Keil MDK 编译器 AC5 和 AC6 优化选项重要内容和区别

使用过Keil MDK （Arm Compiler 6）编译器V6版本的读者应该发现了一个问题，V6版本速度比V5版本编译速度快很多。

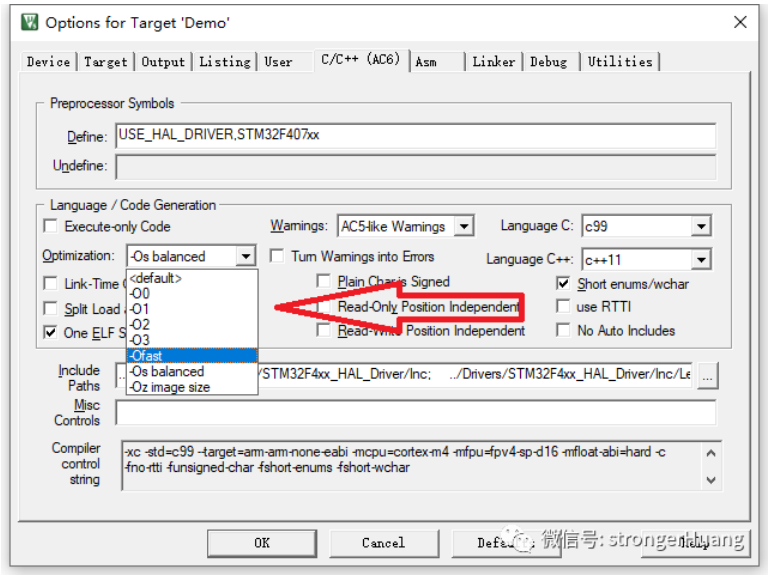
（说明：是V6版本编译器，不是V6版本MDK）

那你发现了Arm Compiler V6和V5有什么区别吗？ 集成在MDK中的优化选项又有哪些区别?

一、关于Arm Compiler 6

Arm Compiler 6（简称AC6）是用于Arm处理器的编译工具链，目前最新版本：Arm Compiler V6.14。

用于编译Coterx-M处理器的编译器很多，Arm Compiler就是其中一个，常用于Keil MDK、 Arm Development Studio（DS-5）中，还可用作独立工具链安装。



当然，除了Arm Compiler，针对Coterx-M的编译器还有很多，比如：GNU Compiler、 IAR Compiler、 CCS Compiler等。

Arm Compiler 6工具链包括：

armclang：基于LLVM和Clang技术的编译器和集成汇编器。

armasm：armasm语法汇编代码的旧版汇编程序。将armclang集成汇编程序用于所有新的汇编文件。

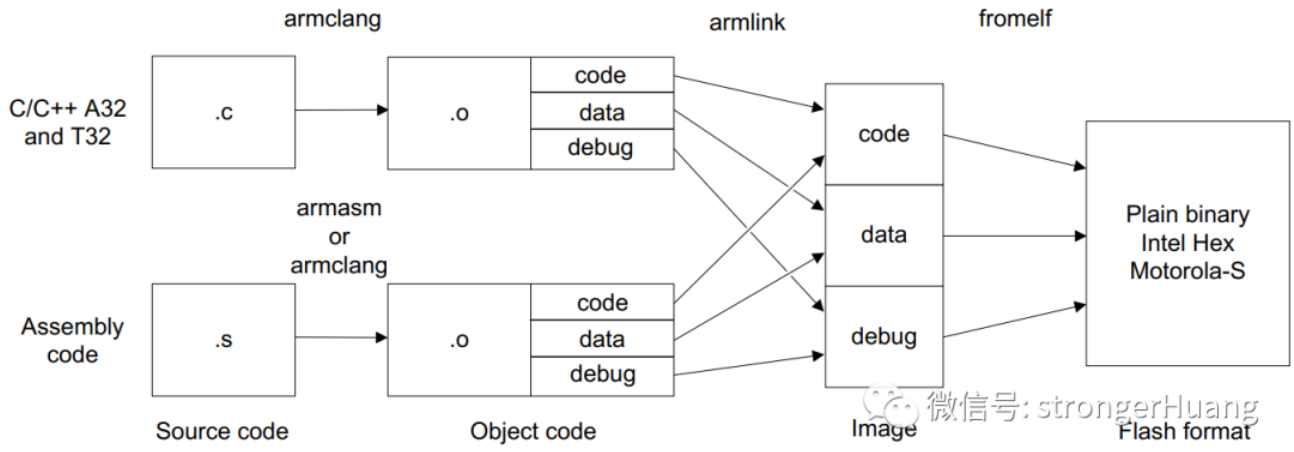
armar：使ELF目标文件集可以一起收集。

armlink：将对象和库组合在一起以生成可执行文件的链接器。

fromelf：镜像转换程序和反汇编程序。

Arm C libraries：嵌入式系统的运行时支持库。

Arm C ++libraries：基于LLVM libc++项目的库。



ARM Compiler 5（和更早版本）使用armcc编译器，而ARM Compiler 6将armcc替换为armclang，armclang基于LLVM，它具有不同的命令行参数、指令等，因此算是一个新的编译器。

更多参考内容和地址：

编译器Clang会代替GCC吗?

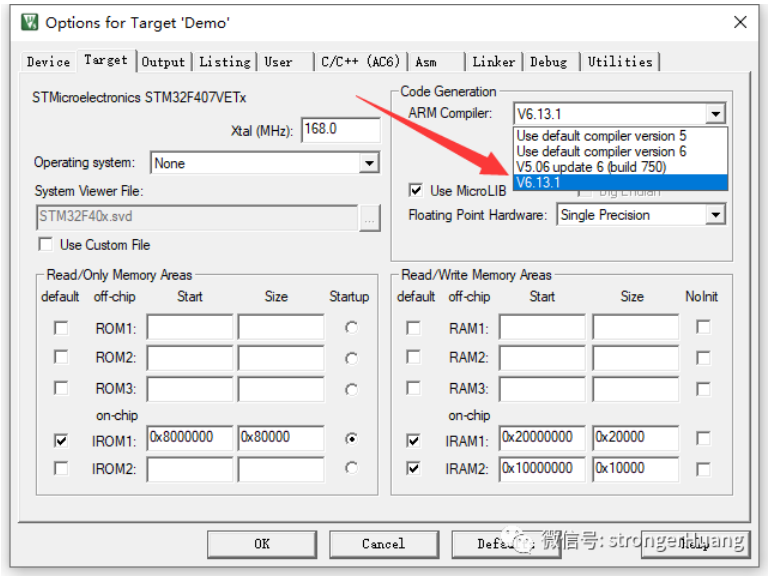
http://www2.keil.com/mdk5/compiler/6/

https://developer.arm.com/tools-and-software/embedded/arm-compiler/downloads/version-6

二、AC5和AC6

Arm Compiler 5（AC5）算是用的比较多的一代编译器，在Keil MDK V4版本及V5早期的版本都是使用AC5。

在2015年的时候，AC6发布了，并在随后新版本的MDK中集成了AC6，直到现在最新版本的MDK集成了AC6.13（可以修改版本）：

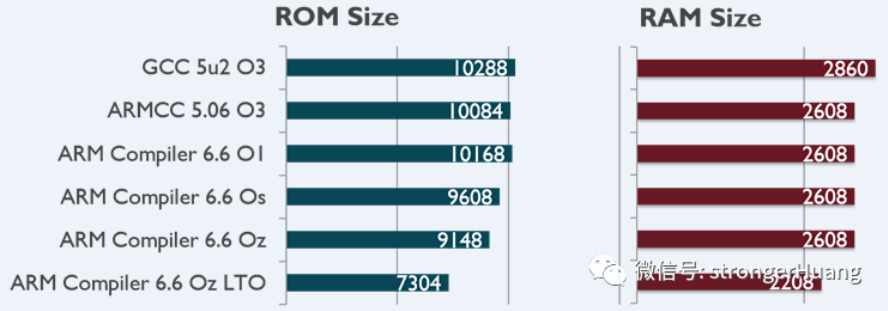


AC6相比AC5优势

AC6相比之前版本的编译器做了很多改动，大家最为直观的感受就是编译速度提高了很多，还有代码大小。

当然除了速度和大小，还有其他很多优势，比如：支持C ++ 14标准、使用TrustZone for Armv8-M为设备创建安全和非安全代码、兼容基于GCC创建的源代码，也就是GCC可以编译的源码它也能编译。

这是官方提供的代码大小对比：



AC5升级到AC6

AC5和AC6是不同的编译器，兼容性方面还是有差异，需要迁移。这个迁移过程官方提供有文档：

https://developer.arm.com/docs/100068/0614/migrating-from-arm-compiler-5-to-arm-compiler-6

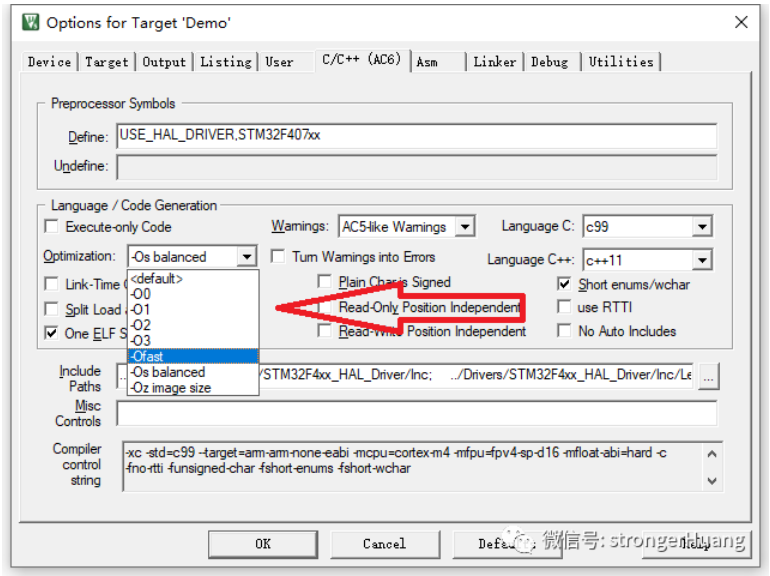
当然，也可以参看我之前分享的文章：

MDK-ARM编译器从V5升级到V6需要做哪些工作?

相关视频：

三、Keil MDK 优化选项

在Keil MDK中，相比AC5，使用AC6会增加几个优化选项：代码大小、速度、平衡等。



优化选项包含：

Optimization goal	Useful optimization levels
Smaller code size	`-Oz`
Faster performance	`-O2`, `-O3`, `-Ofast`, `-Omax`
Good debug experience without code bloat	`-O1`
Better correlation between source code and generated code	`-O0`
Faster compile and build time	`-O0`
Balanced code size reduction and fast performance	`-Os`

优化级别-O0

-O0禁用所有优化。此优化级别是默认设置。使用-O0 结果可以加快编译和构建时间，但比其他优化级别生成的代码要慢。与-O0其他优化级别相比，代码大小和堆栈使用率明显更高 。生成的代码与源代码紧密相关，但是生成的代码量更大，包括无用的代码。

优化级别-O1

-O1在编译器中启用核心优化。此优化级别提供了良好的调试体验，并具有比-O0更好的代码质量，堆栈使用率也提高了。Arm建议使用此选项以获得良好的调试体验。

-O1与-O0相比，使用时的区别是：

- 启用优化，这可能会降低调试信息的完整度。
- 启用了内联和尾调用，这意味着回溯可能无法提供打开功能激活的堆栈。
- 不会调用没有使用，或没有预期调用的函数，代码量更小。
- 变量的值在不使用后可能在其范围内不可用。例如，它们的堆栈位置可能已被重用。。

优化级别-O2

-O2与-O1相比，有更高的性能优化。增加了一些新的优化，并更改了优化的启发式方法。这是编译器可能生成矢量指令的第一个优化级别。它还会降低调试体验。

-O2与-O1相比使用时的差异是：

- 编译器认为内联调用站点可获利的阈值可能会增加。
- 执行的循环展开数量可能会增加。
- 可以为简单循环和独立标量运算的相关序列生成矢量指令。

可以使用armclang命令行选项禁止创建矢量指令-fno-vectorize。

优化级别-O3

-O3与-O2相比，有更高的性能优化。此优化级别允许进行需要大量编译时分析和资源的优化，并且与-O2相比更改了优化的启发式方法。-O3指示编译器针对生成的代码的性能进行优化，而忽略生成的代码的大小，这可能会导致代码大小增加。

-O3与-O2相比使用时的差异是：

- 编译器认为内联调用站点是有利可图的阈值增加。
- 执行的循环展开量增加。
- 在编译器管道中启用更积极的指令优化。

优化级别-Os

-Os目的是在不显着增加代码大小的情况下提供高性能。根据你的应用程序，提供的性能可能类似于 -O2或-O3。

-Os与-O3相比，可减少代码大小。但会降低调试体验。

-Os与-O3相比使用时的差异是：

- 降低编译器认为内联调用站点可获利的阈值。
- 显着降低了执行的循环展开量。

优化级别-Oz

-Oz目的是提供尽可能小的代码量。Arm 建议使用此选项以获得最佳代码大小。此优化级别会降低调试体验。

-Oz与-Os相比使用时的差异是：

- 编译器仅针对代码大小进行优化，而忽略性能优化，这可能会导致代码变慢。
- 未禁用功能内联。在某些情况下，内联可能会整体上减少代码大小，例如，如果一个函数仅被调用一次。仅当预期代码大小会减小时，才将内联启发式方法调整为内联式。
- 禁用可能会增加代码大小的优化，例如循环展开和循环矢量化。
- 循环是作为while循环而不是do-while循环生成的。

优化级别-Ofast

-Ofast从级别执行优化，包括使用 -ffast-math armclang选项执行的优化。

该级别还执行其他进一步的优化，可能会违反严格遵守语言标准的要求。

与-O3相比，该级别会降低调试体验，并可能导致代码大小增加。

优化级别-Omax

-Omax是最大程度的优化，并专门针对性能优化。它支持从级别进行的所有优化，以及链接时间优化（LTO）。

在此优化级别上，Arm Compiler可能会违反严格遵守语言标准的规定。使用此优化级别可获得最快的性能。

与-Ofast相比，该级别会降低调试体验，并可能导致代码大小增加。

如果你使用-Omax进行编译，并具有单独的编译和链接步骤，你还必须在armlink命令行中包括-Omax。