

## Omniscient Code Generation Technology Stretches MCU Memories 利用 OCG 技术扩展 MCU 内存

by Clyde Stubbs  
CEO and Founder, HI-TECH Software  
HI-TECH Software 公司 CEO 兼创始人 Clyde Stubbs

Engineers often unwittingly specify microcontrollers with more flash and SRAM than is really necessary. This is because traditional C-compilation technology often results in object code, pointers and stacks that are much larger than they need to be. In a worst-case situation, a designer may have to migrate the design to a more complex, expensive, and power-hungry 32-bit device - not because he needs the extra performance, but because his otherwise ideal microcontroller is not available with enough memory resources to accommodate the final program.

工程师通常会无意识地在 MCU 中加入多于其真正所需的闪存和 SRAM。这是因为，传统的 C 编译技术通常会产生远大于工程师需要的结果代码、指针和堆栈。在最坏情况下，设计人员可能不得不将设计转向更为复杂、昂贵且功耗更大的 32 位产品，这并不是因为他需要额外的性能，而是因为他理想的 MCU 没有足够多的内存资源来存放最终程序。C-language programs are usually developed by teams of engineers who are often geographically dispersed. This process can lead to redundant code in multiple modules and inconsistent variable declarations between modules. Traditional compilation technology shadows the design process, compiling each module separately with no regard for what goes on in the other modules. The compiler never has an opportunity to look at how pointers, stacks, variables and functions are used throughout the whole program. Rather than risking program failure from stack overflows and the like, they tend to over-allocate memory space for pointers, stacks and registers. Traditional compilers also are unable to see redundant code or inconsistent variable declarations. Without the ability to see the whole program, traditional compilers cannot take advantage of the wide variety of memory maps, and register and stack configurations available in today's microcontrollers. All too often, the programmer must resort to manual optimizations that compromise portability. Figure\_1 (Conventional compilation)

组成一个 C 语言程序开发团队的工程师们，通常分布在全球各个角落。这样的流程会产生多个模块中的冗余码以及模块间不一致的变量声明。传统的编译器技术会对设计流程进行映像(shadow)，对每个模块进行独立编译，而不去考虑其它模块中发生的事情。编译器从来不会想要去了解整个程序中到底使用了多少个指针、堆栈、变量以及函数。与其冒着让程序由于堆栈溢出而失败的风险，他们更愿意为指针、

堆栈和寄存器留出足够的空间。传统的编译器还无法获知冗余码或不一致的变量声明。由于没有观察整个程序的能力，传统的编译器无法充分发挥内存映射多样性带来的优势，并利用当前 MCU 中寄存器和栈的配置。通常情况下，程序员必须手工进行优化，以获得性能上的折衷。

**“Omniscient Code Generation”**. A new compiler technology, called Omniscient Code Generation (OCG), can effectively double the capacity of on-chip flash memory by cutting the code footprint by as much as half. OCG also frees up 10% to 15% of SRAM resources.

**“全知代码生成”**。一种被成为全知代码生成(OCG)的新型编译器技术，通过将程序代码量减少一半，有效实现了片上 flash 存储能力的翻番。

### **Figure 1 – Compilation Using OCG**

图 2：使用 OCG 技术进行编译。

Rather than relying completely on the linker to uncover errors between the independently compiled modules, an OCG compiler defers object code generation until a view of the whole program is available. It completes the initial stages of compilation for each module separately. Then, instead of compiling to machine code instructions in an object file, it compiles each module to an intermediate code file that represents a more abstract view of each module *before* producing any machine instructions or allocating any registers.

并不是完全依赖链接器来发现完全独立的被编辑模块间出现的错误，OCG 编译器会延迟结果代码的产生，直到对整个程序有了全局认识。首先，它会对每个模块进行单独编译；下一步，不是编译成目标文件中的机器代码指令，而是将每一个模块编译成中间代码文件，从而在产生机器指令或分配寄存器之前对每个模块有更直观的认识。

**Call Graph: A Clearer View.** The intermediate code files generated by the OCG compiler are loaded into a *call graph* structure. Any library functions referenced by the program are also located and extracted.

**调用关系图：更为清晰的认识。**由 OCG 编译器产生的中间代码文件被置于调用关系图架构中，对程序中引用到的任意库函数进行定位和提取。

A traditional compiler uses fixed calling conventions for register usage; when a function is being compiled, any other functions called by it will be assumed to use all the registers that the calling

convention decrees, which may result in sub-optimal use of registers, and thus more use of stack memory than really required.

传统的编译器使用固定的呼叫惯例进行寄存器使用；当一个函数被编译时，任何被它调用的其它函数都假设使用了呼叫惯例中规定的所有寄存器，这样可能会导致不理想的寄存器使用情况，从而使用了高于实际需求的栈存储。

The OCG code generator uses the Call Graph to look at all the program modules and identify any functions that are called recursively or re-entrantly, such as those called from both main-line code and interrupt functions. It allocates dynamic stack space for storage of local variables to prevent re-entrant function calls from overwriting existing data. It also searches the call graph for any functions that are never called by and removes them.

OCG 代码生成器使用了调用关系图技术，来观察所有的程序模型，并辨识被递归或可重入调用的任意函数，例如发生在主代码行(main-line code)和中断函数间的调用。它分配动态栈空间来存储局部变量，以防止现有数据重写所发生的重入函数调用。它还搜索调用关系图寻找从未被调用的函数，并将其移除。

For MCUs that do not have hardware stacks, making it difficult to implement re-entrant or recursive code. OCG gets supports re-entrant function calls in these architectures by building separate call graphs for both main-line and interrupt code. Any functions that appear in more than one call graph can be replicated, each with its own local variable area. Using this technique reentrancy can be implemented without a conventional stack.

对没有硬件堆栈的 MCU 来说，实现重入或递归代码很困难。通过建立独立的主代码行和中断代码调用关系图，OCG 在这些架构中获得了重入函数调用的支持。出现超过一次调用关系图的任意函数都可以被复制，每一个都带自己的局部变量区域。通过使用这种技巧，重入可以在不需要传统堆栈的情况下实现。

For non-re-entrant, non-recursive functions, the OCG compiler generates optimally sized function stacks with exactly enough memory to accommodate each function's maximum depth. Because the call graph for all functions has already been determined, functions which are executed at different times can share the same SRAM space for their static compiled stack. This feature reduces stack space to the absolute minimum required and frees up more SRAM for data. A compiled static stack

also reduces the likelihood of stack overflows that occur when a dynamic stack expands into the data variables' space in the SRAM.

对于不可重入、非递归函数来说，OCG 编译器产生了尺寸优化的功能栈，能够提供恰到好处的存储容量来容纳每个函数的最大深度。由于针对所有函数的调用关系图已经确定下来，那么在不同时间内执行的函数就可以针对其静态编译栈共享同一个 SRAM 空间。这样的特性使得栈空间降低到必需的绝对最小值，并且为数据存储释放了更多的 SRMA 空间。一个被编译的静态栈还能够减少栈溢出的可能性，当动态栈扩展到 SRAM 中的数据变量空间时，就会发生栈溢出现象。At the end of this optimization, compiled stack space can be allocated before any machine code has been generated. OCG knows exactly how big the stack needs to be and where it is located before it generates any code.

在优化的最后环节，当机器码被产生之前，对编译过的栈空间进行分配。OCG 很确切的知道需要多大的栈，以及代码产生之前它会被置于何处。

**Pointer Reference Graph Identifies Inconsistent Variable Declarations.** Once the stacks have been optimized, an OCG algorithm uses each instance of a variable having its address taken, plus each instance of an assignment of a pointer value to a pointer (either directly, via function return, function parameter passing, or indirectly via another pointer) and builds a "pointer reference graph" that identifies all objects that can possibly be referenced by each pointer. This information is used to determine which memory space each pointer will be required to access.

利用指针引用图识别不一致的变量声明。一旦堆栈被优化，OCG 算法就能够利用每一次的变量取址，加上每一次的指针值分配(无论通过函数回归、函数参数传递的直接分配，还是通过其它指针的非直接分配)，构建“指针引用图”，来识别所有可能被指针引用的对象，这一信息被用来决定每一个指针将被要求访问存储空间的哪部分。

Any conflicting declarations of the same object from different modules are detected an informative error message issued to the user. Variables that are never referenced are deleted.

来自不同模块对相同对象的任何冲突性声明，都会被发现，并且形成错误信息提示汇报给用户。从来没有被引用的变量也将会被发现并被删除。Once the set of used variables and pointers is complete, OCG allocates memory of both the stack (compiled or dynamic) based on the call graph and allocates global and static variables based on the pointer reference graph. Where there are multiple memory spaces (e.g. an architecture with banked RAM), the variables accessed most often in the

program can be allocated to the memory spaces that are cheapest to access. On an 8051, for example, this would be internal, directly addressable RAM, rather than external RAM which must be accessed via a pointer.

一旦被使用的变量和指针设置完成，OCG 就会基于调用关系图，对编译过的栈或者是动态栈所使用的内存进行分配，并以指针引用图为基础分配全局和静态变量。在有多个内存空间的地方(例如带有分组 RAM 的架构)，那些在程序中被经常访问到的变量，可能会被分配到最容易访问到的存储空间。例如，8051 上应该内嵌可直接寻址的 RAM，而不是必须通过一个指针来读取的外部 RAM。

The OCG code generator also sizes each pointer and encodes it in a way that is optimally efficient for the target architecture. Each pointer is automatically assigned an optimized set of address spaces, without any intervention from the programmer.

OCG 代码生成器也会按照每个指针的大小排列顺序，并且以最终架构最优化为前提对其进行编码。每一个指针自动被分配一个最优化的地址空间，不需要程序员的任何干涉。

### Figure 2 – Pointer Reference Graph

图 4：指针引用图。

**Optimizes Non-contiguous Memory Addresses.** Standard C assumes a single linear address space while in reality many embedded processors have complex, non-linear memory spaces, often with different address widths. Conventional compilers are as likely as not to assign variables to pages that require cycle-intensive access – particularly during interrupts. In contrast, the OCG compiler knows which variables will be used in interrupt routines and assigns them to the appropriate memory space, relieving the programmer of tedious assembly programming often required to accommodate non-linear memory architectures.

**优化非邻接存储器地址。** 标准 C 所假设的是一个单独的线性地址空间，而实际上许多嵌入式处理器所拥有的是复杂、非线性存储器空间，通常它们还拥有不同的地址宽度。传统的编译器不太可能将变量分配到页，这需要循环增强型(cycle-intensive)访问，特别是在中断过程中。与之相反，OCG 编译器知道哪些变量将被用于中断程序，并且会将其分配在合适的存储空间，帮助程序员从单调乏味的汇编编程(通常需要提供非线型的存储器架构)中解放出来，。

Figure 3- SRAM- Utilization with Conventional and OCG Compilation

图 5: 传统和 OCG 编译的 SRAM 使用情况。

**Bottom-up Code Generation.** Once the pointers and variables are assigned memory space, machine code generation can begin. The OCG code generator begins at the bottom of the call graph, starting with those functions that do not call any other functions. Automatic in-lining of these functions may be performed if desired. In any case, the code can be generated without the constraints of rigid calling conventions. Code generation then proceeds up the call graph so that for each function the code generator knows exactly which functions are called by the current function. The code generator knows exactly which registers and other resources are available at each point. Calling conventions can be tailored to the register usage and argument type of a function, instead of following a blind set pattern.

**自下而上生成代码。**一旦指针和变量分配好了存储空间，就轮到生成机器代码。OCG 代码生成器开始于调用关系图底部，以那些不会调用其它函数的函数开始。如果希望的话，这些函数还可以执行自动排列(in-lining)。任何情况下，代码的生成都不需要严格的调用环境限制。代码生成器继续沿着调用关系图上行，所以对任何函数来说，代码生成器都很清楚地知道当前函数将调用哪些函数。代码生成器很确切的了解，在每一个指针处可以获得哪些寄存器和其它资源。呼叫管理可以根据寄存器的使用和函数的自变数进行裁减，而不是盲目跟随一种模式。

**Customized Library Functions.** Programmers often use predefined C-code libraries to performed routine functions. For example, the `sprintf()` and `printf()` functions offer options for formatting text strings or output. They can take up 5 KBytes of memory, even though most programs need only a fraction of the available options. The OCG code generator analyzes all the format strings in the program, determines the exact subset of format specifiers and modifiers required, and creates a customized version. If the output is limited to simple string copying, the C-library code will be automatically reduced to 20 or 30 Bytes.

**定制的库函数。**程序员通常会使用预定义的 C 代码库来实现常规函数。例如，`sprintf()`和 `printf()`提供对字符串格式化或输出的选项。它们可以占用 5KB 的存储空间，虽然大多数程序只需要所有选项中的一部分。OCG 代码生成器对程序中所有的格式串进行分析，确定指定格式化数据的精确子集以及需要进行的修改，并生成一个定制化版本。如果输出被限制为简单的串复制，那么 C 库代码将会被自动减少到 20-30B。

**Customized Runtime Startup Code.** Newer embedded compilers provide canned startup code that clears uninitialized static and global variables to zero. This code is often much larger than necessary for a given program. For example, if the program has no uninitialized global variables, there is no need to include code to clear them. The omniscient code generator knows the state of all global variables and can create the smallest possible runtime startup code. In a minimal case, the startup code may be completely empty.

**定制化的运行启动代码。**最新的嵌入式编译器提供“罐装”好的启动代码，对未初始化的静态和全局变量清零。这个代码通常比给定程序所需的代码长很多。例如，如果程序中没有未初始化的全局变量，那么就没有必要包含需要将其清零的代码。全知的代码生成器知道所有全局变量的状态，并且能够生成最小的可能运行启动代码。在极端情况下，启动代码可能完全是空的。

**No Hand Crafting.** OCG compilers perform an analysis of the whole program every time the program is recompiled and makes optimal decisions about memory placement, pointer scoping, and stack allocation, without any special directives or language extensions from the programmer. OCG allows highly optimized embedded C programs to be written *without* resorting to hand-crafted assembly code.

**无需手动。**在每一次程序重新编译时，OCG 编译器都会执行对整个程序的分析，并且对内存的布置、指针范围，以及堆栈分配进行优化，这些过程都无需程序员的特别指令或语言扩展。OCG 允许写入高度优化的嵌入式 C 程序，而无需诉诸于手动的汇编代码。

**Smaller Code Better Performance.** OCG compilers are commercially available from HI-TECH Software for Microchip's PIC-18 and Cypress' PSoC Mixed Signal Array, and additional OCG-enabled compilers are planned for other 8-, 16- and 32-bit MCUs and DSCs.

**更少的代码，更优的性能。**现在，工程师可以从 HI-TECH 软件公司买到针对 Microchip PIC-18 和 Cypress PSoC 混合信号阵列的 OCG 编译器。此外，附加 OCG 功能的编译器还计划支持其它的 8/16/32 位 MCU 和 DSC。

When compared to conventional compilers, OCG code size is frequently 40% to 50% percent smaller. (Table 1)

与传统编译器相比，OCG 代码量减少了 40-50%。(表 1)

**Conventional vs OCG Compilation**  
**Comparative code sizes (bytes) for ts057.c (with printf)**

<b>Cypress PSoC</b>		<b>Code Size</b>	$\Delta_{\infty}$
ImageCraft Compiler		11957	
HI-TECH C PRO for the PSoC Mixed-Signal Array		6129	-49%
<hr/>			
<b>MicroChip PIC18</b>		<b>Code Size</b>	$\Delta_{\infty}$
IAR EW18		14855	□
HI-TECH PICC-18 PRO		7488	-50%
<hr/>			
MicroChip C18		12620	
HI-TECH PICC-18 PRO		7488	-41%

**传统编译 vs OCG 编译**

**比较 ts057.c (借助 printf)所用到的代码量**

<b>Cypress PSoC</b>		<b>代码量</b>	$\Delta_{\infty}$
ImageCraft 编译器		11957	
面向 PSoC 混合信号阵列的 HI-TECH C PRO		6129	-49%
<hr/>			
<b>MicroChip PIC18</b>		<b>Code Size</b>	$\Delta_{\infty}$
IAR EW18		14855	□
HI-TECH PICC-18 PRO		7488	-50%
<hr/>			
MicroChip C18		12620	
HI-TECH PICC-18 PRO		7488	-41%

By combining all program modules into one large program, the compiler can identify and consolidate re-entrant code, identify inconsistent variable declarations between modules, eliminate redundant code that may exist in more than one module and optimize all the stacks, registers, pointers and memories to exploit the unique advantages of the MCU architecture.

通过将所有程序模块整合到一个大程序中，编译器能够识别并巩固重入代码，辨识模块间出现的不一致代码声明，减少可能存在于多个模块中的冗余码，并优化所有的栈、寄存器、指针以及内存，从而开拓 MCU 架构的独有优势。

Denser code results in faster execution because there is less code to execute. Denser code provides a cost advantage because it allows programmers to use devices with smaller flash and SRAM memories. It also extends the life of existing 8- and 16-bit architectures by allowing them to accommodate up to twice as much code and by freeing up SRAM resources. Equally important it eliminates the need for hand-crafted assembly code.

代码密度的提高使得执行速度加快，因为执行的代码量减少了。代码密度提高提供了成本上的优势，因为它允许程序员使用带有更小 flash 和 SRAM 内存的器件。通过将可存储代码量翻番，以及释放 SRAM 资源，它同事扩展了现有 8/16 位架构的生命周期。同样重要的是，它减少了对手动汇编代码的需求。

Clyde Stubbs is CEO and founder of HI-TECH Software. He received his BA (Hons) in Computer Science from the University of Qld, Australia, in 1982.

Clyde Stubbs 是 HI-TECH 软件公司的 CEO 兼创始人。1982 年，他获得了澳大利亚昆士兰大学计算机科学的优等学士学位。