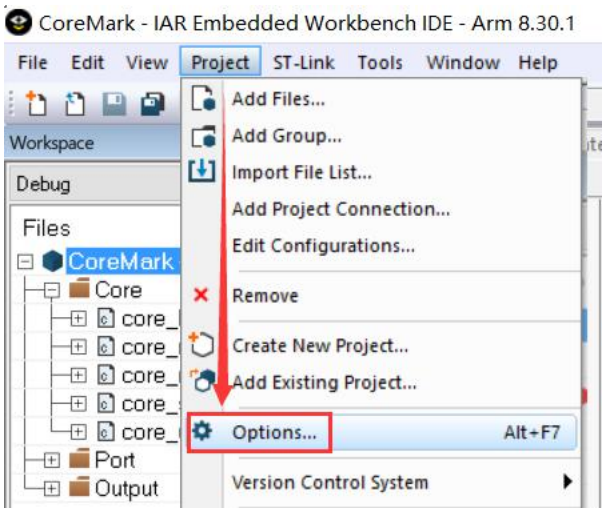
深入探讨代码优化设置

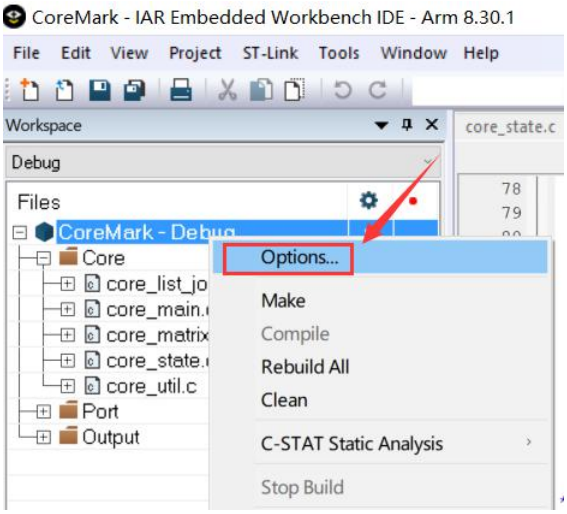
背景介绍

客户在开发嵌入式应用时经常会对应用的大小和性能有不同的要求，IAR 如何来满足不同客户在这方面的需求呢？在 IAR Embedded Workbench 中提供对应用各个层级的优化选项，如何利用好这些优化选项尽可能的满足客户就是我们今天要探讨的内容。

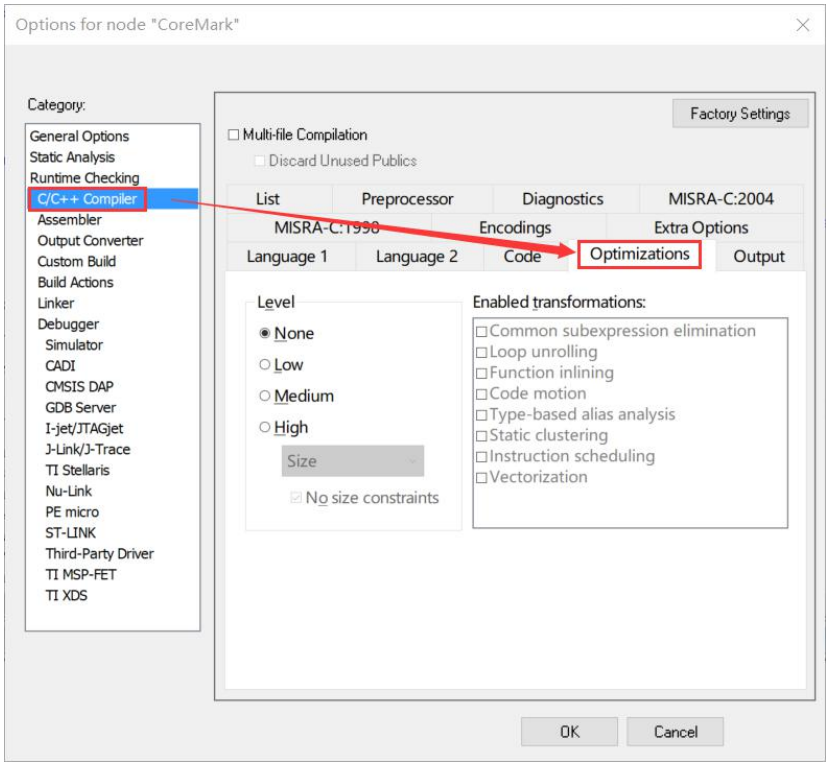
IAR Embedded Workbench 中的优化选项

当你再 IAR Embedded Workbench 中打开一个建立好的工作空间，选择其中的工程项目、文件组或文件从菜单 Project->Options 或 按右键选择 Options，如果你选择了工程项目就对整个工程项目进行优化，同理如果你选择了文件组，就对文件组进行优化，文件组在继承了工程项目优化的基础上在对其中的文件组中的文件进行专门的优化，同理如果你选择了某个文件，就对该文件进行优化，该文件如果在一个文件组中，那么它会在继承该文件组优化的基础上对该文件进行优化。





并在打开的 Options 窗口中的左侧选择 C/C++ Compiler，在右侧选择 **Optimations** 选项卡，IAR 的优化选项就展现在此了。



我们在上图中可以看到各种优化等级：

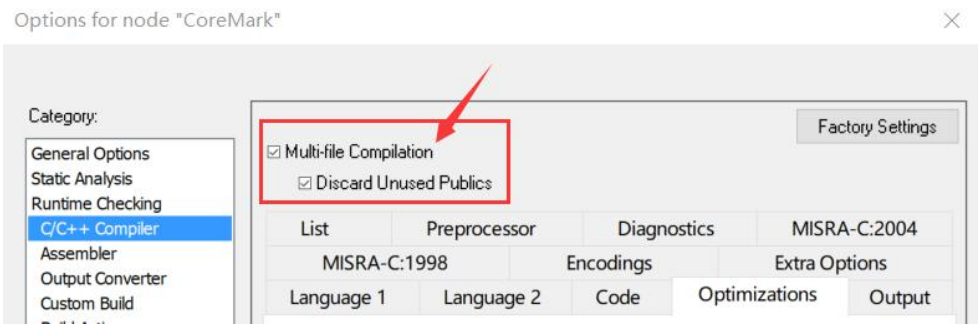
优化等级	说明
None	用于调试
Low	对变量的活动范围有限制
Medium	代码的死活分析和优化 死代码删除，冗余标签删除，冗余分支删除 代码提升，窥孔优化，寄存器内容分析和优化 公共子表达式消除，代码迁移，静态集群
High	指令调度，交叉跳转，高级寄存器内容分析 循环代开，函数内联，基于类型的别名分析
High(Belenced)	在代码大小和速度平衡的基础上进行优化
High (Size)	在代码大小尽可能小的基础上进行优化
High (Speed)	在不太增加代码大小的基础上进行速度优化
High(speed,No Size Constrains)	在不限制代码大小的基础上进行速度优化

我们可以根据不同的需求，选择适合自己应用的优化选项，编译自己的应用代码，满足实际需要。

部分优化内容介绍：

优化内容名称	说明
Common sub-expression elimination	公共子表达式删除：删除冗余的公共子表达式，该优化选项将减小代码长度并提高代码执行速度。默认在中和高优化选项中使用，可能引起调试困难。
Loop unrolling	循环代开：编译时，将可重复使用的小循环体，确定重复次数，以减少循环开销，该优化可以减少代码执行时间，但不能减少代码的大小。默认在高级优化中使用，可能引起调试困难。
Function inlining	函数内联：编译时已知定义的简单函数，将被嵌入到其调用函数的函数体内，以减小调度开销。该优化可以减少代码执行时间，但将增加代码长度，默认在高级优化中使用，可能引起调试困难。
Code motion	代码迁移：固定循环表达式和公共子表达式的赋值被迁移，以避免多重赋值操作。该优化可以减少代码长度和执行时间，可能引起调试困难。
Type-based alias analysis	基于类的别名分析：默认时编译器假定通过已声明的类型或 unsigned char 类型访问对象，但在高级优化选项时采用所谓“基于类的别名分析”优化方法。这意味着优化器认为应用程序遵循 ISO/ANSI 标准，并依照标准规则来决定采用指针间接访问受影响的对象。在高级优化中使用。
Static clustering	静态集群：将同一函数的静态变量和全局变量就近保存。这样编译器将采，用相同基址指针访问不同的对象。变量之间的空白区域件被删除。默认在中高级优化中使用。
Instruction scheduling	指令调度：IAR C/C++指令调度器对指令进行重新安排，使处理器内部资源冲突而导致的流水线停止的数量减至最低。默认在高级优化中使用。
vectorization	矢量化

另外，在 Options 窗口的**顶部**有多文件优化选项，一般编译器将一个源代码文件作为一个编译单元，，而多文件编译会将多个文件或者整个工程项目作为一个编译单位，这样可以提升编译器的视野，提高编译的效率。



Multi-File Compilation 对多个文件作为一个编译单元进行综合优化，进一步提升优化效果

Multi-File Compilation(Discard Unused Publics) 抛弃多个文件中未使用的全局变量

如果上述优化依然不能满足您的要求，您还可以使用 IAR 编译器提供的 #pragma optimize 指令对具体 C 代码某个函数或某几个函数进行精确优化。

IAR 优化指令的格式:

#pragma optimize=[**goal**] [**level**] [no_optimization...]

--**goal**: size, speed, balanced, no_size_constraints

--**level**: none, low, medium, high

例如:

```
#pragma optimize = speed
int SmallAndUsedOften( )
{
/* Do something here */
}
#pragma optimize= size
Int BigAndSeldomUsed( )
{
/* Do something here */
}
```

优化选项使用示例

CoreMark 介绍

CoreMark 是用来衡量嵌入式系统 CPU 或 MCU 性能的标准。由 EEMBC 组织于 2009 年提出，并且试图将其发展成为工业标准，从而代替陈旧的 Dhrystone 标准。CoreMark 的代码使用 C 语言写成，包含如下的算法：列举、数学矩阵操作和状态机，还包括 CRC 校验。通过执行这些算法评测处理器的性能。

CoreMark 数值越高，意味着性能更高。重要的是，CoreMark 测试的设计方式能让处理器内存的影响降到最小。因此，目前 CoreMark 已迅速成为业界量测与比较处理器性能的测试标准的基准。

IAR 提供了在 STM32F401-Necluo 板子上的基于 CoreMark 的测试程序，我们基于此测试程序做了各种优化选项下的实验，记录 IAR EWARM 编译器生代码大小和程序运行的时间和 CoreMark 的得分。

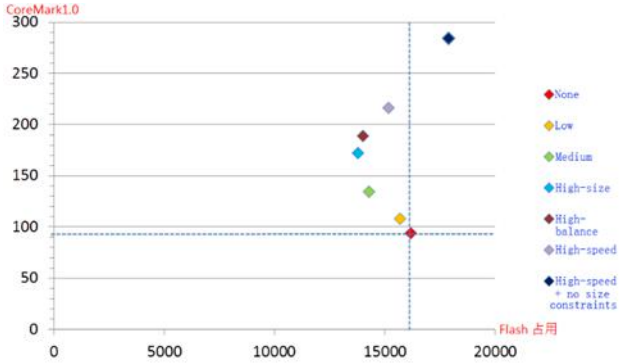
	优化选项设置	CodeMemory (bytes)	ConstMemory (bytes)	Flash Size (bytes)	CoreMark1.0	CoreMark/MHz
1	不启用优化(None)	14612	1492	16104	94.06	1.11
2	低级别优化(Low)	14128	1492	15620	108.06	1.28
3	中级别优化(Medium)	13844	371	14215	134.5	1.6
4	高级别优化(Size)	13344	371	13715	172.22	2.05
5	高级别优化(Balanced)	13572	371	13943	188.87	2.24
6	高级别优化(Speed)	14732	370	15102	216.34	2.57
7	高级别优化(Speed) +不限制尺寸	17456	370	17826	284.14	3.38

从实验数据可以发现：

①优化级别越高，代码的性能越好，性能最小提升了约 15%，最大则提升了约 200%；
②多数优化设定情况下，代码性能提高的同时，代码的尺寸也是减小的；
③ 对代码的 size 进行优化，代码尺寸减小，同时代码的性能也是提高的；
④ 只有在启用了“No size constrains”选项才会让代码尺寸明显的增大。

因此，我们得出结论：

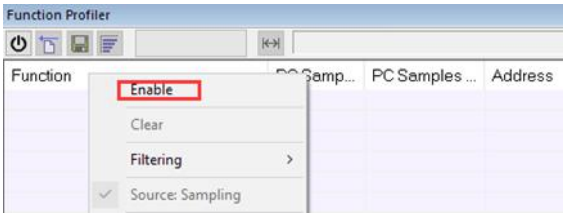
IAR 编译器的代码优化功能会使程序的性能获得明显提升。IAR 的优化技术充分考虑了 MCU 的存储空间有限这一特性，没有极端地追求高性能或者小代码，而是尽力兼顾二者之间的平衡。多数情况下，在代码性能提升的同时，代码的尺寸也是缩小的；只有在打开特定开关(不考虑代码大小)时，编译器才会以牺牲存储空间为代价，全力提升代码性能；



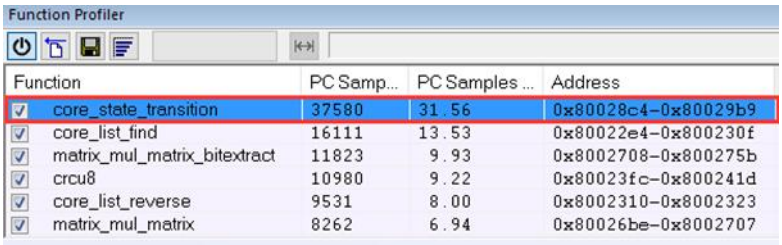
除此之外，我们还用此程序作了另外一个实验：

(1).先将实验工程针对 size 优化，然后编译下载到开发板运行，得到 CoreMark 跑分。

(2).通过 EWARM 的 Function Profiler 功能找出占用 CPU 时间最长的两个函数，然后通过预处理指令，对这两个函数进行 speed 优化。从菜单栏上的 ST-Link 下拉菜单打开 Function Profiler 窗口，然后在 Function Profiler 窗口空白处单击右键，从下级菜单选择“Enable”，或者直接单击 图标开启；



启用 Function Profiler 功能后执行 CoreMark 跑分程序，先点击 PC Samples 标题栏进行排序，完成后得到如下窗口的结果，可以看到占用 cpu 时间最长的两个函数分别是“core_state_transition”和”“core_list_find”，这两个函数占 cpu 时间总和有 45%左右；



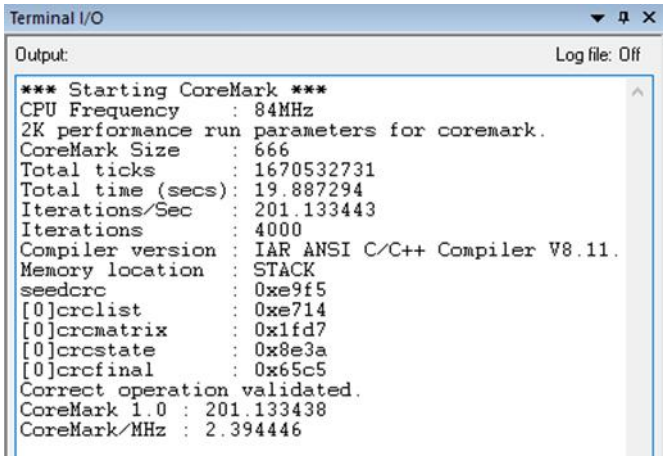
(3).在函数名上鼠标左键双击会跳转到函数源代码的位置，在函数定义的前面添加下面的预处理指令，对函数进行性能上的优化，然后重新编译代码；

```
183  #pragma optimize = no_size_constraints
184  enum CORE_STATE core_state_transition(
185      ee_u8 *str=*instr;
186      ee_u8 NEXT_SYMBOL;
187      enum CORE_STATE state=CORE_START;
```

```
370  #pragma optimize = no_size_constraints
371  list_head *core_list_find(list_head *list
372      if (info->idx>=0) {
373          while (list && (list->info->idx !
374              list=list->next;
375          return list;
```

从 build 窗口记录 Code Memory、Data Memory 的占用存储空间的大小值，并下载到开发板上运行得到 CoreMark 的跑分。

13 524 bytes of readonly code memory
371 bytes of readonly data memory
6 348 bytes of readwrite data memory



(4).将实验数据列在一起进行比较;

	优化选项设置	CodeMemory (bytes)	Const Memory (bytes)	FlashSize (bytes)	CoreMark 1.0	CoreMark/MHz
1	High(Size)	13344	371	13715	172.22	2.05
2	High(Balanced)	13572	371	13943	188.87	2.24
3	High(Speed)	14732	370	15102	216.34	2.57
4	High(Size)+函数优化	13524	371	13895	201.13	2.39

从实验数据来看，针对占用 CPU 时间最长的两个函数进行速度优化之后，程序的性能提升了约 16%，高于全局 Balance 优化；而代码的体积却比全局 Balance 优化时更小，可以说在两方面都得到了更好的结果；这也说明在嵌入式系统中，系统的性能往往是由执行得最频繁的几个任务决定的。针对最关键的任务进行速度优化，能够有效地提升系统的性能，同时付出最小的空间代价。

附录

本文编译自 IAR EWARM_DevelopmentGuide.ENU.pdf。
参考 IAR EWARM 实验指导手册