Omniscient Code Generation
A whole-program compilation technology for superior code density and performance

by Clyde Stubbs
CEO and Founder, HI-TECH Software

Introduction

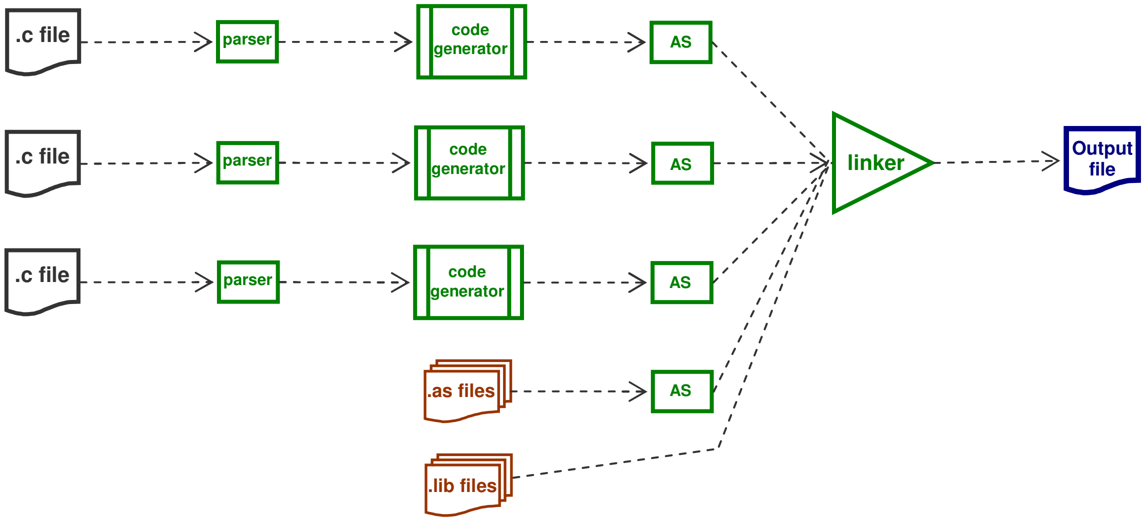
The C and C++ programming languages are the heart and soul of embedded systems programming. To a large extent, these high-level languages, and the compilers that translate them into machine code, free programmers from the tedium of assembly language programming. They also provide a measure of code portability between processor architectures. However, while computing hardware has made huge advances in the last 30 years, compiler technology has basically stood still.

The original C language, and the compilers that support it, were developed in the 1970s, when computers could not run, much less compile, the large programs of today. Programs were both developed and run on the target platform. Typically computer memories rarely exceeded 1 MB RAM and a processor clock frequency of 400 kHz was pushing the envelope.

In order to conform to the limited memories and processing power of the time, programs were split into multiple small modules, each of which was compiled *independently* into a sequence of low-level machine instructions. A *linker* joined these object modules, along with code extracted from pre-compiled libraries, to create the final program. This approach enabled quite large programs to be compiled on machines with limited memory.

Memory size and processor speed are now no longer constraints.

Independent Compilation Sequence



Today, 8-bit embedded microcontrollers exceed the typical 1970s computer, in terms of clock frequency — 40 MHz-plus — and they may integrate a megabyte or more of on-chip memory. However, MCU architectures have evolved with radically different memory maps, and register and stack configurations. All too often, conventional C compilers cannot make good use of these vastly different architectures requiring manual optimizations that compromise portability.

While processor architectures have evolved to meet the needs of 21st century applications, compilers have barely changed at all. Embedded programs are still broken up into smaller modules, each of which is compiled independently, with little regard for cross-module register optimization, re-entrant code or variable declarations.

There are good reasons to break embedded code into modules. Code is frequently developed by teams of engineers, each of whom is responsible for one or more modules. This approach speeds system development and also makes the application easier to manage. However, there is no reason today to compile each module independently. There are many reasons not to do so.

Inconsistent Definitions between Modules Conventional compilers that compile program modules independently can introduce several types of problem into the program because each module is compiled without knowledge of the other modules. For one, C has always presented difficulties in ensuring that variables and other objects used in multiple modules have consistent definitions. Although it is possible to have the linker check for incompatible re-declarations of variables by different modules, this approach adds complexity and doesn't always solve the problem due to the fact that the linker may not have enough information to detect the human error.

Sub-optimal register allocation The way in which arguments are passed and values returned, is another potential drawback in independent module compilation. Each calling convention contains a set of rules that defines which CPU registers are to be preserved across calls. *All* functions must adhere to the same calling conventions. Since it is impossible to know at the time of compilation which registers will and will not actually be used by a called function, these rules can result in sub-optimal register allocation. This is particularly true with small embedded processors.

Non-portable code and subtle bugs Processor memory architectures pose additional problems. Many embedded processors have a complex and non-linear set of memory spaces, often with different address widths. Standard C is difficult to map onto these architectures because it is often impossible to know at compile time in which memory space a variable will be located, or which spaces a given pointer will be required to access.

For example, a pointer might need to address memories of different address widths, such as 8-bit wide RAM and 16-bit, or wider, ROM. The programmer can achieve efficient pointer usage in these architectures by explicitly declaring the address spaces that a pointer will access. However, this solution is architecture-specific and results in non-portable code. It also increases the likelihood of subtle bugs being introduced**.**

Compiled Stack Limitations Many small embedded processors do not have a fully or efficiently addressable stack for the storage of local variables. This situation is usually handled by a *compiled stack* where local variables are statically allocated in memory, based on a *call graph*. Unfortunately independent compilation cannot know the call graph until link time. Using a compiled stack requires the programmer to nominate any functions that are called re-entrantly, so that they can be dealt with accordingly. If the programmer nominates these incorrectly, the error is not revealed until link time, by which time it is too late to make changes.

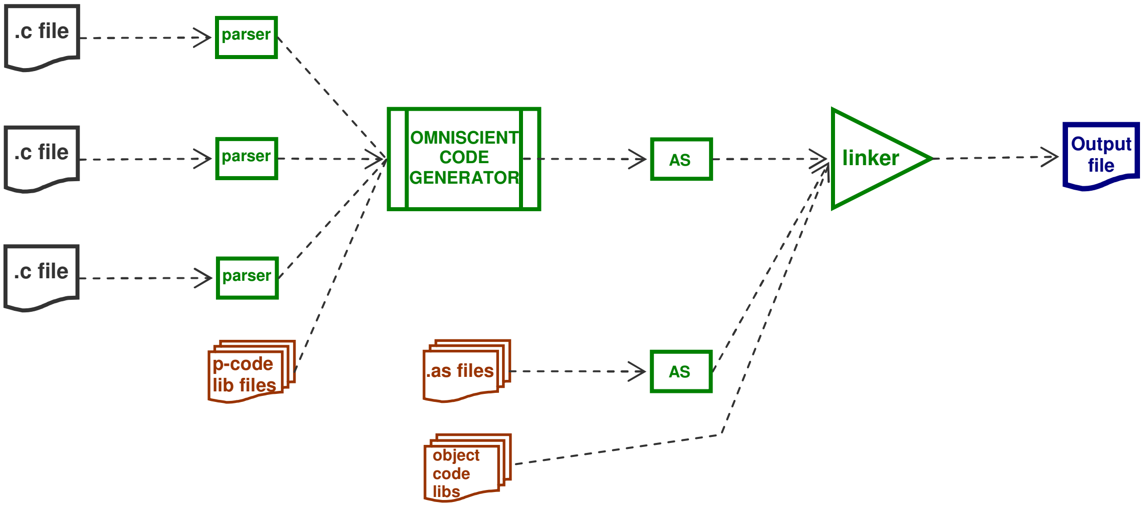
A New Approach - OMNISCIENT CODE GENERATION ™

Today the CPU speed and memory size of a typical host desktop or server computer exceed those of a typical embedded target by several orders of magnitude. Consequently, it is now quite practicable to generate code for the entire program at once, allowing all routines, variables, stacks and registers to be optimized based on the entire program. Although it is theoretically possible to compile a very large program from a single source, the fact is that most programs are written by teams of engineers, each of whom is responsible for one or more modules of functionality. As a result, embedded systems will always be compiled from multiple files.

A new compiler technology, called Omniscient Code Generation (OCG), takes advantage of the capabilities of today's host systems to achieve fully optimized object code that takes into account all the variables, functions, stacks and registers represented in all program modules. Rather than relying completely on the linker to uncover errors in independently compiled modules, an OCG compiler completes the initial stages of compilation for each module separately, but defers object code generation until the point at which a view of the whole program is available. It uses a global view of the compiled program, along with various other optimization techniques made possible by the complete information that is then available to the code generator. This approach solves most of the problems associated with independent compilation, described above.

An OCG compiler still compiles each module separately. However, 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. It does not produce actual machine instructions or allocate registers at this time.

Compilation Sequence with OMNISCIENT CODE GENERATION^{TM}



Call Graph: A Clearer View

Once all the intermediate code files are available, they are loaded by the code generator into a *call graph* structure. The code generator also searches intermediate code libraries and extracts, as required, any library functions that are referenced by the program. Once the call graph is complete, functions that are never called may be removed, thus saving space.

The call graph also allows identification of any functions called recursively (not common in embedded systems) or re-entrantly, such as those called from both main-line code and interrupt functions. These functions must either use dynamic stack space for storage of local variables, or be managed in other ways to ensure that a re-entrant call of the function does not overwrite existing data. This is achieved in a way that is completely transparent to the programmer, and without requiring non-standard extensions to the language.

If a compiled stack is to be used, it can be allocated at this point, before any machine code has been generated. OCG knows exactly how big the stack is and where it is located, before the code is generated.

Pointer Reference Graphs

With a complete Call Graph the compiler then builds reference graphs for objects and pointers in the program. Any conflicting declarations of the same object from different modules can be detected at this point and an informative error message issued to the user. Any variables never referenced can be deleted.

Determining the memory space for each pointer is one of the most important features of OCG. An 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 data reference graph (Pointer Reference Graph). The graph is completed and then 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.

Once the set of used variables and pointers is complete, OCG allocates memory for the stack (compiled or dynamic: the call graph is used to determine the stack size) at the same time as it allocates global and static variables. 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.

Each pointer variable now has a set of address spaces that it will be required to access, without any specific directions provided by the user. This allows each pointer to be sized and encoded in a way which is optimally efficient for the particular architecture, while still being accurate.

Bottom-up Code Generation

Generation of machine code now 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, and therefore also knows exactly what 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 set pattern.

Customized Library Functions

An omniscient code generator has a truly global[1] view of the program. Therefore, complex library functions can be implemented in a way that is specific to each particular program. A good example of this is the C library functions `sprintf()` and `printf()`. These workhorse routines for formatting text strings or output are enormously useful, but can occupy a large code footprint (5 KBytes or more), if implemented in their entirety.

The OCG code generator can analyze the format strings supplied to these functions, and determine exactly the set of format specifiers and modifiers that are used. These can then be used to create a customized version of either function as required. The code size saving can be immense. Code for a minimal version of `sprintf()` implementing simple string copying can be as little as 20 or 30 bytes, whereas a version providing real number formats with specific numbers of digits could occupy 5000 bytes or more. No programmer input, other than writing the program itself, is required to benefit from this customization and optimization.

[1] In compiler parlance the meaning of the adjective "global" has been diluted by the use of the term *global optimization* to refer to optimization within one function - OCG takes a wider view than this, implementing *whole-program* optimization.

Unused Return Values

Many library functions return values that are not necessarily checked by the calling code. A compiler with OCG can establish whether the return value of a function is ever used. If the code generator determines that the return value for a particular function is never used, it can remove the code implementing the return value in that function, saving both code and cycles.

Re-entrancy Without a Conventional Stack

A *conventional stack* is implemented in the hardware of the target MCU. – However, not all MCUs have a hardware stack. In these implementations the compiler must implement a *compiled stack* built at compile time rather than runtime. Although this is not a great problem for most embedded applications, it makes writing re-entrant code difficult — a *compiled stack* cannot implement recursion or re-entrant function calls. However implementing a *compiled stack* transparently addresses re-entrant code 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 re-entrancy can be implemented without a conventional stack.

Customized Runtime Startup Code

Once upon a time, all embedded systems programmers were accustomed to writing their own *runtime startup code* (often in assembly language). Startup code is executed immediately after reset to perform housekeeping, such as clearing the uninitialized RAM area. The C language requires that uninitialized static and global variables be cleared to zero on startup. Many newer embedded compilers relieve the engineer of this task by providing canned startup code. However, canned startup code is often much larger than necessary for a given program — if the program has no uninitialized global variables, there is no need to include code to clear them. OCG makes this information available to the code generator, which then creates custom runtime startup code. In a minimal case, the startup code may be completely empty.

Smaller Code, Better Performance One obvious advantage of OCG is smaller, faster code. More importantly, OCG allows embedded C programs to be written ***without*** the use of architecture-specific extensions. By performing at compile time an analysis of the whole program, the code generator can make decisions about memory placement, pointer scoping etc. that would otherwise be made by the programmer and specified through special directives or language extensions. Since this analysis is performed every time the program is recompiled, it is always accurate and up-to-date.

Performance comparisons of OCG to conventional compilation focus on code size — typically the most important parameter for embedded systems, but also a valid proxy for execution speed. A reduction in code size always results in an increase in execution speed unless specific optimization techniques are used that sacrifice code speed for size (such as code-threading).

A single C-language source file was compiled for the Microchip's PIC18 and Cypress' PSoC™, using both traditional and OCG compilation technologies. OCG compilation resulted in 17.2% less code for Microchip's PIC18 architecture compared to Hi-Tech's PICC-18™ STD that shares much of the same compilation technology, except OCG.
OCG-compiled code for Cypress' PSoC was 41.4% smaller, than that compiled by ImageCraft.

Comparative code sizes (bytes) for ts057.c

Target Chip	Compiler	OCG	Code Size	∆%
Microchip PIC18	PICC-18 STD.	no	8038	
	PICC-18 PRO.	yes	6652	-17.2%
Cypress PSoC	ImageCraft	no	11957	
	PSoC PRO Beta version	yes	7001	-41.4%

Additional tests to evaluate OCG’s library optimization functions on the PIC18C242 target resulted 50% better code density than IAR’s EW18 compiler and 40% better code density than Microchip’s compiler.

Comparative PIC-18 code sizes (bytes) for ocg-test.c		
Compiler	**Code size with `printf()`**	**Code size without `printf()`**
IAR EW18	14855	8416
Microchip C18	12620	8873
HI-TECH PICC-18 STD	9775	8377
HI-TECH PICC-18 PRO	7488	6738

Tests conducted on other source files showed similar results..

Conclusion

The somewhat irregular architectures of embedded microcontrollers are often an awkward fit with the standard C language, frequently requiring substantial architecture-specific hand crafting to achieve efficient code. Omniscient Code Generation simplifies and streamlines the programmer's job by abstracting and hiding the underlying architecture, while simultaneously delivering reduced code size and increased execution speed.

<ends>

Clyde Stubbs, CEO and Founder of HI-TECH Software.
BA (Hons) majoring in Computer Science - University of Qld, Australia 1982.

About HI-TECH Software

Hi-Tech Software is a world class developer of development tools for embedded systems, offering compilers, RTOS and an Eclipse based IDE (HI-TIDE) for 8-, 16-, and 32-bit microcontroller and DSP chip architectures. Its products support Microchip PICmicro® MCUs and DSPs, ARM, 8051, TI MSP430, HOLTEK, ARClite, XA, Z80, and PSoC architectures, to name a few. Its customers include tens of thousands of embedded system developers including General Motors, Whirlpool, Qualcomm, and John Deere.

Founded by Clyde Stubbs, in 1984 in Brisbane, Australia, Hi-Tech Software has an office in Gilroy, California, and an extensive network of distributors around the Globe.