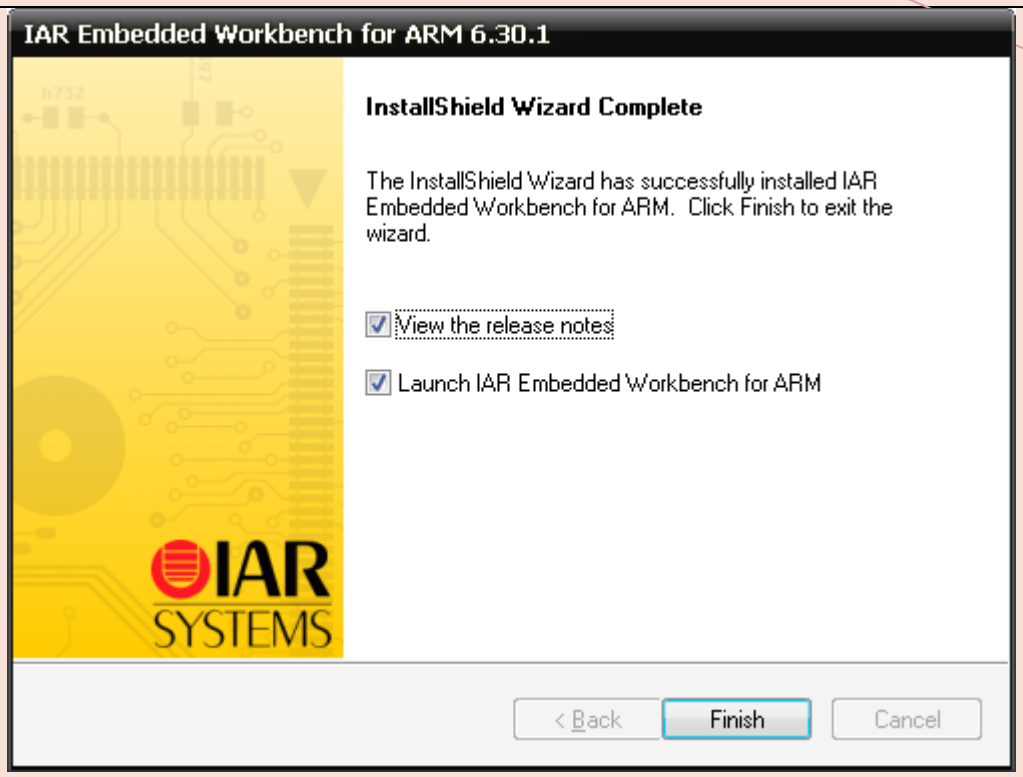


9. 设置「开始」菜单文件夹，还是继续 Next……

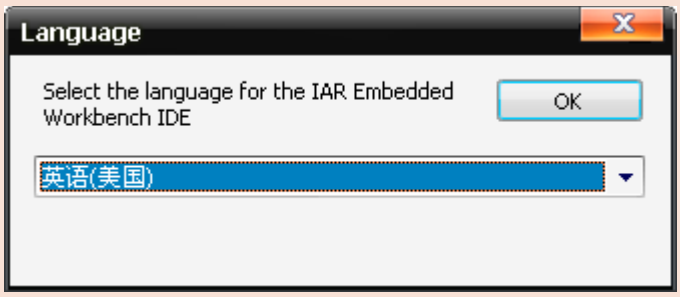


经过漫长的等待……

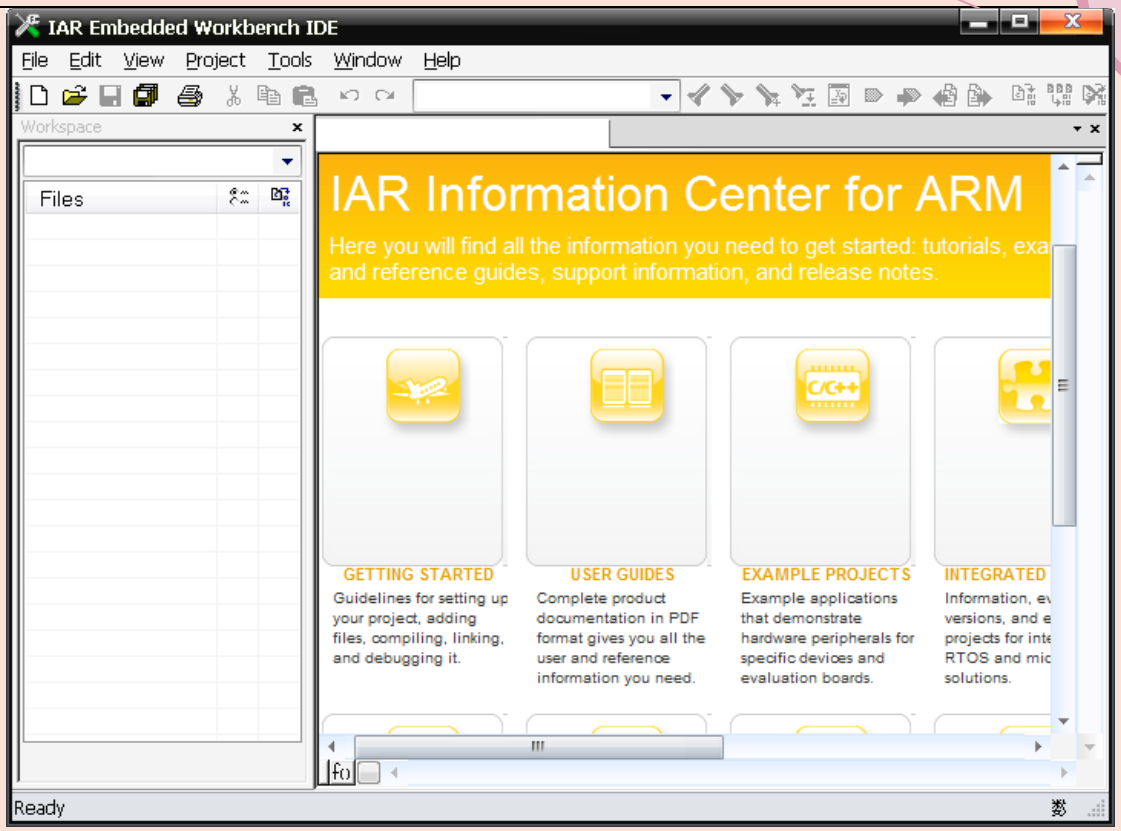
10. 噔噔噔噔……华丽丽地按个 finish 结束安装了



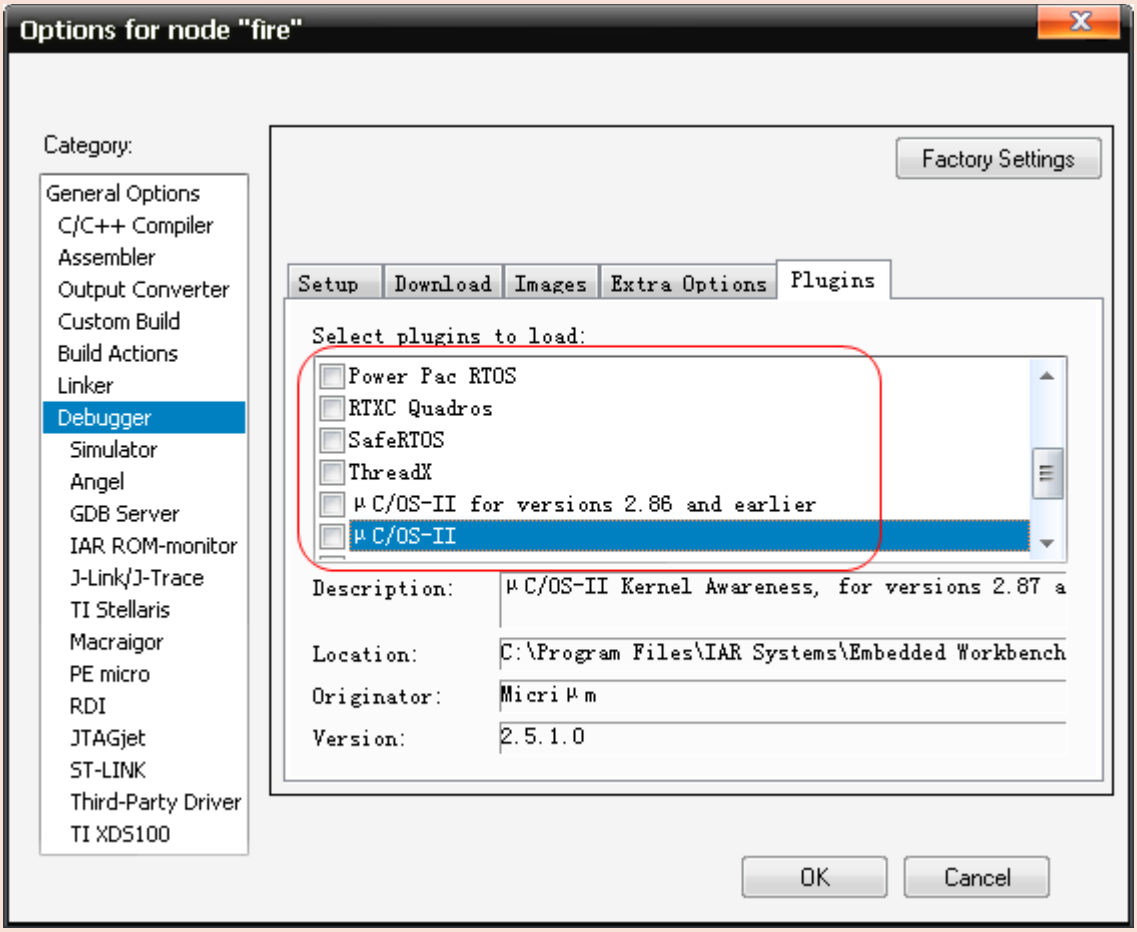
11. 运行 IAR，语言选择，按 OK 结束



12. IAR 的编程界面：

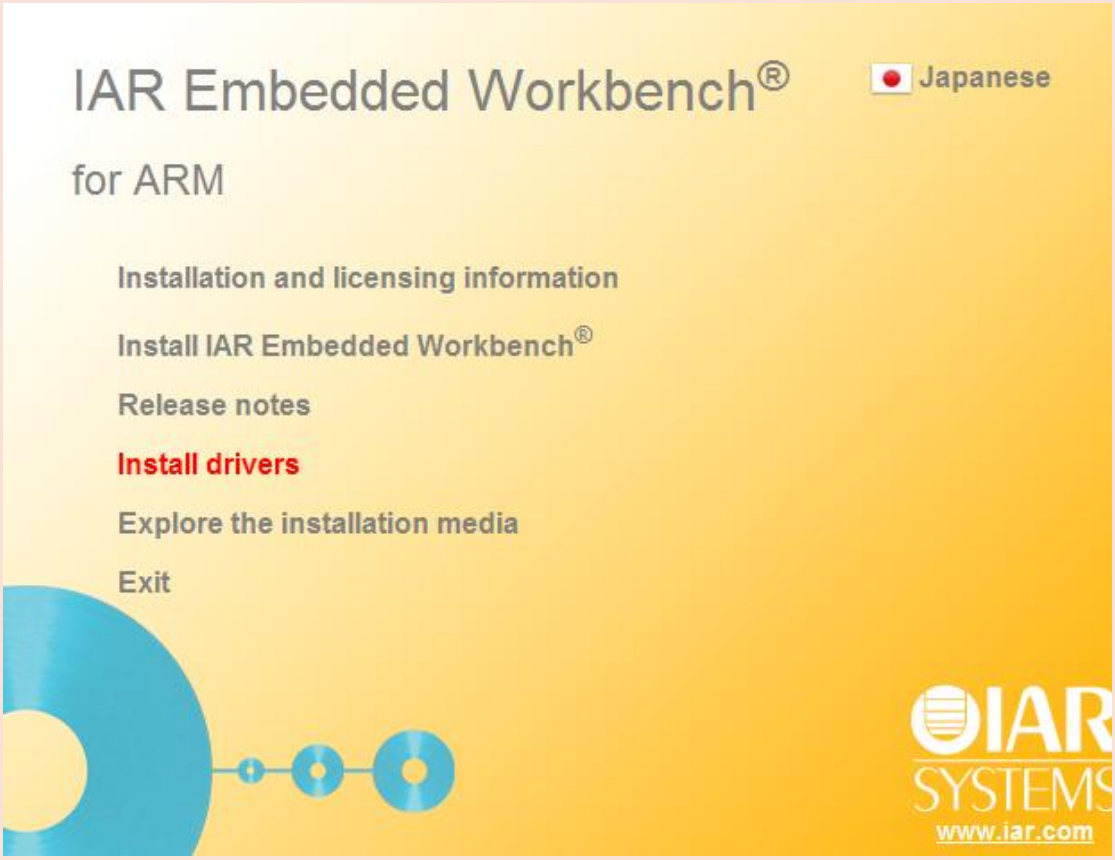


相对于 Keil For ARM、CodeWarrior 而言，IAR for ARM 的编程界面是最简单的，编译效率高，在嵌入式系统的调试方面提供了可供调试的插件：



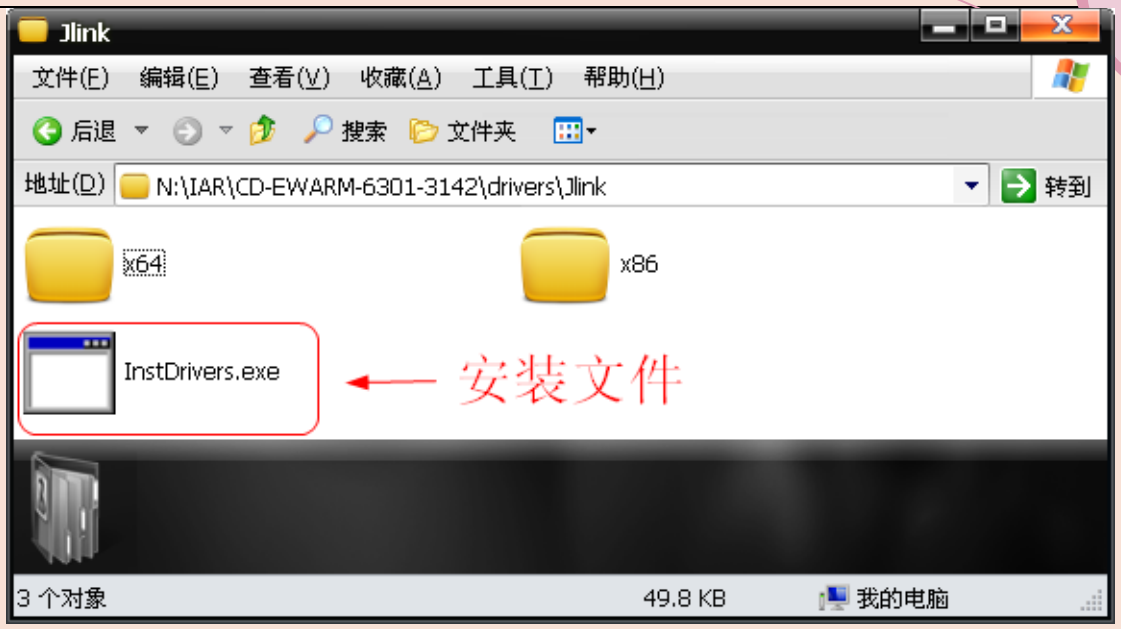
三、　安装仿真器驱动

1. 又换会原来的安装导航界面：选择 Install drivers



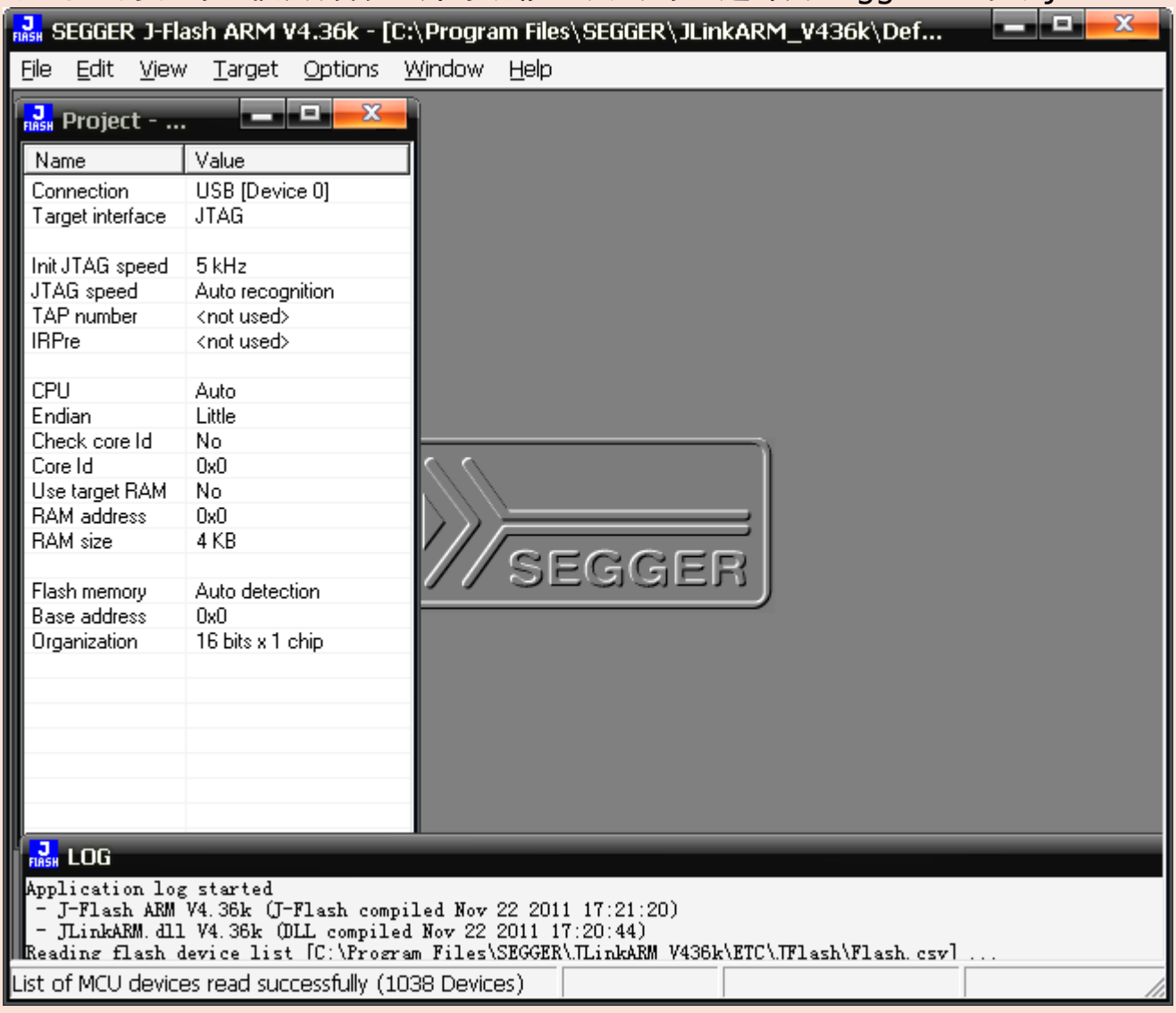
2. 野火 Kinetics 开发板，选择了用 jlink 作为仿真调试器，因此这里选择 jlink。

运行安装文件



运行后，会自动安装驱动，不会有其他提示。

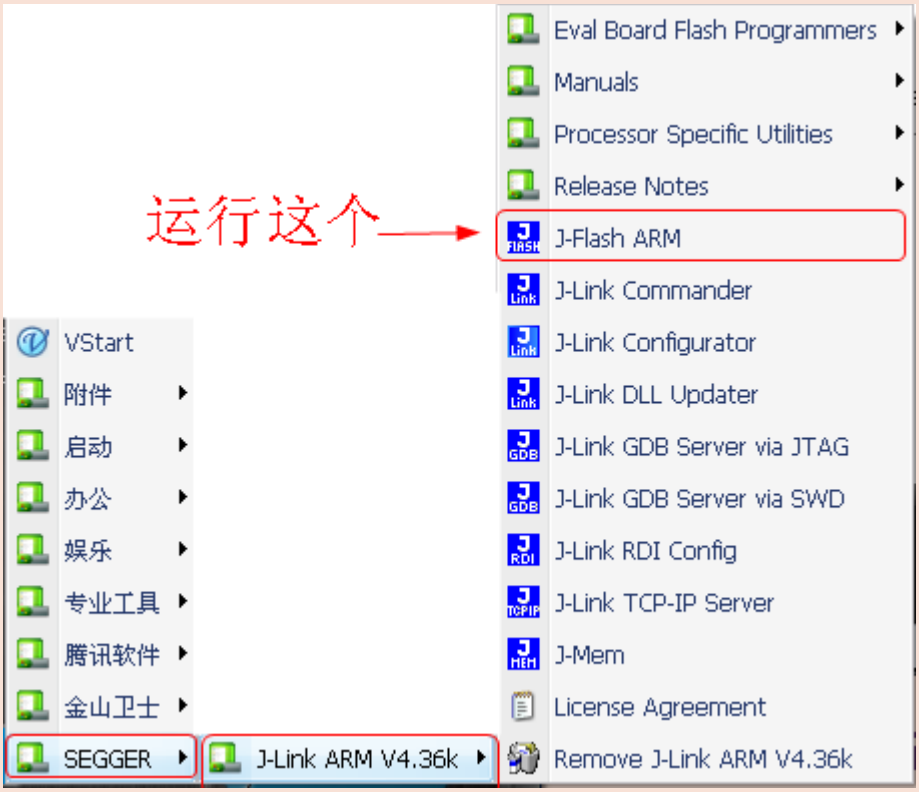
另外，为了可以使用擦除芯片等功能，我们可以选择用 segger 公司的 j-link：



下载地址：Setup_JLinkARM_V436k.zip

安装时，一路狂 Next 就行...

安装后，在开始菜单里找到 J-Flash ARM，运行它即可



建立 IAR 工程

创建工程文件

一、　准备材料

在建立 IAR 工程前，需要下载官方的 Kinetis 例程，提取里面的配置文件和官方写好的库。

下载地址：KINETIS512_SC.zip

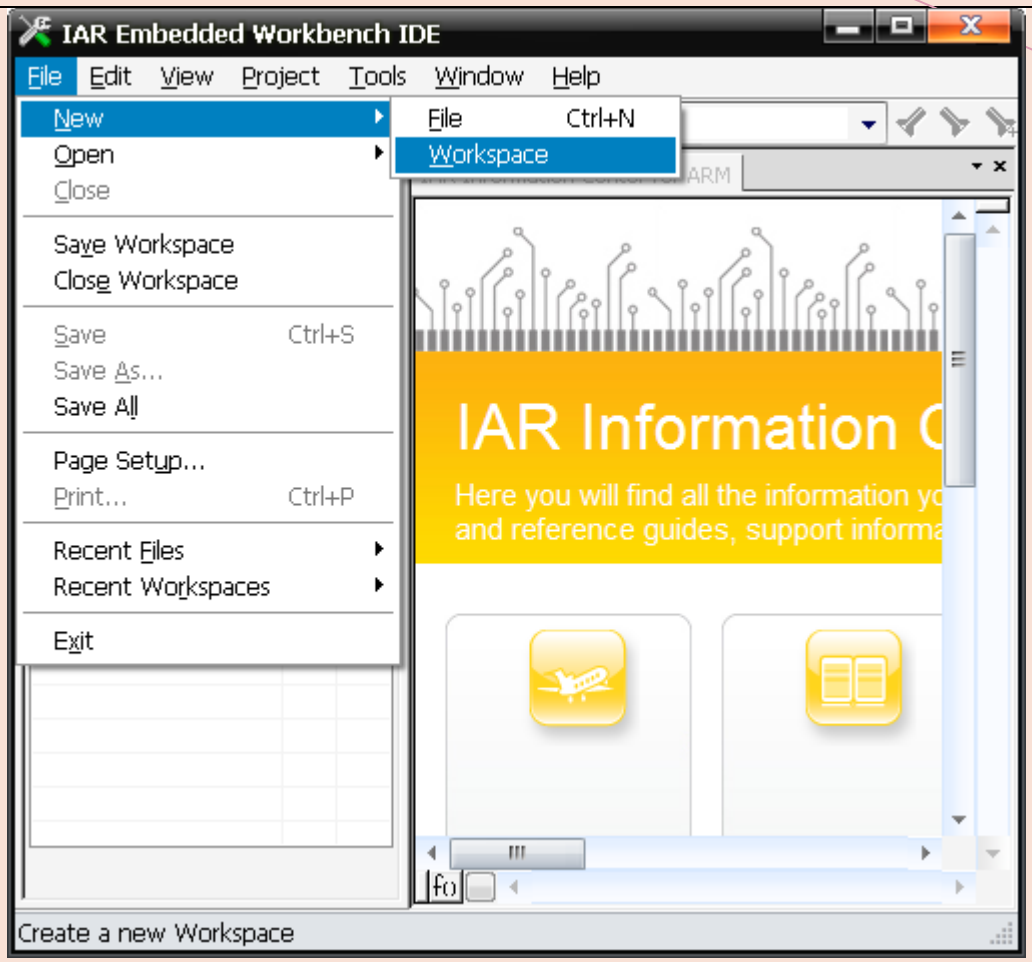
解压后，打开目录，会看到两个文件夹

二、　建立 IAR 工程过程

在这里，我们以 GPIO 为例子，点亮 LED，一步步操作，讲解建立工程的详细过程。

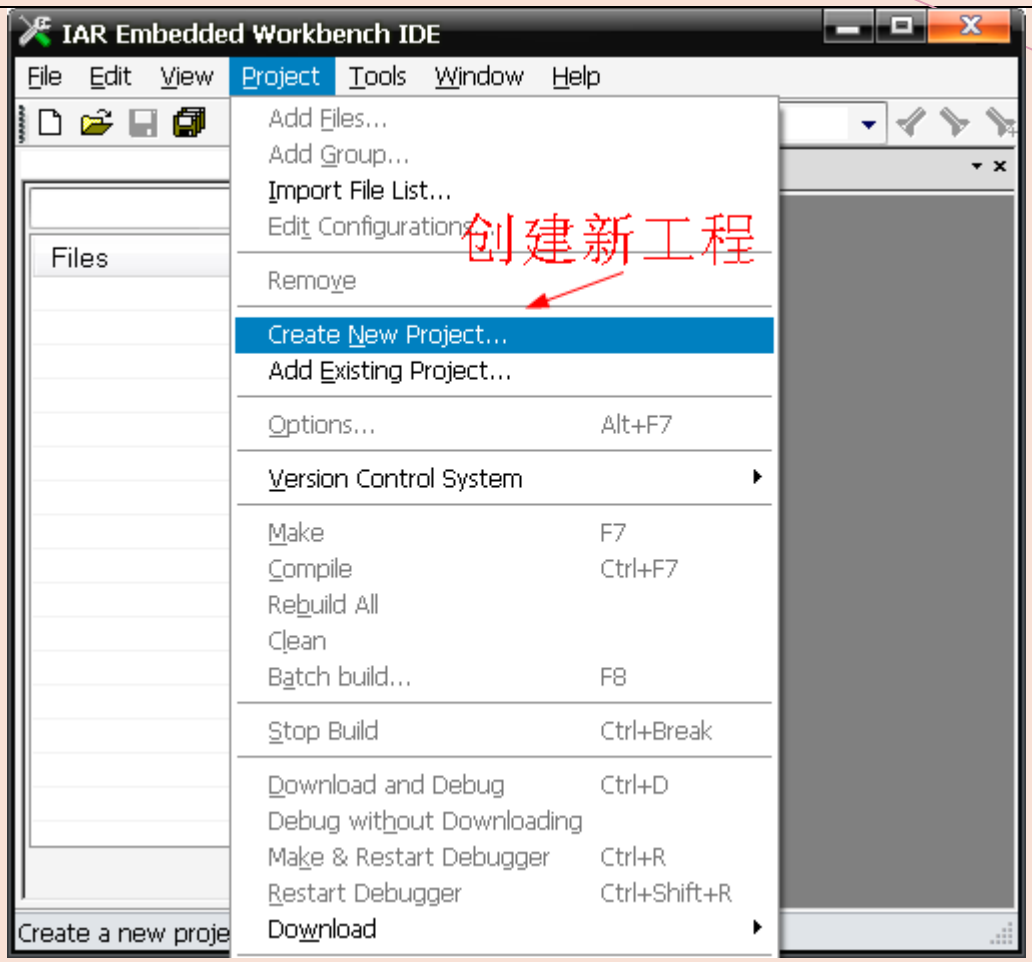
1. 建立工作空间

File——New——Workspace

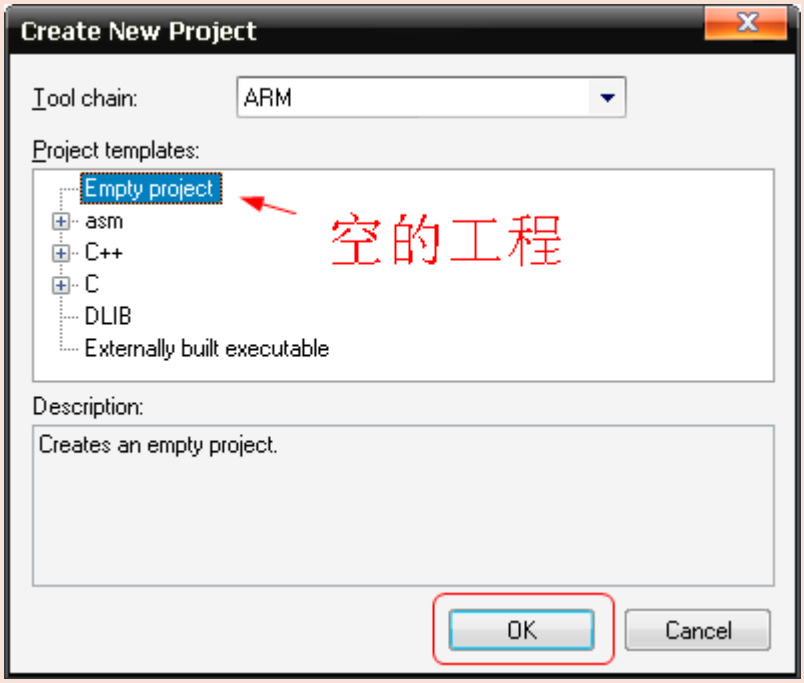


2. 建立工程

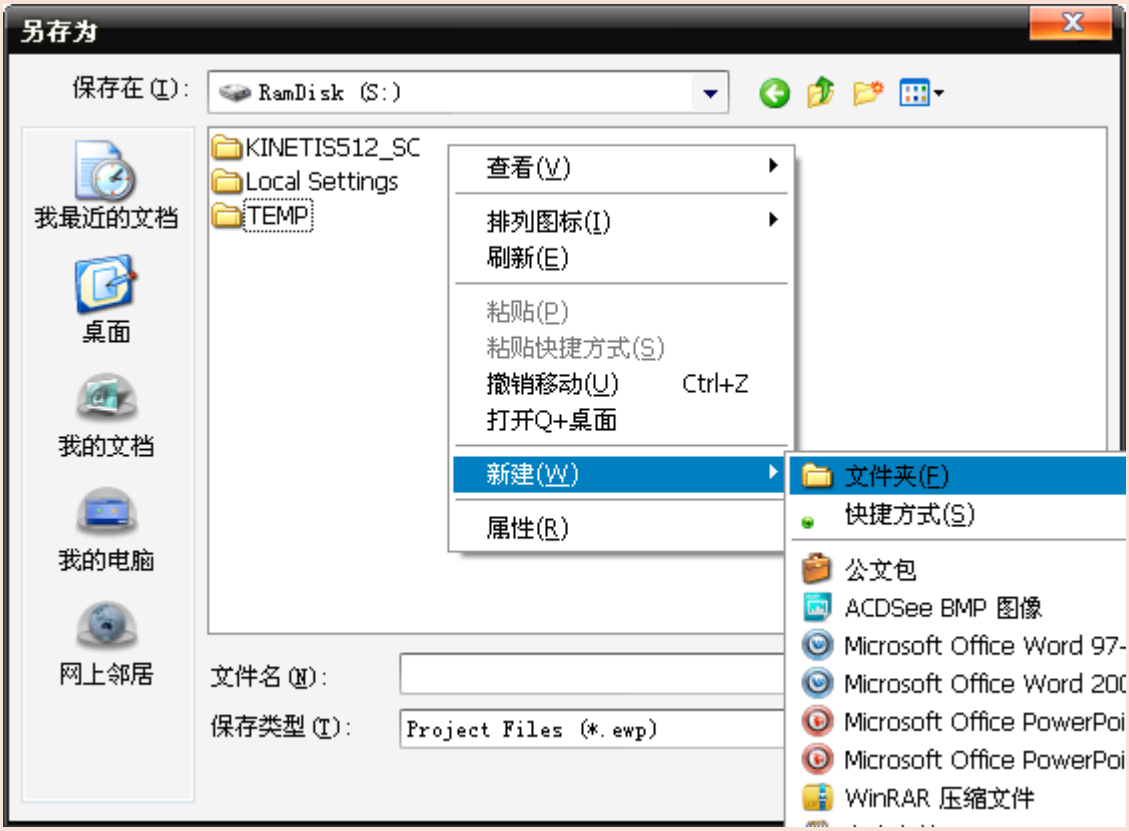
在工具栏里选择 Project——Create New Project



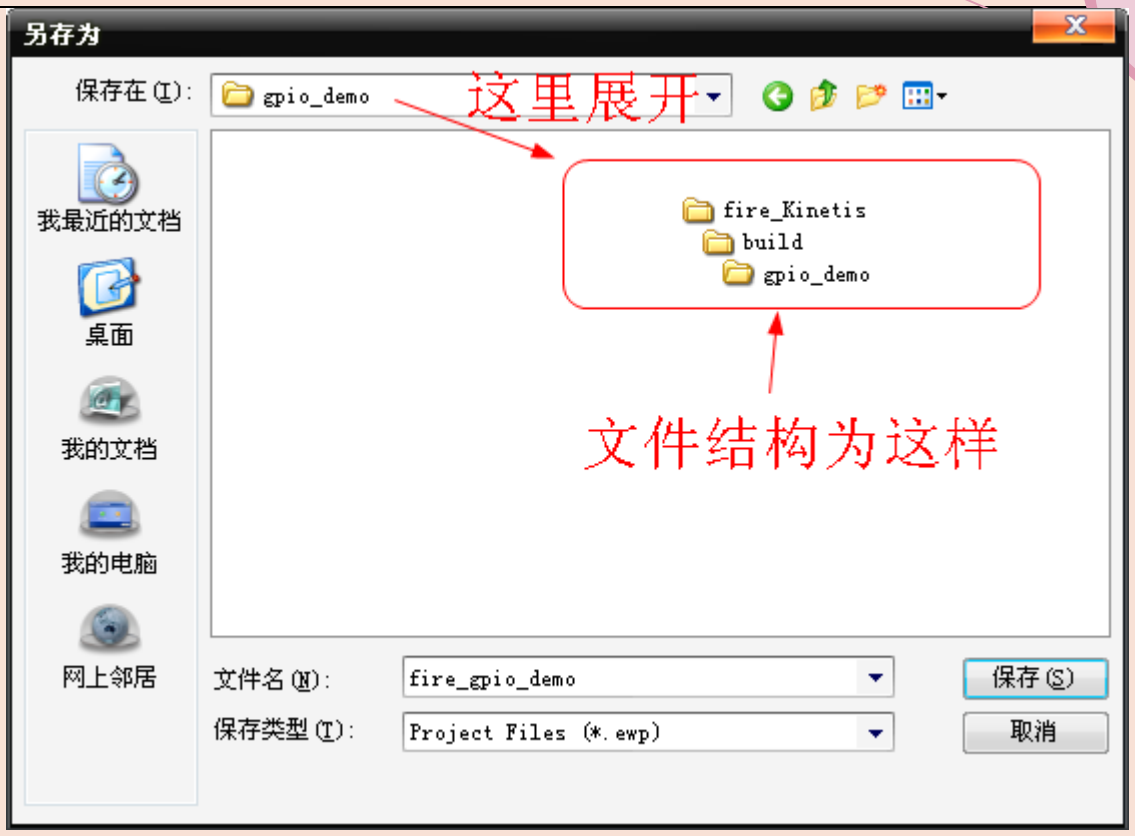
接着在弹出来的对话框里选择空的工程，点击确定



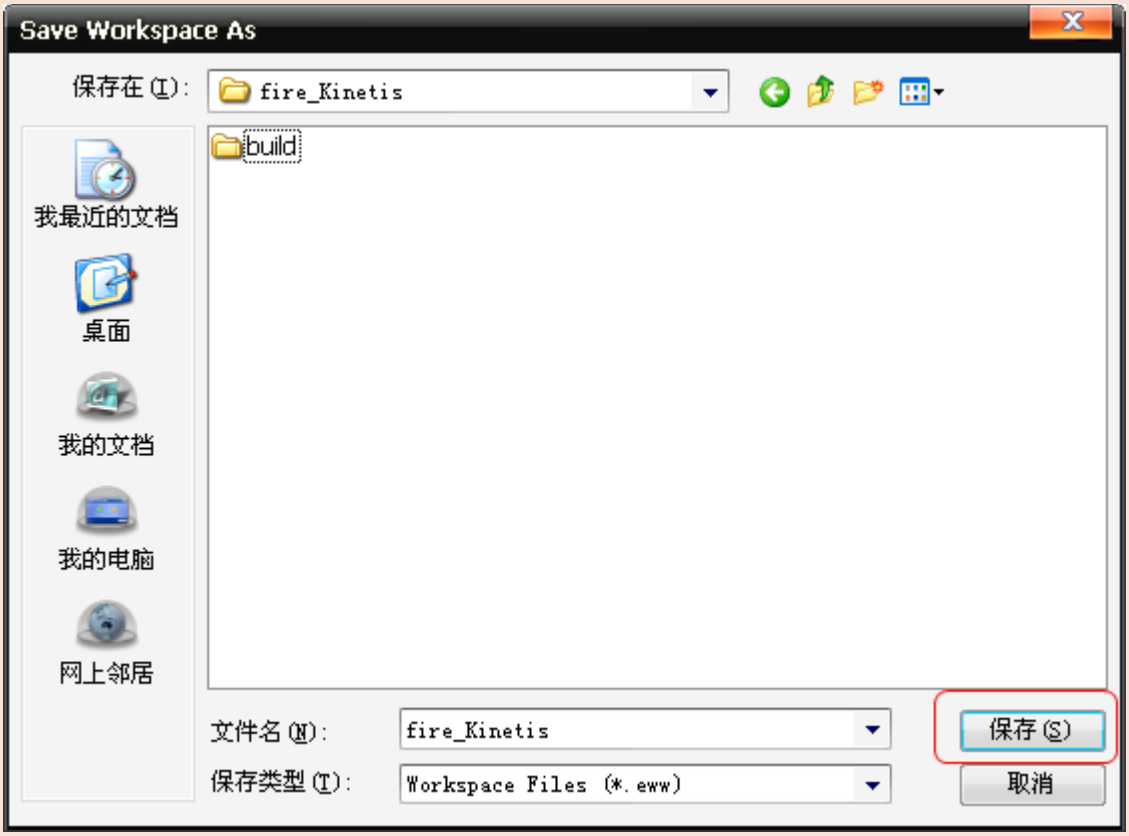
接着弹出选择保存工程的对话框。由于我们还没建保存工程的文件夹，我们就直接在对话框里新建：右键——新建——文件夹——重命名为：fire_Kinetis



打开新建的 fire_Kinetis 文件夹，再建一个 build 文件夹，build 文件里再建一个 gpio_demo 文件夹，把工程文件保存在 gpio_demo 文件夹里面，工程文件名为：fire_gpio_demo，文件结构如下图：



保存后，进入 IAR 界面，别忘了工作区还没有保存哦。在菜单栏里找到图标  ，点击保存全部文件，把 工作区文件保存在 fire_Kinetis 文件夹下，名字为 fire_Kinetis ：



这样，一个空的 IAR 工程骨架就建立完毕...需要我们往里面添加自己的模块。

3. 往工程添加 Kinetis 官方自带函数库

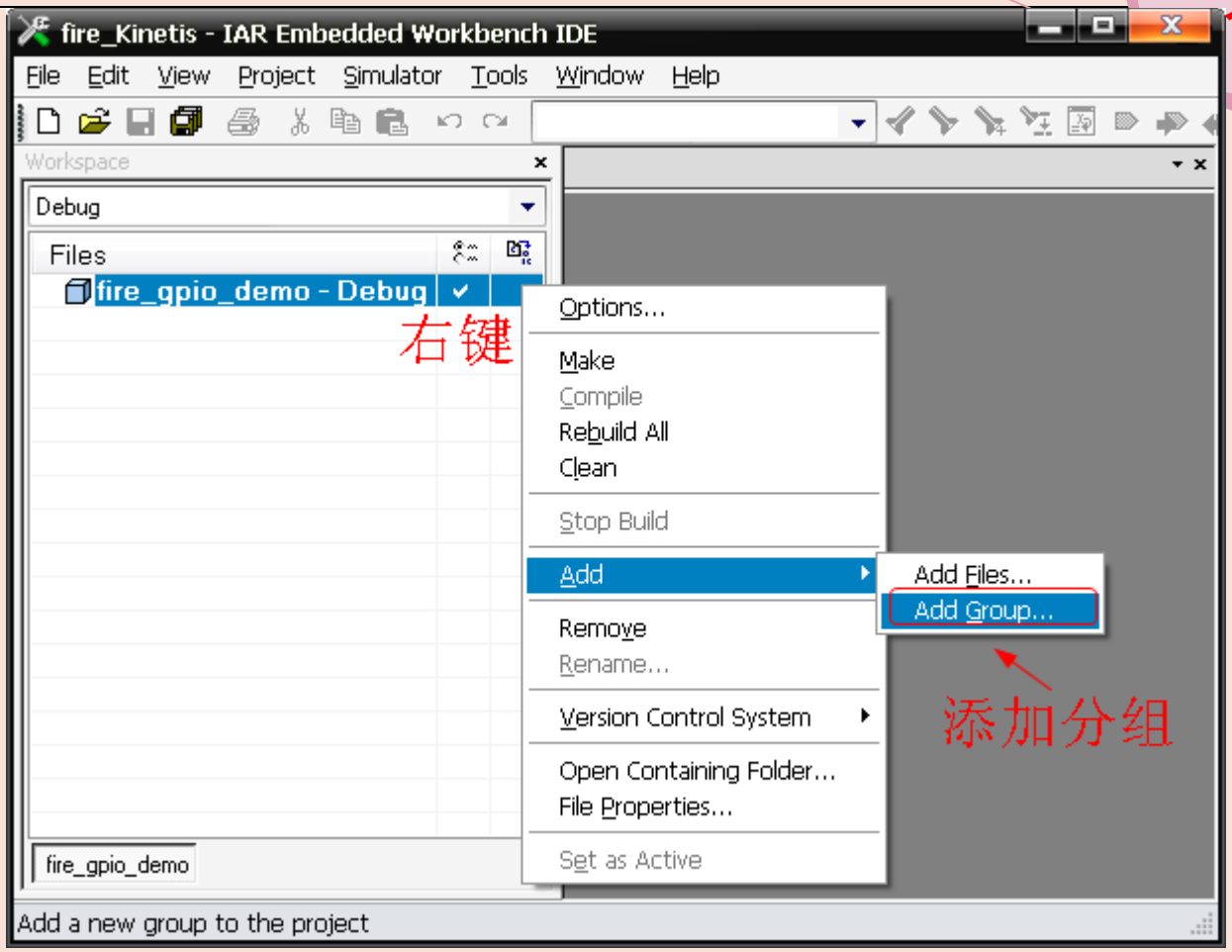
复制官方例程文件夹下的文件夹 src 到 fire_Kinetis 文件夹下 ：

另外，还需要把官方例程文件夹 KINETIS512_SC\build\iar 下的文件 iar.h 和 文件夹 config files 复制到 fire_Kinetis\build 文件夹下。

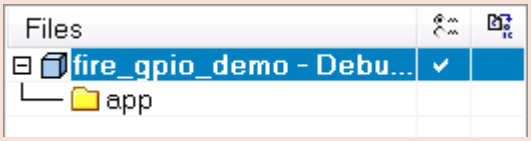
呵呵，怎么会多了个 settings 呢？这个是保存工作区后 IAR 自动创建的设置文件，我们不需要管他。

4. 添加分组，方便管理代码

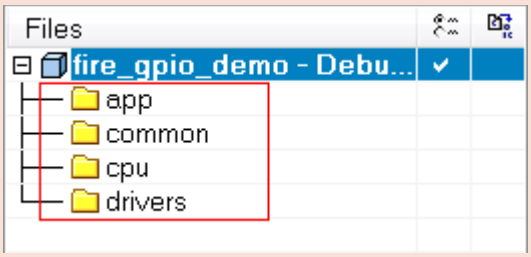
返回 IAR 界面，在工程里创建分组：



在弹出来的对话框里填入 “app”，这样就添加了一个 app 分组：



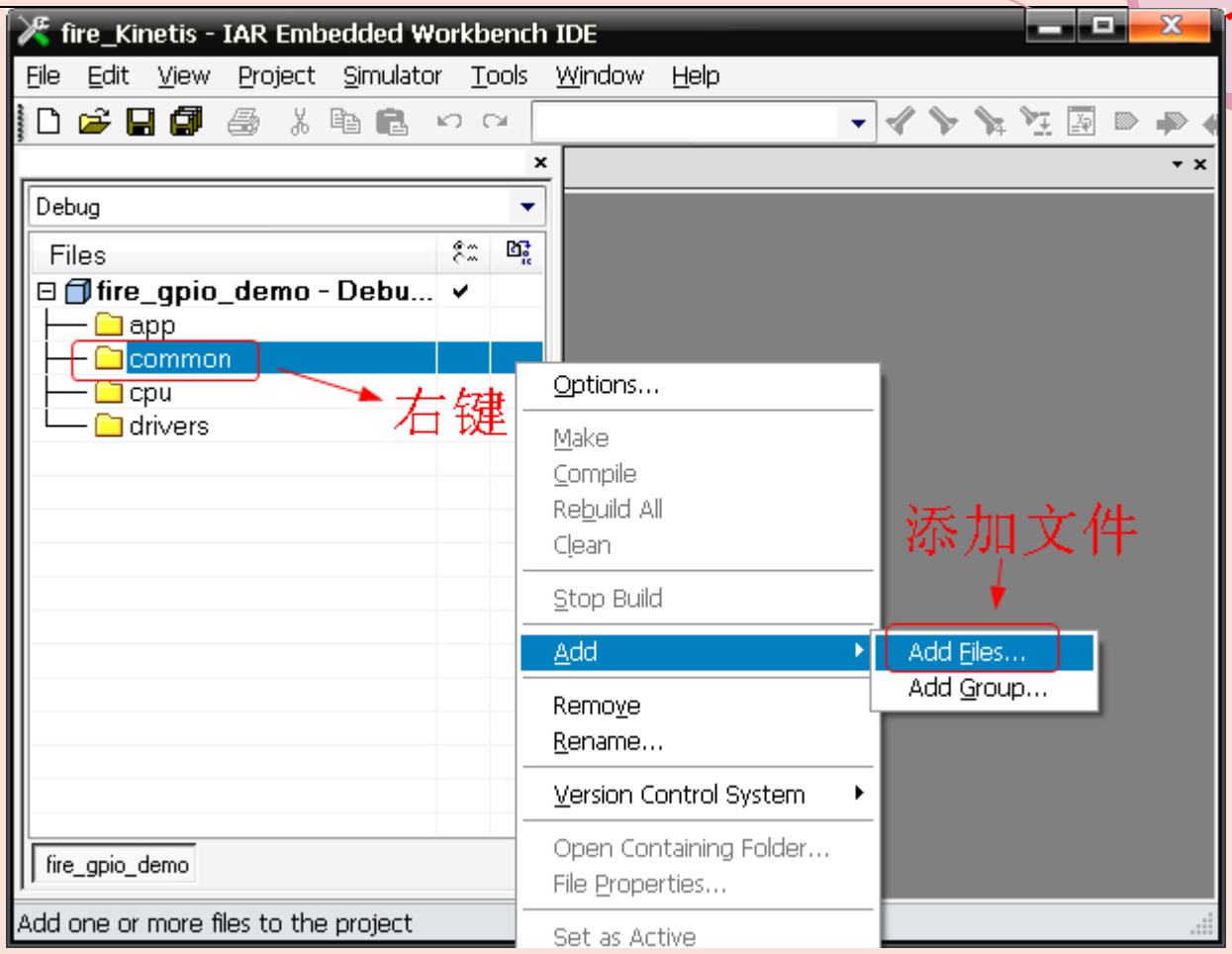
如此继续添加其他分组：common、cpu、drivers



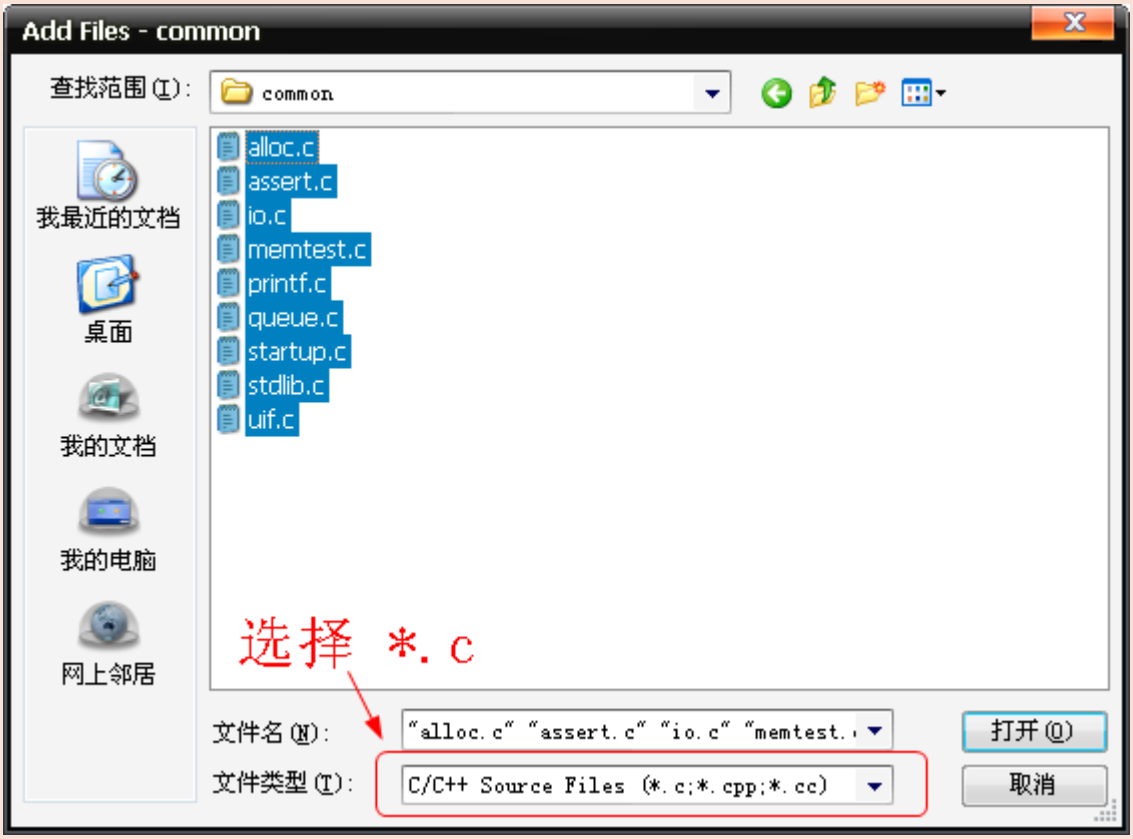
这些组有什么用呢？现在就往里面添加源代码文件，添加后，你就开始明白的。

5. 把代码文件放进分组

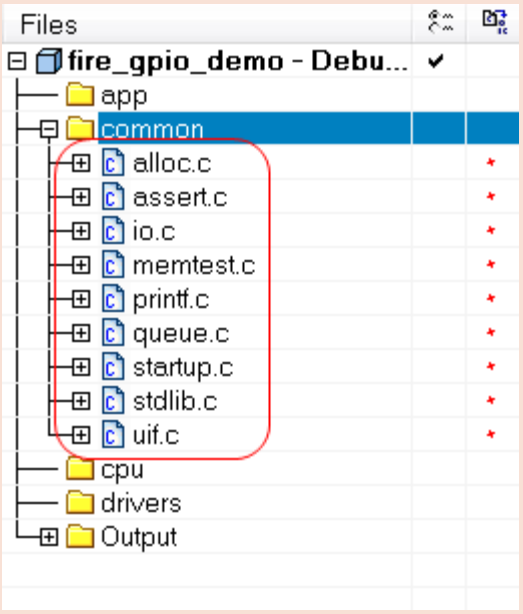
把 fire_Kinetis\src\common 文件夹 下的 *.c 文件全部添加到 common 分组里：



选中所有的 *.c 文件。如果你想把头文件也放在工程里，也可以把 *.h 文件选中。

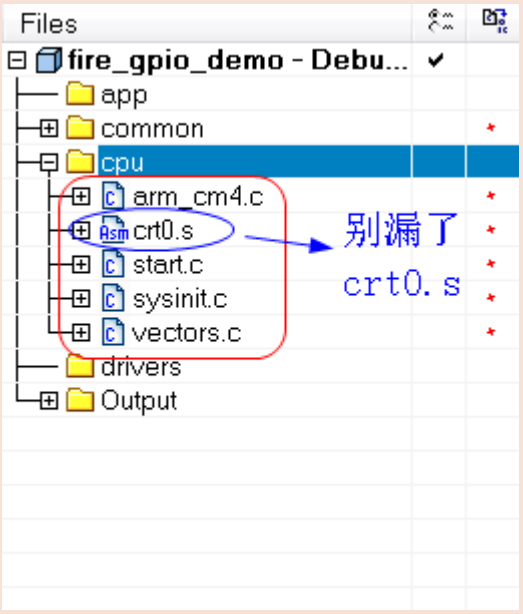


添加后：



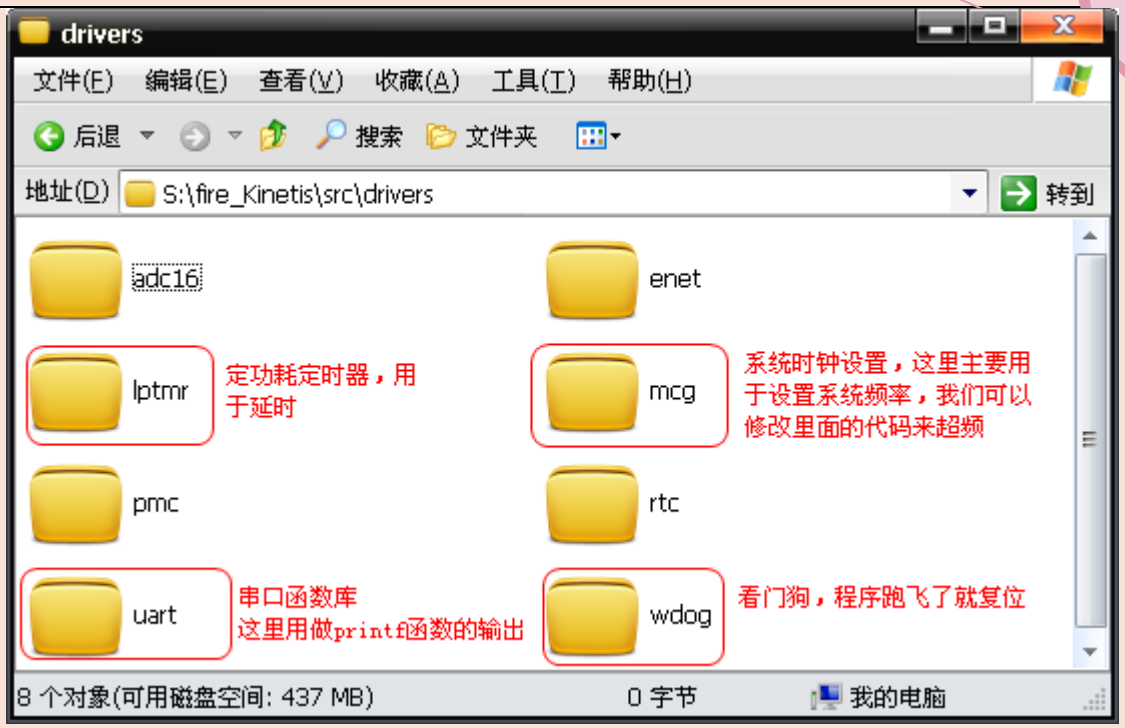
common 里的都是官方提供的通用库。

再把 fire_Kinetis\src\cpu 文件夹 下的 *.c 和 crt0.s 文件添加到 cpu 分组里：

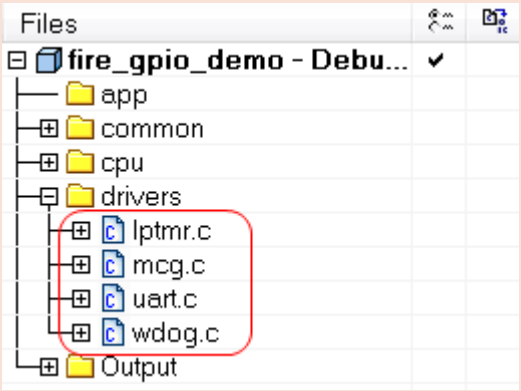


别漏了还有 crt0.s 哦。crt0.s 是启动文件，单片机上电复位后，就会执行里面的汇编代码。

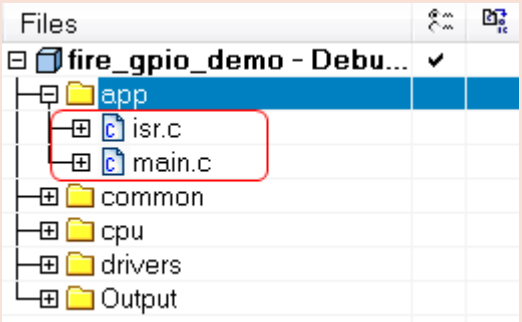
再把 fire_Kinetis\src\drivers 文件夹下的 lptmr、 mcg、uart、wdog 子文件 里 *.c 文件添加进工程了：

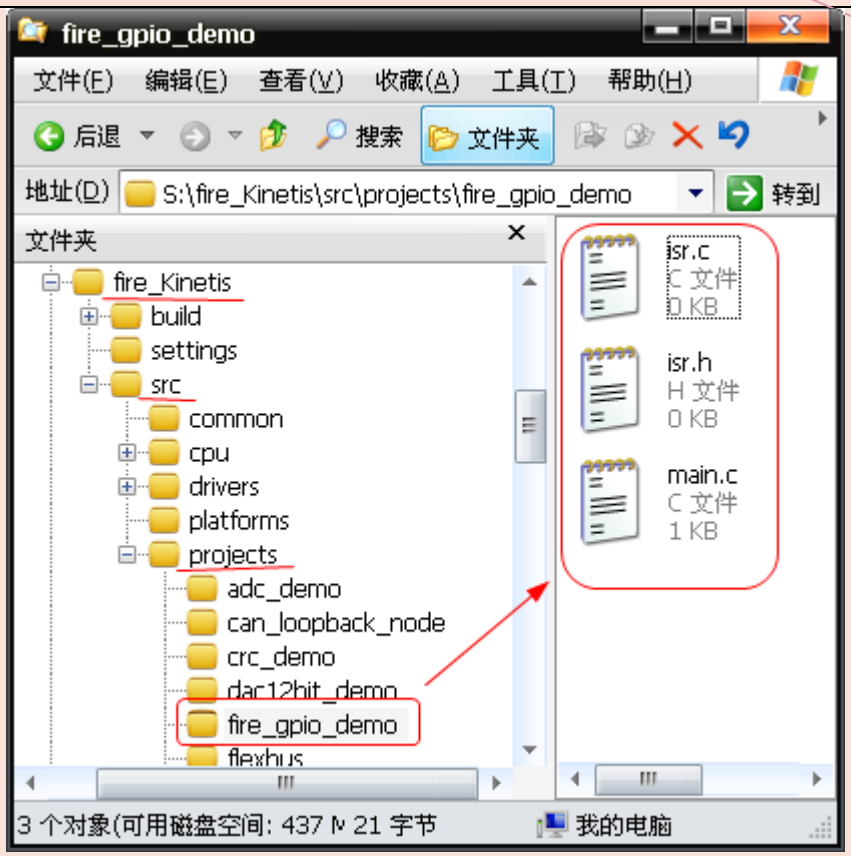


添加后：



再在 fire_Kinetis\ src\projects 下创建 fire_gpio_demo 文件夹，在 fire_gpio_demo 文件下继续创建 main.c 、isr.c 和 isr.h 文件，往 app 分组添加 main.c 、isr.c





然后在 main.c 里写个空的 main 函数：

```
18. void main()
19. {
20.
21. }
```

在 isr.h 文件里添加一下代码：

```
22. /******************** (C) COPYRIGHT 2011 野火嵌入式开发工作室 ********************
23.  * 文件名        : isr.h
24.  * 描述          : 重新宏定义中断号，重映射中断向量表里的中断函数地址，
25.  *                 使其指向我们所定义的中断服务函数。声明中断服务函数
26.  *                 警告：只能在"vectors.c"被包含，而且必须在"vectors.h"包含的后面！！！
27.  *
28.  * 实验平台      : 野火 kinetis 开发板
29.  * 库版本        :
30.  * 嵌入系统      :
31.  *
32.  * 作者          : 野火嵌入式开发工作室
33.  * 淘宝店        : http://firestm32.taobao.com
34.  * 技术支持论坛 : http://www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
35. *********************************************************************************/
36. #ifndef __ISR_H
37. #define __ISR_H 1
38.
39. /*                                重新定义中断向量表
40.  *   先取消默认的中断向量元素宏定义       #undef  VECTOR_xxx
41.  *   在重新定义到自己编写的中断函数       #define VECTOR_xxx    xxx_IRQHandler
```

```
42.  *  例如：
43.  *  #undef  VECTOR_003                    先取消映射到中断向量表里的中断函数地址宏定义
44.  *  #define VECTOR_003 HardFault_Handler   重新定义硬件上访中断服务函数
45.  */
46.
47. #endif  //__ISR_H
48.
49. /* "isr.h" 结尾 */
```

VECTOR_003 在 vectors.h 里已经定义了：

```
#define VECTOR_003      default_isr
```

默认中断向量表里大部分中断地址都是指向 default_isr ，所以我们要重新映射到自己的中断服务函数。

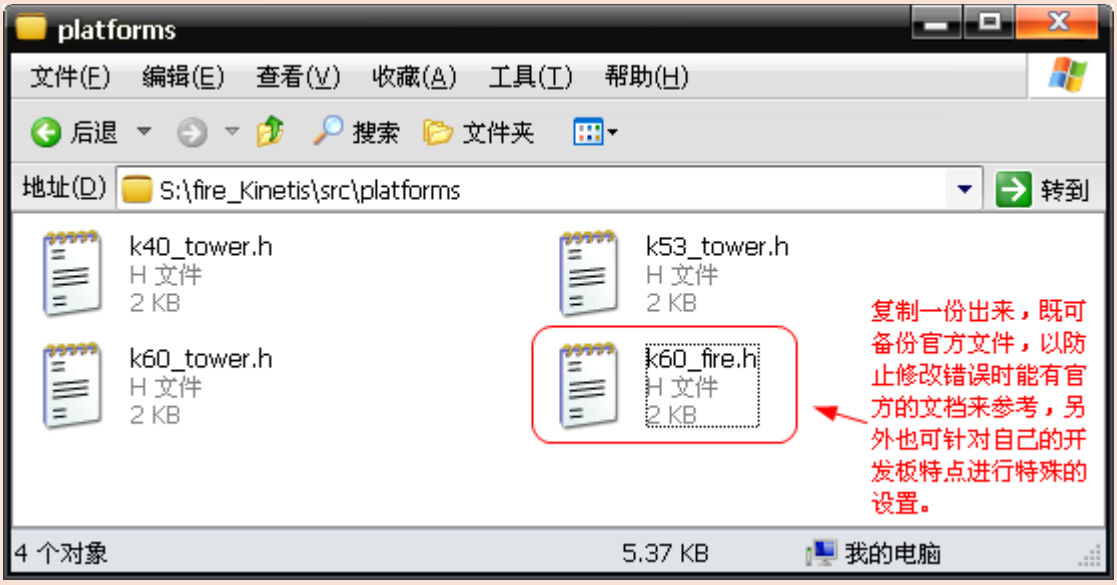
这个头文件是用来重映射中断向量表里的中断函数地址，使其指向我们所定义的中断服务函数。

注意：这个头文件只能在"vectors.c"中被包含，而且必须在"vectors.h"包含的后面被包含！！！即在"vectors.c"里，头文件的包含顺序为：

```
50. #include "vectors.h"   //两者顺序不能换，如果换了，就不能重映射到我们指定的中断函数地址
51. #include "isr.h"
```

关于中断函数的编程和 isr.h 的作用，可以看：中断函数的编写方法

最后，在 fire_Kinetis\src\platforms 文件夹下，复制一份 k60_tower.h 文件出来，命名为 k60_fire.h ：



在 fire_Kinetis\src\common\common.h 里找到下面的代码：

```
52. /*
```

```
53.  * Include the platform specific header file
54.  */
55. #if (defined(TWR_K40X256))
56.   #include "k40_tower.h"
57. #elif (defined(TWR_K60N512))
58.  #include "k60_tower.h"
59. #elif (defined(TWR_K53N512))
60.  #include "k53_tower.h"
61. #else
62.   #error "No valid platform defined"
63. #endif
```

注释掉，并插入#include "k60_fire.h"，即

```
64.
65. /*
66.  * Include the platform specific header file
67.  */
68. #if (defined(TWR_K40X256))
69.   #include "k40_tower.h"
70. #elif (defined(TWR_K60N512))
71. // #include "k60_tower.h"
72.   #include "k60_fire.h"
73. #elif (defined(TWR_K53N512))
74.  #include "k53_tower.h"
75. #else
76.   #error "No valid platform defined"
77. #endif
```

另外，DEBUG_PRINT 在 fire_Kinetis\src\common\common.h 和 fire_Kinetis\src\platforms\ k60_fire.h 里都有定义，应该注释掉 k60_fire.h 里的定义

```
78. /* Global defines to use for all boards */
79. //#define DEBUG_PRINT
```

在 fire_Kinetis\src\common\common.h 添加数据类型的 typedef：

```
1.  /*****************************设置数据类型*****************************/
2.  typedef     unsigned    char        u8;    //无符号型
3.  typedef     unsigned    short int   u16;
4.  typedef     unsigned    long  int   u32;
5.
6.  typedef             char        s8;    //有符号型
7.  typedef             short int   s16;
8.  typedef             long  int   s32;
```

添加 GPIO 驱动和点亮 LED

Kinetis 的编程步骤：开启时钟、复用管脚、设置频率、设置功能...

在 fire_Kinetis\src\drivers 文件夹下 建立 gpio 文件夹，再在 gpio 文件夹建立 gpio.c 和 gpio.h

把 gpio.c 添加进工程分组的 drivers 里，

在 gpio.c 里添加如下 gpio 的驱动代码：

```
/****************** (C) COPYRIGHT 2011 野火嵌入式开发工作室 ****************
 * 文件名        : gpio.c
 * 描述          : gpio 驱动函数
 *
 * 实验平台      : 野火 kinetis 开发板
 * 库版本        :
 * 嵌入系统      :
 *
 * 作者          : 野火嵌入式开发工作室
 * 淘宝店        : http://firestm32.taobao.com
 * 技术支持论坛 : http://www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
*********************************************************************/

#include "common.h"
#include "gpio.h"


volatile struct GPIO_MemMap *GPIOx[5]={PTA_BASE_PTR,PTB_BASE_PTR,PTC_BASE_PTR,PTD_BASE_PTR,PTE_BASE_PTR}; //定义五个指针数组保存 GPIOx 的地址
volatile struct PORT_MemMap *PORTX[5]={PORTA_BASE_PTR,PORTB_BASE_PTR,PORTC_BASE_PTR,PORTD_BASE_PTR,PORTE_BASE_PTR};

/*********************************************************************
*                                 野火嵌入式开发工作室
*
*   函数名称：gpio_init
*   功能说明：初始化 gpio
*   参数说明：PORTx       端口号（PORTA,PORTB,PORTC,PORTD,PORTE）
*             n           端口引脚
*             IO          引脚方向,0=输入,1=输出
*             data        输出初始状态,0=低电平,1=高电平  （对输入无效）
*   函数返回：无
*   修改时间：2012-1-15    已测试
*   备    注：
*********************************************************************/
void gpio_init (PORTx portx, u8 n,IO cfg,u8 data)
{
    ASSERT( (n < 32u)  && (data < 2u) );//使用断言检查输入、电平 是否为 1bit
```

```
37.
38.     //选择功能脚 PORTx_PCRx ，每个端口都有个寄存器 PORTx_PCRx
39.     PORT_PCR_REG(PORTX[portx],n)=(0|PORT_PCR_MUX(1));
40.
41.     //端口方向控制输入还是输出
42.     if(cfg == GPO)//output
43.     {
44.         GPIO_PDDR_REG(GPIOx[portx]) |= (1<<n);        //端口方向为输出
45.         if(data == 1)//output
46.         {
47.             GPIO_PDOR_REG(GPIOx[portx]) |= (1<<n); //对端口输出控制，输出为 1
48.         }
49.     else
50.         {
51.             GPIO_PDOR_REG(GPIOx[portx]) &= ~(1<<n);//对端口输出控制，输出为 0
52.         }
53.     }
54.     else
55.         GPIO_PDDR_REG(GPIOx[portx]) &= ~(1<<n);        //端口方向为输入
56. }
57.
58. /*********************************************************************
59. *                              野火嵌入式开发工作室
60. *
61. *   函数名称：gpio_set
62. *   功能说明：设置引脚状态
63. *   参数说明：PORTx        端口号（PORTA,PORTB,PORTC,PORTD,PORTE）
64. *             n            端口引脚
65. *             data         输出初始状态,0=低电平,1=高电平
66. *   函数返回：无
67. *   修改时间：2012-1-16    已测试
68. *   备    注：
69. *********************************************************************/
70. void gpio_set (PORTx portx, u8 n, u8 data)
71. {
72.     ASSERT( (n < 32u)  && (data < 2u) );//使用断言检查输入、电平 是否为 1bit
73.
74.     if(data == 1)
75.         GPIO_PDOR_REG(GPIOx[portx]) |= (1<<n);       //输出高电平
76.     else
77.         GPIO_PDOR_REG(GPIOx[portx]) &= ~(1<<n);      //出去低电平
78. }
```

在 gpio.h 里添加变量和函数的声明：

```
1.  #ifndef __GPIO_H__
2.  #define __GPIO_H__
3.
4.  typedef enum PORTx    //端口宏定义
5.  {
6.    PORTA,
7.    PORTB,
8.    PORTC,
9.    PORTD,
10.   PORTE
```

```
}PORTx;

typedef enum IO    //定义管脚方向
{
  _I=0u,          //定义管脚输入方向
  GPO=1u           //定义管脚输出方向
}IO;

#define HIGH  1u
#define LOW   0u

extern volatile  struct GPIO_MemMap *GPIOx[5];
extern volatile struct  PORT_MemMap *PORTX[5];

void    gpio_init   (PORTx, u8 n,IO,u8 data);     //初始化 gpio
void    gpio_set    (PORTx,  u8 n,      u8 data);    //设置引脚状态

#endif
```

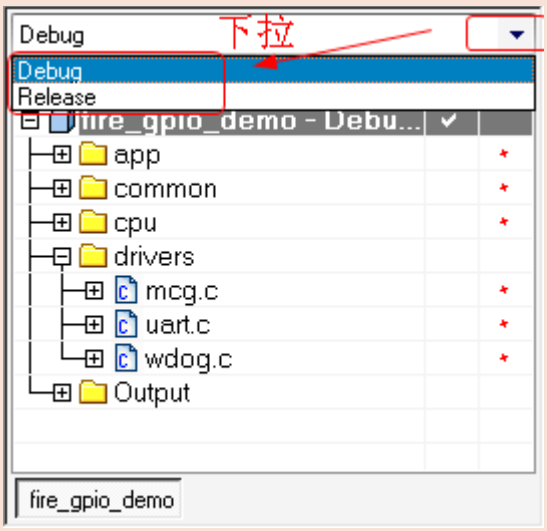
然后修改 main 函数

```
#include "gpio.h"
#include "lptmr.h"  //延时
/*********************************************************************
*                                  野火嵌入式开发工作室
*                                    GPIO 实验简单测试
*   实验说明：野火 GPIO 实验，利用 LED 来显示电平高低
*
*   实验操作：无
*
*   实验效果：LED0 每隔 500ms
*
*   实验目的：明白如何设置 IO 口电平
*
*   修改时间：2012-2-28      已测试
*
*   备     注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
*********************************************************************
/
void  main(void)
{
    gpio_init(PORTD,15,GPO,HIGH);//初始化 PTD15：输出高电平 ,即 初始化 LED0，灭
    while(1)
    {
        gpio_set(PORTD,15,LOW);    //设置 PTD15 输出 低电平 , 即 LED0 亮
        time delay ms(500);        //延时 500ms
        gpio_set(PORTD,15,HIGH);   //设置 PTD15 输出 高电平 , 即 LED0 灭
        time delay ms(500);        //延时 500ms
    }
}
```

O(∩_∩)o 哈哈，添加好文件了，现在需要配置工程了，不然编译出错。

IAR 工程选项设置

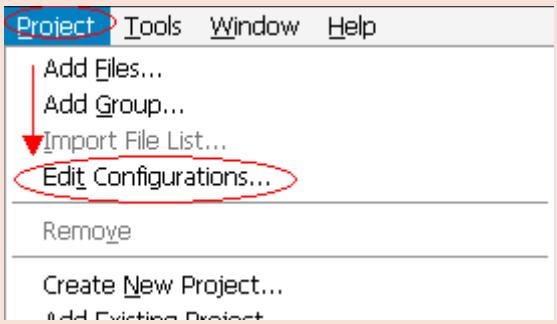
IAR 的工程选项 分成两个：Debug 模式 和 Release 模式，一个是调试模式，一个是发布模式。

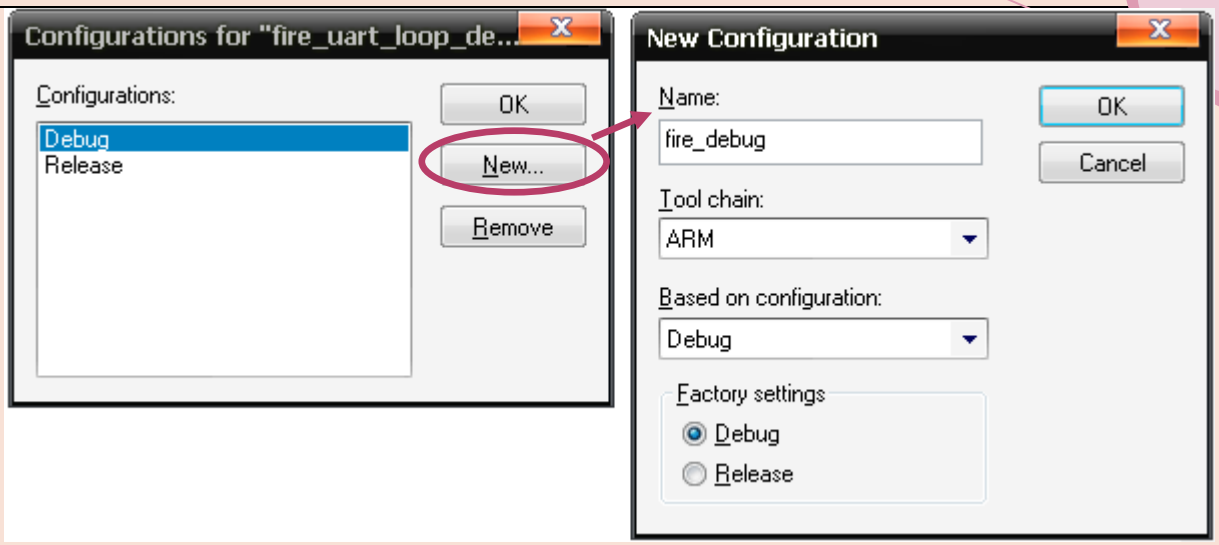


这样，在调试的时候，我们选择 Debug；发布产品或者比赛的时候，我们选择 Release ，切换起来非常方便。

网上共享的一些工程，你们有时会发现他们并没有 Debug 模式 和 Release 模式，取而代之的是 ROM 模式、RAM 模式、flash 模式等等，这些都是可以自行修改的：

Project —— Edit Configurations —— 在弹出来的界面里选择：new —— 然后命名和设置成自己所需要的。



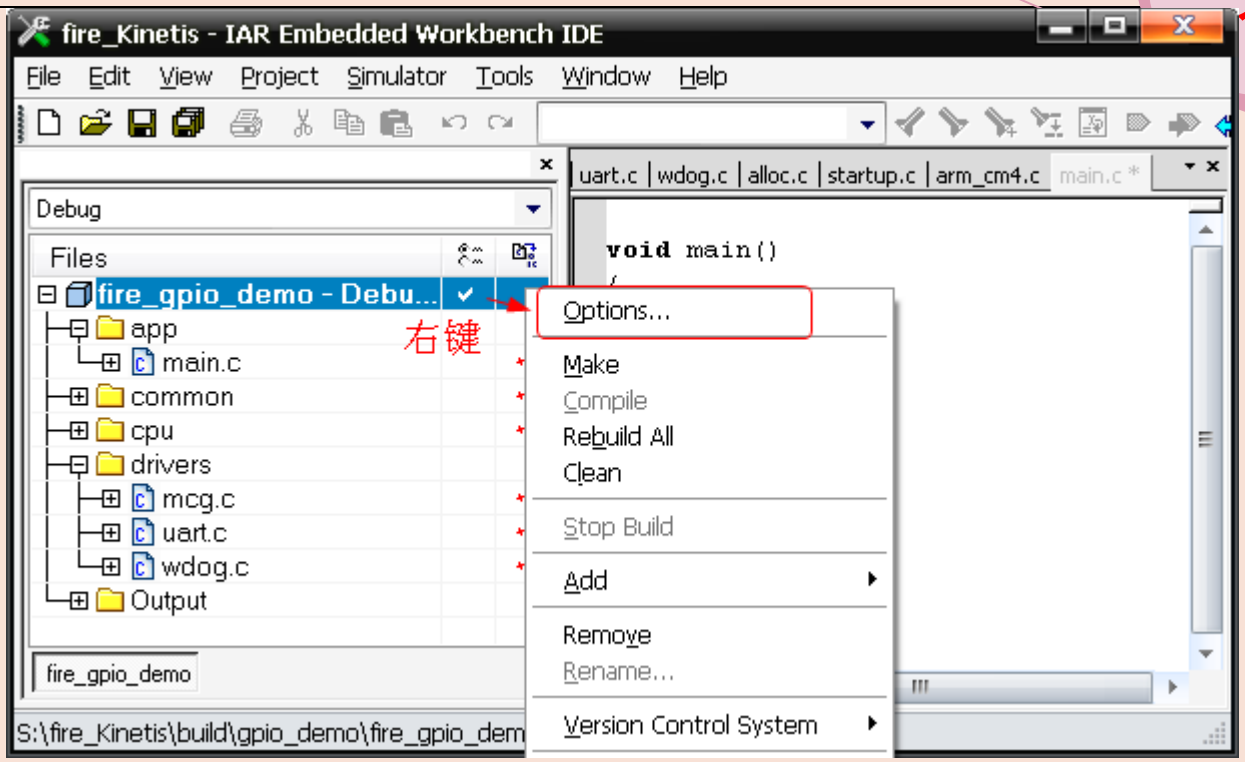


注意，从 Debug 模式切换到 Release 模式，会经常出现各种异常的编译错误或者运行异常，这是因为 Release 有更严格的检查，详细情况，可看：在 IAR 的 Workspace 窗口顶部的下拉菜单中有两个选项.pdf

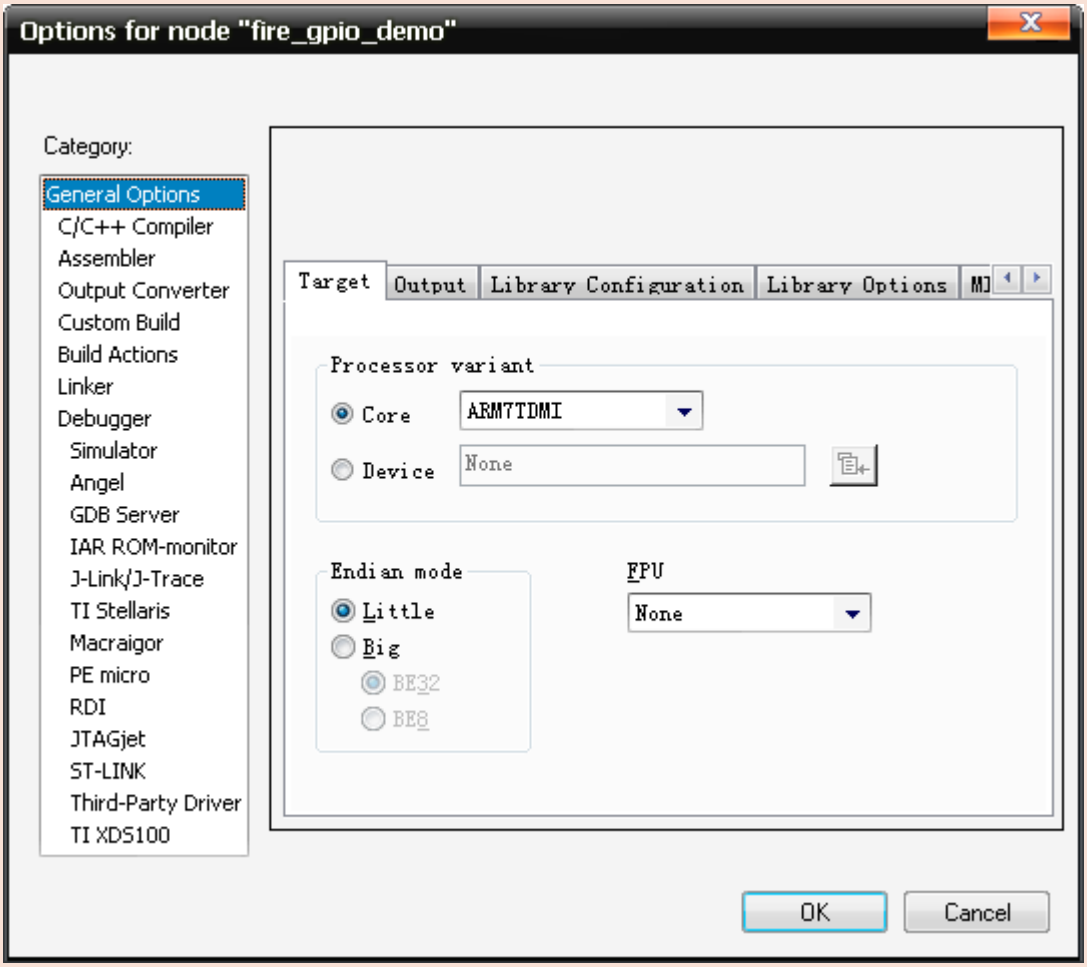
网上共享的工程模版，改名字后的，他们大部分都是基于 Debug 模式的，就是怕用户编译容易出错，但 Debug 模式的优化效果不如 Release 模式，如果我们的程序编程是没 Debug，就不怕把模式调到 Release 模式，只不过是在 Debug 模式下，会隐藏了一些错误而让我们觉得编译通过了。也有时，debug 模式里会添加一些延时，所以有时候在 Release 模式下延时。

在建工程的时候，我们需要根据两种不同的模式进行不同的配置。在这里，我们就用 Debug 模式来讲解：

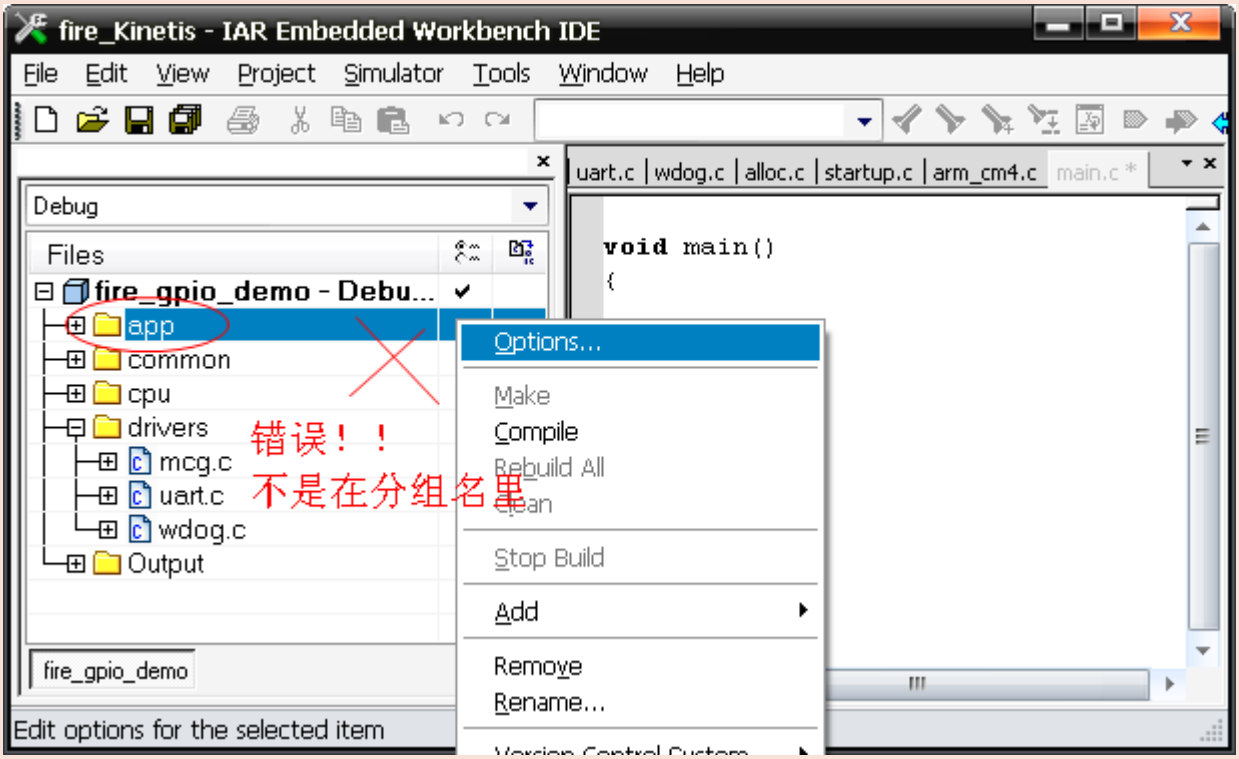
在 工作区 的 工程名上：右键——Options


fire_Kinetis - IAR Embedded Workbench IDE
File Edit View Project Simulator Tools Window Help
Debug
Files
fire_gpio_demo - Debu...
app
main.c
common
cpu
drivers
mcg.c
uart.c
wdog.c
Output
fire_gpio_demo
右键
uart.c | wdog.c | alloc.c | startup.c | arm_cm4.c | main.c *
void main()
Options...
Make
Compile
Rebuild All
Clean
Stop Build
Add
Remove
Rename...
Version Control System
S:\fire_Kinetis\build\gpio_demo\fire_gpio_dem

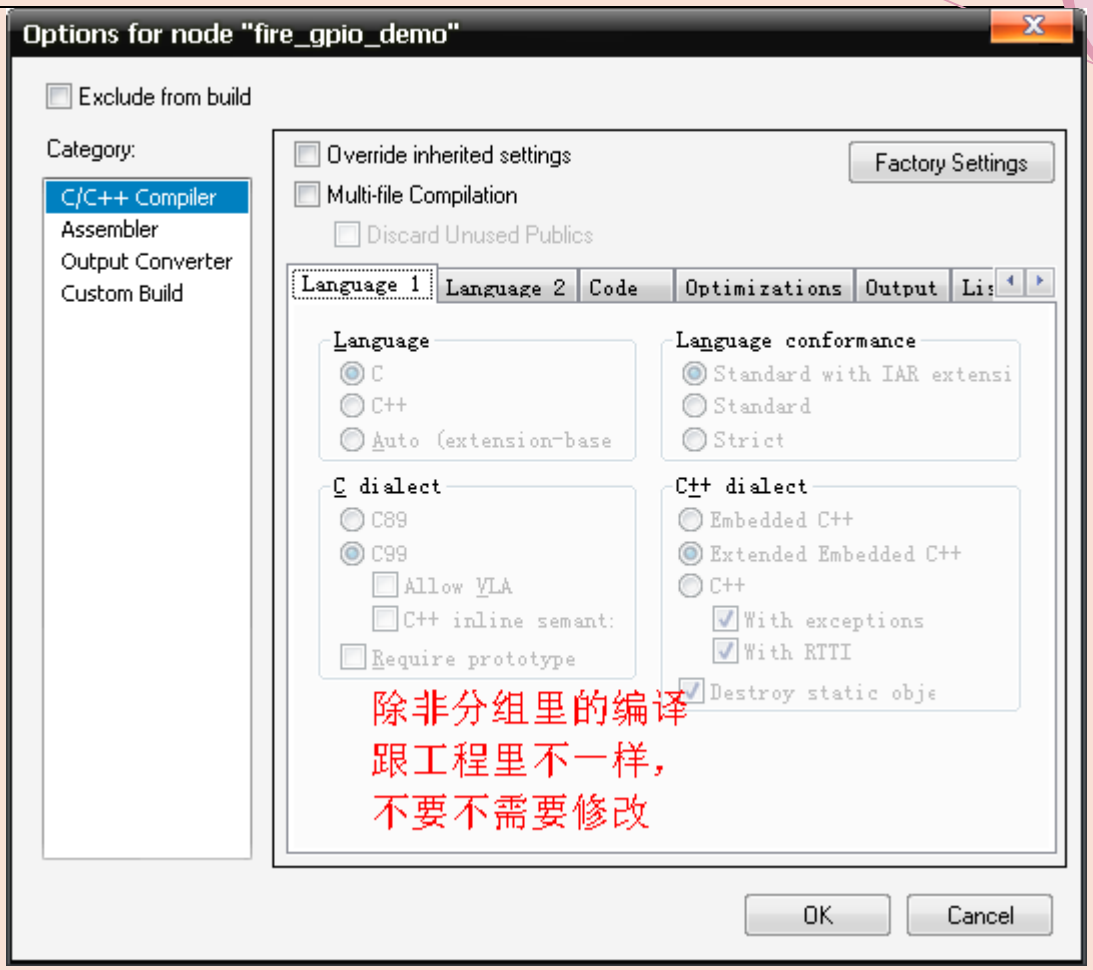
然后就会看到选项框：


Options for node "fire_gpio_demo"
Category:
General Options
C/C++ Compiler
Assembler
Output Converter
Custom Build
Build Actions
Linker
Debugger
Simulator
Angel
GDB Server
IAR ROM-monitor
J-Link/J-Trace
TI Stellaris
Macraigor
PE micro
RDI
JTAGjet
ST-LINK
Third-Party Driver
TI XDS100
Target
Output
Library Configuration
Library Options
Processor variant
Core
ARM7TDMI
Device
None
Endian mode
Little
Big
BE32
BE8
FPU
None
OK
Cancel

记住，是在工程名里右键，不要在分组名里右键哦，不然弹出的对话框不是设置工程的！！！

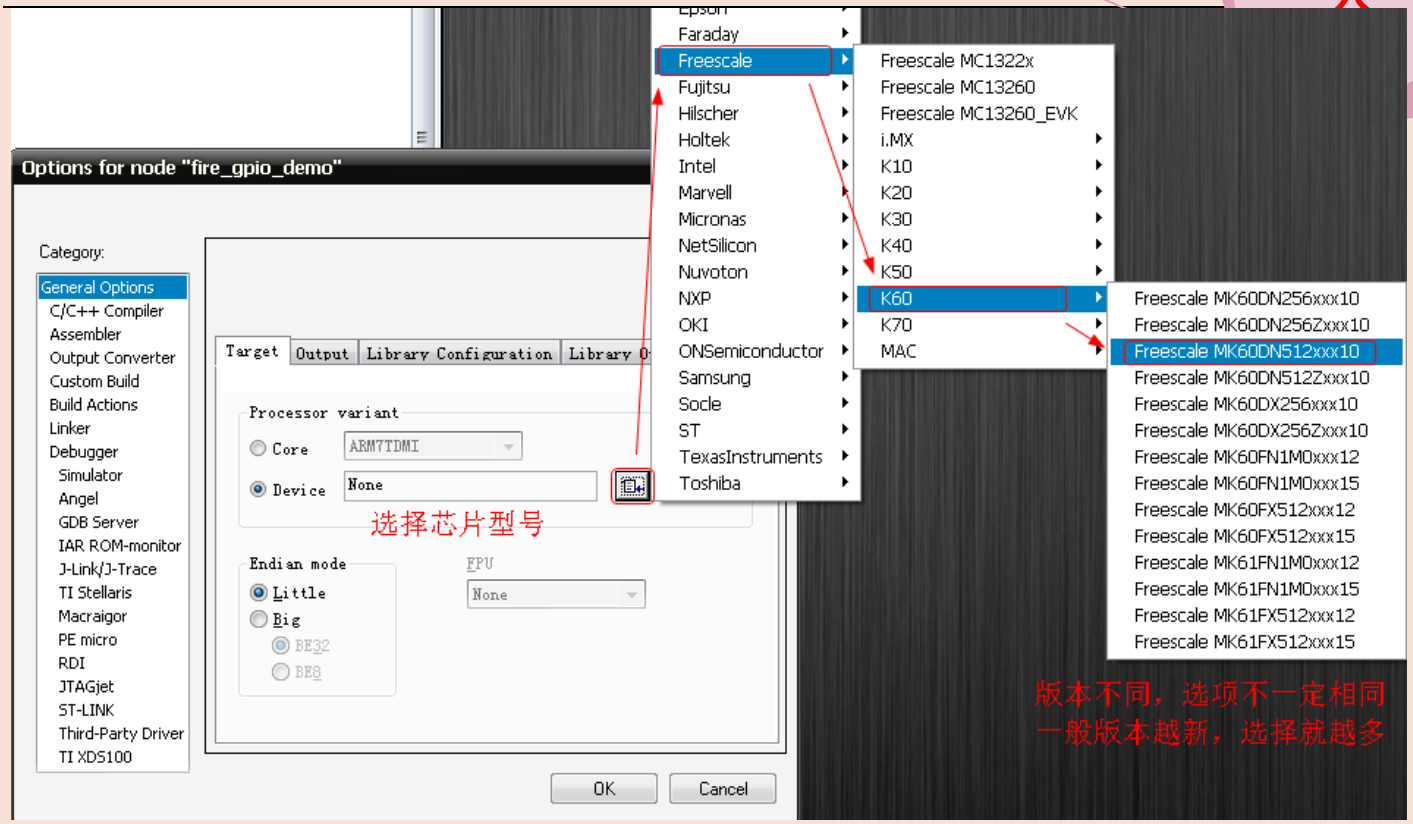


如果选择了分组里的选项，则显示：



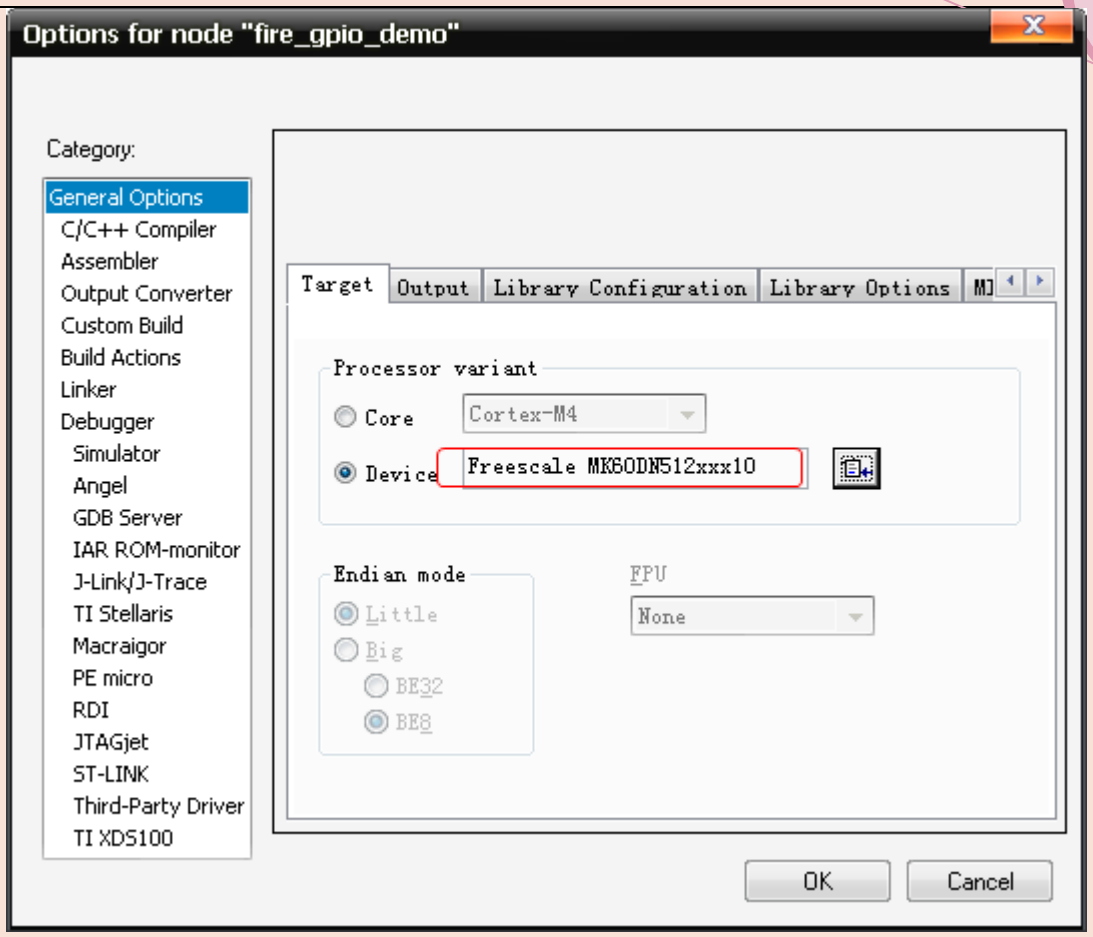
好了，现在言归正传，开始配置工程选项了...

General Options——Target　设置芯片型号

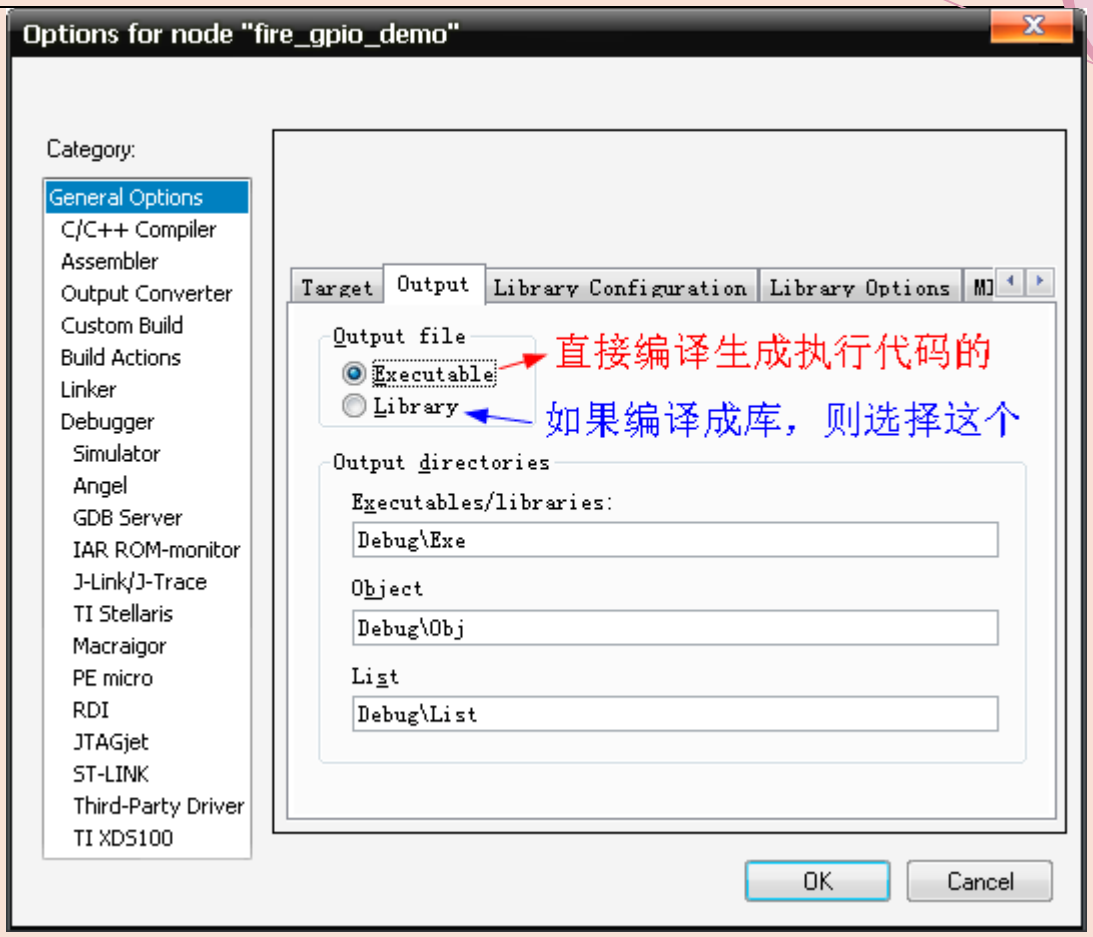


野火 Kinetis 开发板选择的默认芯片为 K60N512LQV100

在这里，我们选择 MK60DN512xxx10



General Options——output　设置输出信息

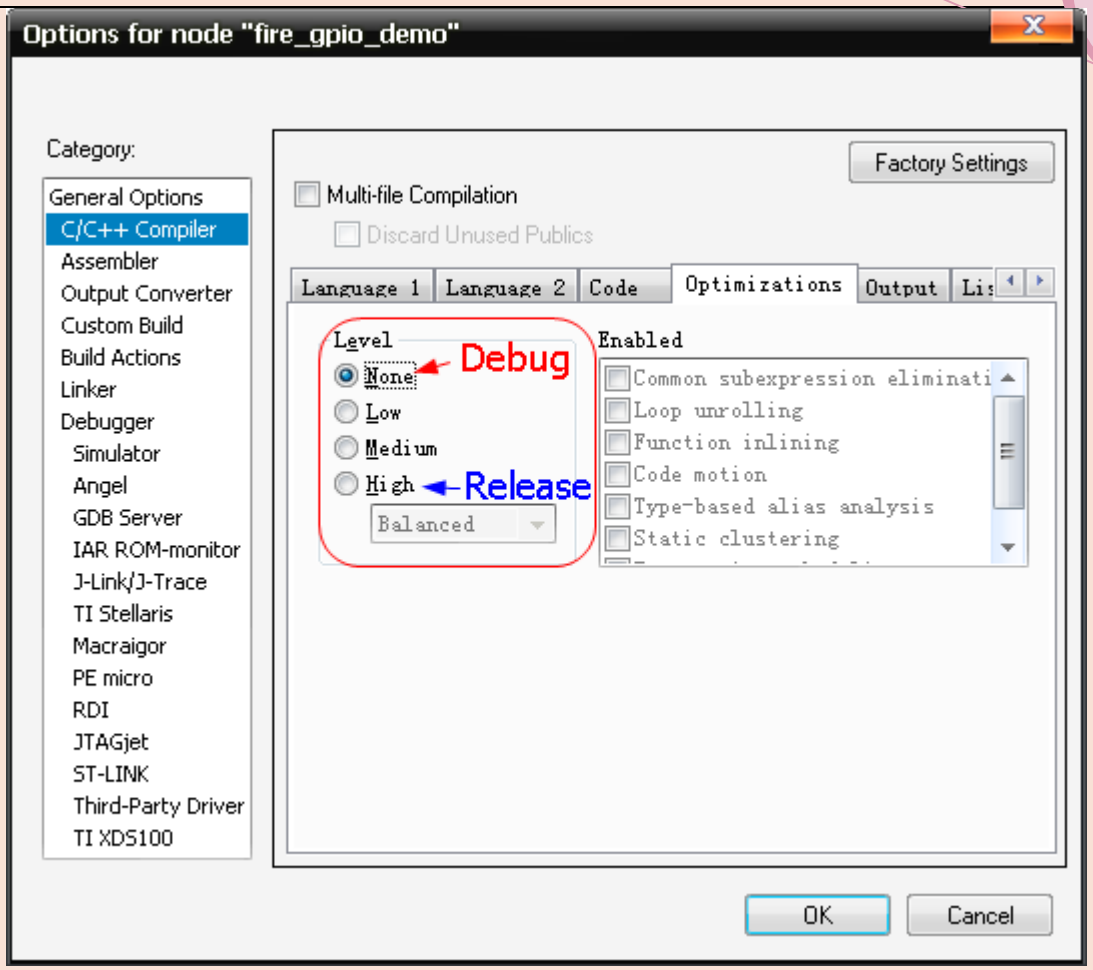


我们编译工程下载到单片机上的，当然选择 Executable

C/C++ Compiler —— Optimizations 优化等级

在 Debug 模式里，这里设为不优化，便于调试方便。事实上，在调试的时候，如果进行了优化，有些变量就不能显示出来，就不便于调试了。

在 Release 模式里，可以选择最大优化，但在发布前，需要对优化后的效果进行验证。因为优化后可能会出现各种异常的错误。



C/C++ Compiler —— Preprocessor 预处理器

在 Additional include directories: (one per line) 文本编辑框里填入：

```
80. $PROJ_DIR$\..\..\src\common
81. $PROJ_DIR$\..\..\src\cpu
82. $PROJ_DIR$\..\..\src\cpu\headers
83. $PROJ_DIR$\..\..\src\platforms
84. $PROJ_DIR$\..\..\src\drivers\mcg
85. $PROJ_DIR$\..\..\src\drivers\uart
86. $PROJ_DIR$\..\..\src\drivers\wdog
87. $PROJ_DIR$\..\..\src\drivers\lptmr
88. $PROJ_DIR$\..\..\src\projects\fire_gpio_demo
89. $PROJ_DIR$\..\..\build
```

在这里填入 头文件 所在的文件夹，预处理器编译的时候，就会搜索这些文件夹，从而找到头文件。

$PROJ_DIR$ 表示 IAR 工程所在的目录

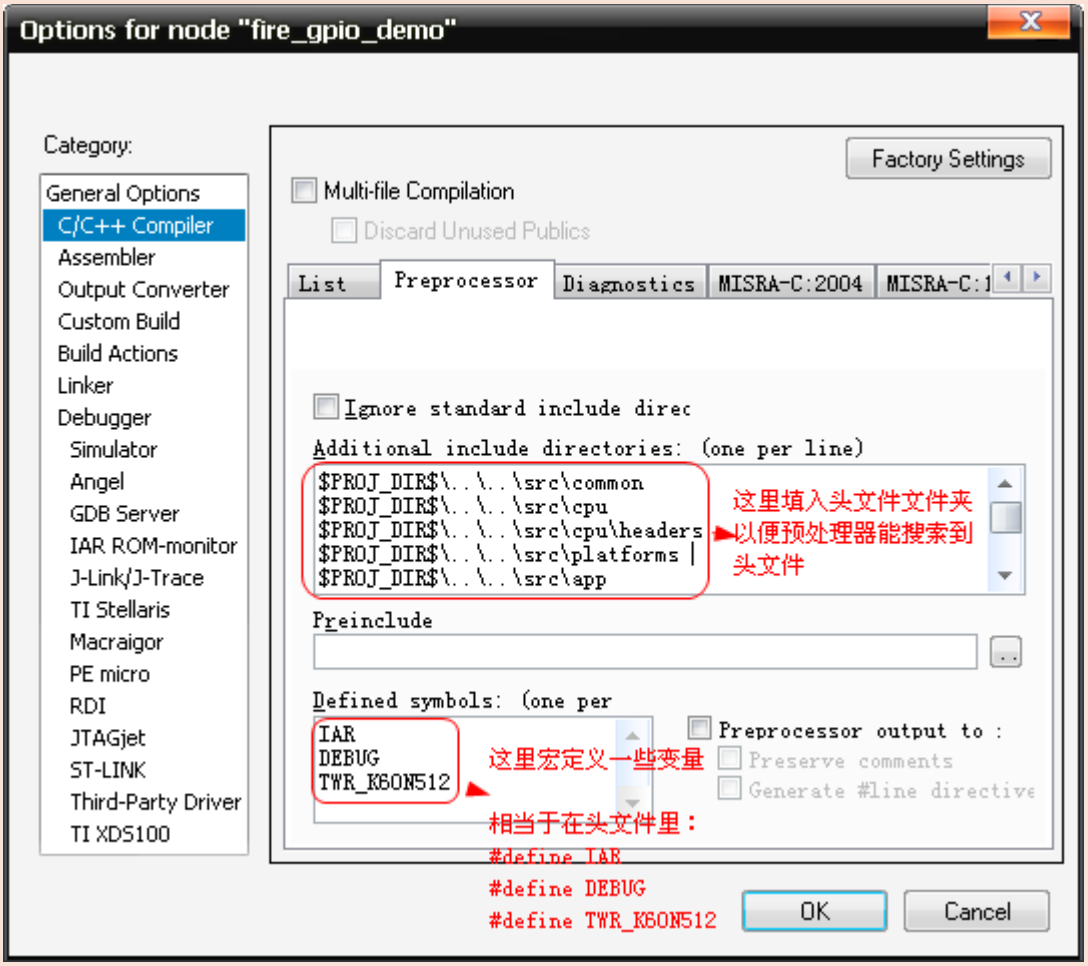
..\ 表示上一层目录

在 Defined symbols: (one per line) 文本编辑框里填入：

```
90. IAR
91. DEBUG
92. TWR_K60N512
```

这里是用来宏定义变量，对整个工程有效，相当于在头文件里：

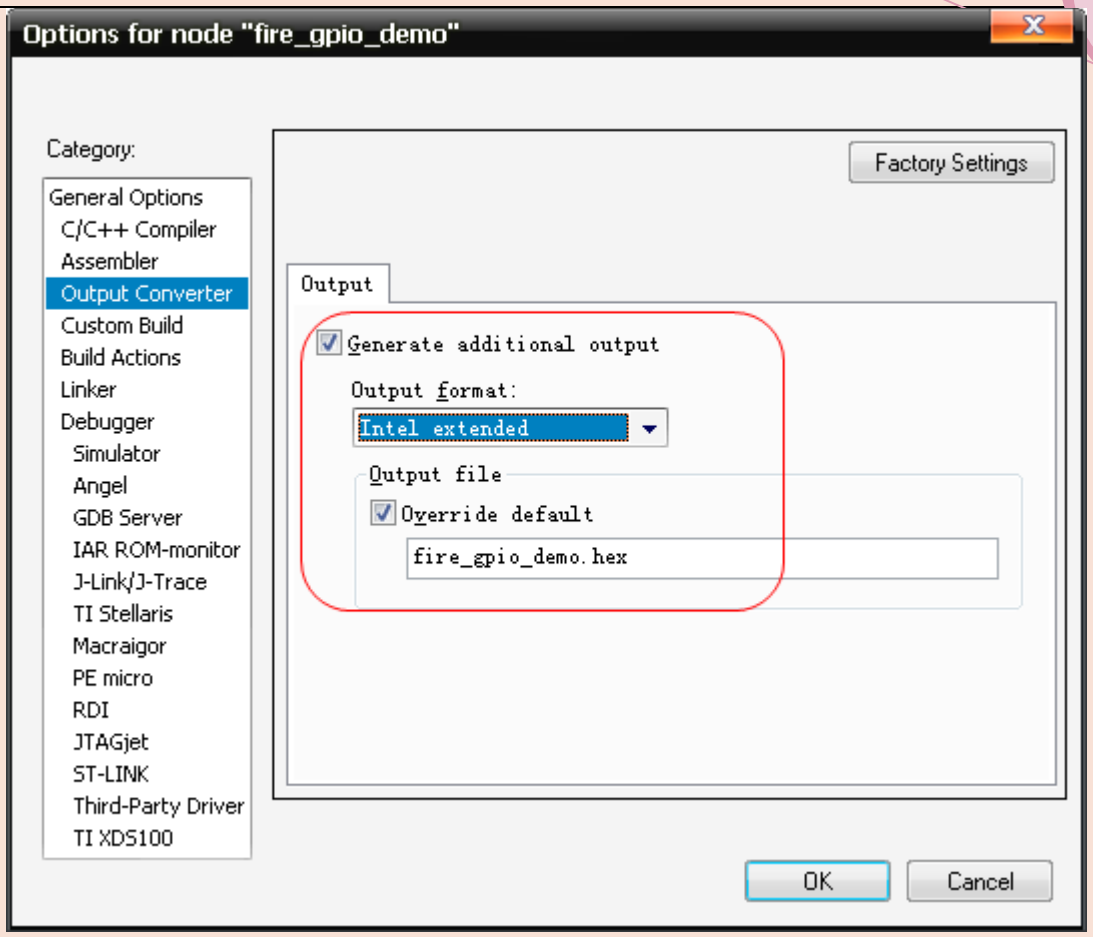
```
93. #define  IAR
94. #define  DEBUG
95. #define  TWR_K60N512
```



注意：如果在 Defined symbols: (one per line) 这里填入了 DEBUG，那就要注释掉 fire_Kinetis\src\common\common.h 里面 DEBUG 的宏定义。

Output Converter —— Output 输出格式转换

这里可以设置编译代码后，把代码转化成其他格式，我们设置转换为 hex 格式



Linker —— Config　链接器配置

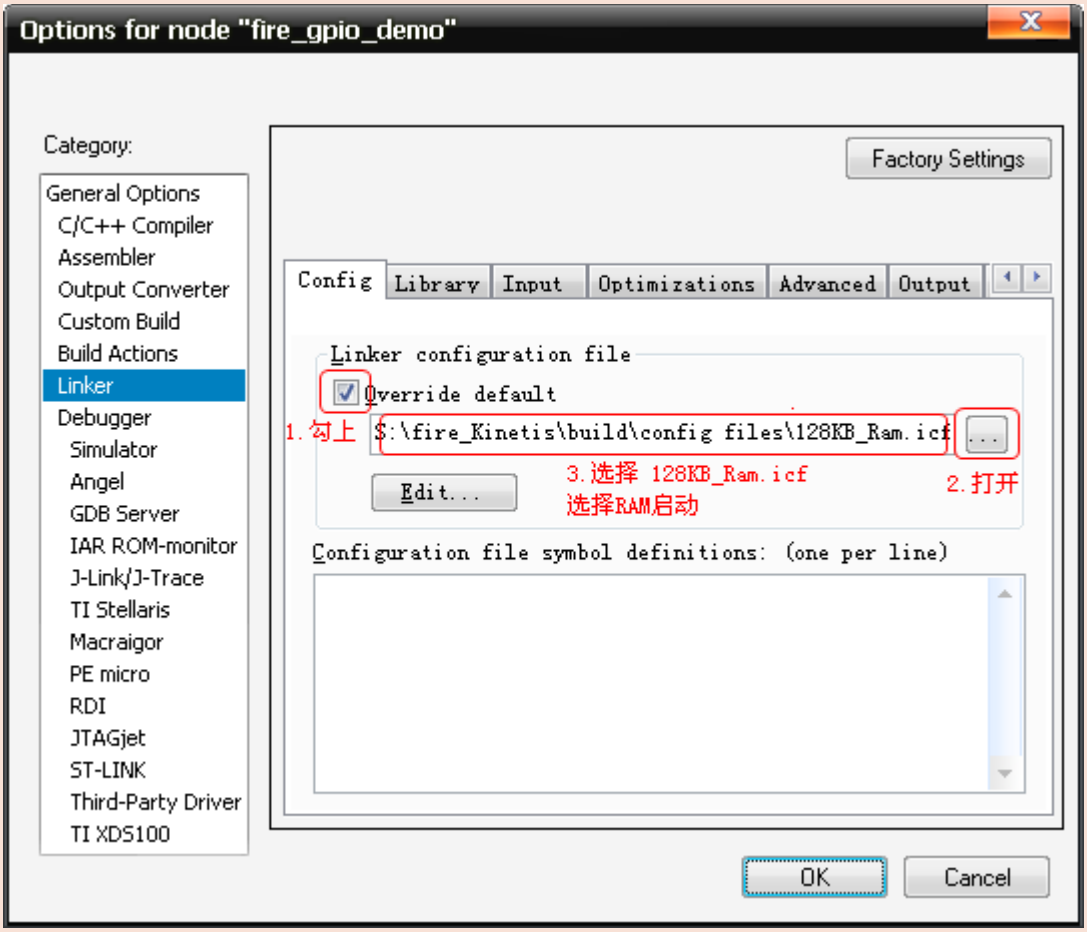
在 Linker configuration file 里勾中 Override default ，重新打开 Linker 配置文件，在 fire_Kinetis\build\config files 文件里就有多个 Linker 配置文件。

Linker 配置文件，里面设置了 Kinetis 芯片是从 ROM 启动还是从 RAM 启动，堆栈的大小等配置。

野火 Kinetis 开发板选择的默认芯片为 K60N512LQV100 ，512KB Flash 和 128KB SRAM 。

在 Debug 模式，如果你想从 RAM 启动，则选择 128KB_Ram.icf ；如果先从 ROM 启动，可以选择 512KB_Pflash.icf 或者 256KB_Pflash_256KB_Dflash.icf ，两者的区别就在于，后者把 512KB Flash 分一半出来，作为保存其他数据用，而不是全部用来保存程序代码。

在 Release 模式，肯定是从 ROM 启动，即可选择 512KB_Pflash.icf 或者 256KB_Pflash_256KB_Dflash.icf 。而且 Release 模式因为没有调试信息，所以是不能使用 jlink 等调试器来 Debug 的。



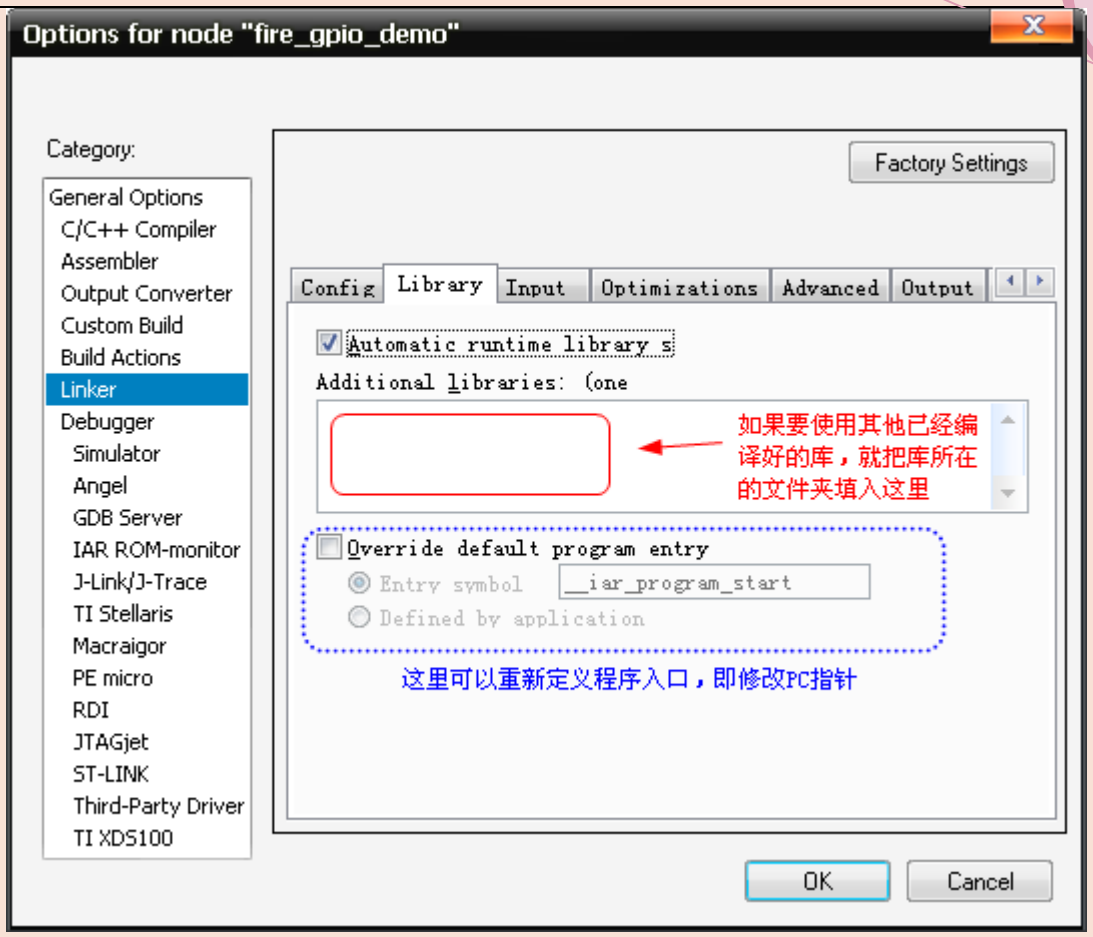
选择的时候，编译器会改成绝对地址的，但为了工程能拿到其他电脑上就能直接运行，建议还是修改成相对地址：

$PROJ_DIR$　表示 IAR 工程所在的目录

..\　　表示上一层目录

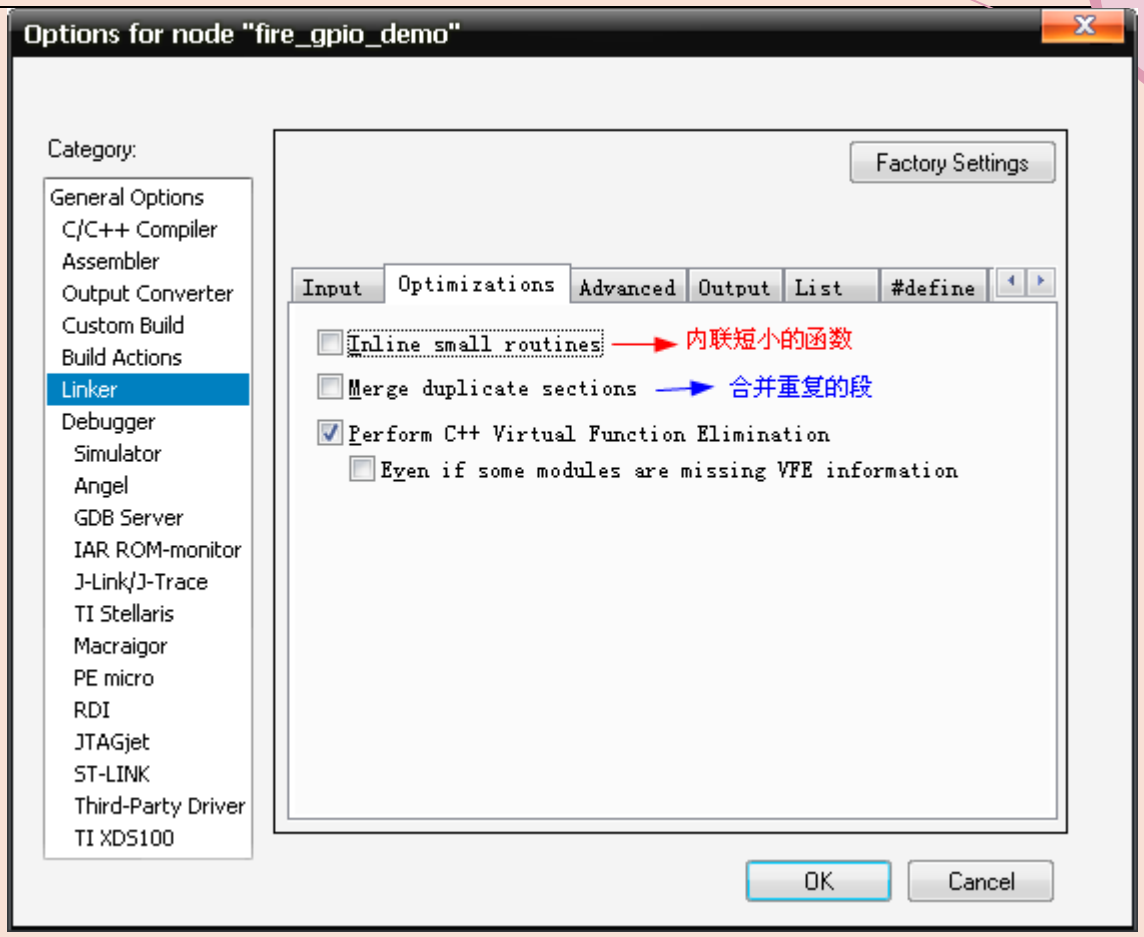
Linker —— Automatic runtime library s　自动运行库

跟预处理时设置头文件所在的文件夹一样，这里是设置库所在的文件夹。



如果用到其他已经编译好的库，那就把库所在的文件夹添加进去。

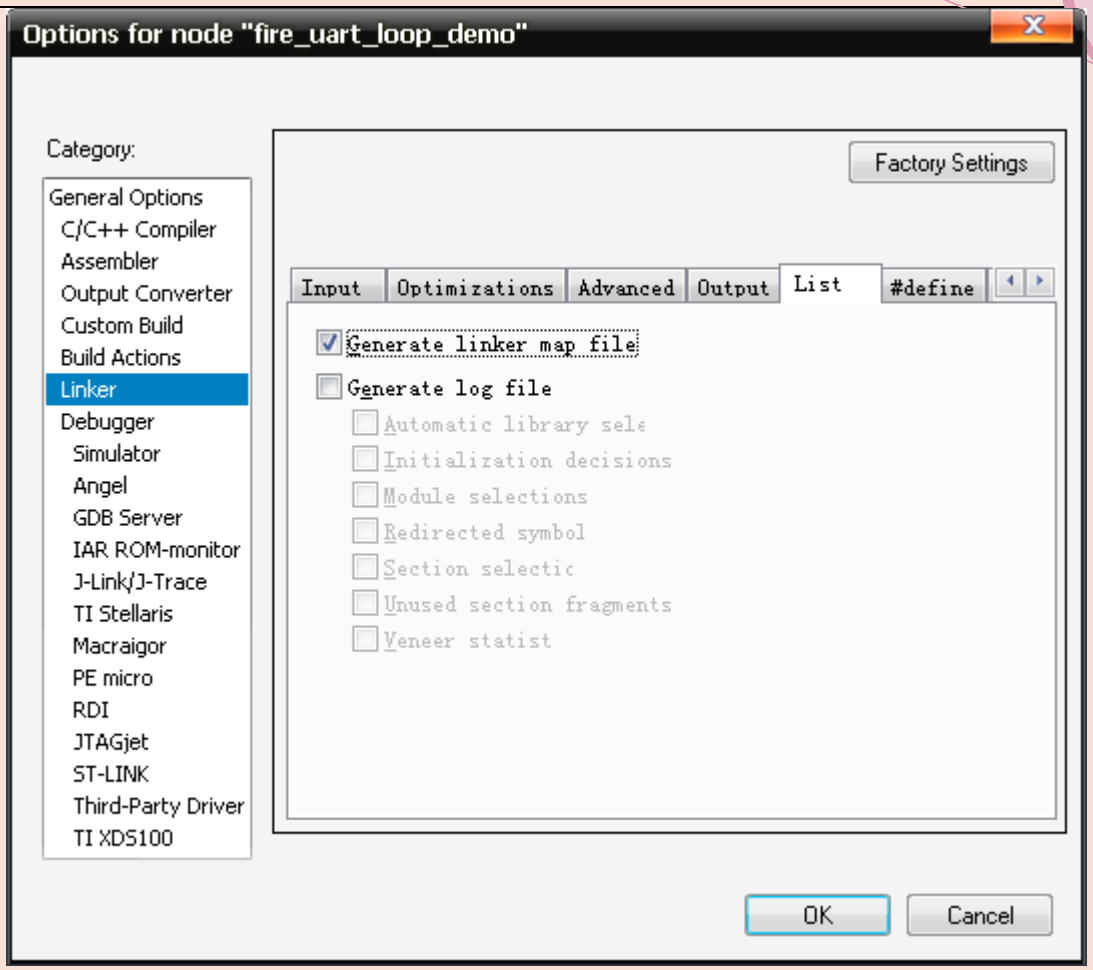
Linker —— Optimizations 优化



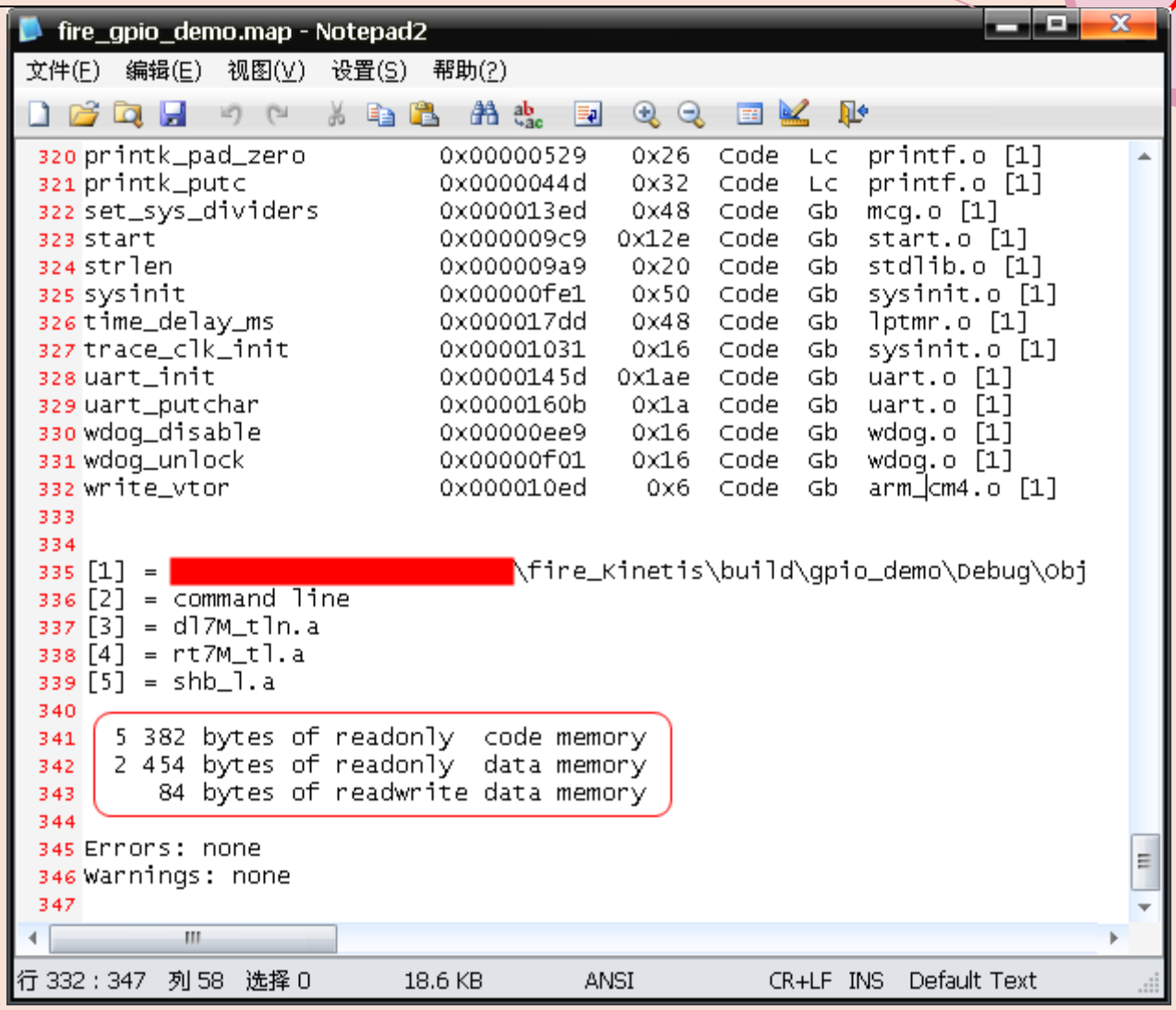
在这里，我就不选择优化了。

Linker —— List　列表

可以生成 关于内存分布、编译后生成文件的大小等各种信息的文件。



勾上 Generate linker map file ，编译后，以 Debug 模式为例，在工程的文件夹下生成：fire_Kinetis\build\gpio_demo\Debug\List\ fire_gpio_demo.map 文件，打开后：

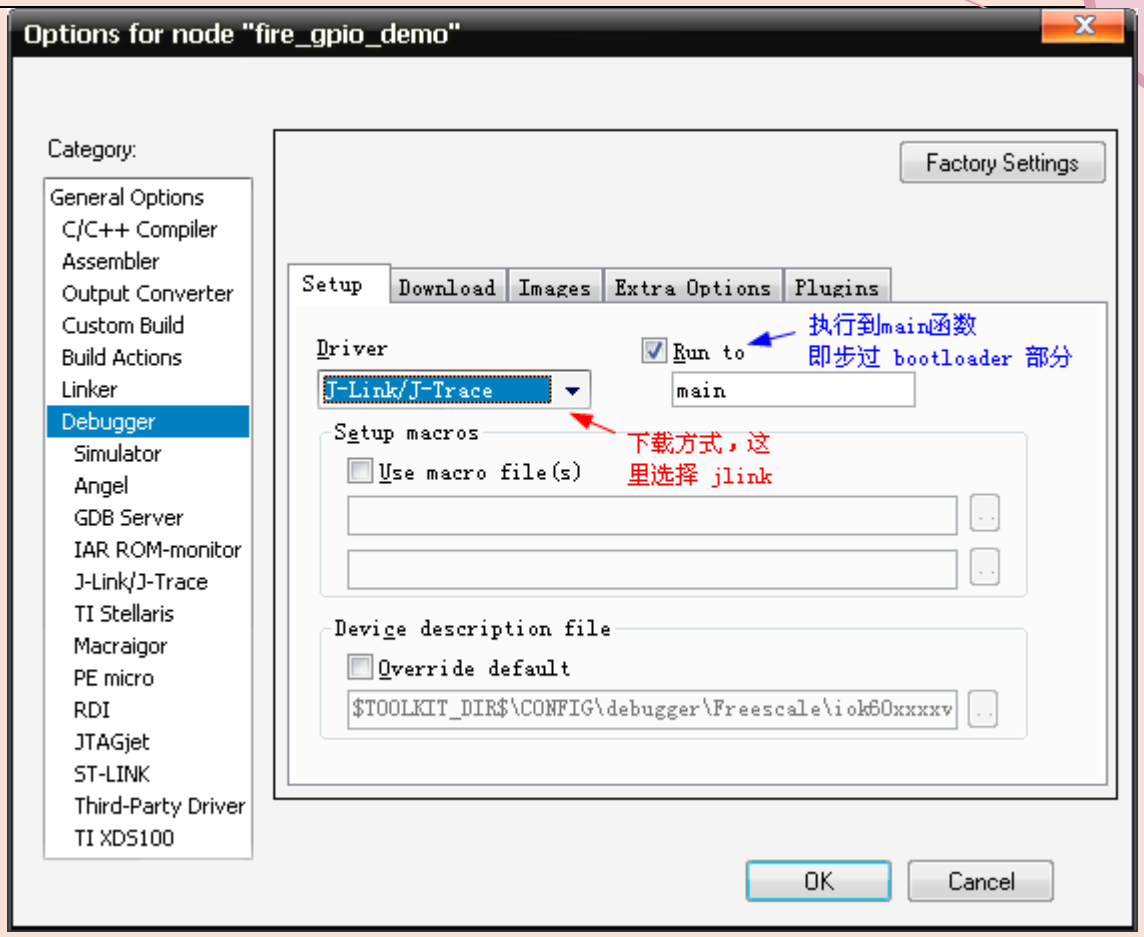


里面有很多详细的信息，方便我们了解工程编译后内存的使用情况等。

Debugger —— Setup　仿真器设置

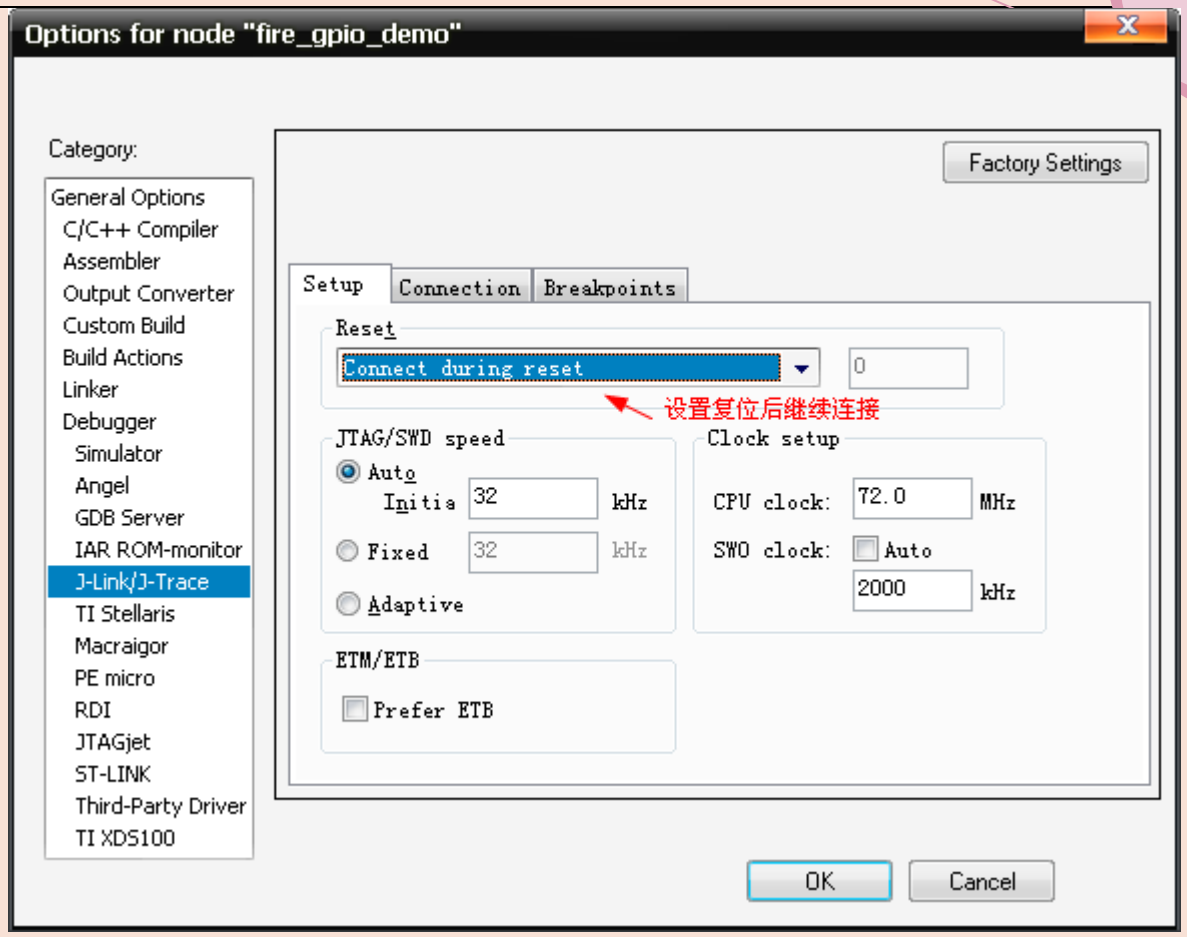
在 Driver 里有多种可供选择的调试方式。野火 Kinetis 开发板配置了 jlink 下载口可供调试，因此可以在这里选择 J-Link/J-Trace；如果想软件仿真，那可以选择 Simulator 。

即硬件调试，选择 J-Link/J-Trace；软件仿真，选择 Simulator 。

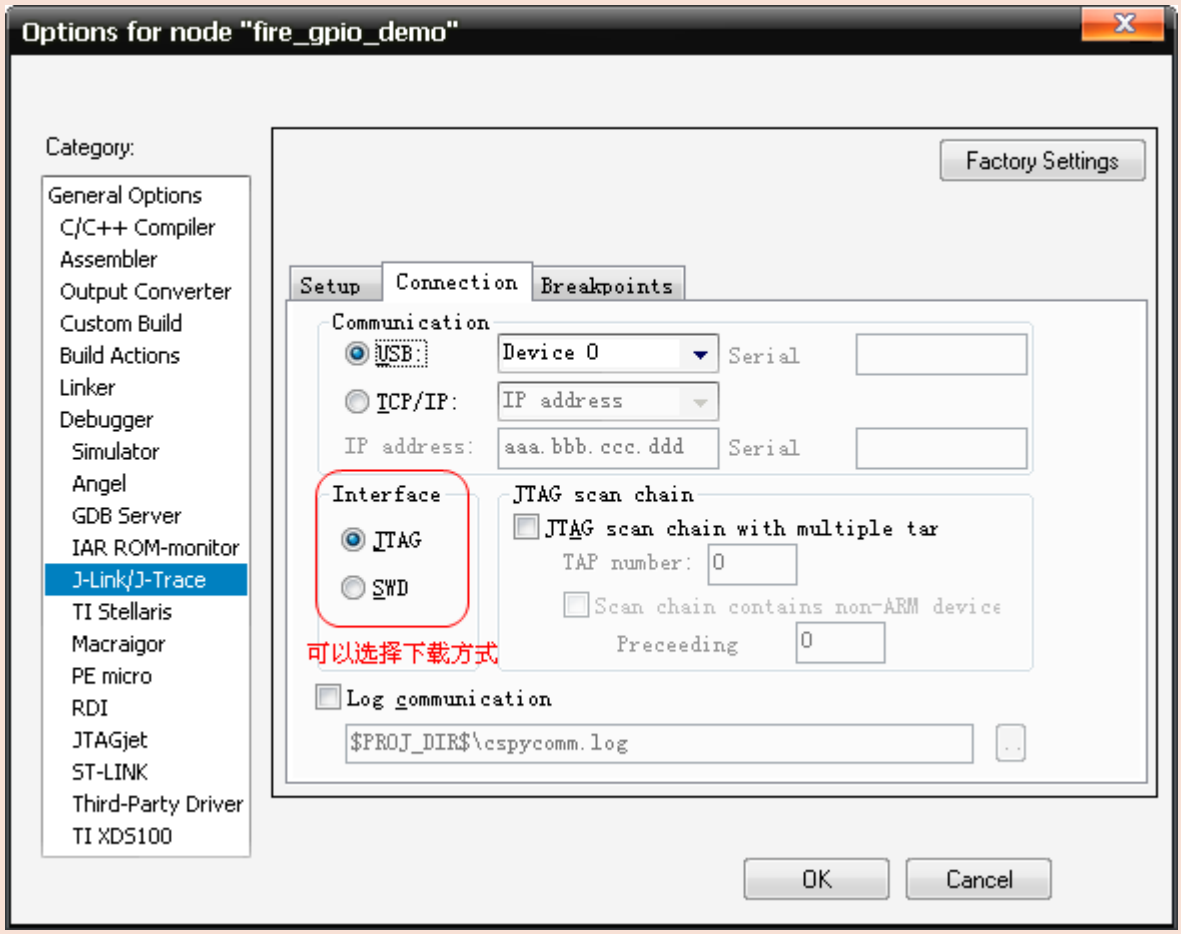


如果是在 Release 模式，那就不要选择软件仿真 Simulator。

J-Link/J-Trace —— Setup　jlink 下载设置



J-Link/J-Trace —— Connection　Jlink 连接设置



这里可以选择 Jlink 的下载方式：JTAG 和 SWD

IAR 重要的设置就介绍得差不多了，相信各位会配置了吧？o(∩_∩)o 哈哈

配置好后，编译一下，就发现编译通过，但出现两个相同的警告：

Warning[Pa082]: undefined behavior: the order of volatile accesses is undefined in this statement

提示警告的代码为：

```
96. printf("Flash parameter version %d.%d.%d.%d\n",
97.                                FTFL_FCCOB4,FTFL_FCCOB5,FTFL_FCCOB6,FTFL_FCCOB7);
```

原因很简单，FTFL_FCCOB4,FTFL_FCCOB5,FTFL_FCCOB6,FTFL_FCCOB7 这些参量都定义为 volatile ，这样调用，读取这些变量的值时不能同时读取，真正的是一个个先后读取，有可能数据不一致。

举个例子

```
98. int square(volatile int *ptr)
99. {
100.        return *ptr * *ptr;
101.    }
```

编译器将产生类似下面的代码：

```
102.    int square(volatile int *ptr)
103.    {
104.        int a,b;
105.        a = *ptr;
106.        b = *ptr;
107.        return a * b;
108.    }
```

两次读取变量，可能出现两次读取结果不一样的问题

正确的代码应该是：

```
109.    long square(volatile int *ptr)
110.    {
111.        int a;
112.        a = *ptr;
113.        return a * a;
114.    }
```

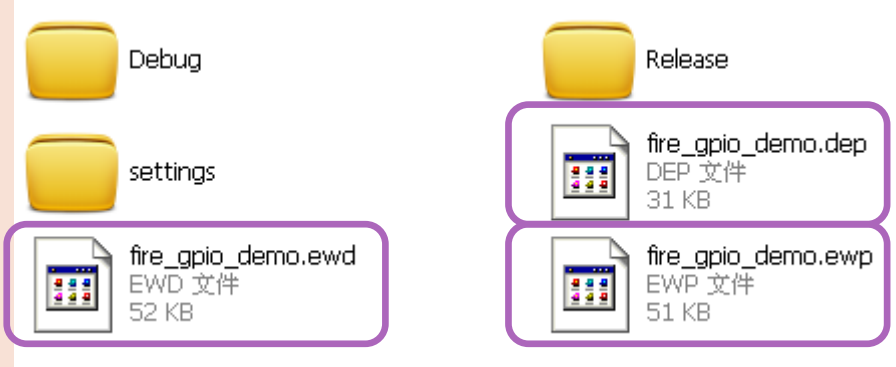
这样读取一次，就能保证相同了。

那现在，明白这个警告的原因了吧？在这个函数里，我们可以不用理它，当然，也可以简单修改即可解决，请参考上面提供的例子自行修改。

快速建 IAR 工程

IAR 的工程，只需要建立一次之后，保存好设置，下次建立工程，仅仅需要修改名字就可以了。

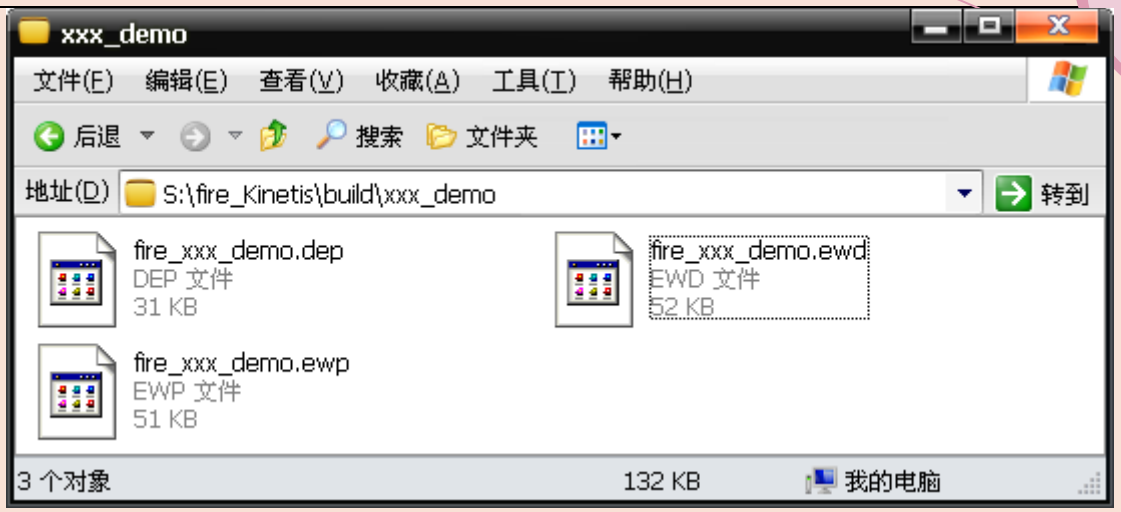
在 fire_Kinetis\build 文件下，找到我们已经建好的 gpio_demo 工程文件夹，里面有三个工程设置文件：



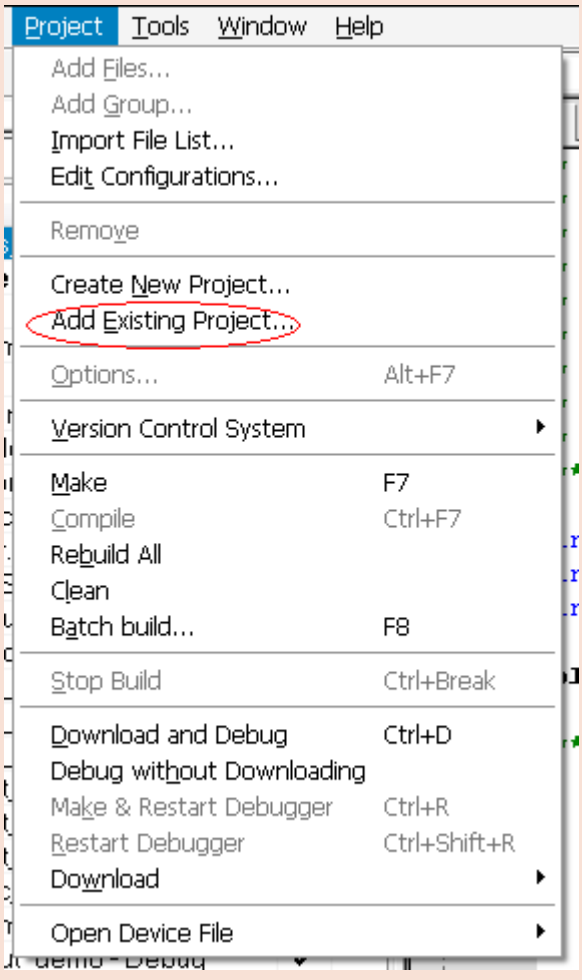
把 gpio_demo 工程文件夹复制一份在原来的文件夹下，改名为自己喜欢的工程名字，例如 xxx_demo：

打开 xxx_demo，删掉里面的文件夹，把其他文件改名为自己喜欢的名字，例如 fire_xxx_demo ，记得保留后缀。

修改后：



然后在 IAR 的工作区里添加工程就行（当然，你也可以新建工作区）。



这样就可以不需要重新设置就建立好新的工程

IAR 使用教程

其实在打开 IAR 软件的时候，已经弹出了 IAR 使用教程界面给你：

飞思卡尔 Kinetis 系列的例程：

IAR Information Center for ARM

IAR Information Center for ARM | EXAMPLES

Examples for Freescale Kinetis evaluation boards

Info	Open project	Name	Description
		ADC	This project is a simple Analog to Digital Converter (ADC) example
		CAN Loopback node	This project is a simple can_loopback_node example.
		CRC	This project is a simple crc example.
		DAC	This project is a simple DAC example.
		Flexbus	This project is a simple example project that uses the FlexBus external bus interface.
		Flex memory	This project is a FlexMemory example.
		Getting Started	Basic use of GPIO and System timer
		GPIO	This project is a simple General Purpose Input/Output (GPIO) example.
		Hello world	This project is a simple hello world example.

提供的都是英文版的，提高你们英语水平的时候有到了……

野火就来个中文版的简单介绍吧，当然远不然官方的介绍那么详细。

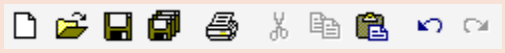
工具栏功能介绍

IAR 的工具栏包含：

新建保存复制功能、查找替换功能、跳转功能、编译下载功能。

与其他编译器项目，IAR 的工具栏功能简单，容易上手

- 新建保存复制功能为：



 ：新建空白文件

 ：复制

 ：打开文件

 ：粘贴

 ：保存文件

 ：打印

 ：保存所有文件

 ：撤销

 ：剪切

 ：恢复

- 查找替换功能为： 

 ：查找内容，这里为查找 fire 文字

 ：查找上一个

 ：查找下一个

 ：查找

 ：替换

- 跳转功能为： 

 ：光标跳到指定行列

 ：设置书签

 ：调到下个书签

 ：跳回上个页面

 ：跳到下个页面

- 编译下载功能为： 

 ：编译当前文件

 ：设置断点

 ：编译整个工程

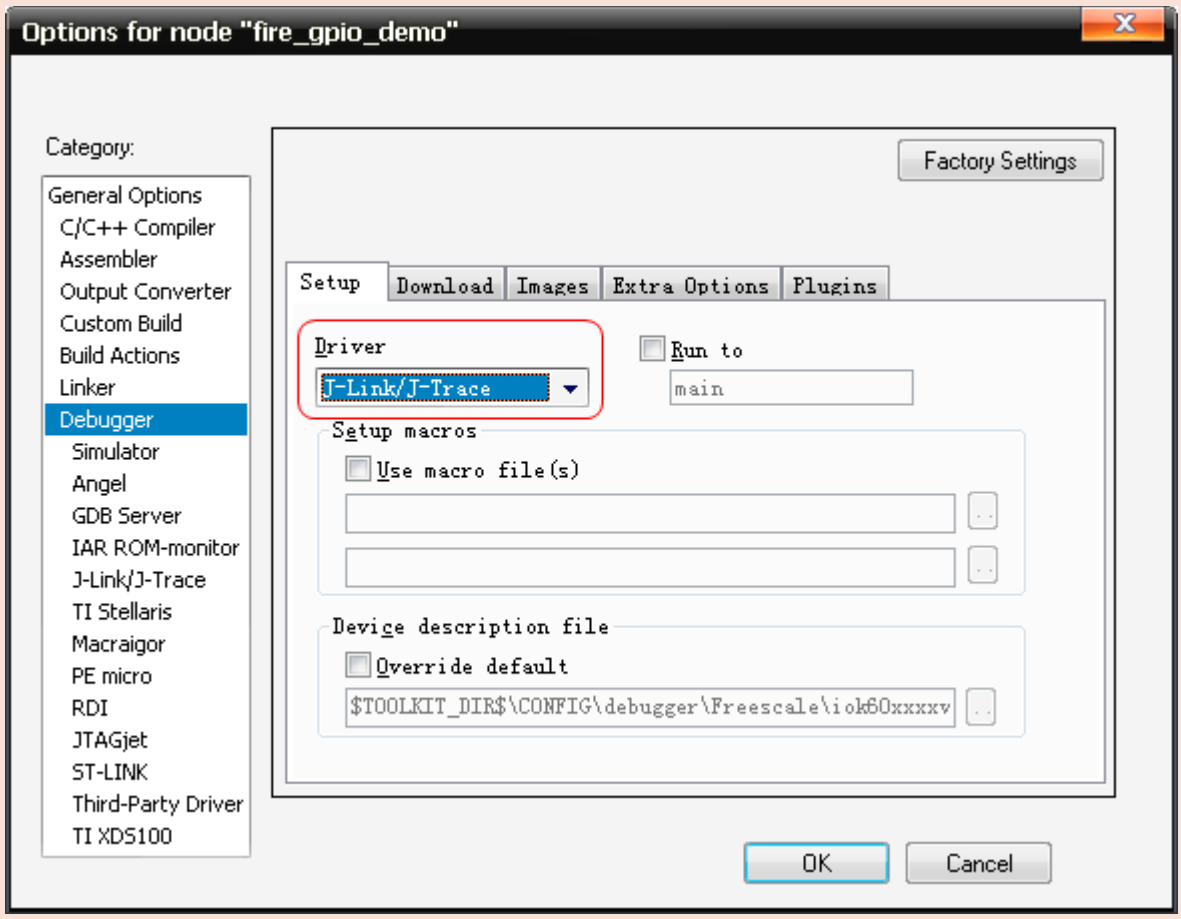
 ：下载并调试

 ：取消编译

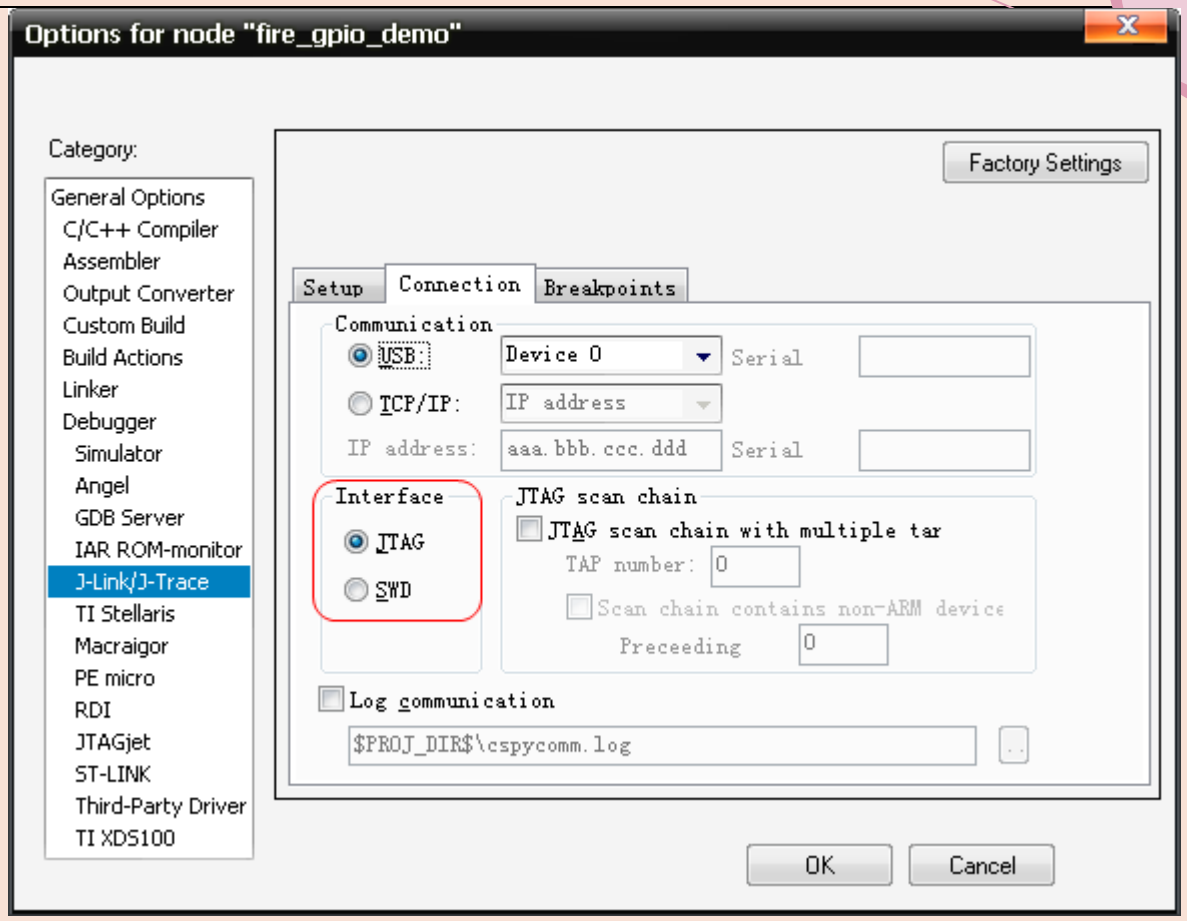
 ：不下载，直接调试

通过 jlink 下载并调试

设置工程选项下载方式为 ：J-Link/J-Trace



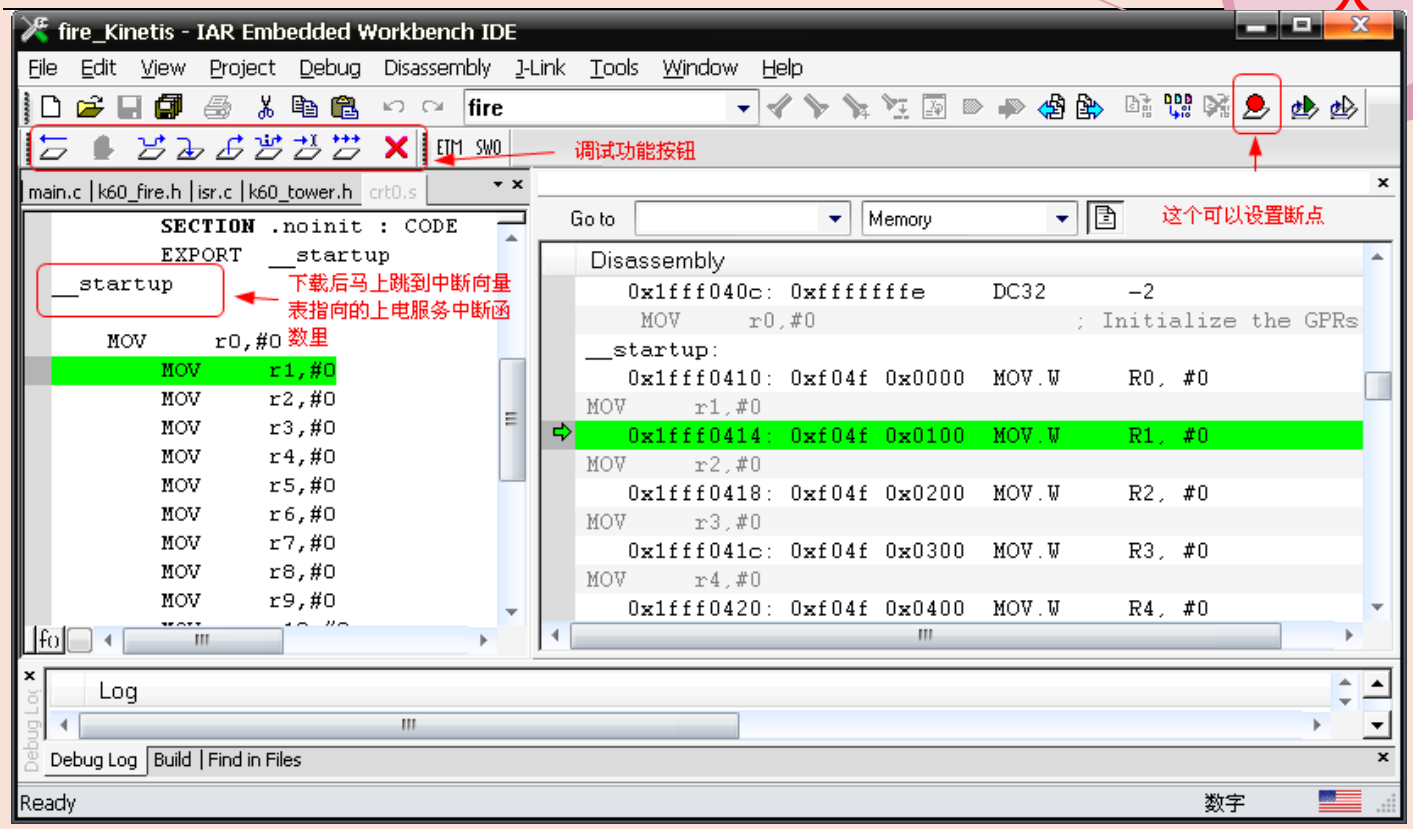
Jlink 有两种下载方式：JTAG 和 SWD



然后点击工具栏的 按钮，下载并调试 或者 按钮，直接调试，不下载：

闪烁一下下载窗口：

然后就进入调试界面：



调试的工具栏：

：复位

：暂停

：步过，执行函数，不进入函数内

：步进，跳进函数内

：步出，跳出函数

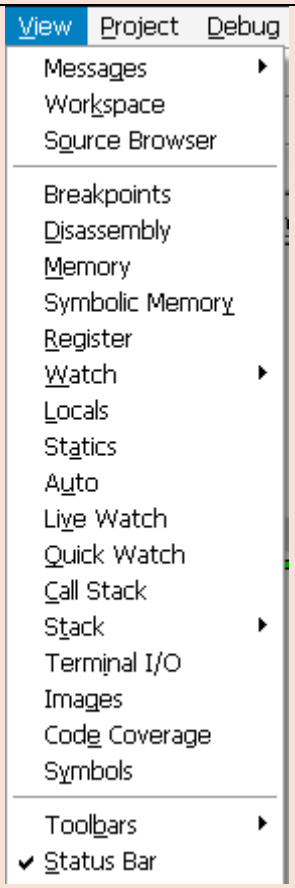
：下一条语句

：跳到光标所指向的语句

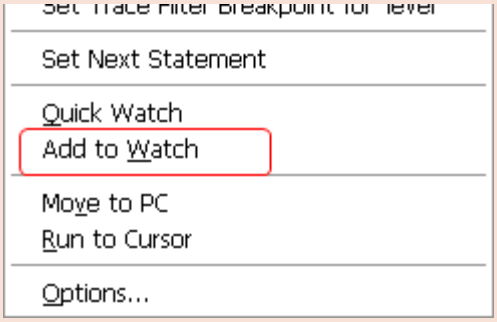
：全速运行

：退出调试

在菜单栏里选择 view ， 就可以看到很多的调试工具。



如果想看变量的值，可以直接光标选中该变量，然后右键——add to watch



调试功能跟其他编译器类似。

用 Jlink 解锁 Kinetis

当芯片进入安全保护状态，就需要擦除芯片就才能继续烧写程序，烧写时会提示：

直接点击是就会自动帮你擦除芯片，然后就能下载程序。

但有时候会锁住芯片的，那时候会提示：

由于野火在使用 Kinetis 过程中，还没出现过这问题。为此，特意网上找到一个可能比较容易解锁的方法：

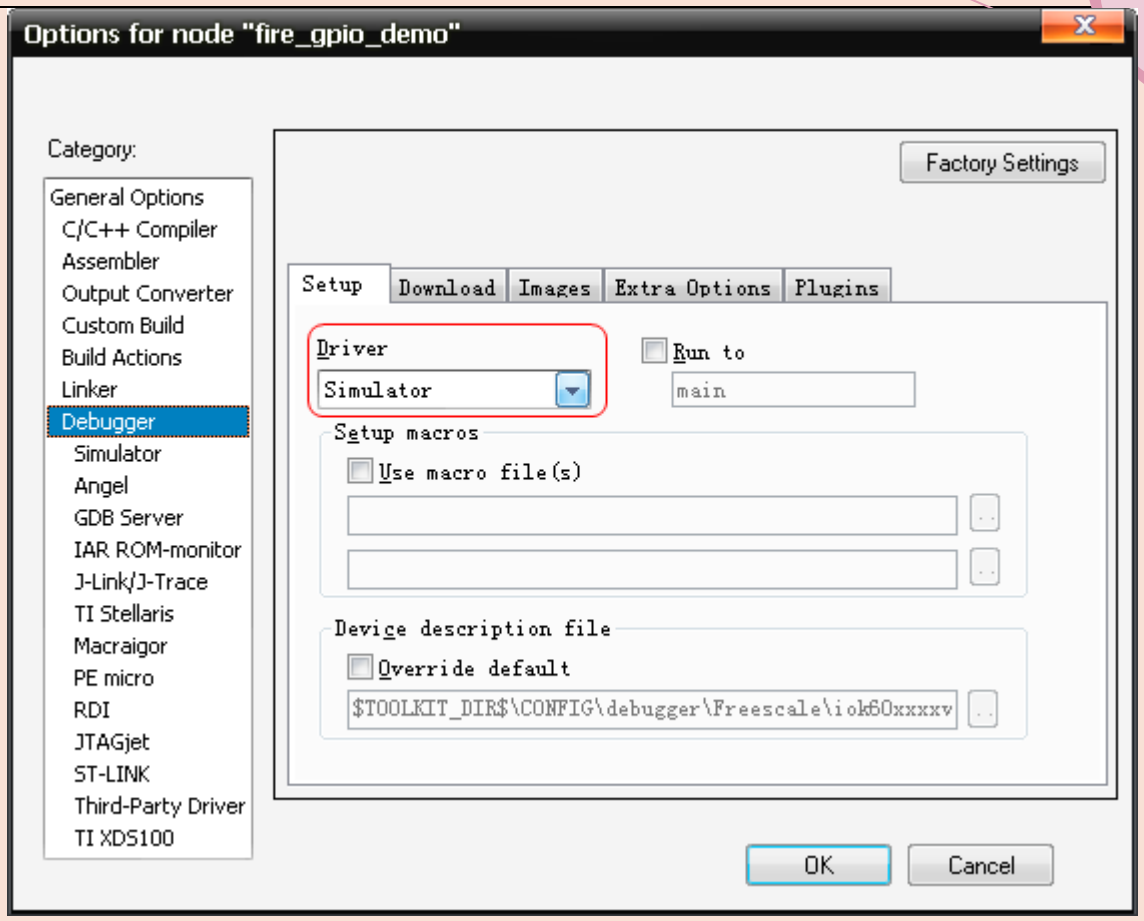
在使用 JLINK 给 K60 烧写程序时，如果芯片被锁如何处理？ 用 J-Link Commander 看看能不能找到 M4 内核，如果能找到，在运行的 J-Link Commander 里面，敲入以下命令：Unlock Kinetis。 然后回车，就能解锁。

原文出处：http://www.eefocus.com/bbs/article_891_256673.html

在 Q 群中，野火与其他人交流时，发现锁住芯片的比较多是用 CW 编译器的，较少是 IAR，而野火用的是 IAR，当然也有可能是用 CW 的人比较多吧。另外，也不要用带 USB3.0 的 USB 接口来接 Jlink、BDM 等下载器，容易出错，甚至下载不了。

使用软件仿真调试

要使用仿真调试，首先就要在选项里设置调试器为软件仿真：



另外还要禁止锁相环时钟设置和串口发送，不然仿真的时候，会卡在死循环里，不能继续调试。

首先，在 common.h 里添加：

```
1. //为使用仿真模式而添加的，仿真模式应该屏蔽串口发送、和 PLL 锁相环设置
2. #define    Simulator
3. #ifdef     Simulator
4.   #define NO_PLL_INIT    //禁用锁相环
5.   #define NPRINTF        //禁用 printf
6. #endif
```

在 printf.h 里修改 printf 函数，即把下面代码：

```
1.  int
2.  printf (const char *fmt, ...)
3.  {
4.      va_list ap;
5.      int rvalue;
6.      PRINTK_INFO info;
7.
8.
9.      info.dest = DEST_CONSOLE;
10.     info.func = &out_char;
11.     /*
```

```
12.       * Initialize the pointer to the variable length argument list.
13.       */
14.      va_start(ap, fmt);
15.      rvalue = printk(&info, fmt, ap);
16.      /*
17.       * Cleanup the variable length argument list.
18.       */
19.      va_end(ap);
20.      return rvalue;
21.  }
```

修改为：

```
1.  #ifdef  NPRINTF      //禁用 printf
2.  int
3.  printf (const char *fmt, ...)
4.  {
5.    return 1;
6.  }
7.  #else
8.  int
9.  printf (const char *fmt, ...)
10. {
11.      va_list ap;
12.      int rvalue;
13.      PRINTK_INFO info;
14.
15.
16.      info.dest = DEST_CONSOLE;
17.      info.func = &out_char;
18.      /*
19.       * Initialize the pointer to the variable length argument list.
20.       */
21.      va_start(ap, fmt);
22.      rvalue = printk(&info, fmt, ap);
23.      /*
24.       * Cleanup the variable length argument list.
25.       */
26.      va_end(ap);
27.      return rvalue;
28. }
29. #endif
```

即添加这些部分

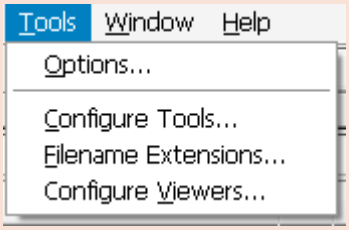
这样就可以使用仿真了。如果不是仿真，而是要烧写到芯片上，记得注释掉 common.h 里的 Simulator 定义。

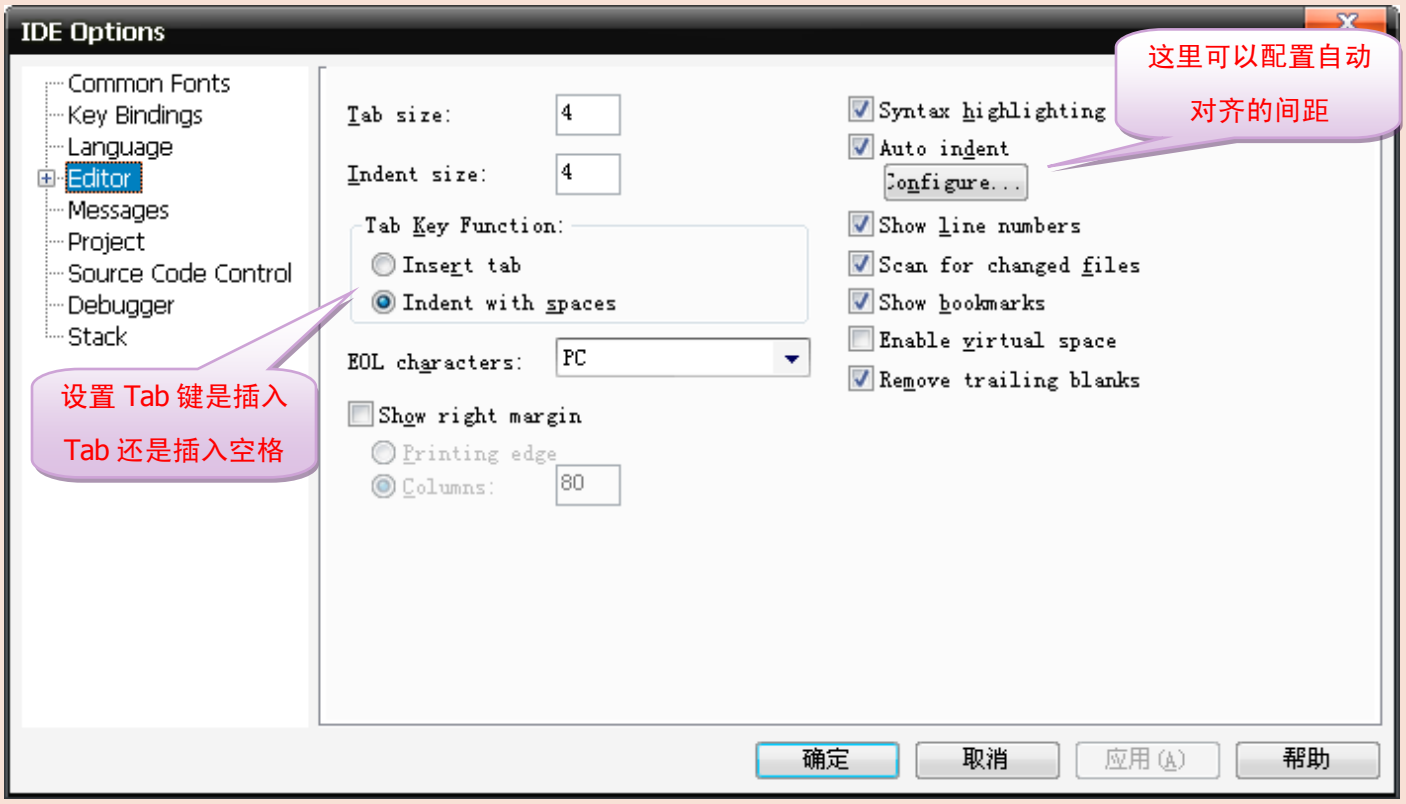
```
30. //为使用仿真模式而添加的，仿真模式应该屏蔽串口发送、和 PLL 锁相环设置
31. //#define   Simulator
```

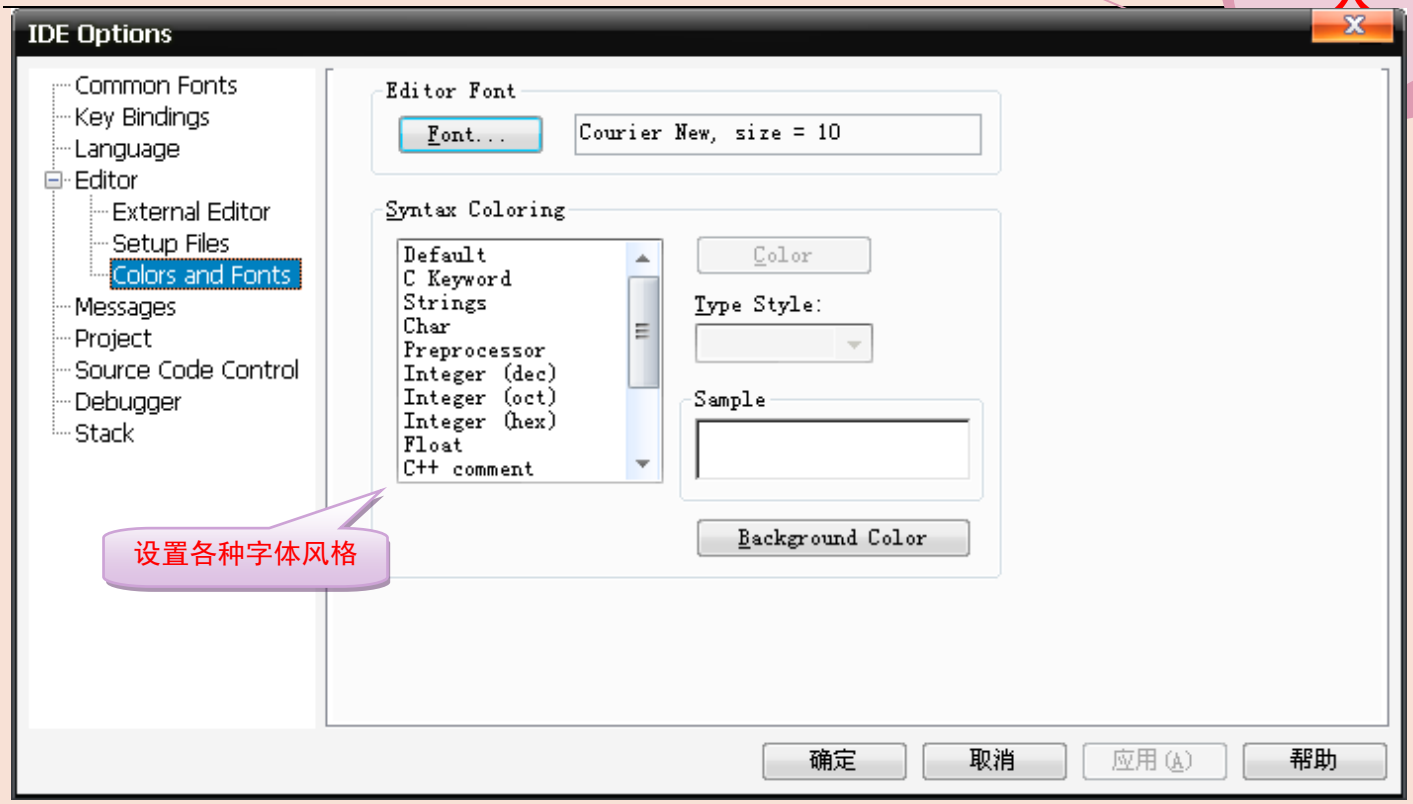
下载方式跟 jlink 一样，直接点击：工具栏的  按钮 。

IAR 界面风格设计

在菜单栏——Tools——Options






IDE Options
Common Fonts
Key Bindings
Language
Editor
External Editor
Setup Files
Colors and Fonts
Messages
Project
Source Code Control
Debugger
Stack
Editor Font
Font...
Courier New, size = 10
Syntax Coloring
Default
C Keyword
Strings
Char
Preprocessor
Integer (dec)
Integer (oct)
Integer (hex)
Float
C++ comment
Color
Type Style:
Sample
Background Color
设置各种字体风格
确定
取消
应用(A)
帮助

野火 Kinetis 核心板实验例程列表

模块	实验号	实验
GPIO	1	51 编程风格的 GPIO 实验输出测试
	2	51 编程风格的 GPIO 实验输入输出测试
	3	GPIO 实验简单测试
	4	GPIO 实验并行读写测试
	5	GPIO 实验综合测试
LED	6	LED 综合测试例程
EXTI	7	EXTI 综合测试例程
UART	8	串口发送例程
	9	串口查询接收例程
	10	串口中断接收例程
ADC	11	ADC 综合测试例程
FTM	12	PWM 实验示波器简单测试
	13	PWM 实验 LED 测试
	14	FTM 输入捕捉中断测试
PIT	15	PIT 定时中断测试例程
FTM&PIT	16	PWM、输入捕捉、PIT 中断综合测试
LPTMR	17	LPT 脉冲累加计数中断实验
	18	PIT 定时读取 LPT 脉冲累加计数实验
I2C	19	I2C 通信实验测试
MCG	20	配置系统频率
RTOS	21	uC/OS

野火 K60 库的使用

前言

想快速上手 K60 单片机吗？野火 Kinetis K60 库，是你最好的选择：简单的调用函数接口，良好的编程风格，让你可以不了解寄存器配置的情况下快速入门 Kinetis 系列单片机。

学习一款单片机，完全没必要深入了解寄存器的配置。现在的单片机更新换代太快，寄存器越来越多，就算你能完全记住 51 的所有寄存器，但你能记得所有的 Kinetis 系列的寄存器吗？现在的发展方向是单片机功能越来越强大，寄存器越来越复杂，底层的驱动由官方写好，方便应用者专注于开发自己的应用程序，而不是搞底层开发。

野火 Kinetics K60 库，尽可能地把底层的驱动完善，用户可以直接调用 API 接口，而不必慢慢对着 datasheet 来研究。例如 UART、FTM、I2C 等模块，函数内部会根据系统时钟频率来自动计算和选择分频系数，用户不必担心更改频率后模块不能使用。当然，如果 main 函数里中途更改时钟频率，就需要重新初始化，以便重新计算和选择分频系数。

野火 Kinetics K60 库，部分功能的实现，提供了两种或两种以上的实现方法，一种是函数调用，一种是宏定义。前者节省空间，后者加快速度。

在编写库的工程中，部分代码参考了其他人例程，在野火 Kinetics K60 库代码中都有备注。代码会不断更新，不一定能及时更新到教程上，如果教程与代码有相互不同的地方，一切以代码为标准。

由于个人能力及时间所限，出错之处，在所难免，欢迎各位指出错误及提出建议：minimcu@foxmail.com

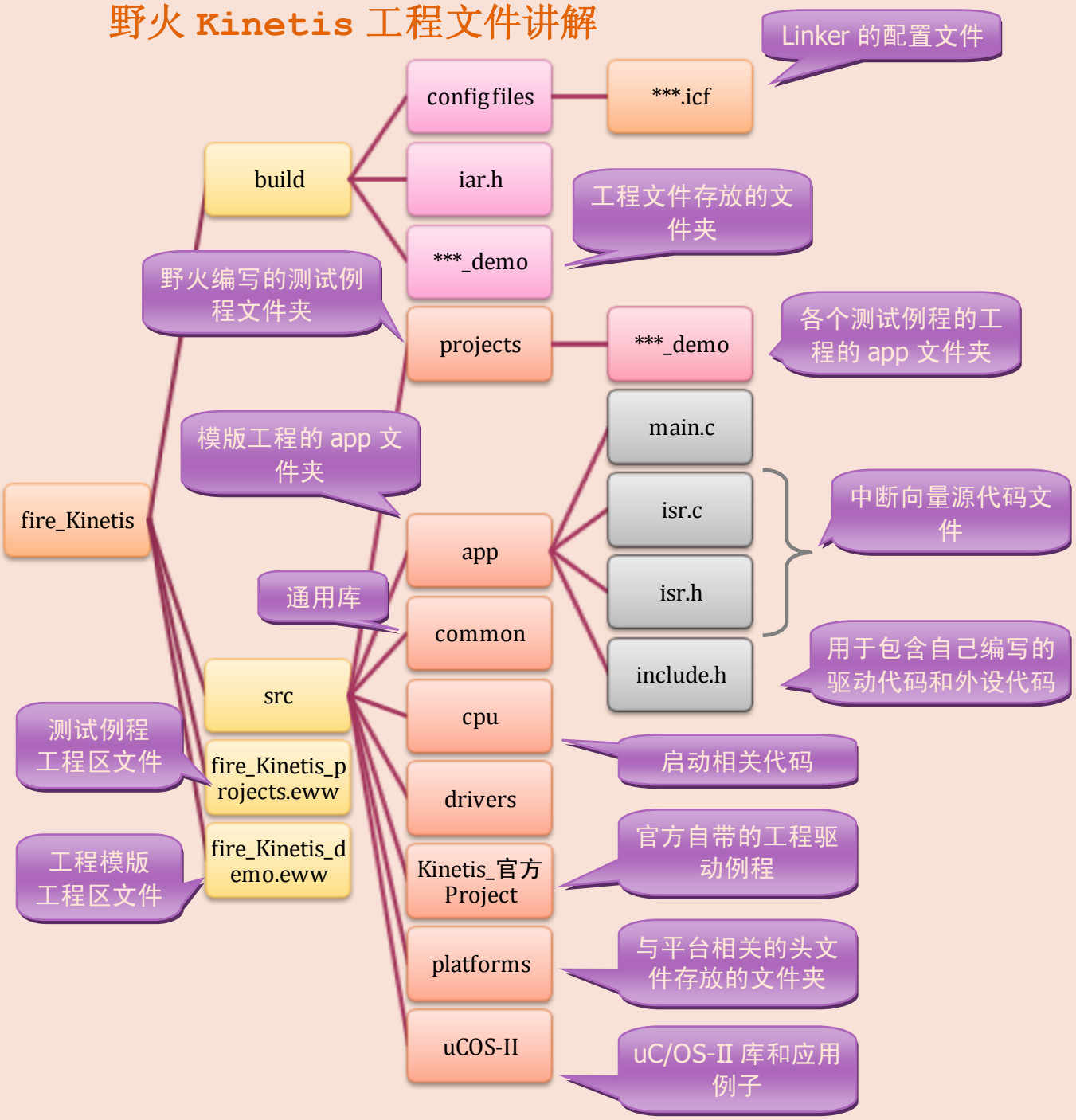
——野火嵌入式开发工作室

淘宝地址：http://firestm32.taobao.com

快速开发指南

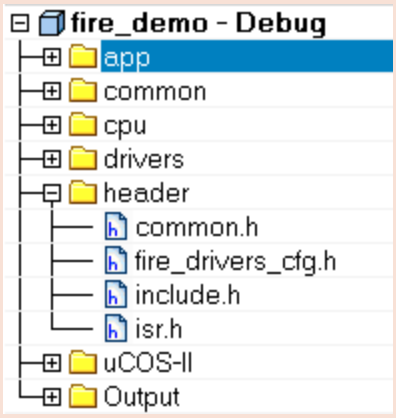
快速入门：了解野火 Kinetis 工程

野火 Kinetis 工程文件讲解

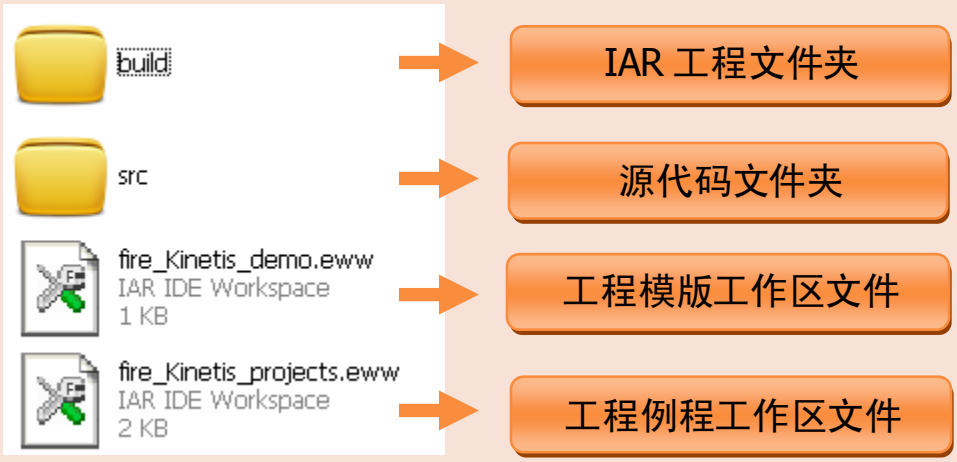


常用的五个头文件

头文件名	作用
common.h	通用配置同文件，与其他工程共用的通用配置
include.h	工程头文件，用于针对特定的工程的配置和包含底层驱动头文件。
isr.h	工程的中断配置头文件
fire_drivers_cfg.h	野火整理出来的快速复用管脚设置头文件
k60_fire.h	配置时钟和开发板选项

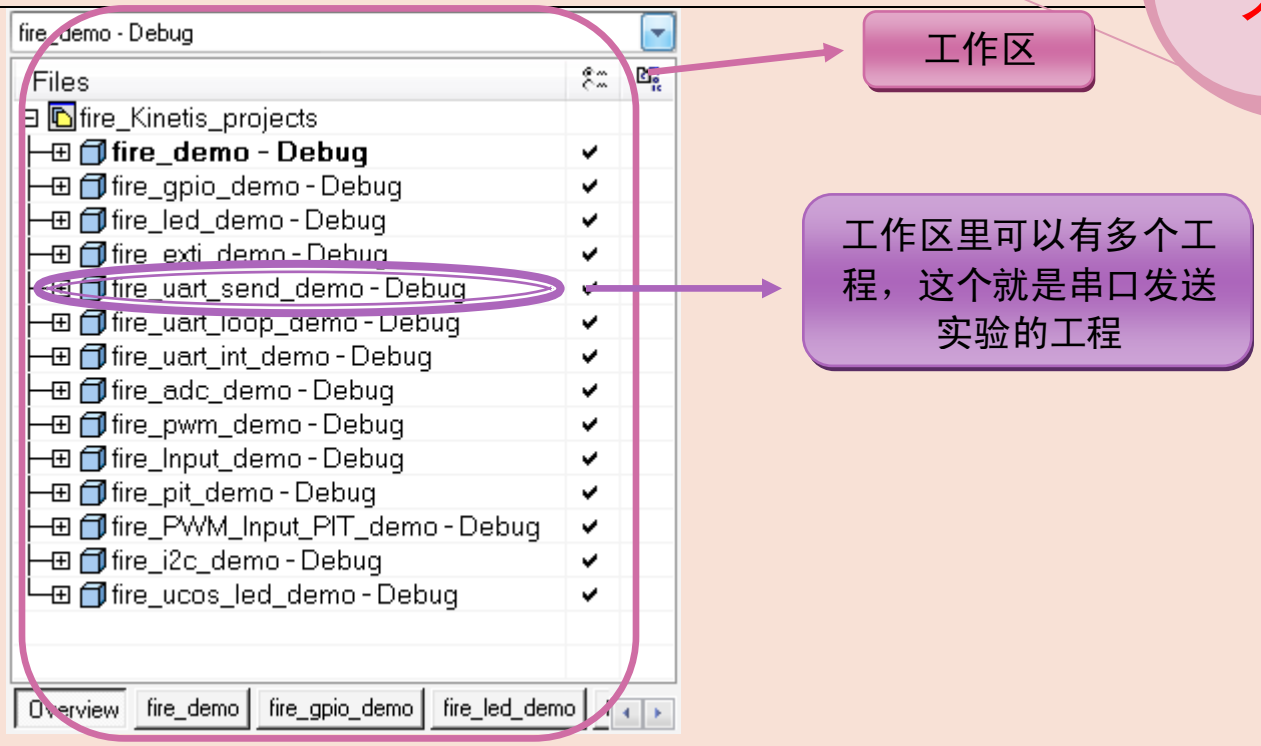


打开 fire_Kinetis 文件夹下：



IAR 工程文件夹

fire_Kinetis\build 为 IAR 工程文件夹。使用过 IAR 的应该知道 IAR 里可以建立一个工作区，工作区里有多个工程：



我们现在就是要把所有的工程文件都放在这个 fire_Kinetis\build 文件夹里：

另外，fire_Kinetis\build 文件夹里还有 config files 文件夹 和 iar.h 文件。

config files 文件夹里放的就是 linker 配置文件：

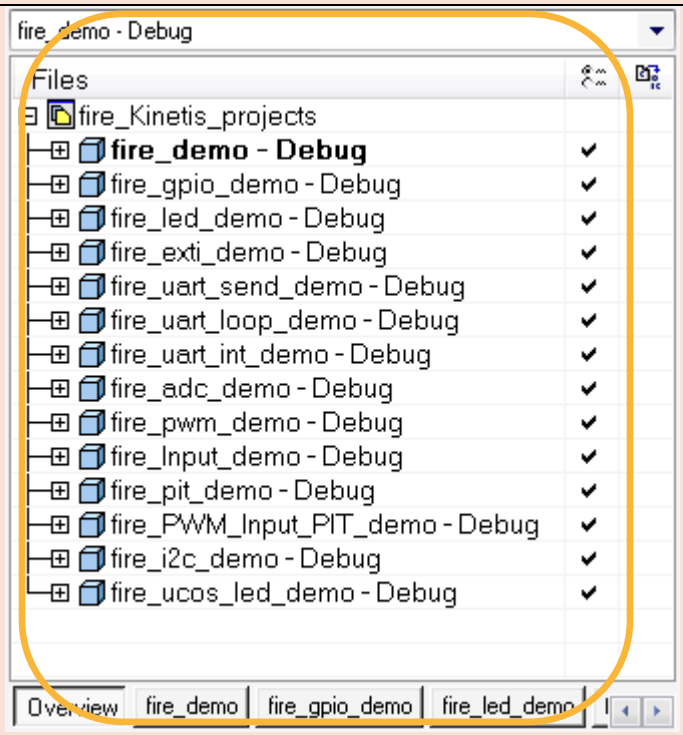
iar.h 是用来专门配置 IAR 编译的头文件，这里实际上没用到。

源代码文件夹

所有的源代码文件都放在这里：

工程模版工作区文件

这个工作区里，包含了多个工程，用 IAR 打开就可以看到：



里面有包含多个工程，按照工程的上下顺序学习，就可以容易熟悉 野火 K60 库的用法。

工程例程工作区文件

这个是给用户自己编写自己的工程模版，工作区里面只有一个工程，与其他工程共用底层驱动，这样就方便改一个底层驱动就能全部都改。

用户应用文件放在 fire_Kinetis\src\app 文件夹下

例程的应用文件放在 fire_Kinetis\src\projects\fire_xxx _demo 文件夹下

快速复用管脚

野火 K60 库，本身已经设置了默认的各个模块通道所对应的管脚。其实就是在 fire_Kinetis\src\drivers \fire_drivers_cfg.h 文件里进行设置：

```
/******************** (C) COPYRIGHT 2011 野火嵌入式开发工作室 ********************
 * 文件名       ：fire_drivers_cfg.h
 * 描述         ：K60复用管家配置
 * 备注         ：野火耗了很久时间整理出来，希望尊重野火的劳动成功，注明野火原创！！
 *
 * 实验平台     ：野火kinetis开发板
 * 库版本       ：
 * 嵌入系统     ：
 *
 * 作者         ：野火嵌入式开发工作室
 * 淘宝店       ：http://firestm32.taobao.com
 * 技术支持论坛 ：http://www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
**********************************************************************************/

#ifndef _FIRE_DRIVERS_CFG_H_
#define _FIRE_DRIVERS_CFG_H_

#include "gpio_cfg.h"


/*************************************   UART   ***************************************/

//      模块通道    端口          可选范围                         建议
#define UART0_RX    PTD6        //PTA1、PTA15、PTB16、PTD6         PTA1不要用（与Jtag冲突）
#define UART0_TX    PTD7        //PTA2、PTA14、PTB17、PTD7         PTA2不要用（与Jtag冲突）

#define UART1_RX    PTC3        //PTC3、PTE1
#define UART1_TX    PTC4        //PTC4、PTE0

#define UART2_RX    PTD2        //PTD2
```

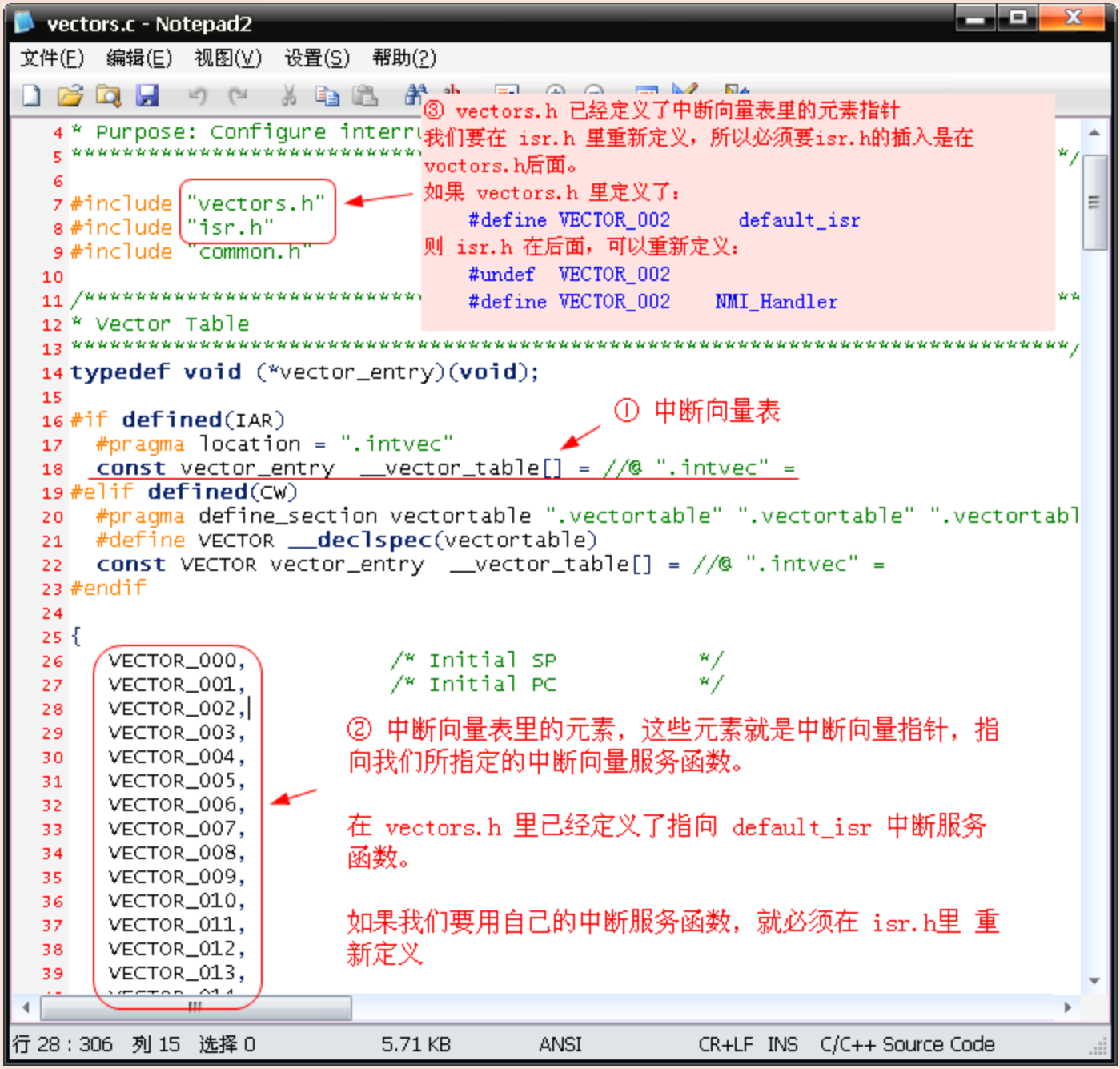
只需按照注释里写着的可选范围进行修改这里的宏定义，就能更改默认的复用管脚，不需要修改函数。

中断函数的编写方法

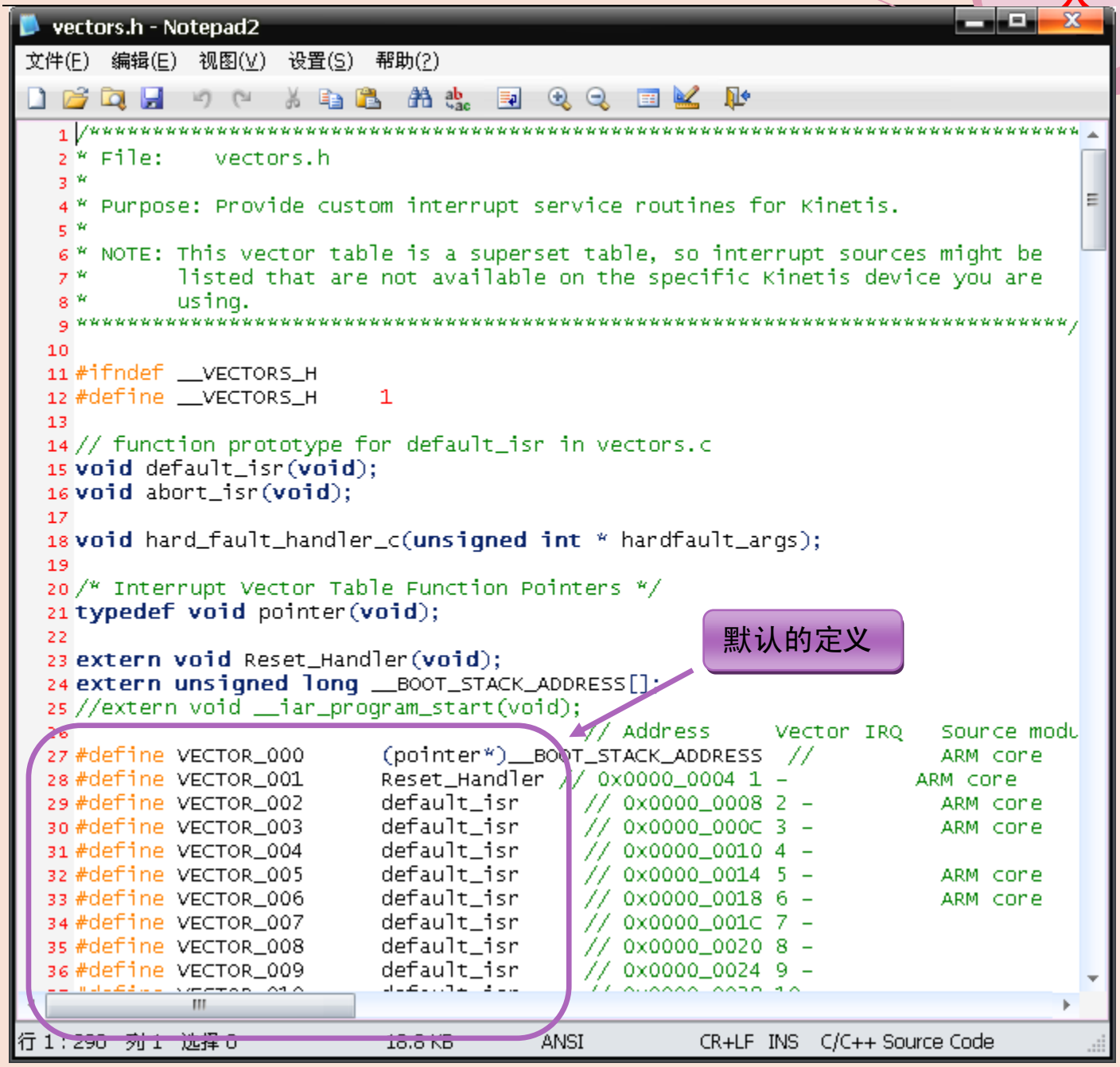
飞思卡尔公司提供的模版里，已经提供有中断函数的参考注释。

主要用到 fire_Kinetis\src\cpu 文件下的 vectors.c 和 vectors.h 文件，这两个用来定义中断向量表 和 定义了一个默认的中断服务函数。

在 vectors.c 里：



在 vectors.h 文件里：




```
/*********************************************************************************
* File:    vectors.h
*
* Purpose: Provide custom interrupt service routines for Kinetis.
*
* NOTE: This vector table is a superset table, so interrupt sources might be
*       listed that are not available on the specific Kinetis device you are
*       using.
*********************************************************************************/

#ifndef __VECTORS_H
#define __VECTORS_H     1

// function prototype for default_isr in vectors.c
void default_isr(void);
void abort_isr(void);

void hard_fault_handler_c(unsigned int * hardfault_args);

/* Interrupt Vector Table Function Pointers */
typedef void pointer(void);

extern void Reset_Handler(void);
extern unsigned long __BOOT_STACK_ADDRESS[];
//extern void __iar_program_start(void);

                                    // Address     Vector IRQ   Source modu
#define VECTOR_000      (pointer*)__BOOT_STACK_ADDRESS  //          ARM core
#define VECTOR_001      Reset_Handler // 0x0000_0004 1 -          ARM core
#define VECTOR_002      default_isr   // 0x0000_0008 2 -          ARM core
#define VECTOR_003      default_isr   // 0x0000_000C 3 -          ARM core
#define VECTOR_004      default_isr   // 0x0000_0010 4 -
#define VECTOR_005      default_isr   // 0x0000_0014 5 -          ARM core
#define VECTOR_006      default_isr   // 0x0000_0018 6 -          ARM core
#define VECTOR_007      default_isr   // 0x0000_001C 7 -
#define VECTOR_008      default_isr   // 0x0000_0020 8 -
#define VECTOR_009      default_isr   // 0x0000_0024 9 -
```

如果我们不重新映射中断向量指针，中断向量指针就会指向 default_isr 默认的中断向量服务函数里。有时候，我们的程序运行异常，不妨打开串口助手，看看是不是跑到 default_isr 服务函数里，如果是，那往往都是代码有 bug。

为了重新映射中断到指向我们指定的中断向量服务函数里，我们建立 isr.h 和 isr.c 两个文件来重映射中断向量指针和编写中断向量服务函数。

在 isr.h 文件里，可以看到野火 K60 模版里已经在注解上写了参考模版，依葫芦画瓢就可以了：

```
#ifndef __ISR_H
#define __ISR_H 1

#include  "include.h"

/*                                重新定义中断向量表
 *   先取消默认的中断向量元素宏定义        #undef  VECTOR_xxx
 *   在重新定义到自己编写的中断函数        #define VECTOR_xxx    xxx_IRQHandler
 *   例如:
 *        #undef  VECTOR_003
 *        #define VECTOR_003    HardFault_Handler     重新定义硬件上访中断服务函数
 *
 *        extren void  HardFault_Handler(void);       声明函数，然后在 isr.c 里定义
 */

#endif  //__ISR_H

/* End of "isr.h" */
```

具体例子，可参考以下例程：

EXTI 综合测试例程

串口中断接收例程

FTM 输入捕捉中断测试

PIT 定时中断测试例程

PWM、输入捕捉、PIT 中断综合测试

重要变量、函数、宏定义一览表

重要的全部变量列表

名称	功能说明
core_clk_khz	内核时钟(KHz)
core_clk_mhz	内核时钟(MHz)
bus_clk_khz	外围总线时钟
__vector_table	中断向量表数组

重要的全局宏定义列表

名称	功能说明
DEBUG	定义 Debug 模式
DEBUG_PRINT	定义输出调试信息
MCG_CLK_MHZ	定义系统频率
Simulator	定义软件仿真

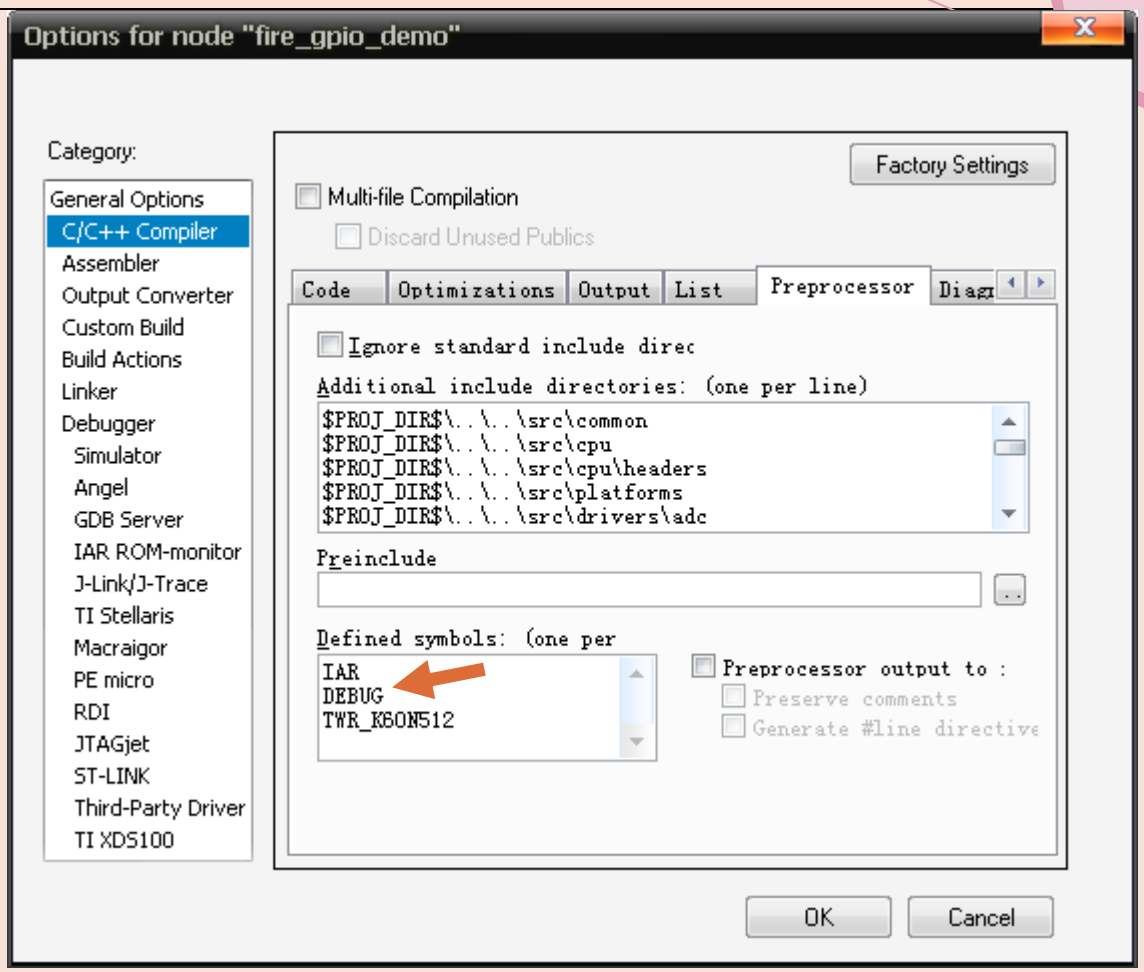
重要的函数列表：

名称	功能说明
printf	串口输出
pll_init	设置锁相环超频

全局宏定义详解

DEBUG

在工程选项那里定义



功能说明

用来定义进入 DEBUG 模式。

如果定义了，则会输出一些调试信息，执行一些防错误的检测，否则取消这些活动。

例如：使用断言，可以帮助我们在调试的时候检查函数传递的参数是否正确，但当我们发布的时候，就需要取消这些检查。

DEBUG_PRINT

在 common.h 里定义

定义

```
1.  #define DEBUG_PRINT
```

功能说明

用来定义是否通过串口显示调试信息。

函数详解

printf

在 printf.c 里定义

函数原型

```
1.  int printf (const char *fmt, ...);
```

要使用 printf，需要定义 printf 的输出信息，在 k60_fire.h 里：

```
1.  /*********************************************************************
2.  *                              野火嵌入式开发工作室
3.  *
4.  *  功能说明：定义 printf 函数 的 串口输出端口 和 串口信息
5.  *  备    注：
6.  *********************************************************************/
7.  #define FIRE_PORT           UART1
8.  #define FIRE_BAUD           19200
```

这里是定义了输出的端口和波特率，然后在 sysinit.c 文件里的 sysinit 函数里调用：

```
1.  void sysinit (void)
2.  {
3.      ……
4.      uart_init(FIRE_PORT, FIRE_BAUD);     //初始化 printf 函数所用到的串口
5.  }
```

功能说明

通过串口发送数据。

调用例子

```
32. printf("欢迎使用野火 Kinetis 开发板\n");
33. printf("%d 年到了\n",2012);
```

重定向输出

printf 函数是可以重映射输出的，在 printf.c 里有 printf 函数。

```
1.  int printf (const char *fmt, ...)
2.  {
3.      PRINTK_INFO info;
4.
5.      info.func = &out_char;              //定义 printf 输出字符的函数为 out_char
6.      ……
7.      rvalue = printk(&info, fmt, ap);    //传递给 printk 函数
8.      ……
```

```
}
```

在 printk 里：

```
int printk (PRINTK_INFO *info, const char *fmt, va_list ap)
{
    ……
    for (p = (char *)fmt; (c = *p) != 0; p++)
    {
        ……
        printk_putc(c, &count, info);    //提取字符串的字符 c ,传递给 printk_putc
        ……
    }
    ……
}
```

在 printk_putc 里：

```
static void printk_putc (int c, int *count, PRINTK_INFO *info)
{
    ……
    info->func((char)c);    //在 printf 里定义 func 为 out_char
                            //即类似执行：out_char((char)c);
    ……
}
```

out_char 函数在 io.c 里定义：

```
void out_char (char ch)
{
    uart_putchar(FIRE_PORT, ch);  //这里就是真正的串口发送字符函数
}
```

综上所述，要重映射输出，有两种方法：

① 在 printf 函数里把 info.func 改为其他函数地址。

```
    info.func = &out_char;              //定义 printf 输出字符的函数为 out_char
```

② 修改 out_char 函数里的输出函数，例如改为 LCD 输出。

安全检查

为了加强程序的健壮性，减少 bug，野火为此而在程序里添加了一些安全措施，方便在调试阶段就能检查出错误，消灭 bug。

断言

相信现在还存在着很多连断言是什么都不知道的软件工程师，更别说是初学者了。

断言，其实就是一个宏定义：

```
/* 注：野火修改过这里的代码，与官方提供例子是不一样的 */
#ifdef DEBUG
#define ASSERT(expr) \
    if (!(expr)) \
        assert_failed(__FILE__, __LINE__)
#else
#define ASSERT(expr)
#endif

//如果断言条件不成立，进入了错误状态，就会打印错误信息和用 LED 来显示状态
void assert_failed(char *file, int line)
{
    LED_init();
    while (1)
    {
#ifdef DEBUG_PRINT
        printf(ASSERT_FAILED_STR, file, line);   //打印错误信息
#endif
        water_lights();                          //用流水灯来指示进入错误状态
    }
}
```

如果定义了 DEBUG，就会对断言里的条件进行检查，条件不成立就提示错误信息，进入死循环。因为会进入死循环，很明显，这个断言是用来检测我们的程序错误，例如传递给函数的变量取值范围是否正确。例如在 ADC 模块里，可以看到很多函数的开头都有这样一个语句：

```
ASSERT( ((adcn == ADC0) && (ch>=AD8 && ch<=AD18)) || ((adcn == ADC1)&& (ch>=AD4a && ch<=AD17)) ) ;
                                                                    //使用断言检测 ADCn_CHn 是否正常
```

如果传递错误的参量，就会出错，在串口里可以看到错误的位置：

```
Assertion failed in G:\大学学习\k60重要文档\fire_Kinetis
\src\drivers\adc\adc.c at line 34
Assertion failed in G:\大学学习\k60重要文档\fire_Kinetis
\src\drivers\adc\adc.c at line 34
Assertion failed in G:\大学学习\k60重要文档\fire_Kinetis
\src\drivers\adc\adc.c at line 34
Assertion failed in G:\大学学习\k60重要文档\fire_Kinetis
\src\drivers\adc\adc.c at line 34
Assertion failed in G:\大学学习\k60重要文档\fire_Kinetis
\src\drivers\adc\adc.c at line 34
Assertion failed in G:\大学学习\k60重要文档\fire_Kinetis
\src\drivers\adc\adc.c at line 34
Assertion failed in G:\大学学习\k60重要文档\fire_Kinetis
\src\drivers\adc\adc.c at line 34
Assertion failed in G:\大学学习\k60重要文档\fire_Kinetis
\src\drivers\adc\adc.c at line 34
```

这样在调试阶段，我们就能发现传递给函数的变量是不是超出了范围，在调试阶段就解决 bug。

关于断言的使用，从 林锐的《高质量 C 编程指南》里提取了一段断言的使用规则：

程序一般分为 Debug 版本和 Release 版本，Debug 版本用于内部调试，Release 版本发行给用户使用。

断言 assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况。

【规则 6-5-1】使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。

【规则 6-5-2】在函数的入口处，使用断言检查参数的有效性（合法性）。

【建议 6-5-1】在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了的假定，就要使用断言对假定进行检查。

【建议 6-5-2】一般教科书都鼓励程序员们进行防错设计，但要记住这种编程风格可能会隐瞒错误。当进行防错设计时，如果“不可能发生”的事情的确发生了，则要使用断言进行报警。

枚举

枚举的使用，本身就是用来限制变量的范围，可以在编译阶段就进行报错，让程序员可以及时发现错误和修正错误。

野火为此也在程序里使用了很多枚举定义，方便在调试阶段就能检查出错误，消灭 bug。

例如 ADC.h 文件里

```
1.  typedef enum ADCn  //ADC 端口
2.  {
3.    ADC0,
4.    ADC1
5.  }ADCn;
```

如果打开其他网络上共享的工程，你们会发现他们往往都是使用宏定义，但宏定义是仅仅进行替换操作，不进行安全检测，这就无法保证安全性。对于一个软件工程

师来说，对程序的安全性进行检测，那是一个基本的要求，但可惜很多程序员都没法做到这点。野火的函数定义里就规定了传递进去的参数是枚举，大大保障了程序的安全性：

```
1.  void adc_init(ADCn adcn,ADC_Ch ch);
```

举个例子：

```
1.  adc_init(0,8);        //报警告
2.  adc_init(ADC0,SE8);   //安全通过检查
```

GPIO 模块

快速入门：GPIO 库使用方法

形参变量的命名及其作用

形参变量	作用	取值
PORTx	端口名	PORTA，PORTB，PORTC，PORTD，PORTE
n	引脚号	0~31
GPIO_CFG	输入输出配置	GPI ，GPO， GPI_DOWN ，GPI_UP ， GPI_PF，GPI_DOWN_PF ， GPI_UP_PF ， GPO_HDS ，GPO_SSR=0x05， GPO_HDS_SSR 。
data	数据	

常用枚举列表

名称	功能说明
PORTx	端口
GPIO_CFG	管脚输入输出配置

函数列表

函数名称	函数功能
gpio_init	初始化 gpio
gpio_set	设置引脚状态
gpio_turn	反转引脚状态
gpio_get	获取 IO 状态

宏定义列表

宏定义名词	功能说明
PTxn_OUT	设置引脚输出电平（这里的 x 要替换成 A~E,n 要替换为 0~31 的数，表示对应的引脚号） 例如：PTA0_OUT
PTxn_INT	读取引脚输入电平（这里的 x 要替换成 A~E,n 要替换为 0~31 的数，表示对应的

	引脚号） 例如：PTA0_IN
DDRxn	设置引脚输入输出方向（这里的 x 要替换成 A~E,n 要替换为 0~31 的数，表示对应的引脚号） 例如：DDRA0
PTx_BYTEn_OUT	设置 8 位引脚输出电平（这里的 x 要替换成 A~E,n 要替换为 0~3 的数，表示对应的 8 位引脚号） 例如： PTA_BYTE0_OUT 表示 PTA7~ PTA0 PTA_BYTE1_OUT 表示 PTA15~ PTA8 PTA_BYTE2_OUT 表示 PTA23~ PTA16 PTA_BYTE3_OUT 表示 PTA31~ PTA24
PTx_BYTEn_IN	设置 8 位引脚输入电平（这里的 x 要替换成 A~E,n 要替换为 0~3 的数，表示对应的 8 位引脚号）
DDRx_BYTEn	设置 8 位引脚输入输出方向（这里的 x 要替换成 A~E,n 要替换为 0~3 的数，表示对应的 8 位引脚号）
PTx_WORDn_OUT	设置 16 位引脚输出电平（这里的 x 要替换成 A~E,n 要替换为 0~1 的数，表示对应的 16 位引脚号） 例如： PTA_WORD0_OUT 表示 PTA15~ PTA0 PTA_WORD1_OUT 表示 PTA31~ PTA16
PTx_WORDn_IN	设置 16 位引脚输入电平（这里的 x 要替换成 A~E,n 要替换为 0~1 的数，表示对应的 16 位引脚号）
DDRx_WORDn	设置 16 位引脚输入输出方向（这里的 x 要替换成 A~E,n 要替换为 0~1 的数，表示对应的 16 位引脚号）
GPIO_SET	设置引脚状态
GPIO_TURN	翻转引脚状态
GPIO_Get	获取 IO 状态
GPIO_SET_1bit	写 1 位数据
GPIO_GET_1bit	读 1 位数据
GPIO_SET_2bit	写 2 位数据
GPIO_GET_2bit	读 2 位数据
GPIO_SET_4bit	写 4 位数据

GPIO_GET_4bit	读 4 位数据
GPIO_SET_8bit	写 8 位数据
GPIO_GET_8bit	读 8 位数据
GPIO_SET_16bit	写 16 位数据
GPIO_GET_16bit	读 16 位数据

枚举详解

PORTx

枚举定义

```
1.  enum PORTx       //端口宏定义
2.  {
3.    PORTA,
4.    PORTB,
5.    PORTC,
6.    PORTD,
7.    PORTE
8.  };
```

枚举作用

用来定义端口号

GPIO_CFG

枚举定义

```
1.  typedef enum GPIO_CFG
2.  {
3.    GPI=0,                                  //定义管脚输入方向
4.    GPO=1,                                  //定义管脚输出方向
5.
6.    GPI_DOWN     =0x02,                     //输入下拉
7.    GPI_UP       =0x03,                     //输入上拉
8.    GPI_PF       =0x10,                     //输入，带无源滤波器
9.    GPI_DOWN_PF =GPI_DOWN | GPI_PF ,        //输入下拉，带无源滤波器
10.   GPI_UP_PF    =GPI_UP   | GPI_PF ,       //输入上拉，带无源滤波器
11.
12.
13.   GPO_HDS      =0x41,                     //输出高驱动能力
14.   GPO_SSR      =0x05,                     //输出慢变化率
15.   GPO_HDS_SSR = GPO_HDS | GPO_SSR         //输出高驱动能力、慢变化率
16. };
```

常用的就这五种

枚举作用

用来定义 GPIO 管脚输入输出的配置。

常用的就前面五种，例如输入滤波，可用着按键上，可以设置为上拉下拉，省去外部电阻。

函数详解

gpio_init

函数原型

```
1.  void    gpio_init(u8 PORTx, u8 n, u8 IO,u8 data);     //初始化 gpio
```

功能说明

初始化 gpio，设置端口的输入输出方向，输出数据。初始化后，才能对 IO 口进行读取或电平设置。

调用例子

```
2.  gpio_init(PORTA,27,GPO,1);     //初始化 PTA27 为输出方向，高电平
3.  gpio_init(PORTA,26,GPI,0);     //初始化 PTA26 为输入方向
```

gpio_set

函数原型

```
4.  void    gpio_set(u8 PORTx,  u8 n,u8 data);          //设置引脚状态
```

功能说明

设置 IO 口电平，需要先初始化 GPIO 端口为输出方向。

调用例子

```
5.  gpio_set(PORTA,27,1);                    //PTA27 = 1
```

gpio_turn

函数原型

```
6.  void    gpio_turn  (u8 PORTx,  u8 n);       //翻转引脚状态
```

功能说明

翻转 IO 口电平

传递参数

参考形参变量的命名及其作用。

函数返回
无

调用例子
```
7.  gpio_turn(PORTA,27);                    //翻转引脚状态,PTA27=~PTA27
```

gpio_get

函数原型
```
8.  u8      gpio_get    (u8 PORTx,  u8 n);      //读取引脚状态
```

功能说明
读取引脚状态

传递参数
参考形参变量的命名及其作用。

函数返回
返回 1bit 的端口电平状态。

调用例子
```
9.  gpio_set(PORTA,27,1);           //PTA27 = 1
```

宏定义

PTxn_OUT

功能说明
设置电平输出。这个是通用显示名字而已，实际上这里的 x 要替换成 A~E 的端口号， n 要替换为 0~31 的数，表示对应的引脚号，例如 PTA0_OUT

调用例子举例
```
10. PTA0_OUT    = 1;    //PTA0 输出高电平
11. PTB16_OUT   = 1;   //PTB16 输出高电平
12. PTE28_OUT   = 0;   //PTE28 输出低电平
```

PTxn_IN

功能说明

读取电平输入。这个是通用显示名字而已，实际上这里的 x 要替换成 A~E 的端口号， n 要替换为 0~31 的数，表示对应的引脚号，例如 PTA0_IN

调用例子举例

```
13. data=PTA0_IN;    //读取 PTA0 输入电平
14. data=PTB16_IN;   //读取 PTB16 输入电平
15. data=PTE28_IN;   //读取 PTE28 输入电平
```

DDRxn

功能说明

设置引脚输入输出方向。这个是通用显示名字而已，实际上这里的 x 要替换成 A~E,n 要替换为 0~31 的数，表示对应的引脚号。例如：DDRA0

调用例子举例

```
16. DDRA0=0;    //设置 PTA0 为输入
17. DDRB16=1;   //读取 PTB16 为输出
18. DDRE28=0;   //读取 PTE28 为输入
```

PTx_BYTEn_OUT

功能说明

设置 8 位引脚输出电平（这里的 x 要替换成 A~E,n 要替换为 0~3 的数，表示对应的 8 位引脚号）

调用例子举例

```
1.  PTA_BYTE0_OUT=0x12   //表示  PTA7~ PTA0    8 位输出 0x12
2.  PTA_BYTE1_OUT=0x12   //表示  PTA15~ PTA8   8 位输出 0x12
3.  PTA_BYTE2_OUT=0x12   //表示  PTA23~ PTA16  8 位输出 0x12
4.  PTA_BYTE3_OUT=0x12   //表示  PTA31~ PTA24  8 位输出 0x12
```

PTx_BYTEn_IN

功能说明

设置 8 位引脚输入电平（这里的 x 要替换成 A~E,n 要替换为 0~3 的数，表示对应的 8 位引脚号）

调用例子举例

```
1.  char data;
2.  data = PTA_BYTE0_IN  //表示  PTA7~ PTA0    8 位数据输入到 data 变量
3.  data = PTA_BYTE1_IN  //表示  PTA15~ PTA8   8 位数据输入到 data 变量
4.  data = PTA_BYTE2_IN  //表示  PTA23~ PTA16  8 位数据输入到 data 变量
5.  data = PTA_BYTE3_IN  //表示  PTA31~ PTA24  8 位数据输入到 data 变量
```

DDRx_BYTEn

功能说明

设置 8 位引脚输入输出方向（这里的 x 要替换成 A~E,n 要替换为 0~3 的数，表示对应的 8 位引脚号）

调用例子举例

```
1.  DDRA_BYTE0=0xFF;  //  表示  PTA7~ PTA0    8 位设置输入输出方向为输出
2.  DDRA_BYTE1=0xFF;  //  表示  PTA15~ PTA8   8 位设置输入输出方向为输出
3.  DDRA_BYTE2=0xFF;  //  表示  PTA23~ PTA16  8 位设置输入输出方向为输出
4.  DDRA_BYTE3=0xFF;  //  表示  PTA31~ PTA24  8 位设置输入输出方向为输出
```

PTx_WORDn_OUT

功能说明

设置 16 位引脚输出电平（这里的 x 要替换成 A~E,n 要替换为 0~1 的数，表示对应的 16 位引脚号）

调用例子举例

```
1.  PTA_WORD0_OUT=0xFF00;  //表示  PTA15~ PTA0   输出 0xFF00
2.  PTA_WORD1_OUT=0xFF00;  //表示  PTA31~ PTA16  输出 0xFF00
```

PTx_WORDn_IN

功能说明

设置 16 位引脚输入电平（这里的 x 要替换成 A~E,n 要替换为 0~1 的数，表示对应的 16 位引脚号）

调用例子举例

```
1.  u16 data = PTA_WORD0_IN;  //表示  PTA15~ PTA0   输入数据到 data 变量
2.  u16 data = PTA_WORD1_IN;  //表示  PTA31~ PTA16  输入数据到 data 变量
```

DDRx_WORDn

功能说明

设置 16 位引脚输入输出方向（这里的 x 要替换成 A~E,n 要替换为 0~1 的数，表示对应的 16 位引脚号）

调用例子举例

```
1.  DDRA_WORD0=0xFFFF;   //  表示  PTA15~ PTA0    16 位设置输入输出方向为输出
2.  DDRA_WORD1=0xFFFF;   //  表示  PTA31~ PTA16   16 位设置输入输出方向为输出
```

GPIO_SET

定义

```
19. #define GPIO_SET(PORTx,n,x)          GPIO_SET_##x(PORTx,n)
```

功能说明

设置输出电平为 x

传递参数

x　电平状态，只能为 0 或 1。 注意，不能为变量 或 其他宏定义。

（如果需要传递变量，可以用 GPIO_SET_1bit 或者 gpio_get ）

展开举例

```
20. GPIO_SET(PORTA,27,1);  //PA27 输出高电平 ，展开后：GPIO_SET_1(PORTA,27);
21. GPIO_SET(PORTA,26,0);  //PA26 输出低电平 ，展开后：GPIO_SET_0(PORTA,26);
```

GPIO_TURN

定义

```
22. #define GPIO_TURN(PORTx,n)       (GPIO_PDOR_REG(GPIOx[PORTx]) ^=  (1<<n))
```

功能说明

翻转指定管脚的输出电平

传递参数

参考形参变量的命名及其作用。

展开举例

```
23. GPIO_TURN(PORTA,17);   //翻转输出电平，如果本来 PTA=1，执行后则 PTA=0。
24.                        //展开后：(GPIO_PDOR_REG(GPIOx[PORTA]) ^=  (1<<17));
```

GPIO_Get

定义

```
25. #define GPIO_Get(PORTx,n)          ((GPIO_PDIR_REG(GPIOx[PORTx])>>n)&0x1)
```

功能说明

读取引脚状态

传递参数

参考形参变量的命名及其作用。

展开举例

```
26. a=GPIO_Get(PORTA,17);        //读取 PTA17 引脚状态,保存在变量 a 里
27.                              // 展开后：((GPIO_PDIR_REG(GPIOx[PORTA])>>17)&0x1)
```

GPIO_SET_nbit

这个是举例说明，n 需要替换为库里提供的 1、2、4、8、16 。

以 n 替换为 8 举例：

定义

```
28. #define GPIO_SET_8bit(PORTx,n,data)    GPIO_PDOR_REG(GPIOx[(PORTx)])=\
29.                 (( GPIO_PDOR_REG(GPIOx[(PORTx)])& ~(0xff<<(n)) )|(data<<(n)))
```

读取 IO 管脚状态

清除要设置电平的 8 位状态

设置电平的 8 位状态为 data

功能说明

设置相邻的 8 位引脚输出状态（n 为最低位引脚号）。

传递参数

参考形参变量的命名及其作用。

展开举例

```
30. GPIO_SET_8bit(PORTA,17,0xff);  //设置从 PTA24 到 PTA17 共 8 位管脚的状态为 0xff
31.                     // 展开后：GPIO_PDOR_REG(GPIOx[PORTA])=\
32.                       (( GPIO_PDOR_REG(GPIOx[PORTA])& ~(0xff<<8) )|(0xff<<8));
```

GPIO_GET_nbit

这个是举例说明，n 需要替换为库里提供的 1、2、4、8、16 。

以 n 替换为 8 举例：

定义

```
#define GPIO_GET_8bit(PORTx,n)          GPIO_PDOR_REG(GPIOx[PORTx])=\
                                 (( GPIO_PDOR_REG(GPIOx[(PORTx)])>>(n)) & 0xff)
```

读取 IO 管脚状态

右移 n 位，即此时 PORTx_n 在 0 位

只保留最低 8 位

功能说明

读取相邻的 8 位引脚输出状态（n 为最低位引脚号）。

传递参数

参考形参变量的命名及其作用。

展开举例

```
a= GPIO_GET_8bit(PORTA,17);   //读取从 PTA24 到 PTA17 共 8 位管脚的状态
                              // 展开后：GPIO_PDOR_REG(GPIOx[PORTA])=
                                \(( GPIO_PDOR_REG(GPIOx[PORTA])>>17 ) & 0xff);
```

GPIO 测试例程

51 编程风格的 GPIO 实验输出测试

```
/**********************************************************************
*                              野火嵌入式开发工作室
*                          51 编程风格的 GPIO 实验输出测试
*
*  实验说明：野火 GPIO 实验，利用 LED 来显示电平高低
*
*  实验操作：无
*
*  实验效果：LED0 每隔 500ms
*
*  实验目的：明白如何用 51 编程风格来设置 IO 口电平
*
*  修改时间：2012-2-28      已测试
*
*  备    注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
**********************************************************************/
void  main(void)
{
    gpio_init  (PORTD,15,GPO,HIGH);          //初始化 PTD15 ：输出高电平 ,即 初始化 LED0，灭

    while(1)
    {
      PTD15_OUT=0;

      time_delay_ms(500);                 //延时 500ms

      PTD15_OUT=1;

      time_delay_ms(500);                 //延时 500ms
    }
}
```

51 编程风格的 GPIO 实验输入输出测试

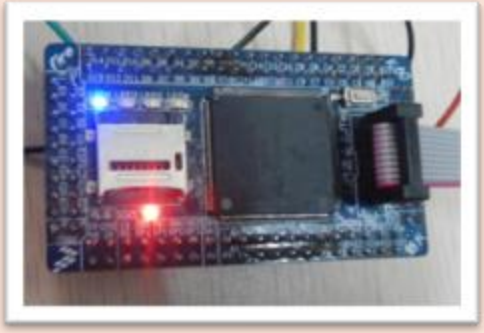
```
/**********************************************************************
```

```
2.  *                              野火嵌入式开发工作室
3.  *                          51 编程风格的 GPIO 实验输入输出测试
4.  *
5.  *  实验说明：野火 51 编程风格的 GPIO 实验，利用 LED 来显示电平高低
6.  *
7.  *  实验操作：短接 PTD15 和 PTD14
8.  *
9.  *  实验效果：LED0、LED1  同时闪烁，间隔 500ms
10. *
11. *  实验目的：明白如何使用 51 编程风格的 GPIO 操作
12. *
13. *  修改时间：2012-2-28       已测试
14. *
15. *  备     注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
16. ******************************************************************************/
17. void  main(void)
18. {
19.      gpio_init   (PORTD,15,GPO,HIGH);          //初始化 PTD15 ：输出高电平 ,即 初始化 LED0，灭
20.      gpio_init   (PORTD,14,GPI,HIGH);          //初始化 PTD14 ：输出高电平 ,即 初始化 LED1，灭
21.
22.      while(1)
23.      {
24.        //DDRD14 = 0;                           //PTD 设为输入（初始化已经设置了，这里就不设置了）
25.
26.        if( PTD14_IN == 1 )
27.        {
28.          PTD15_OUT    =    0;                  //如果 PTD14 输入为 1，即 PTD15 输出为 1 ，那就设置 PTD15 输出为
0
29.        }
30.        else
31.        {
32.          PTD15_OUT    =    1;                  //如果 PTD14 输入为 0，即 PTD15 输出为 0 ，那就设置 PTD15 输出为
1
33.        }
34.        time_delay_ms(500);                    //延时 500ms
35.      }
36. }
```

GPIO 实验简单测试

```
38. /******************************************************************************
39. *                                野火嵌入式开发工作室
40. *                                  GPIO 实验简单测试
41. *  实验说明：野火 GPIO 实验，利用 LED 来显示电平高低
42. *
43. *  实验操作：无
44. *
45. *  实验效果：LED0 每隔 500ms 闪烁
46. *
47. *  实验目的：明白如何设置 IO 口电平
48. *
49. *  修改时间：2012-2-28       已测试
50. *
51. *  备     注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
52. ******************************************************************************/
53. void  main(void)
54. {
55.      gpio_init   (PORTD,15,GPO,HIGH);          //初始化 PTD15 ：输出高电平 ,即 初始化 LED0，灭
56.
57.      while(1)
58.      {
59.          gpio_set(PORTD,15,LOW);               //设置 PTD15 输出 低电平 ，即 LED0 亮
60.
61.          time_delay_ms(500);                   //延时 500ms
62.
63.          gpio_set(PORTD,15,HIGH);              //设置 PTD15 输出 高电平 ，即 LED0 灭
64.
65.          time_delay_ms(500);                   //延时 500ms
66.      }
67. }
```

实验效果：



可以看到 LED0 在闪烁。

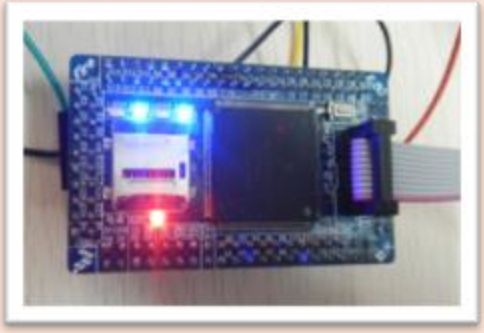
注：图片为旧版本，LED1~3，分别接 PTD15~PTD12。图片的 LED1~LED4 即为新版本的 LED0~3.

GPIO 实验并行读写测试

```
/**********************************************************************
*                                野火嵌入式开发工作室
*                                GPIO 实验并行读写测试
*  实验说明：野火 GPIO 实验，利用 LED 来显示电平高低
*
*  实验操作：PTD15 和 PTD13 短接，PTD14 和 PTD12
*
*  实验效果：LED0、LED2 和 LED1、LED3 轮流间隔 500ms 闪烁
*
*  实验目的：明白如何并行读写数据
*
*  修改时间：2012-3-6       已测试
*
*  备    注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
**********************************************************************/
void  main(void)
{
    u32  value;                                     //保存读取 PTD15~PTD12 的值
    gpio_init  (PORTD,15,GPO,HIGH);                  //初始化 PTD15 ：输出高电平 ,即 初始化 LED0,
灭
    gpio_init  (PORTD,14,GPO,LOW);                   //初始化 PTD14 ：输出高电平 ,即 初始化 LED1,
灭
    gpio_init  (PORTD,13,GPI,HIGH);                  //初始化 PTD13 ：输入
    gpio_init  (PORTD,12,GPI,HIGH);                  //初始化 PTD12 ：输入

    while(1)
    {
        value = GPIO_GET_2bit(PORTD,12);            //读取 PTD13~PTD12 的值

        if(value & (~0x0f))
            while(1);                //如果高位非 0，则死循环。证明 GPIO_GET_4bit 返回的是 4bit 的数据
        else
        {
            GPIO_SET_2bit(PORTD,14,(~value)& 0x03 );     //把读取回来的那两位值取反，清掉高位
        }
        time_delay_ms(500);                 //延时 500ms
    }
}
```

本来这个宏定义就已经会自动清最高位，这里只是示范而加多“& 0x03”

实验效果：

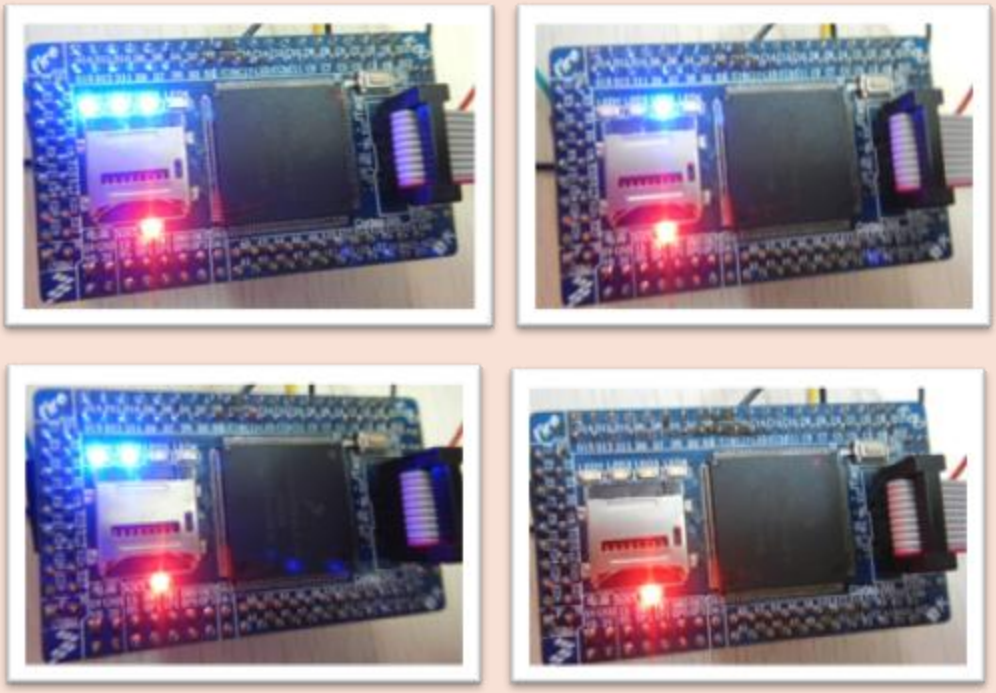


GPIO 实验综合测试

```
/**********************************************************************
*                                野火嵌入式开发工作室
*                                  GPIO 实验综合测试
*  实验说明：野火 GPIO 实验，利用 LED 来显示电平高低
*
*  实验操作：无
*
*  实验效果：LED1 和 LED2 同时亮灭，因为 LED2 的亮灭都是读取 LED1 来控制的。
*            每次 LED1 亮，LED3 都会翻转亮灭。
*
*            可以看到 LED 的亮灭顺序为：
*             LED1   LED2   LED3
*              √      √      √
*              ×      ×      √
*              √      √      ×
*              ×      ×      ×
*              ...                        循环下去
*
*  实验目的：测试 gpio_init、gpio_get、gpio_turn 函数
*
*  修改时间：2012-2-28      已测试
*
*  备    注：野火 Kinetis 开发板的 LED1~4 ，分别接 PTD15~PTD12 ，低电平点亮
**********************************************************************/
void  main(void)
{
    u8  level;          //用来保存读取 PTD15 的电平
    u8  n;              //用来循环计数

    gpio_init  (PORTD,15,GPO,HIGH);          //初始化 PTD15 ：输出高电平 ,即 初始化 LED1，灭
    gpio_init  (PORTD,14,GPO,HIGH);          //初始化 PTD14 ：输出高电平 ,即 初始化 LED2，灭
    gpio_init  (PORTD,13,GPO,HIGH);          //初始化 PTD13 ：输出高电平 ,即 初始化 LED3，灭

    while(1)
    {
        for(n=LOW;n<=HIGH;n++)
        {
            gpio_set(PORTD,15,n);           //设置 PTD15 输出 电平 n ，
                                            //即 n 为控制 LED1 亮灭变量
                                            //这步可以用 PTD15_OUT = n 代替

            level = gpio_get(PORTD,15);     //读取 PTD15 电平
                                            //这步可以用 level=PTD15_IN 代替

            gpio_set(PORTD,14,level);       //设置 PTD14 的电平 设为  读取到 PTD15 的电平
                                            //这步可以用 PTD12_OUT = level 代替

            if(level == LOW)
            {
               gpio_turn(PORTD,13);         //每次读取到 PTD15 为低电平，即 LED1 亮，
```

```
51.                                                 //就翻转 PTD13 电平，即亮灭 LED3
52.                                                 //这步可以用 PTD13_OUT = ~PTD13_OUT 代替
53.                 }
54.
55.                 time_delay_ms(1000);            //延时 1s
56.             }
57.         }
58.     }
```

实验效果：



LED 模块

快速入门：LED 库使用方法

形参变量的命名及其作用

形参变量	作用	取值
LEDn	LED 号	PORTA，PORTB，PORTC，PORTD，PORTE
LED_status	LED 状态	LED_ON ，LED_OFF

常用枚举列表

名称	功能说明
LEDn	LED 号
LED_status	LED 状态

函数列表

函数名称	函数功能
LED_init	初始化 LED
led	翻转 LED 状态
LED_turn	设置 LED 亮灭
water_lights	测试用的流水灯

宏定义列表

宏定义名词	功能说明
LED_PORT	设置 LED 端口号
LED_INIT	初始化 LED
LED_TURN	翻转 LED 状态
LED	设置 LED 亮灭

枚举详解

LEDn

枚举定义

```
1.  enum LEDn
2.  {
3.    LED0     =       12,                 //对应的引脚号
4.    LED1     =       13,
5.    LED2     =       14,
```

```
    LED3     =       15
};
```

枚举作用

用来定义 LED 对应的引脚号

备注：LED_PORT 宏定义 LED 所在的端口号。

LED_status

枚举定义

```
enum LED_status
{
  LED_ON  =   0,          //灯亮(对应低电平)
  LED_OFF =   1           //灯暗(对应高电平)
};
```

枚举作用

用来定义 LED 亮暗的高低电平

函数详解

LED_init

函数原型

```
void    LED_init(void);
```

功能说明

初始化 LED。

---

led

函数原型

```
void    led(LEDn,LED_status);
```

功能说明

设置 LED 亮灭

传递参数

参考形参变量的命名及其作用。

调用例子

```
led(LED1,LED_ON);                    // 设置 LED1 亮
```

LED_turn

函数原型

```
void     LED_turn(LEDn);
```

功能说明

翻转 LED 状态

传递参数

参考形参变量的命名及其作用。

调用例子

```
LED_turn(LED1);                      //翻转 LED1 状态，假设 LED1 本来亮着，则变暗
```

water_lights

函数原型

```
void     water_lights(void);
```

功能说明

用于测试或者指示的流水灯函数

调用例子

```
water_lights();                      //四个 LED 流水灯
```

LED_PORT

定义

```
#define    LED_PORT       PORTE
```

功能说明

定义 LED 所在的端口，根据个人的开发板外设不同而修改。

LED_INIT

定义

```
#define LED_INIT()                   gpio_init(LED_PORT,LED0,GPO,LED_OFF);\
                                     gpio_init(LED_PORT,LED1,GPO,LED_OFF);\
                                     gpio_init(LED_PORT,LED2,GPO,LED_OFF);\
                                     gpio_init(LED_PORT,LED3,GPO,LED_OFF)
```

```
25.                                          // LED 初始化      宏定义     效率高
```

功能说明

初始化 LED

LED_TURN

定义

```
26. #define LED_TURN(LEDn)                  gpio_turn(LED_PORT,LEDn)
```

功能说明

翻转 LED 状态

传递参数

参考形参变量的命名及其作用。

展开举例

```
27. a=LED_TURN(LED1);        //翻转 LED1 状态，假设 LED1 本来亮着，则变暗
```

LED

定义

```
28. #define LED(LEDn,LED_status)    GPIO_SET_1bit(LED_PORT,LEDn,LED_status)
```

功能说明

设置 LED 亮灭

传递参数

参考形参变量的命名及其作用。

展开举例

```
29. LED(LED1,LED_ON);                  //点亮 LED1
30.                                    //展开后：GPIO_SET_1bit(LED_PORT,LED1,LED_ON)
```

LED 综合测试例程

LED 综合测试例程

```
31. /*****************************************************************************
32. *                                  野火嵌入式开发工作室
33. *                                    LED 实验综合测试
```

```
34. *  实验说明：野火 LED 实验
35. *
36. *  实验操作：无
37. *
38. *  实验效果：LED0 和 LED1 同时间隔 500ms 亮灭
39. *
40. *  实验目的：测试 led_init、led、led_turn 函数
41. *
42. *  修改时间：2012-2-28      已测试
43. *
44. *  备     注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
45. ********************************************************************************/
46. void  main(void)
47. {
48.     LED_init();                                 //初始化 4 个 LED
49.
50.     while(1)
51.     {
52.         led(LED0,LED_ON);                       //LED0 亮
53.         LED_turn(LED1);                         //LED1 翻转
54.
55.         time_delay_ms(500);                     //延时 500ms
56.
57.         led(LED0,LED_OFF);                      //LED0 灭
58.         LED_turn(LED1);                         //LED1 翻转
59.
60.         time_delay_ms(500);                     //延时 500ms
61.     }
62. }
```

EXTI 外部 GPIO 中断例程

快速入门：EXTI 库使用方法

形参变量的命名及其作用

形参变量	作用	取值
PORTx	端口名	PORTA，PORTB，PORTC，PORTD，PORTE
n	引脚号	0~31
exti_cfg	exti 配置	zero_down ， //低电平触发，内部下拉 rising_down ， //上升沿触发，内部下拉 falling_down ， //下降沿触发，内部下拉 either_down ， //跳变沿触发，内部下拉 one_down ， //高电平触发，内部下拉 zero_up ， //低电平触发，内部上拉 rising_up ， //上升沿触发，内部上拉 falling_up ， //下降沿触发，内部上拉 either_up ， //跳变沿触发，内部上拉 one_up //高电平触发，内部上拉

常用枚举列表

名称	功能说明
exti_cfg	exti 配置

函数列表

函数名称	函数功能
exti_init	初始化 exti

枚举详解

exti_cfg

枚举定义

```
1.  typedef enum exti_cfg
2.  {
3.    zero_down      = 0x08u,      //低电平触发，内部下拉
4.    rising_down    = 0x09u,      //上升沿触发，内部下拉
5.    falling_down   = 0x0Au,      //下降沿触发，内部下拉
6.    either_down    = 0x0Bu,      //跳变沿触发，内部下拉
7.    one_down       = 0x0Cu,      //高电平触发，内部下拉
```

```
8.
9.    //用最高位标志上拉和下拉
10.   zero_up        = 0x88u,      //低电平触发，内部上拉
11.   rising_up      = 0x89u,      //上升沿触发，内部上拉
12.   falling_up     = 0x8Au,      //下降沿触发，内部上拉
13.   either_up      = 0x8Bu,      //跳变沿触发，内部上拉
14.   one_up         = 0x8Cu       //高电平触发，内部上拉
15. }exti_cfg;
```

枚举作用

配置触发中断模式和 IO 口上拉下拉。

---

exti_init

函数原型

```
16. void  exti_init(PORTx, u8 n,exti_cfg);
```

功能说明

初始化 exti，配置触发中断的条件。

传递参数

参考形参变量的命名及其作用。

调用例子

```
17. void  exti_init(PORTA,17,rising_down);//下拉，上升沿触发中断
```

EXTI 综合测试例程

首先是写个 main 函数初始化 exti:

```
18. /**********************************************************************
19. *                              野火嵌入式开发工作室
20. *                           EXTI 外部中断实验综合测试
21. *
22. *  实验说明：野火 EXTI 外部中断实验
23. *            利用 PTA27 产生方波，触发 PTA26 外部中断，中断服务函数闪烁 LED0。
24. *
25. *            PTA26 是 PORTA 端口，外部触发中断是 PORTA 中断：PORTA_IRQHandler
26. *            我们需要在 isr.c 里编写 PORTA_IRQHandler 中断服务函数
27. *            在 isr.h 里面重新宏定义中断号，重映射中断向量表里的中断函数地址
28. *
29. *  实验操作：用跳线短接 PTA27 和 PTA26
30. *
31. *  实验效果：LED0 闪烁（中断服务函数里控制）
32. *
33. *  实验目的：测试 exti_init 函数
34. *
35. *  修改时间：2012-2-28      已测试
```

```
*
*  备     注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
*************************************************************************/
void main()
{
    LED_INIT();                     //初始化 LED，PORTA_IRQHandler 中断用到 LED

    gpio_init(PORTA,27,GPO,1);    //初始化 gpio ， PTA27 设为输出，以便产生方波

    exti_init(PORTA,26,rising_down);
                                  //PORTA26 端口外部中断初始化 ，上升沿触发中断，内部下拉

    while(1)
    {
        GPIO_TURN(PORTA,27);      //翻转 PTA27，即产生方波

        time_delay_ms(500);       //延时 0.5s
    }
}
```

在 isr.h 里重定向中断向量表的中断函数指针（我们上面用的是 PORTA 口，所以触发的是 PORTA 中断）和声明中断服务函数：

```
#undef  VECTOR_103
#define VECTOR_103    PORTA_IRQHandler     //PORTA 中断

extern void PORTA_IRQHandler();            //PORTA 中断服务函数
```

在 isr.c 里编写中断服务函数：

```

/*************************************************************************
*                              野火嵌入式开发工作室
*
*  函数名称：PORTA_IRQHandler
*  功能说明：PORTA 端口中断服务函数
*  参数说明：无
*  函数返回：无
*  修改时间：2012-1-25     已测试
*  备     注：引脚号需要自己初始化来清除
*************************************************************************/
void PORTA_IRQHandler()
{
    u8  n=0;       //引脚号

/*  根据自己的引脚号，自己编写，这里给出 n= 0 的模版，即 PTAn 产生外部中断
 *  也给出 n=26 例子 ，自行修改 n 即可  ，添加用户任务就行
 */

//========================= n = 0 模版 ======================================
    n=0;
    if(PORTA_ISFR & (1<<n))             //PTA0 触发中断
    {
        PORTA_ISFR  |= (1<<n);          //写 1 清中断标志位
        /*  以下为用户任务  */


        /*  以上为用户任务  */
```

```
30.      }
31.  //=============================================================================
32.
33.      n=26;
34.      if(PORTA_ISFR & (1<<n))              //PTA26 触发中断
35.      {
36.          PORTA_ISFR  |= (1<<n);           //写 1 清中断标志位
37.          /*  以下为用户任务  */
38.
39.          LED_turn(LED0);                  //LED0 反转
40.
41.          /*  以上为用户任务  */
42.      }
43.  }
```

这里 n=0 为模版，给用户来做参考，添加自己的用户代码，这部分可以删掉。

n = 26 才是我们要的执行中断函数内容，因为 PTA26 产生中断。这些用 LED 闪烁来指示执行了中断服务函数。

UART 模块

快速入门：UART 库使用方法

默认通道管脚表格，非常方便查找。

形参变量的命名及其作用

形参变量	作用	取值
UARTn	串口端口号	//初始化默认配置　--TXD--　--RXD-- UART0,　//　PTD7　PTD6 UART1,　//　PTC4　PTC3 UART2,　//　PTD3　PTD2 UART3,　//　PTC17　PTC16 UART4,　//　PTE24　PTE25 UART5　//　PTE8　PTE9
baud	波特率	如 9600、19200、56000、115200 等（不推荐太大）
ch	字符	
str	字符串	
buff	缓冲区	
len	长度	

常用枚举列表

名称	功能说明
UARTn	串口端口号

函数列表

函数名称	函数功能
uart_init	初始化串口
uart_getchar	无限时间等待接受 1 个字节
uart_pendchar	有限时间等待接收一个字符
uart_pendstr	有限时间等待接收字符串
uart_query	查询是否接受到一个字节
uart_putchar	发送 1 个字节
uart_sendN	发送 n 个字节
uart_sendStr	发送字符串
uart_irq_EN	开串口接收中断
uart_irq_DIS	关串口接收中断

宏定义列表

宏定义名词	功能说明
UART_IRQ_EN	开串口接收中断
UART_IRQ_DIS	关串口接收中断

枚举详解

UARTn

枚举定义

```
59. typedef enum  UARTn
60. {     //初始化默认配置   --TXD--       --RXD-- 可以复用其他通道，请自行修改 uart_init
61.   UART0,     //           PTD6          PTD7
62.   UART1,     //           PTC4          PTC3
63.   UART2,     //           PTD3          PTD2
64.   UART3,     //           PTC17         PTC16
65.   UART4,     //           PTE24         PTE25
66.   UART5      //           PTE8          PTE9
67. }UARTn;
```

枚举作用

用来定义串口端口号

函数详解

uart_init

函数原型

```
68. void uart_init (UARTn,u32 baud);
```

功能说明

初始化 uartx 模块

调用例子

```
69. uart_init (UART1,19200);    //初始化 串口 1 波特率为 19200
```

uart_getchar

函数原型

```
70. char uart_getchar    (UARTn);
```

功能说明

无限等待接受 1 个字节。

函数返回

一个字符

调用例子

```
71. ch= uart_getchar    (UART1);                    //等待串口 1 发送数据，接收到后保存在变量 ch
```

uart_pendchar

uart_pendstr 有限时间等待接收字符字符串，也与这类似。

函数原型

```
72. char uart_pendchar   (UARTn,char * ch);
```

功能说明

有限时间等待接收一个字符

传递参数

参考形参变量的命名及其作用。

函数返回

时间到了也接收不了就返回 0，并把 ch 指针指向内容设为 0；成功接收到就返回 1，并把接收到的数据保存在 ch 指针指向的地址。

调用例子

```
73. char ch;
74. if(uart_pendchar(UARTn,&ch)) printf("接收成功，结束数据为：%c\n",ch);
```

uart_query

函数原型

```
75. int uart_query      (UARTn);
```

功能说明

查询是否接收到一个字节

传递参数

参考形参变量的命名及其作用。

函数返回

1 表示接收到一个字节了；　0　表示没有接收到

调用例子

```
76. if(uart_query(UART1)) printf("接收到了\n",ch);
```

uart_putchar

函数原型

```
77. void uart_putchar    (UARTn, char ch);
```

功能说明

发送 1 个字节。

传递参数

参考形参变量的命名及其作用。

调用例子

```
78. uart_putchar (UART1,'F');                    //发送一个字符'F'
```

uart_sendN

函数原型

```
79. void uart_sendN      (UARTn ,uint8* buff,uint16 len);
```

功能说明

发送 len 个字节，包括 NULL 也会发送。

传递参数

参考形参变量的命名及其作用。

调用例子

```
80. uart_sendN (UART1,"fire\n",5);                    //发送 5 个字符
```

uart_sendStr

函数原型

```
81. void uart_sendStr    (UARTn ,const u8* str);
```

功能说明

发送字符串。

传递参数

参考形参变量的命名及其作用。

调用例子

```
82. uart_sendStr (UART1,"uart_str 函数发送数据");          //发送字符串
```

uart_irq_EN

函数原型

```
83. void uart_irq_EN     (UARTn);
```

功能说明
开串口接收中断。

传递参数
参考形参变量的命名及其作用。

调用例子
```
84. uart_irq_EN     (UART1);                    //开串口 1 接收中断
```

uart_irq_DIS

函数原型
```
85. void uart_irq_DIS    (UARTn);
```

功能说明
关串口接收中断。

传递参数
参考形参变量的命名及其作用。

调用例子
```
86. uart_irq_DIS    (UART1);                    //关串口 1 接收中断
```

宏定义

UART_IRQ_EN

定义
```
87. #define UART_IRQ_EN(UARTn)    UART_C2_REG(UARTx[UARTn])|=UART_C2_RIE_MASK;\
88.                                                         enable_irq((UARTn<<1)+45)
```

功能说明
开串口接收中断

传递参数
参考形参变量的命名及其作用。

展开举例
```
89. UART_IRQ_EN(UART1);  //开串口接收中断
```

UART_IRQ_DIS

定义

```
90. #define UART_IRQ_DIS(UARTn)  UART_C2_REG(UARTx[UARTn])&=~UART_C2_RIE_MASK;\
91.                                                          disable_irq((UARTn<<1)+45)
```

功能说明

关串口接收中断

传递参数

参考形参变量的命名及其作用。

展开举例

```
92. UART_IRQ_DIS(UART1);                    //宏定义关接收引脚的 IRQ 中断
```

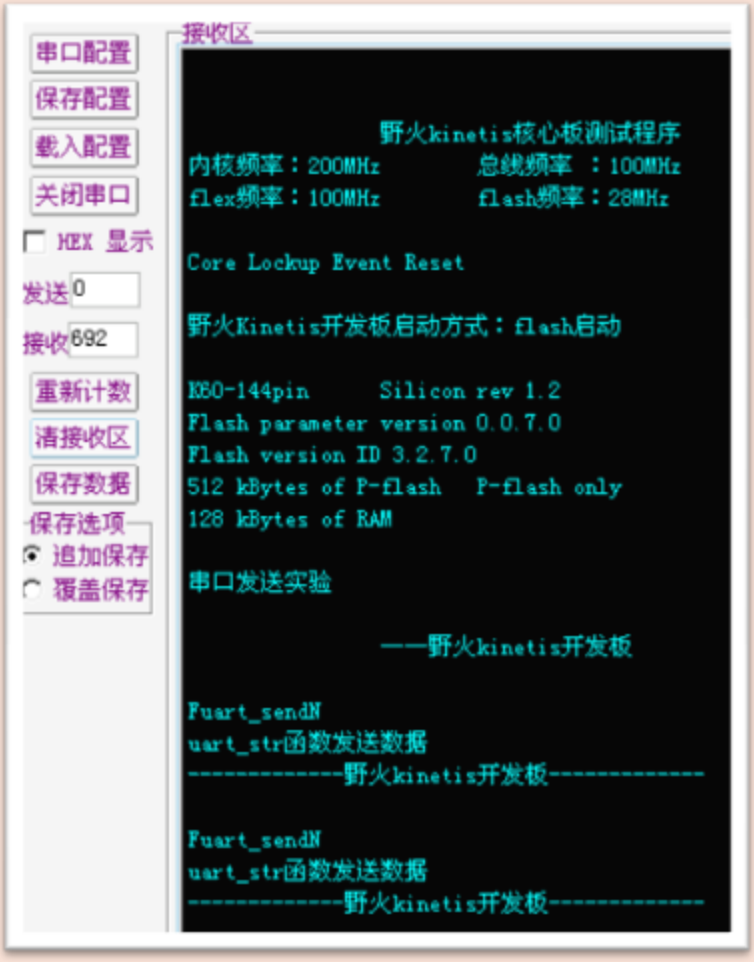
UART 综合测试例程

串口发送例程

```
1.
2.  /**********************************************************************
3.  *                              野火嵌入式开发工作室
4.  *                                串口发送实验测试
5.  *
6.  *  实验说明：野火串口发送实验
7.  *            野火串口默认为： UART1，TX 接 PTC4，RX 接 PTC3
8.  *            k60_fire.h 里定义了 printf 函数的输出设置：
9.  *                #define FIRE_PORT           UART1
10. *                #define FIRE_BAUD           19200
11. *            这里的串口发送实验也使用 UART1 ，这样方便显示。
12. *
13. *  实验操作：打开串口助手 ，设置波特率为 19200 。
14. *            串口线（经过 MAX3232 电平转换）：TX 接 PTC4，RX 接 PTC3
15. *
16. *  实验效果：在串口助手里，可以看到输出如下信息：
17. *
18. *                        野火 kinetis 核心板测试程序
19. *                内核频率：200MHz     总线频率 ：66MHz
20. *                flex 频率：66MHz    flash 频率：28MHz
21. *
22. *                Software Reset
23. *
24. *                K60-144pin      Silicon rev 1.2
25. *                Flash parameter version 0.0.7.0
26. *                Flash version ID 3.2.7.0
27. *                512 kBytes of P-flash  P-flash only
28. *                128 kBytes of RAM
29. *
30. *                串口发送实验
31. *
32. *                                —野火 kinetis 开发板
33. *
34. *                Fuart_sendNuart_str 函数发送数据
35. *                -------------野火 kinetis 开发板-------------
36. *
37. *
38. *  实验目的：测试串口发送的各个函数：printf、uart_putchar、uart_sendN、uart_sendStr
39. *
```

```
*  修改时间：2012-2-29       已测试
*
*  备     注：printf 函数的底层是调用 uart_putchar 来发送。
*********************************************************************/
void  main(void)
{
    uart_init(UART1, 19200);                                    //初始化串口

    printf("串口发送实验\n\n");                                  //使用 printf 来发送
    printf("\t\t--野火 kinetis 开发板\n\n");

    while(1)
    {
        uart_putchar (UART1,'F');                               //发送一个字符'F'
        uart_sendN (UART1,"uart_sendN\n",11);                   //发送 11 个字符
        uart_sendStr (UART1,"uart_str 函数发送数据");            //发送字符串
        printf("\n-------------野火 kinetis 开发板-------------\n\n");  //使用 printf 来发送
        time_delay_ms(500);                                     //延时
    }
}
```

实验效果：



注：图片跟注释的实验效果描述有所差别，是因为图片是在修改启动代码后补上去的。

串口查询接收例程

```
/*********************************************************************
*                                 野火嵌入式开发工作室
*                                 串口查询接收实验测试
*
*  实验说明：野火串口查询接收实验
```

```
 *             野火串口默认为：  UART1，TX 接 PTC4，RX 接 PTC3
 *             k60_fire.h 里定义了 printf 函数的输出设置：
 *                 #define FIRE_PORT           UART1
 *                 #define FIRE_BAUD           19200
 *             这里的串口接收实验也使用 UART1 ，这样就可以使用 printf 函数。
 *
 *  实验操作：打开串口助手 ，设置波特率为 19200 。
 *             串口线（经过 MAX3232 电平转换）：TX 接 PTC4，RX 接 PTC3 。
 *             运行程序后，要在串口助手里，根据提示才发送数据，才能接收到数据。
 *
 *
 *  实验效果：在串口助手里，可以看到输出如下信息：
 *
 *                       野火 kinetis 核心板测试程序
 *       内核频率：200MHz 总线频率 ：66MHz
 *       flex 频率：66MHz     flash 频率：28MHz
 *
 *       Software Reset
 *
 *       K60-144pin      Silicon rev 1.2
 *       Flash parameter version 0.0.7.0
 *       Flash version ID 3.2.7.0
 *       512 kBytes of P-flash   P-flash only
 *       128 kBytes of RAM
 *
 *       串口查询接收实验
 *
 *              ——野火 kinetis 开发板
 *
 *       请发送数据：
 *       你发送的字符为：f
 *
 *       快点发送字符哦，不等你的哦：
 *       哈哈，赶得及哦！你发送的字符为：i
 *
 *       快点发送字符串哦，不等你的哦：
 *       哈哈，赶得及哦！你发送的字符为：fire
 *
 *
 *  实验目的：测试串口查询接收的函数
 *
 *  修改时间：2012-2-29      已测试
 *
 *  备    注：
 ********************************************************************************/

void  main(void)
{
    char ch;
    char str[20];

    uart_init(UART1,19200);                 //初始化串口 1，波特率为 19200 ,波特率太大，容易不稳定
    printf("串口查询接收实验\n\n");          //发送提示信息
    printf("\t\t——野火 kinetis 开发板\n\n");

    while(1)
    {
        /**********     测试无限等待接收     **********/
        printf("请发送数据：\n");
        ch=uart_getchar(UART1);                    //从串口 1 中等待接收数据
        printf("你发送的字符为：%c\n\n",ch);      //从串口 1 中发送出去

        time_delay_ms(1000);                       //延时 1s

        /**********     测试有限等待接收一个字符     **********/
        printf("快点发送字符哦，不等你的哦：\n");
        if(uart_pendchar (UART1,&ch))
            printf("哈哈，赶得及哦！你发送的字符为：%c\n\n",ch);         //从串口 1 中发送出去
```

```
74.          else
75.              printf("o(￣へ￣)o 唉，赶不及了！收不到你的数据。\n\n");      //从串口 1 中发送出去
76.
77.          time_delay_ms(1000);
78.
79.
80.          /**********     测试有限等待接收字符串     **********/
81.          printf("快点发送字符串哦，不等你的哦：\n");
82.          if(uart_pendstr(UART1,str))
83.              printf("哈哈，赶得及哦！你发送的字符为：%s\n\n",str);      //从串口 1 中发送出去
84.          else
85.              printf("o(￣へ￣)o 唉，赶不及了！收不到你的数据。\n\n");      //从串口 1 中发送出去*/
86.      }
87.  }
```

实验效果：

串口中断接收例程

用到中断服务，那就要写中断服务函数，先在 main.c 里写 main 函数：

```
1.  /*********************************************************************
2.  *                              野火嵌入式开发工作室
3.  *                              串口中断接收实验测试
4.  *
5.  *  实验说明：野火串口中断接收实验
6.  *            野火串口默认为： UART1，TX 接 PTC4，RX 接 PTC3
7.  *            k60_fire.h 里定义了 printf 函数的输出设置：
8.  *                #define FIRE_PORT           UART1
9.  *                #define FIRE_BAUD           19200
10. *            这里的串口接收实验也使用 UART1 ，这样就可以使用 printf 函数。
11. *            这里用到串口 1 的中断服务函数 USART1_IRQHandler();
12. *            我们需要在 isr.c 里编写 USART1_IRQHandler 中断服务函数
13. *            在 isr.h 里面重新宏定义中断号，重映射中断向量表里的中断函数地址
14. *
15. *  实验操作：打开串口助手 ，设置波特率为 19200 。
16. *            串口线（经过 MAX3232 电平转换）：TX 接 PTC4，RX 接 PTC3 。
```

```
17.  *              运行程序后，要在串口助手里，根据提示发送数据，才能接收到数据。
18.  *
19.  *
20.  *  实验效果：可以看到流水灯正常运行着。
21.  *              在串口助手里，可以看到输出如下信息：
22.  *
23.  *
24.  *                          野火 kinetis 核心板测试程序
25.  *        内核频率：200MHz 总线频率 ： 66MHz
26.  *        flex 频率：66MHz     flash 频率：28MHz
27.  *
28.  *        Software Reset
29.  *
30.  *        K60-144pin      Silicon rev 1.2
31.  *        Flash parameter version 0.0.7.0
32.  *        Flash version ID 3.2.7.0
33.  *        512 kBytes of P-flash   P-flash only
34.  *        128 kBytes of RAM
35.  *
36.  *        串口中断接收实验
37.  *
38.  *                ——野火 kinetis 开发板
39.  *
40.  *        请发送数据：
41.  *
42.  *        你发送的数据为：f
43.  *        你发送的数据为：i
44.  *        你发送的数据为：r
45.  *        你发送的数据为：e
46.  *
47.  *
48.  *  实验目的：测试串口中断接收
49.  *
50.  *  修改时间：2012-2-29      已测试
51.  *
52.  *  备     注：
53.  ************************************************************************/
54.
55.
56.
57.  /*
58.  * 这里用到串口 1 的中断服务函数 UART1_ISR();
59.  * 接收到数据后就发送出去
60.  */
61.  void  main(void)
62.  {
63.
64.      LED_INIT();                                     //LED 初始化
65.
66.      UART_IRQ_DIS(UART1);                            //串口 1    关接收中断
67.
68.      uart_init(UART1, 19200);                        //初始化串口 1
69.
70.      printf("串口中断接收实验\n\n");
71.      printf("\t\t——野火 kinetis 开发板\n\n");
72.      printf("请发送数据： \n");
73.
74.      UART_IRQ_EN(UART1);                             //串口 1    开接收中断
75.
76.      while(1)
77.      {
78.          water_lights();                             //流水灯,用来指示系统正在运行
79.      }
80.  }
```

然后在 isr.h 里重定向中断向量表的中断函数指针和声明中断服务函数：

```
#undef  VECTOR_063                      //要先取消了，因为在 vectors.h 里默认是定义为 default_isr
#define VECTOR_063    USART1_IRQHandler    //重新定义 63 号中断的 ISR：UART1

extern void USART1_IRQHandler();          //串口 1 中断接收函数
```

最后在 isr.c 里编写中断服务函数：

```
/*********************************************************************
*                              野火嵌入式开发工作室
*
*  函数名称：USART1_IRQHandler
*  功能说明：串口 1 中断 接收 服务函数
*  参数说明：无
*  函数返回：无
*  修改时间：2012-2-14     已测试
*  备     注：
*********************************************************************/
void USART1_IRQHandler(void)
{
    uint8 ch;

    DisableInterrupts;          //关总中断

    //接收一个字节数据并回发
    ch=uart_getchar (UART1);                  //接收到一个数据
    printf("\n 你发送的数据为：%c",ch);       //发送出去


    EnableInterrupts;           //开总中断
}
```

ADC 模块

快速入门：ADC 库使用方法

形参变量的命名及其作用

默认通道管脚表格，非常方便查找。

形参变量	作用	取值
ADCn	ADC 模块号	ADC0，ADC1
ADC_Ch	ADC 通道	见下方详细说明
ADC_nbit	字符	ADC_8bit ， ADC_12bit ， ADC_10bit ， ADC_16bit

```
                //  当 SC1n[DIFF]= 0 为 0
                //  ------ADC0--------对应管脚------|--------ADC1--------对应管脚----
DAD0=0,         //    ADC0_DP0                      ADC1_DP0
DAD1=1,         //    ADC0_DP1                      ADC1_DP1
DAD2=2,         //    PGA0_DP                       PGA1_DP
DAD3=3,         //    ADC0_DP3                      ADC1_DP3

//ADCx_CFG2[MUXSEL] 位决定 ADCx_SEn 通道为 a 或 b.
AD4a=4,         //    保留                          ADC1_SE4a    -- PTE0
AD5a=5,         //    保留                          ADC1_SE5a    -- PTE1
AD6a=6,         //    保留                          ADC1_SE6a    -- PTE2
AD7a=7,         //    保留                          ADC1_SE7a    -- PTE3
//也是 4、5、6、7
AD4b=AD4a,      //    ADC0_SE4b    -- PTC2          ADC1_SE4b    -- PTC8
AD5b=AD5a,      //    ADC0_SE5b    -- PTD1          ADC1_SE5b    -- PTC9
AD6b=AD6a,      //    ADC0_SE6b    -- PTD5          ADC1_SE6b    -- PTC10
AD7b=AD7a,      //    ADC0_SE7b    -- PTD6           ADC1_SE7b   -- PTC11

AD8=8,          //    ADC0_SE8     -- PTB0          ADC1_SE8     -- PTB0
AD9=9,          //    ADC0_SE9     -- PTB1          ADC1_SE9     -- PTB1
AD10=10,        //    ADC0_SE10    -- PTA7          ADC1_SE10    -- PTB4
AD11=11,        //    ADC0_SE11    -- PTA8          ADC1_SE11    -- PTB5
AD12=12,        //    ADC0_SE12    -- PTB2          ADC1_SE12    -- PTB6
AD13=13,        //    ADC0_SE13    -- PTB3          ADC1_SE13    -- PTB7
AD14=14,        //    ADC0_SE14    -- PTC0          ADC1_SE14    -- PTB10
AD15=15,        //    ADC0_SE15    -- PTC1          ADC1_SE15    -- PTB11
AD16=16,        //    ADC0_SE16                     ADC1_SE16
AD17=17,        //    ADC0_SE17    -- PTE24         ADC1_SE17    -- PTA17
AD18=18,        //    ADC0_SE18    -- PTE25          VREF Output
AD19=19,        //    ADC0_DM0                      ADC1_DM0
AD20=20,        //    ADC0_DM1                      ADC1_DM1
AD21=21,        //                                  保留
AD22=22,        //
AD23=23,        //    DAC0_OUT-- DAC0_OUT           DAC1_OUT(12-bit)
AD24=24,        //    保留                          保留
AD25=25,        //    保留                          保留
AD26=26,        //    Temperature Sensor (S.E)      Temperature Sensor (S.E)
AD27=27,        //    Bandgap (S.E)                 Bandgap (S.E)
AD28=28,        //    保留                          保留
AD29=29,        //    VREFH (S.E)                   VREFH (S.E)
AD30=30,        //    VREFL                         VREFL
AD31=31         //    禁用 ADC0                     禁用 ADC1
```

注：调用时直接写模块”_”后面的通道号就可以。如 SE16、SE4a 等。
软件触发不支持 B 通道，目前库只支持软件触发。

常用枚举列表

名称	功能说明

ADCn	ADC 模块号
ADC_Ch	ADC 通道
ADC_nbit	精度位数

函数列表

函数名称	函数功能
adc_init	
ad_once	
ad_mid	
ad_ave	
adc_start	
adc_stop	

枚举详解

ADCn

枚举定义

```
1.  //ADC 端口
2.  typedef enum ADCn
3.  {
4.    ADC0,
5.    ADC1
6.  }ADCn;
```

枚举作用

用来定义 ADC 模块号

ADC_Ch

枚举定义

```
1.  //ADC 通道
2.  //提醒，用的时候，可以直接用 DP0  代替 DAD0 ，其他的也类似，因为有个宏定义：#define DP0   DAD0
3.  typedef enum ADC_Ch
4.  {
5.              //SC1n[DIFF]= 0
6.              //  ------ADC0------野火开发板丝印层---|-----ADC1------野火开发板丝印层----
7.    DAD0=0,   //      ADC0_DP0                        ADC1_DP0
8.    DAD1=1,   //      ADC0_DP1                        ADC1_DP1
9.    DAD2=2,   //      PGA0_DP                         PGA1_DP
10.   DAD3=3,   //      ADC0_DP3                        ADC1_DP3
11.
12.   //ADCx_CFG2[MUXSEL] 位决定 ADCx_SEn 通道为 a 或 b.
13.   AD4a=4,   //      保留                             ADC1_SE4a -- PTE0
14.   AD5a=5,   //      保留                             ADC1_SE5a -- PTE1
15.   AD6a=6,   //      保留                             ADC1_SE6a -- PTE2
```

```
    AD7a=7,    //       保留                                ADC1_SE7a -- PTE3
    //也是 4、5、6、7
    AD4b=AD4a, //       ADC0_SE4b    -- PTC2              ADC1_SE4b  -- PTC8
    AD5b=AD5a, //       ADC0_SE5b    -- PTD1              ADC1_SE5b -- PTC9
    AD6b=AD6a, //       ADC0_SE6b    -- PTD5              ADC1_SE6b  -- PTC10
    AD7b=AD7a, //       ADC0_SE7b    -- PTD6              ADC1_SE7b  -- PTC11

    AD8=8,     //       ADC0_SE8     -- PTB0              ADC1_SE8 -- PTB0
    AD9=9,     //       ADC0_SE9     -- PTB1              ADC1_SE9 -- PTB1
    AD10=10,   //       ADC0_SE10    -- PTA7              ADC1_SE10 -- PTB4
    AD11=11,   //       ADC0_SE11    -- PTA8              ADC1_SE11 -- PTB5
    AD12=12,   //       ADC0_SE12    -- PTB2              ADC1_SE12 -- PTB6
    AD13=13,   //       ADC0_SE13    -- PTB3              ADC1_SE13 -- PTB7
    AD14=14,   //       ADC0_SE14    -- PTC0              ADC1_SE14 -- PTB10
    AD15=15,   //       ADC0_SE15    -- PTC1              ADC1_SE15 -- PTB11
    AD16=16,   //       ADC0_SE16                         ADC1_SE16
    AD17=17,   //       ADC0_SE17    -- PTE24             ADC1_SE17 -- PTA17
    AD18=18,   //       ADC0_SE18    -- PTE25             VREF Output
    AD19=19,   //       ADC0_DM0                          ADC1_DM0
    AD20=20,   //       ADC0_DM1                          ADC1_DM1
    AD21=21,   //                                         保留
    AD22=22,   //
    AD23=23,   //       DAC0_OUT(12-bit) -- DAC0_OUT      DAC1_OUT(12-bit)
    AD24=24,   //       保留                              保留
    AD25=25,   //       保留                              保留
    AD26=26,   //       Temperature Sensor (S.E)          Temperature Sensor (S.E)
    AD27=27,   //       Bandgap (S.E)                     Bandgap (S.E)
    AD28=28,   //       保留                              保留
    AD29=29,   //       VREFH (S.E)                       VREFH (S.E)
    AD30=30,   //       VREFL                             VREFL
    AD31=31    //       禁用 ADC0                         禁用 ADC1
}ADC_Ch;
```

注：调用时直接写模块” _”后面的通道号就可以。如 SE16、SE4a 等。
软件触发不支持 B 通道，目前库只支持软件触发。

枚举作用

用来定义 ADC 通道。

ADC_nbit

枚举定义

```
//精度位数
typedef enum ADC_nbit
{
  ADC_8bit   =0x00,
  ADC_10bit  =0x02,
  ADC_12bit  =0x01,
  ADC_16bit  =0x03
}ADC_nbit;
```

枚举作用

用来定义 ADC 模数转换的精度。

函数详解

adc_init

函数原型

```
1.  void      adc_init   (ADCn,ADC_Ch);
```

功能说明

初始化 ADC 模块

调用例子

```
1.  adc_init   (ADC1,SE16);   //初始化 ADC1_SE16 通道
```

ad_once

函数原型

```
2.  u16       ad_once    (ADCn,ADC_Ch,ADC_nbit);
```

功能说明

采样一次一路模拟量的 AD 值

函数返回

返回采样数值。按采集精度来返回不同大小，16 位精度最大返回：65535

调用例子

```
3.  result= ad_once   (ADC1,SE16,ADC_16bit);        //在 ADC1_SE16 上采样一次 16 位精度
```

ad_mid

函数原型

```
4.  u16       ad_mid     (ADCn,ADC_Ch,ADC_nbit);
```

功能说明

采样三次取中值

函数返回

返回采样三次的中值数值。按采集精度来返回不同大小，16 位精度最大返回：65535

调用例子

```
5.  result= ad_mid   (ADC1,SE16,ADC_16bit);   //在 ADC1_SE16 上采样三次的中值，16 位精度
```

ad_ave

函数原型

```
6.  u16       ad_ave     (ADCn,ADC_Ch,ADC_nbit,u8 N);
```

功能说明

采样 N 次的 ADC 值平均值

函数返回

返回采样 N 次的 ADC 值平均值。按采集精度来返回不同大小，16 位精度最大返回：65535

调用例子

```
7.  result= ad_ave     (ADCn,ADC_Ch,ADC_nbit,u8 N);
8.                  //在 ADC1_SE16 上采样 N 次的平均值，16 位精度
```

---

adc_start

函数原型

```
9.  void      adc_start  (ADCn,ADC_Ch,ADC_nbit);
```

功能说明

开始 adc 转换

调用例子

```
10. adc_start  (ADC1,SE16,ADC_nbit);                    //启动 ADC1_SE16 通道采样
```

---

adc_stop

函数原型

```
11. void      adc_stop   (ADCn);
```

功能说明

停止 ADC 模块的 AD 转换，同一个模块的所有通道都会停止。

调用例子

```
12. adc_stop   (ADC1);                                  //停止 ADC1 上采样
```

ADC 综合测试例程

```
13. /*********************************************************************
14. *                              野火嵌入式开发工作室
15. *                               ADC 模数转换实验测试
16. *
```

```
17.  *  实验说明：野火 ADC 模数转换实验，用串口发送转换后结果。
18.  *            野火串口默认为： UART1，TX 接 PTC4，RX 接 PTC3
19.  *            k60_fire.h 里定义了 printf 函数的输出设置：
20.  *                #define FIRE_PORT           UART1
21.  *                #define FIRE_BAUD           19200
22.  *
23.  *  实验操作：这里用 ADC1_SE4a ，所以 ADC1_SE4a 还要接一个 0~3.3V 的可调电路
24.  *            打开串口助手，设置波特率为 19200 。
25.  *            串口线（经过 MAX3232 电平转换）：TX 接 PTC4，RX 接 PTC3
26.  *
27.  *  实验效果：在串口助手里，可以看到输出如下信息：
28.  *
29.  *                      野火 kinetis 核心板测试程序
30.  *       内核频率：200MHz 总线频率 ： 66MHz
31.  *       flex 频率：66MHz     flash 频率：28MHz
32.  *
33.  *       Software Reset
34.  *
35.  *       K60-144pin      Silicon rev 1.2
36.  *       Flash parameter version 0.0.7.0
37.  *       Flash version ID 3.2.7.0
38.  *       512 kBytes of P-flash   P-flash only
39.  *       128 kBytes of RAM
40.  *
41.  *       AD 转换一次的结果为:36983
42.  *       AD 转换三次的中值结果为:37143
43.  *       AD 转换十次的平均值结果为:36912
44.  *
45.  *
46.  *
47.  *  实验目的：测试 ADC 转换的各个结果
48.  *
49.  *  修改时间：2012-2-29      已测试
50.  *
51.  *  备    注：adc.h 有 各个 ADC 通道所对应管脚的表格，方便查看
52.  ********************************************************************/
53.  void  main()
54.  {
55.      u16 ADresult;                          //保存 ADC 转换结果
56.
57.      uart_init(UART1,19200);                 //初始化串口，用来发送转换数据
58.
59.      adc_init(ADC1,SE4a);      //初始化 ADC1_SE4a ,从 adc.h 里的表格就可以看到 ADC1_SE4a 对应为 PTE0
60.
61.      while(1)
62.      {
63.          /****** 读取一次 ******/
64.          ADresult     =    ad_once(ADC1,SE4a,ADC_16bit);        //读取 ADC1_SE4a ，16 位精度
65.          printf("AD 转换一次的结果为:%d\n\n",ADresult);
66.
67.          time_delay_ms(500);                  //延时 500ms
68.
69.          /****** 读取三次，取中值 ******/
70.          ADresult     =    ad_mid(ADC1,SE4a,ADC_16bit);          //读取 ADC1_SE4a ，16 位精度
71.          printf("AD 转换三次的中值结果为:%d\n\n",ADresult);
72.
73.          time_delay_ms(500);                  //延时 500ms
74.
75.          /****** 读取十次次，取平均值 ******/
76.          ADresult     =    ad_ave(ADC1,SE4a,ADC_16bit,10);          //读取 ADC1_SE4a ，16 位精度
77.          printf("AD 转换十次的平均值结果为:%d\n\n",ADresult);
78.
79.          time_delay_ms(500);                  //延时 500ms
80.      }
81.  }
```

FTM PWM 模块

快速入门：PWM 库使用方法

形参变量的命名及其作用

默认通道管脚表格，非常方便查找。

形参变量	作用	取值
FTMn	模块名	FTM0，FTM1，FTM2
CHn	通道号	// --FTM0-- --FTM1-- --FTM2-- CH0, // PTC1 PTA8 PTA10 CH1, // PTC2 PTA9 PTA11 CH2, // PTC3 × × CH3, // PTC4 × × CH4, // PTD4 × × CH5, // PTD5 × × CH6, // PTD6 × × CH7 // PTD7 × ×
freq	频率	
duty	通道占空比	通道占空比为：duty /FTM_PRECISON FTM_PRECISON 为精度宏定义，定义为 100

常用枚举列表

名称	功能说明
FTMn	模块号
CHn	通道号

函数列表

函数名称	函数功能
FTM_PWM_init	初始化 FTM 的边沿 PWM 模式，设置频率和占空比
FTM_PWM_Duty	设置通道占空比,占空比为 （duty / FTM_PRECISON）*100% 。
FTM_PWM_freq	设置 FTM 的频率。设置后，需要重新调用 FTM_PWM_Duty 设置占空比。 建议直接使用 FTM_PWM_init 。

宏定义列表

宏定义名词	功能说明
FTM_PRECISON	精度（影响变量 duty 的取值，通道占空比为：duty /FTM_PRECISON ）

枚举详解

FTMn

枚举定义

```
1.  enum FTMn
2.  {
3.    FTM0,
4.    FTM1,
5.    FTM2
6.  }
```

枚举作用

用来定义端口号

CHn

枚举定义

```
7.  enum CHn
8.  {
9.            //    --FTM0--   --FTM1--   --FTM2--
10.   CH0,     //      PTC1       PTA8       PTA10
11.   CH1,     //      PTC2       PTA9       PTA11
12.   CH2,     //      PTC3        ×          ×
13.   CH3,     //      PTC4        ×          ×
14.   CH4,     //      PTD4        ×          ×
15.   CH5,     //      PTD5        ×          ×
16.   CH6,     //      PTD6        ×          ×
17.   CH7      //      PTD7        ×          ×
18.           // ×表示不存在
19. }CHn;
```

枚举作用

用来定义通道号。

函数详解

FTM_PWM_init

函数原型

```
20. void  FTM_PWM_init(FTMn,CHn,u32 freq,u32 duty);
```

功能说明

初始化 FTM 的边沿 PWM 模式，设置频率和占空比。

调用例子

```
21. FTM_PWM_init(FTM1,CH0,35000,50);//初始化 FTM1_CH0 频率为 35kHZ,占空比为 50%的 PWM
22.                                 //从通道的枚举旁边的注解里，可以查到 FTM1_CH0 对应 PTA8 口
```

FTM_PWM_Duty

函数原型

```
23. void  FTM_PWM_Duty(FTMn,CHn,          u32 duty);
```

功能说明

设置通道占空比,占空比为 （duty / FTM_PRECISON）*100% 。

传递参数

参考形参变量的命名及其作用。

调用例子

```
24. FTM_PWM_Duty(FTM1,CH0,80);        // FTM1_CH0 输出占空比为(80/FTM_PRECISON)*100%
```

FTM_PWM_freq

函数原型

```
25. void  FTM_PWM_freq(FTMn,    u32 freq);
```

功能说明

设置 FTM 的频率。设置后，需要重新调用 FTM_PWM_Duty 设置占空比。

建议直接使用 FTM_PWM_init 。

传递参数

参考形参变量的命名及其作用。

调用例子

```
26. FTM_PWM_freq(FTM1,1000);                    //设置 FTM1_CH0 频率为 1kHZ
```

宏定义

FTM_PRECISON

定义

```
27. #define FTM_PRECISON 100u
```

功能说明

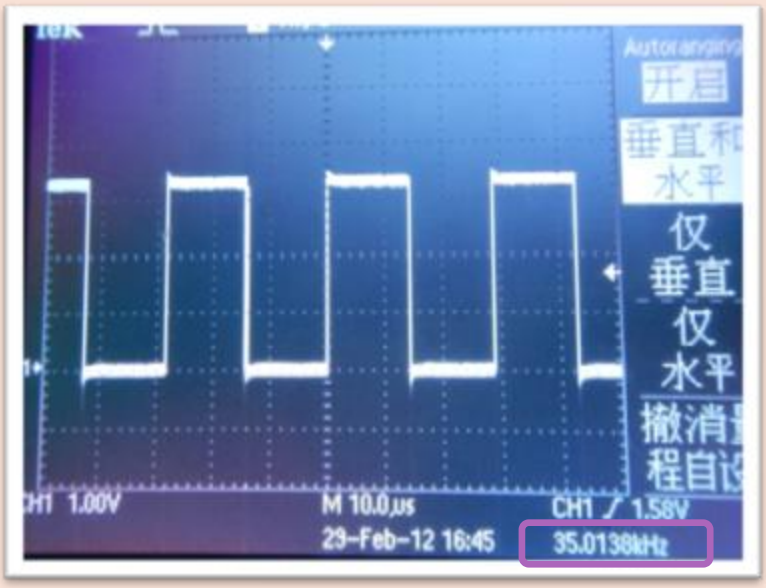
定义占空比精度，100 即精度为 1%，1000u 则精度为 0.1%。用于占空比 duty 形参传入，即占空比为 duty / FTM_PRECISON

PWM 综合测试例程

PWM 实验示波器简单测试

```
/**********************************************************************
*                              野火嵌入式开发工作室
*                              PWM 实验示波器简单测试
*
*  实验说明：野火 PWM 实验，用示波器测输出频率。
*
*  实验操作：这里用 FTM1_CH0 产生 PWM 脉冲波 ，即 PTA8 管脚。
*            把 PTA8 接入示波器
*
*  实验效果：可以测出频率为 35kHz
*
*  实验目的：测试 PWM 频率是否正确
*
*  修改时间：2012-2-29       已测试
*
*  备    注：FTM.h 里有各个 FTM 通道所对应管脚的表格，方便查看
*            占空比传递进入的参数，要根据 FTM_PRECISON 的定义来选择
**********************************************************************/
void  main()
{
    FTM_PWM_init(FTM1,CH0,35000,50);      //初始化 FTM1_CH0 输出频率为 35kHZ,占空比为 50%的 PWM
                                          //FTM1_CH0 对应 PTA8 口
    while(1)
    {
    }
}
```

实验效果：



PWM 实验 LED 测试

```
/**********************************************************************
*                              野火嵌入式开发工作室
*                                PWM 实验 LED 测试
*
*  实验说明：野火 PWM 实验，用 LED 来测试占空比的变化。
*
*  实验操作：这里用 FTM1_CH0 产生 PWM 脉冲波
*            在 FTM.h 里，可以查到 FTM1_CH0 对应管脚为 PTA8
*            把 PTA8 接入 LED0，即 PTA8 和 PTD15 短接
*
*  实验效果：可以看到 LED0 由暗变亮，再突然暗，如此下去……
*
*  实验目的：测试 PWM 频率是否正确
*
*  修改时间：2012-2-29      已测试
*
*  备     注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
*            FTM.h 里有各个 FTM 通道所对应管脚的表格，方便查看
*            占空比传递进入的参数，要根据 FTM_PRECISON 的定义来选择
**********************************************************************/
void  main()
{
    u32 i;
    FTM_PWM_init(FTM1,CH0,35000,100);         //FTM1_CH0 初始化 PWM : PA8
    while(1)
    {
        for(i=10;i>1;i--)
        {
            FTM_PWM_Duty(FTM1,CH0,i*10);      //改变占空比，逐渐变小 ,LED 逐渐变亮 （低电平点亮）
            time_delay_ms(100);                //延时 100ms
        }
    }
}
```

FTM 输入捕捉 模块

快速入门：FTM 输入捕捉库使用方法

默认通道管脚表格，非常方便查找。

形参变量的命名及其作用

形参变量	作用	取值
FTMn	模块名	FTM0，FTM1，FTM2
CHn	通道号	//　--FTM0--　--FTM1--　--FTM2-- CH0,　//　PTC1　PTA8　PTA10 CH1,　//　PTC2　PTA9　PTA11 CH2,　//　PTC3　×　× CH3,　//　PTC4　×　× CH4,　//　PTD4　×　× CH5,　//　PTD5　×　× CH6,　//　PTD6　×　× CH7　//　PTD7　×　×
Input_cfg	输入捕捉模式配置	Rising，Falling，Rising_or_Falling

常用枚举列表

名称	功能说明
FTMn	模块号
CHn	通道号
Input_cfg	输入捕捉模式配置

函数列表

函数名称	函数功能
FTM_Input_init	初始化 FTM 的输入捕捉模式

宏定义列表

宏定义名词	功能说明
FTM_IRQ_EN	开启 FTMn_CHn 中断
FTM_IRQ_DIS	关闭 FTMn_CHn 中断

枚举详解

Input_cfg

枚举定义

```
1.  enum Input_cfg
2.  {
3.    Rising,                 //上升沿捕捉
4.    Falling,                //下降沿捕捉
5.    Rising_or_Falling       //跳变沿捕捉
```

```
6.  };
```

枚举作用

输入捕捉模式配置

函数详解

FTM_Input_init

函数原型

```
28. void FTM_Input_init(FTMn,CHn,Input_cfg);
```

功能说明

初始化 FTM 的输入捕捉模式。

调用例子

```
29. FTM_Input_init(FTM1,CH0,Rising);                    //初始化 FTM 输入捕捉模式，上升沿触发
```

宏定义

FTM_IRQ_EN

定义

```
30. #define FTM_IRQ_EN(FTMn,CHn)     FTM_CnSC_REG(FTMx[FTMn],CHn) |=  \
31.                                      FTM_CnSC_CHIE_MASK   //开启 FTMn_CHn 中断
```

功能说明

开启 FTMn_CHn 中断

传递参数

参考形参变量的命名及其作用。

展开举例

```
32. FTM_IRQ_EN(FTM1,CH0);      //开启 FTM1_CH0 中断
33.                 //展开后：FTM_CnSC_REG(FTMx[FTM1],CH0) |= FTM_CnSC_CHIE_MASK
```

FTM_IRQ_DIS

定义

```
34. #define FTM_IRQ_DIS(FTMn,CHn)        FTM_CnSC_REG(FTMx[FTMn],CHn) &= \
35.                                      ~FTM_CnSC_CHIE_MASK    //关闭 FTMn_CHn 中断
```

功能说明

关闭 FTMn_CHn 中断

传递参数

参考形参变量的命名及其作用。

展开举例

```
36.  FTM_IRQ_DIS(FTM1,CH0);        //关闭 FTM1_CH0 中断
```

FTM 输入捕捉中断测试

中断服务例程，就要编写中断服务函数，先在 main.c 里写 main 函数：

```
1.   /*********************************************************************
2.   *                              野火嵌入式开发工作室
3.   *                              FTM 输入捕捉中断测试
4.   *
5.   *  实验说明：野火 FTM 输入捕捉中断实验，用 LED 显示是否进入了中断。
6.   *
7.   *  实验操作：这里用 FTM1_CH0 输入捕捉 ，即 PTA8 管脚。
8.   *            PTA9 产生方波 ，把 PTA9 和 PTA8 短接 ，
9.   *            即 PTA9 产生的上升沿来触发 FTM1_CH0 中断
10.  *
11.  *  实验效果：LED0 闪烁
12.  *
13.  *  实验目的：测试  FTM1_CH0 输入捕捉功能
14.  *
15.  *  修改时间：2012-2-29      已测试
16.  *
17.  *  备     注：FTM.h 里有各个 FTM 通道所对应管脚的表格，方便查看
18.  *
19.  *********************************************************************/
20.  void main()
21.  {
22.      DisableInterrupts;             //禁止总中断
23.
24.      gpio_init(PORTA,9,GPO,HIGH);                  //初始化 PTA9，输出高电平
25.      LED_INIT();                                  //初始化 LED，FTM1_IRQHandler 中断函数里闪烁 LED0
26.
27.      FTM_Input_init(FTM1,CH0,Rising);             //初始化 FTM 输入捕捉模式，通道 0 上升沿触发
28.
29.      EnableInterrupts;                            //开总中断
30.
31.      while(1)
32.      {
33.          gpio_set(PORTA,9,LOW);                  //PTA9 产生低电平
34.
35.          time_delay_ms(250);                     //延时一下
36.
37.          gpio_set(PORTA,9,HIGH);                 //PTA9 产生高电平
38.
39.          time_delay_ms(250);                     //延时一下
40.      }
41.  }
```

在 isr.h 里重定向中断向量表的中断函数指针和声明中断服务函数（main 函数里用 FTM1 触发中断，所以用 FTM1 的中断服务函数。在 vectors.h 里的注解列表的 Source module 查到 FTM1 的中断向量号为 VECTOR_079）：

```
1.   #undef  VECTOR_079
2.   #define VECTOR_079    FTM1_IRQHandler      //FTM1 输入捕捉中断
3.
```

```
4.  extern void FTM1_IRQHandler();              //FTM1 输入捕捉中断
```

在 isr.c 里编写中断函数：

```
1.  /*************************************************************************
2.  *                              野火嵌入式开发工作室
3.  *
4.  *  函数名称：FTM1_IRQHandler
5.  *  功能说明：FTM1 输入捕捉中断服务函数
6.  *  参数说明：无
7.  *  函数返回：无
8.  *  修改时间：2012-2-25
9.  *  备    注：引脚号需要根据自己初始化来修改，参考现有的代码添加自己的功能
10. *************************************************************************/
11. void FTM1_IRQHandler()
12. {
13.     u8 s=FTM1_STATUS;                   //读取捕捉和比较状态
14.                                         //All CHnF bits can be checked using only one read of STATUS.
15.     u8 CHn;
16.     FTM1_STATUS=0x00;                   //清中断标志位
17.
18.     CHn=0;
19.     if( s & (1<<CHn) )
20.     {
21.         FTM_IRQ_DIS(FTM1,CHn);          //禁止输入捕捉中断
22.         /*      用户任务          */
23.         LED_turn(LED0);                 //翻转 LED0
24.
25.
26.         /**********************/
27.         FTM_IRQ_EN(FTM1,CHn);   //开启输入捕捉中断
28.     }
29.
30.     /* 这里添加 n=1 的模版，根据模版来添加 */
31.     CHn=1;
32.     if( s & (1<<CHn) )
33.     {
34.         FTM_CnSC_REG(FTM1_BASE_PTR,CHn) &= ~FTM_CnSC_CHIE_MASK; //禁止输入捕捉中断
35.         /*      用户任务          */
36.
37.
38.         /**********************/
39.         FTM_IRQ_EN(FTM1,CHn);   //开启输入捕捉中断
40.     }
41.
42. }
```

通道 0 上升沿触发了中断

参考模版

用户代码

PIT 定时中断模块

快速入门：PIT 定时中断库使用方法

形参变量的命名及其作用

形参变量	作用	取值
PITn	模块号	PIT0，PIT1，PIT2，PIT3
cnt	定时器计数	
ms	时间,单位为 ms	

常用枚举列表

名称	功能说明
PITn	模块号

函数列表

函数名称	函数功能
pit_init	初始化 pit 定时器

宏定义列表

宏定义名词	功能说明
pit_init_ms	初始化 pit 定时器
PIT_Flag_Clear	清中断标志

枚举详解

PITn

枚举定义

```
1.  enum PITn
2.  {
3.    PIT0,
4.    PIT1,
5.    PIT2,
6.    PIT3
7.  };
```

枚举作用

用来定义模块号

函数详解

pit_init

函数原型

```
8.  void      pit_init(PITn,u32 cnt);
```

功能说明

初始化 PITn，并设置定时时间(单位为 bus 时钟周期)

调用例子

```
9.  pit_init(PIT0,100000); //初始化 PIT0,定时 100000 个时钟周期
```

宏定义

pit_init_ms

定义

```
10. #define   pit_init_ms(PITn,ms)          pit_init(PITn,ms * bus_clk_khz);
```

功能说明

初始化 PITn，并设置定时时间(单位为 ms)

展开举例

```
11. pit_init_ms(PIT0,1000); //初始化 PIT0,定时 1s
12.                         //展开后: pit_init(PIT0,1000* bus_clk_khz);
13.
```

PIT_Flag_Clear

定义

```
14. #define   PIT_Flag_Clear(PITn)            PIT_TFLG(PITn)|=PIT_TFLG_TIF_MASK
```

功能说明

清 PITn 中断标志。一般在中断服务函数和 pit 初始化函数里用到。

展开举例

```
15. PIT_Flag_Clear(PIT0)      //展开后: PIT_TFLG(PIT0)|=PIT_TFLG_TIF_MASK
```

PIT 定时中断测试例程

中断服务例程，就要编写中断服务函数，先在 main.c 里写 main 函数：

```
/*********************************************************************
*                               野火嵌入式开发工作室
*                                 PIT 定时中断测试
*
*  实验说明：野火 PIT 定时中断实验，在中断函数了用 LED0 显示进入了中断函数。
*
*  实验操作：无
*
*  实验效果：LED0 间隔 1s 闪烁
*
*  实验目的：测试 PIT 是否定时产生中断
*
*  修改时间：2012-2-29      已测试
*
*  备    注：野火 Kinetis 开发板的 LED0~3 ，分别接 PTD15~PTD12 ，低电平点亮
*
*********************************************************************/
void main()
{

    DisableInterrupts;                                  //禁止总中断

    LED_INIT();                                         //初始化 LED，PIT0 中断用到 LED
    pit_init_ms(PIT0,1000);                             //初始化 PIT0，定时时间为：  1000ms

    EnableInterrupts;                               //开总中断
    while(1)
    {
    }
}
```

在 isr.h 里重定向中断向量表的中断函数指针和声明中断服务函数（main 函数里用 PIT0 触发中断，所以用 PIT0 的中断服务函数。在 vectors.h 里的注解列表的 Source module 查到 PIT0 的中断向量号为 VECTOR_084）：

```
#undef  VECTOR_084
#define VECTOR_084    PIT0_IRQHandler      //重新定义 84 号中断为 PIT0_IRQHandler 中断

extern void PIT0_IRQHandler();             //PIT0 定时中断服务函数
```

在 isr.c 里编写中断函数：

```
/*********************************************************************
*                               野火嵌入式开发工作室
*
*  函数名称：PIT0_IRQHandler
*  功能说明：PIT0 定时中断服务函数
*  参数说明：无
*  函数返回：无
*  修改时间：2012-2-18     已测试
*  备    注：
*********************************************************************/

```

```
61. void PIT0_IRQHandler(void)
62. {
63.     LED_turn(LED0);                 //LED0 反转
64.     PIT_Flag_Clear(PIT0);           //清中断标志位
65. }
```

PWM、输入捕捉、PIT 中断综合测试

PWM、输入捕捉、PIT 中断综合测试，测试各个模块计数是否正常、频率是否准确。也提供一个例子讲解如何使用这三个模块。

先在 main.c 里写 main 函数：

```
/**********************************************************************
*                              野火嵌入式开发工作室
*                         PWM、输入捕捉、PIT 中断综合测试
*
*  实验说明：野火 PWM、输入捕捉、PIT 中断综合测试。
*           利用 FTM2_CH0 产生 PWM 脉冲波，
*           FTM1_CH0 输入捕捉 FTM2_CH0 产生的 PWM 脉冲波，每次捕捉到就计数。
*           PIT 每隔 1s 产生一次中断，用串口发送脉冲的计数，并清空计数。
*
*  实验操作：短接 PTA8 和 PTA10 ，即 FTM2_CH0 和 FTM1_CH0 短接。
*           打开串口助手，设置波特率为 19200
*
*  实验效果：串口输出脉冲计数：
*
*       野火 kinetis 核心板测试程序
*       内核频率：100MHz 总线频率 ：50MHz
*       flex 频率：50MHz     flash 频率：25MHz
*
*       Software Reset
*
*       K60-144pin      Silicon rev 1.2
*       Flash parameter version 0.0.7.0
*       Flash version ID 3.2.7.0
*       512 kBytes of P-flash   P-flash only
*       128 kBytes of RAM
*
*
*       接收到 99 个脉冲
*       接收到 100 个脉冲
*       接收到 100 个脉冲
*       ......
*
*
*  实验目的：综合测试 PWM、输入捕捉、PIT 中断
*
*  修改时间：2012-2-29      已测试
*
*  备     注：FTM.h 里有各个 FTM 通道所对应管脚的表格，方便查看
*           这里用到了中断，在 isr.h 和 isr.c 可以看到中断服务函数
**********************************************************************/
volatile u32 pwmtest=0;                             //用来计数
void main()
{
    DisableInterrupts;                              //禁止总中断

    FTM_Input_init(FTM1,CH0,Rising);                //初始化 FTM 输入捕捉模式，上升沿触发

    FTM_PWM_init(FTM2,CH0,100,50);                  //PTA10       100Hz

    pit_init_ms(PIT0,1000);                         //1s 产生一次中断

    EnableInterrupts;                               //开总中断

    while(1)
    {
    }
}
```

在 isr.h 里重定向中断向量表的中断函数指针和声明中断服务函数（main 函数里用 PIT0 定时触发中断 和 FTM0 输入捕捉中断）：

```
#undef  VECTOR_079
#define VECTOR_079    FTM1_IRQHandler  //FTM0 输入捕捉中断

#undef  VECTOR_084
#define VECTOR_084    PIT0_IRQHandler  //重新定义 84 号中断为 PIT0_IRQHandler 中断

extern void PIT0_IRQHandler();         //PIT0  定时中断服务函数
extern void FTM1_IRQHandler();         //FTM0 输入捕捉中断
```

在 isr.c 里编写中断函数：

```
extern volatile u32 pwmtest;          //用来计数

/*************************************************************************
*                               野火嵌入式开发工作室
*
*  函数名称：FTM1_IRQHandler
*  功能说明：FTM1 输入捕捉中断服务函数
*  参数说明：无
*  函数返回：无
*  修改时间：2012-2-25
*  备    注：引脚号需要根据自己初始化来修改，参考现有的代码添加自己的功能
*************************************************************************/
void FTM1_IRQHandler()
{
    u8 s=FTM1_STATUS;       //读取捕捉和比较状态
                           // All CHnF bits can be checked using only one read of STATUS.
    u8 CHn;
    FTM1_STATUS=0x00;               //清中断标志位

    CHn=0;
    if( s & (1<<CHn) )
    {
        FTM_IRQ_DIS(FTM1,CHn);      //禁止输入捕捉中断
        /*     用户任务        */
        pwmtest++;                                  //计数+1


        /*********************/
        FTM_IRQ_EN(FTM1,CHn);  //开启输入捕捉中断
    }

    /* 这里添加 n=1 的模版，根据模版来添加 */
    CHn=1;
    if( s & (1<<CHn) )
    {
        FTM_CnSC_REG(FTM1_BASE_PTR,CHn) &= ~FTM_CnSC_CHIE_MASK; //禁止输入捕捉中断
        /*     用户任务        */


        /*********************/
    }

}


/*************************************************************************
*                               野火嵌入式开发工作室
```

```
113. *
114. *  函数名称：PIT0_IRQHandler
115. *  功能说明：PIT0 定时中断服务函数
116. *  参数说明：无
117. *  函数返回：无
118. *  修改时间：2012-2-18     已测试
119. *  备     注：
120. ************************************************************************/
121. void PIT0_IRQHandler(void)
122. {
123.     DisableInterrupts;              //禁止总中断
124. 
125.     printf("\n 接收到 %d 个脉冲",pwmtest);
126.     pwmtest=0;                      //清计数
127. 
128.     PIT_Flag_Clear(PIT0);           //清中断标志位
129.     EnableInterrupts;               //开总中断
130. }
```

I2C 模块

快速入门：I2C 通信库使用方法

形参变量的命名及其作用

形参变量	作用	取值
I2Cn	模块号	I2C0、I2C1
SlaveID	7 位从机地址	7bit
Addr	从机的寄存器地址	8bit
Data	数据	8bit

常用枚举列表

名称	功能说明
I2Cn	模块号

函数列表

函数名称	函数功能
I2C_init	初始化 I2C 模块
I2C_WriteAddr	读取地址里的内容
I2C_ReadAddr	往地址里写入内容

枚举详解

I2Cn

枚举定义

```
1.  enum I2Cn
2.  {
3.    I2C0   = 0,
4.    I2C1   = 1
5.  };
```

枚举作用

用来定义模块号

函数详解

I2C_init

函数原型

```
6.  void  I2C_init(I2Cn);
```

功能说明

初始化 I2C

调用例子

```
7.  I2C_init(I2C0);  //初始化 I2C0
```

I2C_WriteAddr

函数原型

```
8.  void  I2C_WriteAddr (I2Cn,u8 SlaveID,u8 Addr,u8 Data);
```

功能说明

写入一个字节数据到 I2C 设备指定寄存器地址

调用例子

```
9.  I2C_WriteAddr (I2C0,0x47,0x00,1); //向设备号为 0x47 的芯片写入数据 1 到地址 0x00 里
```

I2C_ReadAddr

函数原型

```
10. u8    I2C_ReadAddr(I2Cn,u8 SlaveID,u8 Addr);
```

功能说明

读取 I2C 设备指定寄存器地址的一个字节数据

调用例子

```
11. d=I2C_ReadAddr (I2C0,0x47,0x00); //向设备号为 0x47 的芯片读取地址 0x00 里的数据
```

I2C 通信实验测试

```
1.  /*********************************************************************
2.  *                              野火嵌入式开发工作室
3.  *                                I2C 通信实验
4.  *
5.  *  实验说明：野火 I2C 通信实验，往 AT24C02 写入数据，再读出来，通过串口来显示。
6.  *            野火串口默认为： UART1，TX 接 PTC4，RX 接 PTC3
7.  *            k60_fire.h 里定义了 printf 函数的输出设置：
8.  *                #define FIRE_PORT           UART1
9.  *                #define FIRE_BAUD           19200
```

```
*
*  实验操作：打开串口助手 ，设置波特率为 19200 。
*            串口线（经过 MAX3232 电平转换）：TX 接 PTC4，RX 接 PTC3 。
*            运行程序后，在串口助手里看到。
*
*  实验效果：串口助手里显示数据：
*
*           野火 kinetis 核心板测试程序
*       内核频率：200MHz     总线频率 ：66MHz
*       flex 频率：66MHz     flash 频率：28MHz
*
*       Software Reset
*
*       K60-144pin      Silicon rev 1.2
*       Flash parameter version 0.0.7.0
*       Flash version ID 3.2.7.0
*       512 kBytes of P-flash   P-flash only
*       128 kBytes of RAM
*
*       AT24C02 I2C 实验
*
*               ——野火 kinetis 开发板
*
*       接收到的数据为：0
*
*       接收到的数据为：1
*
*       ......
*
*
*  实验目的：测试 I2C 通信功能
*
*  修改时间：2012-2-29      已测试
*
*  备    注：
*
*************************************************************************/
void main()
{
#define ADDR    0x00
    u8 i=0;
    u8 Data;
    uart_init(UART1, 19600);                        //初始化串口

    I2C_init(I2C0);                                 //初始化 I2C0

    printf("AT24C02 I2C 实验\n\n");
    printf("\t\t——野火 kinetis 开发板\n\n");

    while(1)
    {
        for(i=0;i<255;i++)
        {
            I2C_WriteAddr(I2C0,AT24C02_I2C_ADDRESS,ADDR,i);
                            //I2C 向 AT24C02_I2C_ADDRESS 芯片写入数据 i 到地址为 ADDR 的寄存器
            Data    =   I2C_ReadAddr(I2C0,AT24C02_I2C_ADDRESS,ADDR);
                            //I2C 向 AT24C02_I2C_ADDRESS 芯片读取寄存器地址为 ADDR 的数据

            printf("接收到的数据为：%d\n\n",Data);                  //发送到串口显示出来

            time_delay_ms(1000);                                    //延时 1s
        }
    }
#undef  ADDR
}
```

为了方便操作 AT24C02，在 AT24C02.h 里定义：

```
#define AT24C02_I2C_ADDRESS              0x50
#define AT24C02_PageSize                 8          /* AT24C02 每页有 8 个字节 */

#define AT24C02_init()                   I2C_init(I2C0)

#define AT24C02_WriteByte(Addr,Data)     I2C_WriteAddr(I2C0,AT24C02_I2C_ADDRESS,Addr,Data)
                                                                          //读取地址里的内容
#define AT24C02_ReadByte(Addr)           I2C_ReadAddr(I2C0,AT24C02_I2C_ADDRESS,Addr)
                                                                          //从地址读取内容
```

则上面的测试例程可以改为：

```
void main()
{
#define ADDR     0x00
    u8 i=0;
    u8 Data;
    uart_init(UART1, 19600);                        //初始化串口

    AT24C02_init();                                 //初始化 AT24C02，启动 I2C 总线

    printf("AT24C02 I2C 实验\n\n");
    printf("\t\t——野火 kinetis 开发板\n\n");
    while(1)
    {
        for(i=0;i<255;i++)
        {
            AT24C02_WriteByte(ADDR,i);              //向地址 ADDR 写入数据 i

            Data=AT24C02_ReadByte(ADDR);            //读取地址 ADDR 的数据

            printf("接收到的数据为：%d\n\n",Data);//发送到串口显示出来

            time_delay_ms(1000);                    //延时 1s
        }
    }
#undef  ADDR
}
```

lptmr 低功耗定时器模块

定功耗定时器 LPTMR 可以用来作为定时计数器或者脉冲计数器。

快速入门：lptmr 低功耗定时器库使用方法

形参变量的命名及其作用

形参变量	作用	取值
LPT0_ALTn	管脚号	LPT0_ALT1、LPT0_ALT2
count	计数值	16bit
PrescaleValue	滤波延时值	4bit
LPT_CFG	触发条件配置	LPT_Rising 、LPT_Falling

常用枚举列表

名称	功能说明
LPT0_ALTn	管脚号： LPT0_ALT1　　// PTA19 LPT0_ALT2　　// PTC5
LPT_CFG	LPT 中断配置 LPT_Rising　　//上升沿触发 LPT_Falling　　//下降沿触发

函数列表

函数名称	函数功能
time_delay	低功耗定时器延时函数
lptmr_counter_init	计数器初始化

宏定义列表

宏定义名词	功能说明
lptmr_counter_clean	计数器计数清 0

枚举详解

LPT0_ALTn

枚举定义

```
1.  typedef enum LPT0_ALTn
2.  {
```

```
3.      //只有 1、2 管脚，并没有 0、3 管脚
4.      LPT0_ALT1=1,          // PTA19
5.      LPT0_ALT2=2           // PTC5
6.  }LPT0_ALTn;
```

枚举作用

用来定义管脚

LPT_CFG

枚举定义

```
1.  typedef enum LPT_CFG
2.  {
3.      LPT_Rising  = 0,     //上升沿触发
4.      LPT_Falling = 1      //下降沿触发
5.  }LPT_CFG;
```

枚举作用

用来定义触发方式

函数详解

time_delay

函数原型

```
1.  void time_delay(uint32 cnt);
```

功能说明

延时（官方自带例程）

调用例子

```
2.  time_delay (1000);          //延时 1000 个 bus 周期
```

lptmr_counter_init

函数原型

```
1.  void lptmr_counter_init(LPT0_ALTn,u16 count,u8 PrescaleValue,LPT_CFG);
```

功能说明

计数器初始化设置

调用例子

```
3.  lptmr_counter_init(LPT0_ALT2,INT_COUNT,2,LPT_Rising);
4.                          //初始化脉冲计数器，用 LPT0_ALT2，即 PTC5 输入
```

```
5.                                  //每隔 INT_COUNT 个脉冲产生中断，延时 2 个时钟滤波，上升沿触发
```

宏定义

lptmr_counter_clean

定义

```
1.  #define  lptmr_counter_clean()      LPTMR_CSR_REG(LPTMR0_BASE_PTR)   \
2.                                                  &= ~LPTMR_CSR_TEN_MASK;\
3.                                      LPTMR_CSR_REG(LPTMR0_BASE_PTR)   \
4.                                                  |=  LPTMR_CSR_TEN_MASK
5.                                                          //计数器计数清 0
```

功能说明

计数器计数清 0

调用例子举例

```
6.  lptmr_counter_clean();        //清空 LPTMR Counter 和 Timer Compare Flag
```

lptmr 低功耗定时器测试例程

LPT 脉冲计数中断实验

这里，FTM 产生 PWM 脉冲，LPT 脉冲计数，LPT 脉冲到一定数目的时候，就产生中断。

在 main.c 里：

```
1.  extern volatile u32 LPT_INT_count;
2.  extern volatile u8  pit_flag;
3.
4.  /*********************************************************************
5.  *                              野火嵌入式开发工作室
6.  *                    LPT 脉冲计数中断实验（利用 FTM 产生 PWM 脉冲波）
7.  *
8.  *  实验说明：利用 FTM 产生 PWM 脉冲，LPT 脉冲计数
9.  *
10. *  实验操作：短接 PTC5 和 PTA8
11. *
12. *  实验效果：串口输出计数值，效果现象：
13. *                  野火 kinetis 核心板测试程序
14. *        内核频率：200MHz     总线频率 ：100MHz
15. *        flex 频率：100MHz   flash 频率：28MHz
16. *
17. *        Core Lockup Event Reset
18. *
19. *        野火 Kinetis 开发板启动方式：flash 启动
20. *
21. *        K60-144pin      Silicon rev 1.2
22. *        Flash parameter version 0.0.7.0
23. *        Flash version ID 3.2.7.0
24. *        512 kBytes of P-flash  P-flash only
```

```
*         128 kBytes of RAM
*
*         LPT 产生一次中断啦:100
*         LPT 产生一次中断啦:100
*         ……
*
*  实验目的：明白如何用脉冲计数函数
*
*  修改时间：2012-3-16      已测试
*
*  备     注：可以修改 FTM 频率来发现频率越快，LPT 产生中断越快
*****************************************************************************/
#define INT_COUNT  100          //LPT 产生中断的计数次数
void  main(void)
{
    u16 count;

    DisableInterrupts;      //禁止总中断

    FTM_PWM_init(FTM1,CH0,100,50);
                                        //FTM 模块产生 PWM，用 FTM1_CH0 ，即 PTA8 ，频率为 100

    lptmr_counter_init(LPT0_ALT2,INT_COUNT,2,LPT_Rising);
                                        //初始化脉冲计数器，用 LPT0_ALT2，即 PTC5 输入，
                                        //每隔 INT_COUNT 产生中断，延时 2 个时钟滤波，上升沿触发
    EnableInterrupts;                   //开总中断

    while(1)
    {
        if( LPT_INT_count > 0 )
        {
            count           =   LPTMR0_CNR;     //保存脉冲计数器计算值
            lptmr_counter_clean();
                                //清空脉冲计数器计算值（马上清空，这样才能保证计数值准确）
            printf("LPT 产生一次中断啦:%d\n",LPT_INT_count*INT_COUNT +  count);
                                                        //打印计数值
            LPT_INT_count   =   0;                      //清空 LPT 中断次数
        }
    }
}
```

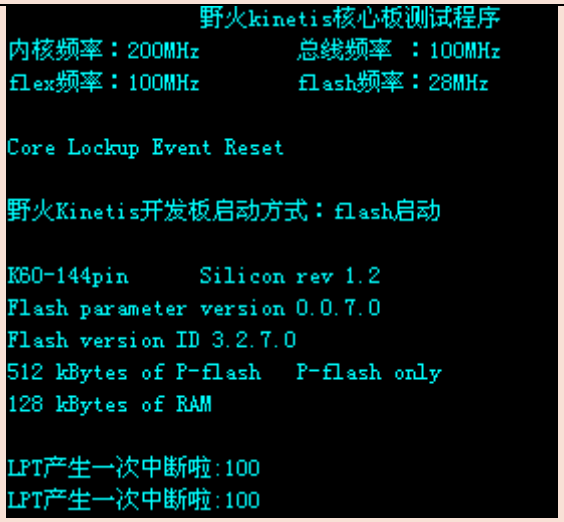
在 isr.h 里重映射中断向量表：

```
#undef      VECTOR_101
#define     VECTOR_101  LPT_Handler

extern  void  LPT_Handler(void);
```

然后在 isr.c 里编辑 LPT 的中断服务函数：

```
volatile u32 LPT_INT_count=0;

void  LPT_Handler(void)
{
    LPTMR0_CSR|=LPTMR_CSR_TCF_MASK;     //清除 LPTMR 比较标志
    LPT_INT_count++;                    //中断溢出加 1
}
```

实验效果：



注：接收到的脉冲数为：LPT_INT_count * INT_COUNT + LPTMR0_CNR，LPTMR0_CNR 一般情况下应该为 0。

PIT 定时读取 LPT 脉冲计数实验

这里，FTM 产生 PWM 脉冲，LPT 脉冲计数，PIT 定时中断读取脉冲值。

在 main.c 里：

```
1.  extern volatile u32 LPT_INT_count;
2.  extern volatile u8  pit_flag;
3.
4.  /*********************************************************************
5.  *                              野火嵌入式开发工作室
6.  *                   PIT 定时读取 LPT 脉冲计数实验（利用 FTM 产生 PWM 脉冲波）
7.  *
8.  *  实验说明：利用 FTM 产生 PWM 脉冲，LPT 脉冲计数，PIT 定时中断读取计数
9.  *
10. *  实验操作：短接 PTC5 和 PTA8
11. *
12. *  实验效果：串口输出计数值，效果现象：
13. *                野火 kinetis 核心板测试程序
14. *        内核频率：200MHz     总线频率 ：100MHz
15. *        flex 频率：100MHz  flash 频率：28MHz
16. *
17. *        Core Lockup Event Reset
18. *
19. *        野火 Kinetis 开发板启动方式：flash 启动
20. *
21. *        K60-144pin      Silicon rev 1.2
22. *        Flash parameter version 0.0.7.0
23. *        Flash version ID 3.2.7.0
24. *        512 kBytes of P-flash  P-flash only
25. *        128 kBytes of RAM
26. *
27. *        1 秒钟 LPT 读取脉冲:81 个                  第一个不稳定
28. *        1 秒钟 LPT 读取脉冲:199 个
29. *        1 秒钟 LPT 读取脉冲:199 个
30. *        1 秒钟 LPT 读取脉冲:199 个
```

```
*          ……
*
*  实验目的：明白如何定时统计脉冲计数
*
*  修改时间：2012-3-16      已测试
*
*  备     注：可以修改 FTM 频率，1 秒时间的脉冲计数与频率非常相近（有误差）
*********************************************************************/
#define INT_COUNT  100
void main()
{
    u16 count;

    DisableInterrupts;                                                  //禁止总中断

    FTM_PWM_init(FTM1,CH0,200,50);        //FTM 模块产生 PWM，用 FTM1_CH0 ，即 PTA8 ，频率为 200

    lptmr_counter_init(LPT0_ALT2,INT_COUNT,2,LPT_Rising);
                                //初始化脉冲计数器，用 LPT0_ALT2
                                //即 PTC5 输入，每隔 INT_COUNT 产生中断，延时 2 个时钟滤波，上升沿触发

    pit_init_ms(PIT0,1000);                                             //定时 1 秒 中断

    EnableInterrupts;                                                   //开总中断

    while(1)
    {
        if( pit_flag > 0 )                                             /*   1 秒中断了   */
        {
            count=LPTMR0_CNR;               //保存脉冲计数器计算值
            lptmr_counter_clean();          //清空脉冲计数器计算值（马上清空，这样才能保证计数值准确）
            printf("1 秒钟 LPT 读取脉冲:%d 个\n",LPT_INT_count*INT_COUNT +  count);
                                            //读取间隔 1 秒的脉冲次数
            LPT_INT_count=0;                //清空 LPT 中断次数
            pit_flag=0;                     //清空 pit 中断标志位
        }
    }
}
```

在 isr.h 里重映射中断向量表：

```
#undef      VECTOR_101
#define     VECTOR_101  LPT_Handler

#undef  VECTOR_084
#define VECTOR_084    PIT0_IRQHandler          //重新定义 84 号中断为 PIT0_IRQHandler 中断

extern  void  LPT_Handler(void);
extern  void PIT0_IRQHandler(void);             //PIT0 定时中断服务函数
```

然后在 isr.c 里编辑 LPT 的中断服务函数：

```
volatile u32 LPT_INT_count=0;
volatile u8  pit_flag    = 0;

void  LPT_Handler(void)
{
    LPTMR0_CSR|=LPTMR_CSR_TCF_MASK;      //清除 LPTMR 比较标志
    LPT_INT_count++;                     //中断溢出加 1
}

void PIT0_IRQHandler()
{
    pit_flag=1;                  //标记进入 PIT 中断
    PIT_Flag_Clear(PIT0);        //清中断标志位
```

```
14.  }
```

实验效果：


野火kinetis核心板测试程序
内核频率：200MHz　　总线频率 ：100MHz
flex频率：100MHz　　flash频率：28MHz
Core Lockup Event Reset
野火Kinetis开发板启动方式：flash启动
K60-144pin　　Silicon rev 1.2
Flash parameter version 0.0.7.0
Flash version ID 3.2.7.0
512 kBytes of P-flash　P-flash only
128 kBytes of RAM
1秒钟LPT读取脉冲:199个
1秒钟LPT读取脉冲:199个
1秒钟LPT读取脉冲:199个

MCG 模块超频

快速入门：MCG 库使用方法

全局变量列表

全局变量列表	作用	取值
mcg_div	分频因子结构体	
core_clk_khz	内核频率 kHz	
core_clk_mhz	内核频率 MHz	
bus_clk_khz	总线频率 kHz	

常用枚举列表

名称	功能说明

clk_option	PLLUSR , //自定义设置分频系数模式，直接加载 全局变量 mcg_div 的值
	PLL48 =48,
	PLL50 =50,
	PLL96 =96,
	PLL100 =100,
	PLL110 =110,
	PLL120 =120,
	PLL125 =125,
	PLL130 =130,
	PLL140 =140,
	PLL150 =150,
	PLL160 =160,
	PLL170 =170,
	PLL180 =180,
	PLL200 =200, //绝大部分芯片都成超到这个程度
	PLL225 =225, //不同芯片，不同板子，超频能力不一样，不一定全部都能超到这个水平
	PLL250 =250

函数列表

函数名称	函数功能
pll_init	初始化锁相环，可用于超频

宏定义列表

宏定义名词	功能说明
MCG_CLK_MHZ	定义系统频率（在 k60_fire.h 定义）
MAX_BUS_CLK	定义总线最大频率（MCG_CLK_MHZ 定义为 PLLUSR 时设置无效）
MAX_FLASH_CLK	定义 flash 最大频率（MCG_CLK_MHZ 定义为 PLLUSR 时设置无效）

PRDIV	分频因子选项
VDIV	倍频因子选项
CORE_DIV	内核时钟分频因子
BUS_DIV	总线时钟分频因子
FLEX_DIV	flex 时钟分频因子
FLASH_DIV	flash 时钟分频因子

全局变量详解

mcg_div

结构体定义

```
1.  struct mcg_div
2.  {
3.    u8 prdiv;                 //外部晶振分频因子选项   mcg = 外部晶振频率
4.    u8 vdiv;                  //外部晶振倍频因子选项
5.    u8 core_div;              //内核时钟分频因子
6.    u8 bus_div;               //总线时钟分频因子
7.    u8 flex_div;              //flex 时钟分频因子
8.    u8 flash_div;             //flash 时钟分频因子
9.  } mcg_div = { PRDIV , VDIV , CORE_DIV , BUS_DIV , FLEX_DIV , FLASH_DIV };
```

结构体作用

时钟分频因子。用于 pll_init 设置系统频率。

系统时钟

在 sysinit.c 里定义。

变量定义

```
10. u32 core_clk_khz;         //内核时钟(KHz)
11. u32 core_clk_mhz;         //内核时钟(MHz)
12. u32 bus_clk_khz;          //外围总线时钟
```

变量作用

用于方便外部函数查询系统时钟频率，以便在外部函数内部根据系统时钟频率的不同而设置不同情况。

调用 pll_init 后应该更改这些参数的值，以保证这些值的正确性。例：

```
1.  //通过 pll_init 函数的返回值来计算内核时钟和外设时钟，便于其他函数可查询时钟频率
2.  core_clk_khz=core_clk_mhz * 1000;
3.  bus_clk_khz =core_clk_khz/ (((SIM_CLKDIV1 & SIM_CLKDIV1_OUTDIV2_MASK)>>24)+1);
```

枚举详解

clk_option

枚举定义

```
13. enum clk_option
14. {
15.   PLLUSR      ,  //自定义设置分频系数模式，直接加载全局变量 mcg_div 的值
16.   PLL48    =48,
17.   PLL50    =50,
18.   PLL96    =96,
19.   PLL100   =100,
20.   PLL110   =110,
21.   PLL120   =120,
22.   PLL125   =125,
23.   PLL130   =130,
24.   PLL140   =140,
25.   PLL150   =150,
26.   PLL160   =160,
27.   PLL170   =170,
28.   PLL180   =180,
29.   PLL200   =200,  //绝大部分芯片都成超到这个程度
30.   PLL225   =225,  //不同芯片，不同板子，超频能力不一样，不一定全部都能超到这个水平
31.   PLL250   =250
32. };
```

枚举作用

用于 MCG_CLK_MHZ 的宏定义和 pll_init 的形参。

函数详解

pll_init

函数原型

```
33. unsigned char  (clk_option);  //锁相环初始化
```

功能说明

初始化锁相环，设置系统频率。超频就是调用这函数。

调用例子

```
34. pll_init(PLL100);                 //初始化锁相环为 100MHz
```

宏定义

MCG_CLK_MHZ

（在 k60_fire.h 定义）

功能说明

设置系统频率，用于系统初始化时传递给 pll_init

如果 MCG_CLK_MHZ 定义为 PLLUSR，则使用自定义的分频系数来进行系统初始化分频。

举例

```
35. #define MCG_CLK_MHZ         PLL200      //设置初始化频率为 200MHz
36. #define MCG_CLK_MHZ         PLLUSR      //设置初始化频率为自定义的频率
```

最大时钟频率

MAX_BUS_CLK 、MAX_FLASH_CLK （在 k60_fire.h 定义）

宏定义

```
37. #define  MAX_BUS_CLK     100              //bus 不要超过 200M，这里设为 100M
38. #define  MAX_FLASH_CLK   30              //flash 不能超过 35M，这里设为不超过 30M
```

功能说明

设置系统 bus 频率和 flash 频率的最大值，用于使用默认配置时，即 MCG_CLK_MHZ 不是定义为 PLLUSR 时，pll_init 自动计算分频因子。

由于内核时钟设置为等于 MCG 时钟，flex 频率设置为等于 bus 频率，所以没有这两项的宏定义选择。

如果 MCG_CLK_MHZ 定义为 PLLUSR，则使用自定义的分频系数来进行系统初始化分频。这里的宏定义设置就无效。

举例

有效

```
39. #define MCG_CLK_MHZ         PLL200      // 设置时钟频率
40. #define MAX_BUS_CLK         100         // bus 不要超过 200M，这里设为 100M
41. #define MAX_FLASH_CLK       30          // flash 不能超过 32M，这里设为不超过 30M
```

无效

```
42. #define MCG_CLK_MHZ         PLLUSR      // 使用自定义分频
```

```
43. #define MAX_BUS_CLK          100          // 无效
44. #define MAX_FLASH_CLK        30           // 无效
```

系统频率分频因子选项

PRDIV、VDIV、CORE_DIV、BUS_DIV、FLEX_DIV、FLASH_DIV

宏定义

```
45. #define PRDIV                14           // MCG_CLK_MHZ = 50u/(PRDIV+1)*(VDIV+24)
46. #define VDIV                 6
47. #define CORE_DIV             0            //  core = mcg/ ( CORE_DIV  + 1 )
48. #define BUS_DIV              1            //  bus  = mcg/ ( BUS_DIV   + 1 )
49. #define FLEX_DIV             1            //  flex = mcg/ ( FLEX_DIV  + 1 )
50. #define FLASH_DIV            3            //  flash= mcg/ ( FLASH_DIV + 1 )
```

可以在 Fire_kinetis_MCG_CFG.h 里选择适当的 PRDIV、VDIV 超频 MCG，然后根据 MCG 的频率来选择适当的分频因子。

《野火 kinetis 开发板超频配置一览表 》已经枚举了所有可能的 PRDIV、VDIV 选项，并计算出其频率，非常方便查询、自定义频率。

具体的配置教程，请看：配置频率

功能说明

自定义系统频率分频。如果 MCG_CLK_MHZ 定义为 PLLUSR，则使用这些自定义的分频系数来进行系统初始化分频。否则按照 MCG_CLK_MHZ 的定义使用已经配置好的默认配置。

举例

```
51. #define MCG_CLK_MHZ          PLL200       //设置初始化频率为 200MHz
52. #define MCG_CLK_MHZ          PLLUSR       //设置初始化频率为自定义的频率
```

快速入门：配置频率

野火 Kinetis 系列核心板，采用 50MHz 外部晶振。野火 Kinetics K60 库，配置频率也非常简单，有两种方法：

①　使用默认配置

在 k60_fire.h 文件里找到 **MCG_CLK_MHZ** 的宏定义：

```
53. #define MCG_CLK_MHZ         PLL200          // 设置时钟频率
```

在这里可以修改 PLL200 为其他选项，参考 **clk_option** 的配置：

```
54. enum clk_option
55. {
56.   PLLUSR       ,   //自定义设置分频系数模式，直接加载 全局变量 mcg_div 的值
57.   PLL48      =48,
58.   PLL50      =50,
59.   PLL96      =96,
60.   PLL100     =100,
61.   PLL110     =110,
62.   PLL120     =120,
63.   PLL125     =125,
64.   PLL130     =130,
65.   PLL140     =140,
66.   PLL150     =150,
67.   PLL160     =160,
68.   PLL170     =170,
69.   PLL180     =180,
70.   PLL200     =200,   //绝大部分芯片都成超到这个程度
71.   PLL225     =225,   //不同芯片，不同板子，超频能力不一样，不一定全部都能超到这个水平
72.   PLL250     =250
73. };
```

从 clk_option 里选择一个选项作为 MCG_CLK_MHZ 的宏定义

另外，总线频率 、flex 频率、flash 频率的最大值，在 pll_init 函数的内部，默认的最大分频设置为：

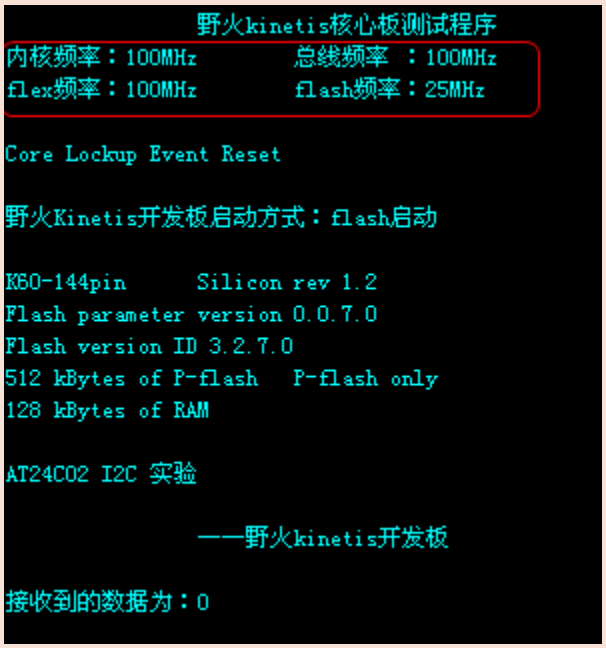
```
74. #define  MAX_BUS_CLK     100             //bus 不要超过 200M，这里设为 100M
75. #define  MAX_FLASH_CLK   30              //flash 不能超过 35M，这里设为不超过 30M
```

因为代码里设置了 flex　= bus，所以就没有宏定义 MAX_FLEX_CLK。

举例说明

```
76. #define MCG_CLK_MHZ         PLL100      // 设置时钟频率为 100MHz
77. #define MAX_BUS_CLK         100         // bus 不要超过 200M，这里设为 100M
78. #define MAX_FLASH_CLK       30          // flash 不能超过 32M，这里设为不超过 30M
```

这里烧写 I2C 实验为例，打开串口助手后，可看到各时钟频率的大小：

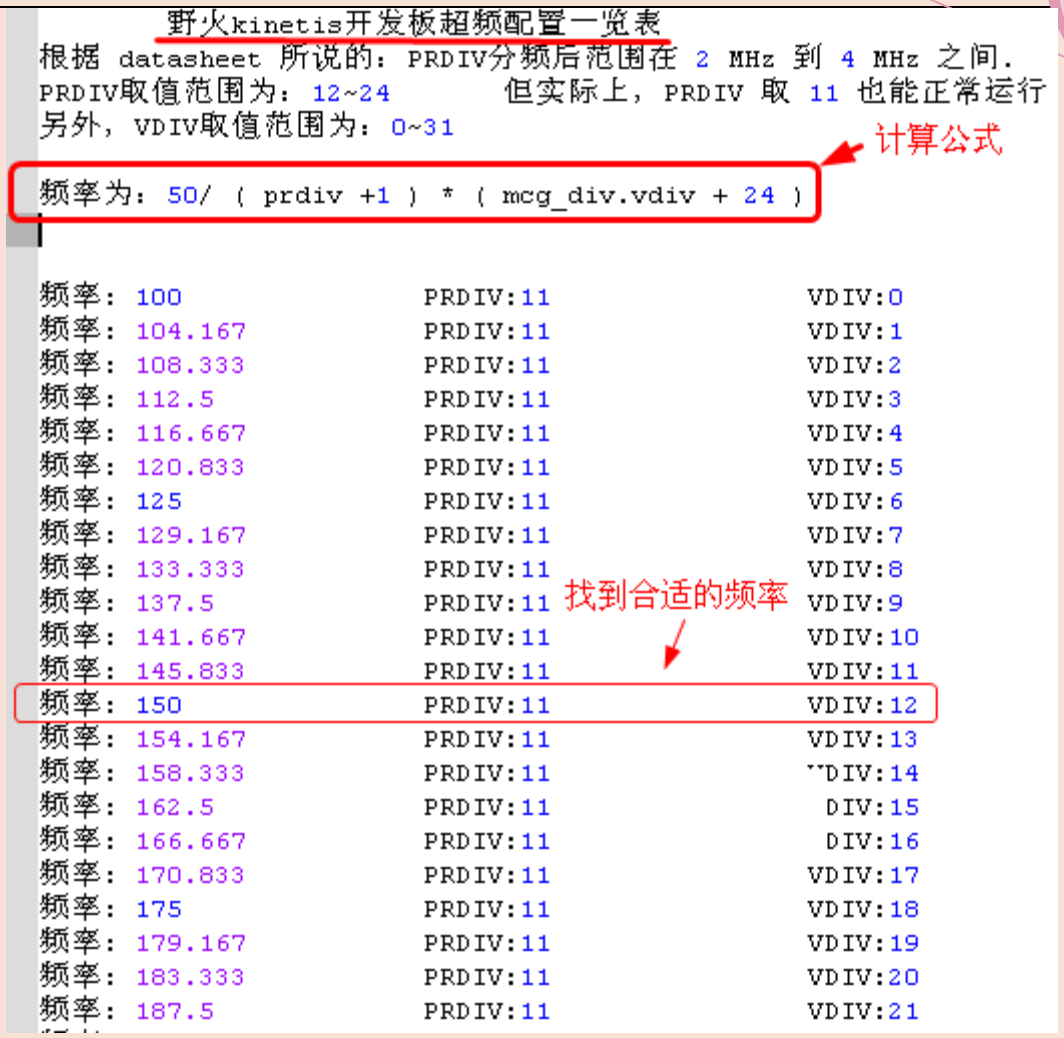


②　使用自定义分频因子

首先先把 MCG_CLK_MHZ 定义为 PLLUSR：

```
79. #define MCG_CLK_MHZ        PLLUSR      // 用户自定义
```

再在 Fire_kinetis_MCG_CFG.h 里的《野火 kinetis 开发板超频配置一览表 》中选择适当的 MCG 频率：



这里以频率为 150MHz 为例，频率: 150　　PRDIV:11　　VDIV:12

```
80. #define PRDIV          11      // MCG_CLK_MHZ = 50u/(PRDIV+1)*(VDIV+24)
81. #define VDIV           12
```

然后考虑各时钟分频。关于各时钟分频的最大超频频率，野火嵌入式软件开发工作室测试到的最大极限：内核频率：229MHz，总线频率 ：200MHz ，flex 频率：200MHz ，flash 频率：32MHz。

但绝不推荐这样做，不同的芯片，不同的板子，不同的外设，导致其最大频率都会有所不同，而且会有很可能会产生不稳定情况。我们在使用中发现，芯片频率高了，明显感受到芯片温度升高。

推荐各时钟最大频率为：内核频率：200MHz，总线频率 ：100MHz ，flex 频率：100MHz ，flash 频率：30MHz 。

这里以 150MHz 为例：

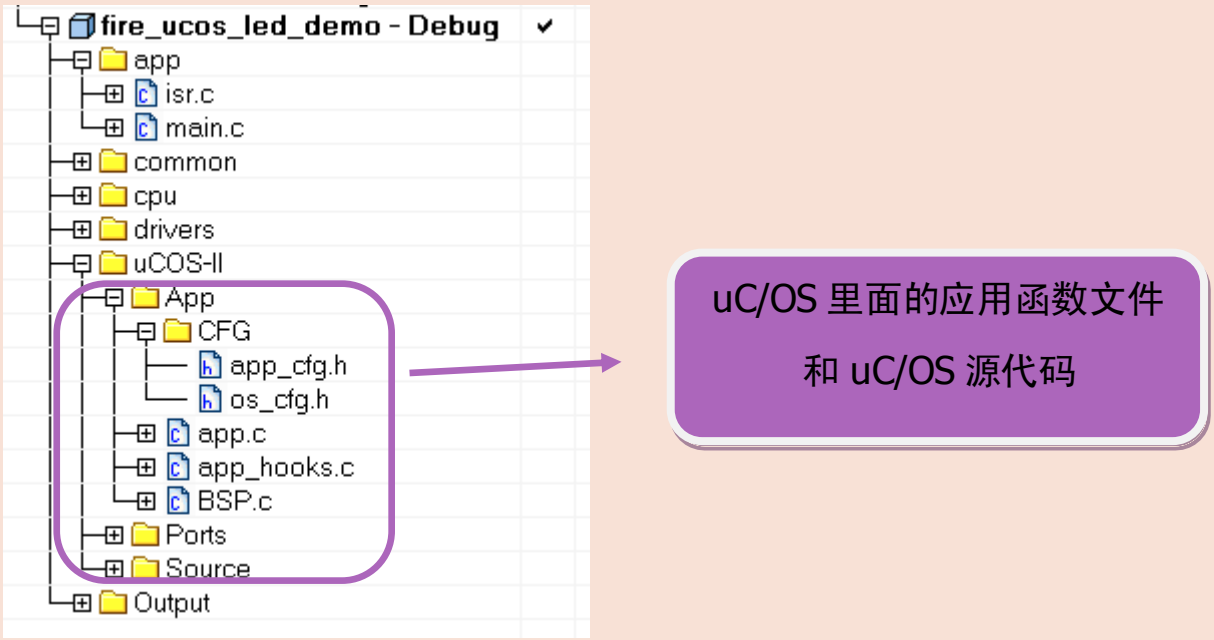
```
82. #define CORE_DIV          0     //  core = 150/ ( CORE_DIV  + 1 ) = 150 MHz
83. #define BUS_DIV           1     //  bus  = 150/ ( BUS_DIV   + 1 ) = 75  MHz
84. #define FLEX_DIV          1     //  flex = 150/ ( FLEX_DIV  + 1 ) = 75  MHz
85. #define FLASH_DIV         4     //  flash= 150/ ( FLASH_DIV + 1 ) = 30  MHz
```

在串口助手里可以看到启动信息：

uC/OS

关于 uC/OS，野火 K60 库里已经移植成功。虽然 uC/OS-II 系统比较容易入门，关于 uC/OS 系统的使用，对于之前接触过 uC/OS 的用户而言，会很快上手，对于刚接触的朋友来说，就需要点时间。这里是简单的库讲解，就不详细讲解 uC/OS 的使用，如果刚接触的朋友想学习，不妨先看看野火写的在 stm32 平台下的 uC/OS 教程：

http://www.ourdev.cn/bbs/bbs_content.jsp?bbs_sn=5233926&bbs_page_no=1&bbs_id=1008



如果你之前已经接触过 uC/OS，那么这部分不用讲解也可以马上上手。

Digitally signed by □□
DN: cn=□□, c=CN, o=□□□□□, ou=□□□□□□□□□□, email=minimcu@foxmail.com
Reason: □□□□□□□□
Date: 2012.03.17 11:46:45 +08'00'