IDE 设置编译优化等级使用指南

版本 A/0 – 2023 年 9 月

概述

在编译代码时，使用不同的编译优化等级会对生成的代码质量产生不同的影响。在实际使用时，需要根据具体的业务场景，使用不同的编译优化等级。除此之外，也可能需要对某一部分的代码设置编译优化等级。本文将介绍不同的编译器使用优化等级的方式。

目录

图片列表

表格列表

版本历史

版本	日期	作者	状态	变更
A/0	2023-09-19	TQ.Hou	Released	首次发布。

术语或缩写

术语或缩写	描述

1 优化等级介绍

不同的编译器对于不同优化等级的行为存在差异，为了说明优化等级对于代码的影响，在此以 KEIL(ARMCC 编译器)为例进行说明，其它编译器的具体行为请读者自行查阅相关编译器手册。对于 KEIL(ARMCC 编译器)而言，优化等级 O0、O1、O2、O3 的具体含义如下：

- O0：最低优化级别。关闭大多数优化。启用调试时，此选项提供最佳的调试视图，因为生成的代码结构直接对应源代码。所有干扰调试视图的优化都被禁用。具体而言：
  - 可以在任何可达点上设置断点，包括无效代码。
  - 变量的值在其作用域内随处可用，除非它未初始化。
  - 回溯提供了符合源代码预期的开放函数调用堆栈。

  请注意：

  a) 尽管-O0 生成的调试视图最接近源代码，但用户可能更喜欢-O1 生成的调试视图，因为它提高了代码的质量而不改变其基本结构。

  b) 无效代码包括对局部变量的赋值等不会影响程序结果的可达代码。不可达代码具体指的是无法通过任何控制流路径到达的代码，例如紧随返回语句之后的代码。
- O1：受限制的优化。编译器仅执行可以通过调试信息描述的优化。删除未使用的内联函数和未使用的静态函数。关闭会严重降低调试视图的优化。如果与--debug 一起使用，可以得到效果不错的调试试图，具有良好的代码密度，其与-O0 得到的调试视图的差异包括：
  - 不能在无效代码上设置断点。
  - 变量的值在初始化后可能在其作用域内不可用。比如，当它们的分配位置已被重用的情况。
  - 无副作用的函数可能会按非正常顺序调用，或者如果不需要的话可能会被省略。
  - 由于存在尾调用，回溯可能不会提供符合源代码预期的开放函数调用堆栈。

  优化级别-O1 在源代码和目标代码之间产生很好的对应关系，特别是当源代码不包含无效代码时。生成的代码可以显著小于-O0 的代码，这可以简化目标代码的分析。
- O2：高级优化。如果与--debug 一起使用，调试视图可能不太令人满意，因为目标代码到源代码的映射并不总是清晰。编译器可能执行无法用调试信息描述的优化，这是默认的优化级别。与-O1 的调试视图的差异包括：
  - 源代码到目标代码的映射可能是多对一的，因为多个源代码位置可能映射到文件中的一个点，并且使用更激进的指令调度。
  - 允许指令调度跨越序列点。这可能导致在特定点报告的变量值与从源代码中读取的期望值不匹配。
  - 编译器会自动内联函数。
- O3：最大优化。启用调试时，此选项通常提供较差的调试视图。ARM 建议在较低的优化级别下进行调试，如果同时使用-O3 和-Otime，编译器会使用更激进的额外优化，如：
  - 高级标量优化，包括循环展开。这可以在稍微增加代码大小的前提下提供显著的性能优势，但可能会导致更长的构建时间。

- 更激进的内联和自动内联。

这些优化实际上重新编写了输入的源代码，导致目标代码与源代码的对应性最低，调试视图最差。--loop_optimization_level=option 控制了在-O3 -Otime 时执行的循环优化的数量。循环优化的数量越多，源代码与目标代码之间的对应性就越差。

要获取有关在-O3 -Otime 情况下，对源代码执行高级转换的额外信息，请使用--remarks 命令行选项。

由于优化影响了目标代码到源代码的映射，因此使用-Ospace 和-Otime 的优化级别选择通常会影响调试视图。如果需要简单的调试视图，-O0 选项是最佳选择。选择-O0 通常会使 ELF 映像的大小增加 7%到 15%。要减小调试表的大小，请使用--remove_unneeded_entities 选项。

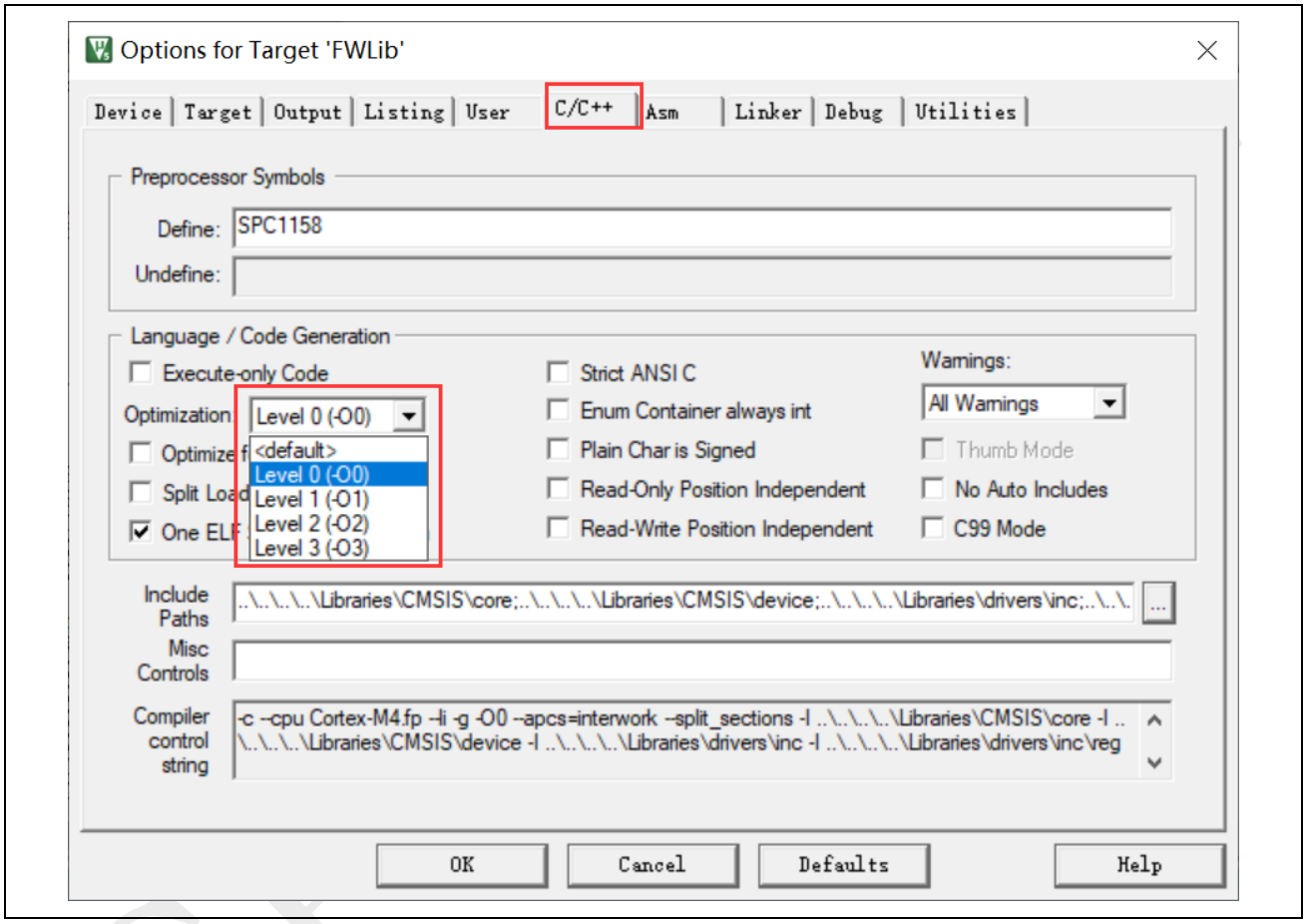
2 全局代码优化设置方式

在实际的业务场景，用户需要根据具体的情况设置 IDE 的编译优化等级。本章节将介绍 KEIL 和 IAR 的编译优化等级设置方式。

2.1 KEIL

打开 KEIL 选项，选择 C/C++选项卡，里面有设置优化等级的选项，如图 2-1 所示。

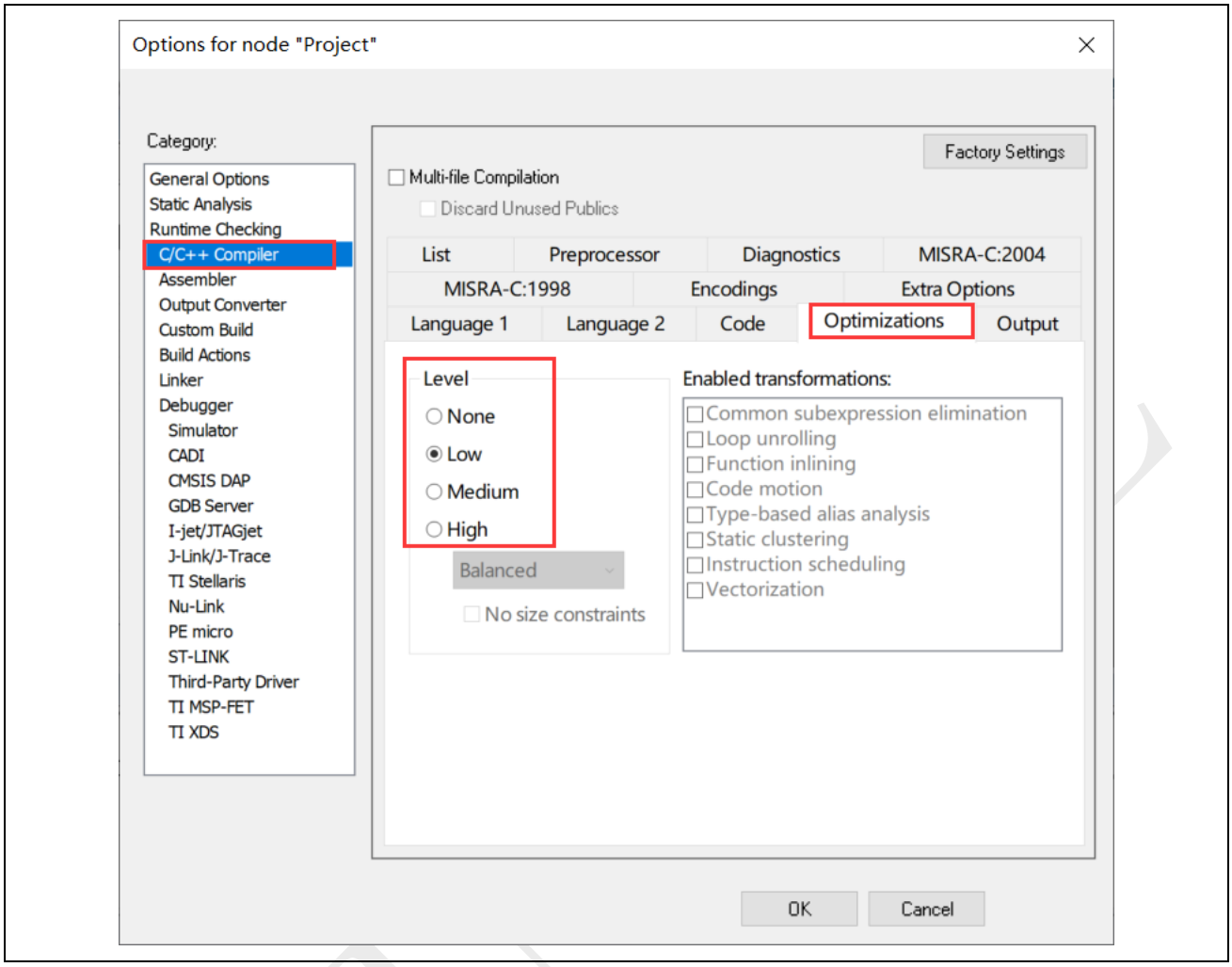
图 2-1 KEIL 优化设置



2.2 IAR

在 IAR 中，打开 Options，设置优化等级，如图 2-2 所示。

图 2-2 IAR 优化设置



3 局部代码优化

在实际工程中，除了对全部代码使用编译优化选项之外，还可以针对某一部分代码，指定编译优化等级。下面分别介绍 KEIL、IAR 对某一段代码，使用局部编译优化等级的方式。

3.1 KEIL

KEIL 可以使用许多编译指令，用于指示编译器使用特定功能。使用#pragma pop 和#pragma push 这对指令，将包含指定的代码，同时，使用#pragma On 指令指定这段代码的优化等级，On 可为 O0、O1、O2、O3 中的任意一个，下面举例说明。

KEIL 对指定的代码设置优化等级

```
#pragma push
#pragma O0

int Func1(void)
{
    printf("hello world 1\n");
}
int Func2(void)
{
    printf("hello world 2\n");
}

#pragma pop
```

上面的代码就将 Func1 和 Func2 函数的优化等级设置为 O0，其余的代码编译优化等级使用 KEIL 中的设置。在#pragma pop 和#pragma push 这对指令之间的所有代码优化等级都会设置为 O0，因此，二者之间可以写多行代码，而不是只能对一个函数起作用。

3.2 IAR

IAR 的局部代码优化使用指令是#pragma optimize=param，其中 param 可为如下参数的一个：none、low、medium、high。IAR 中的#pragma optimize=param 指令只能修饰某一个函数，因此，若是多个函数进行优化，需要分别使用#pragma optimize=param 配置。

IAR 对指定的代码设置优化等级

```
#pragma optimize=none
int Func1(void)
{
    printf("hello world 1\n");
}

#pragma optimize=low
int Func2(void)
{
    printf("hello world 2\n");
}
```

上述代码，将 Func1 设置的优化等级为 none、Func2 为 low。其余的代码使用 IAR 中设置的编译优化等级不变，不受影响。

4 示例

以 SPC1168 平台下的 clock.c 文件中的 CLOCK_ConfigurePLL 函数为例，说明对某些代码设置编译优化等级，IDE 使用 KEIL。

CLOCK_ConfigurePLL 为初始化 PLL 时钟的函数，在 CLOCK_ConfigurePLL 中，若是使用外部晶振时，需要等待 XO 的稳定才能继续运行，代码部分如图 4-1 所示：

图 4-1 等待 XO 稳定的代码

```
318
319     /* Enable XO */
320     if (u8RefClkFromXO == 1)
321     {
322         CLOCK->XOCTL.all =  XOCTL_ALL_PRECNT_(0xFFU)                /* Set pre-counted target value            */
323                           | XOCTL_ALL_FASTEN_ENABLE                 /* Enable noise injection for fast startup */
324                           | XOCTL_ALL_EXTRFB_DISABLE;               /* Use internal feedback resistor          */
325
326         /* Enable XO */
327         CLOCK->XOCTL.bit.EN = XOCTL_BIT_EN_ENABLE;
328
329         /* Wait XO ready */
330         u32TimeOut = 4000000U;
331         while (CLOCK->CLKSTS.bit.XORDY == CLKSTS_BIT_XORDY_NOT_READY)
332         {
333             if ((u32TimeOut--) == 0)
334             {
335                 return ERROR;
336             }
337         }
338     }
```

红框部分的代码功能为等待 XO 稳定，需要等待一段时间。在使用高等级优化选项（如-O3）时，可能会出现编译器执行激进的优化策略导致该部分的代码被优化掉，等待时间优化掉，不能得到预期的结果，导致初始化时钟失败，芯片无法正常工作。因此，为了防止此种情况的产生，需要对这部分代码进行处理，保证编译器不会优化该部分的代码。

将该部分代码的编译优化等级设置为-O0，不进行优化，可以避免上述情况。这样无论 KEIL 选择哪一种优化等级，该部分的代码使用的是指定的编译优化等级，从而保证代码在任意编译优化等级情况下都能产生预期的指令。

将该部分代码的编译优化等级设置为-O0，改动后的代码如下所示。

对指定的代码设置编译优化等级

```
#pragma push
#pragma O0
/*******************************************************************************
 * @brief      Configure PLL clock
 *
 * @param[in]  u8RefClkFromXO:  Reference clock selection
 *                              - \ref 1 -> clock source(reference) is from
external crystal clock
 *                              - \ref 0 -> clock source(reference) is from
internal RCO(32MHz)
 * @param[in]  u32RefClkHz   :  Reference clock frequency (Hz)
 * @param[in]  u32PLLClk     :  PLL output clock frequency (Hz)
 *
 * @return     SUCCESS or ERROR
 *

*******************************************************************************/
ErrorStatus CLOCK_ConfigurePLL(uint8_t u8RefClkFromXO, uint32_t u32RefClkHz,
uint32_t u32PLLClk)
{
    uint32_t u32TimeOut;
    uint32_t u32NOUT, u32NIN, u32NFB;
    uint32_t u32VCOTrim;
```

```
    uint64_t u64NFB;


    /* Check PLL frequency */
    if (u32PLLClk > PLL_MAX_FREQ)
    {
        return ERROR;
    }

    /* Enable XO */
    if (u8RefClkFromXO == 1)
    {
        CLOCK->XOCTL.all =  XOCTL_ALL_PRECNT_(0xFFU)                /* Set pre-
counted target value           */
                          | XOCTL_ALL_FASTEN_ENABLE                /* Enable noise
injection for fast startup */
                          | XOCTL_ALL_EXTRFB_DISABLE;              /* Use internal
feedback resistor          */

        /* Enable XO */
        CLOCK->XOCTL.bit.EN = XOCTL_BIT_EN_ENABLE;

        /* Wait XO ready */
        u32TimeOut = 4000000U;
        while (CLOCK->CLKSTS.bit.XORDY == CLKSTS_BIT_XORDY_NOT_READY)
        {
            if ((u32TimeOut--) == 0)
            {
                return ERROR;
            }
        }
    }
    else
    {
        /* Enable RCO0 */
        CLOCK->RCO0CTL.all |= RCO0CTL_ALL_EN_ENABLE;

        /* Wait RCO0 ready */
        u32TimeOut = 10000;
        while (CLOCK->CLKSTS.bit.RCO0RDY == CLKSTS_BIT_RCO0RDY_NOT_READY)
        {
            if ((u32TimeOut--) == 0)
            {
                return ERROR;
            }
        }
    }


    /* NIN number */
    u32NIN = u32RefClkHz / 8000000;
    if ((u32RefClkHz % 8000000) != 0)
    {
        u32NIN++;
    }


    /* NOUT number (VCO range is [400 : 600] MHz) */
    if (u32PLLClk > 150000000U)
    {
        u32NOUT = 3U;      /* (150 : 200] MHz */
    }
    else if (u32PLLClk > 100000000U)
```

```
    {
        u32NOUT = 4U;      /* (100 : 150] MHz    */
    }
    else if (u32PLLClk > 75000000U)
    {
        u32NOUT = 6U;      /* (75 : 100] MHz  */
    }
    else if (u32PLLClk > 50000000U)
    {
        u32NOUT = 8U;      /* (50 : 75] MHz  */
    }
    else if (u32PLLClk > 37500000U)
    {
        u32NOUT = 12U;     /* (37.5 : 50] MHz  */
    }
    else if (u32PLLClk >= 25000000U)
    {
        u32NOUT = 16U;     /* [25 : 37.5] MHz */
    }
    else
    {
        return ERROR;
    }


    /* NFB Number */
    u64NFB = (uint64_t)u32PLLClk * u32NOUT * u32NIN * 65536;
    u32NFB = (uint32_t)(u64NFB / (uint64_t)u32RefClkHz);


    /* Set NIN, NOUT and NFB */
    CLOCK->PLLCTL1.bit.NIN  = u32NIN;
    CLOCK->PLLCTL1.bit.NOUT = (u32NOUT - 1);
    CLOCK->PLLCTL1.bit.NFB  = u32NFB;

    /* Set ICP */
    u32VCOTrim = CLOCK->PLLCTL0.bit.VCOTRIM;
    CLOCK->PLLCTL0.bit.ICP = (96 * u32NFB) / (14 - u32VCOTrim) / 65536 / 100;

    /* Select PLL reference clock */
    CLOCK->PLLCTL0.bit.RCLKSELXO = u8RefClkFromXO;

    /* Enable PLL and output */
    CLOCK->PLLCTL0.all |= (PLLCTL0_ALL_EN_ENABLE | PLLCTL0_ALL_OE_ENABLE);


    /* Wait PLL clock ready */
    u32TimeOut = 100000;
    while (CLOCK->CLKSTS.bit.PLLRDY == CLKSTS_BIT_PLLRDY_NOT_READY)
    {
        if ((u32TimeOut--) == 0)
        {
            return ERROR;
        }
    }

    return SUCCESS;
}
#pragma pop
```

在 CLOCK_ConfigurePLL 函数的前后部分，使用#pragma 指令，设置 CLOCK_ConfigurePLL 函数的编译优化等级，因此，无论 KEIL 选择哪一个编译优化等级(-O0 至-O3)，该部分的代码使用的都是-O0 编译优化等级，从而保证 CLOCK_ConfigurePLL 正常工作。

IAR 同理，在 clock.c 中，对 KEIL 和 IAR 均进行了处理。