

Let Sleeping Chips Lie

Software techniques for low power

Clyde Stubbs, CEO, HI-TECH Software

The increasing emphasis on green technologies has focused attention on power consumption in all devices, even those that are connected to a line power supply. The proliferation of USB for barcode scanners, printers, pin pads, signature capture devices, external storage devices, antennas for WiFi, or wireless keyboards mean that appliances that used to be plugged into the wall must now conform to the parsimonious power budgets of the USB standard.

There are two main areas of power consumption in digital logic: leakage and switching currents. Leakage currents in CMOS logic are negligible, particularly in the more conservative process technologies used for low-power 8- and 16-bit microcontrollers.

The main culprit where power consumption is concerned is switching transients. Every time a digital output changes state, it has to charge and discharge various elements of stray capacitance associated with the output itself and anything connected to it – either external or internal to the chip. Each transition on a signal line costs a more-or-less fixed amount of energy. So reducing the rate of transitions (the clock rate) will reduce the power consumption. However, if the chip has to run longer to achieve the same result, this doesn't necessarily translate into total energy savings.

The real challenge is to reduce the number of instructions that the CPU has to execute to get its job done. This needs to be addressed in a number of ways - careful design of the application software structure is the first, followed by tight coding, and proper choice of development tools (e.g. compilers). The compiler itself can have a significant effect on the number of instructions required to execute the application and the resulting power consumption.

Sleep modes

Most microcontrollers have sleep or halt states that can be entered to reduce the power consumption of the chip. In these states the CPU will be stopped, and various other resources on the chip will be suspended or shut down. The type of sleep states available on a given chip and the requirements of the application will determine which sleep states are usable.

The objective in designing software for low power is to make use of the deepest sleep state the application allows, and to spend as much time as possible in that state. Compilers can have a subtle but significant effect on how much time the microcontroller spends in sleep mode. All too often the method of compilation wastes CPU cycles on interrupt routines, in devices with banked memory, on locating addresses in memory.

Interrupt Routines.

In any application it's important to keep interrupt routines small and fast. This is especially true when speed and/or power consumption are critical. Keeping interrupt routines small minimizes interrupt overhead.

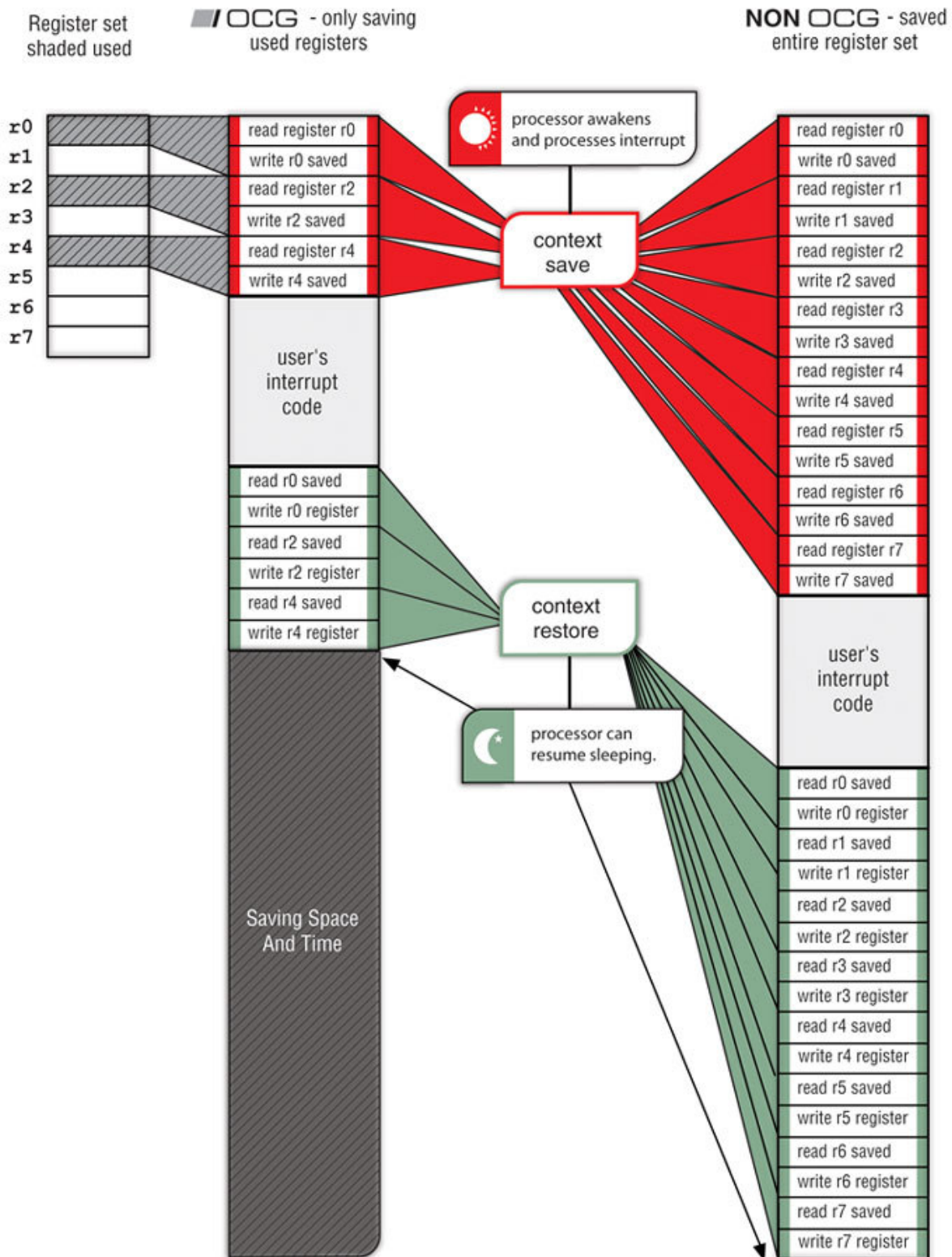
The compiler's contribution to minimizing interrupt overhead is in the number of registers it saves in response to an interrupt – the context. In order to prevent memory overwrites, conventional compilers save every register that *might* be used by an interrupt. Most compilers have no way of knowing which registers will or will not be used by a given interrupt, so they save them all. The problem is that the number of cycles used is a direct function of the number of registers that are saved and restored. The more cycles spent saving and restoring the context, the most power is consumed.

For example, one compiler for Microchip's PIC16 *always* saves 8 bytes of data for every interrupt, regardless of what data will be required. Saving 8 bytes of data requires a total of 42 instruction cycles (23 for the context save and 19 for the restore). This may not seem like much, but in an interrupt intensive application, the CPU could spend thousands of extra cycles "awake", unnecessarily consuming power.

Newer compilers are now available with "omniscient code generation" (OCG) technology that has the intelligence to save only those registers that are required for each particular interrupt. Omniscient code generation works by collecting comprehensive data on register, stack, pointer, object and variable declarations from *all* program modules *before* compiling the code. An OCG compiler combines all the program modules into one large program which it loads into a call graph structure. Based on the call graph, the OCG code generator creates a pointer reference graph that shows 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). It then identifies all objects that can possibly be referenced by each pointer. This information is used to determine exactly how much memory space each pointer will be required to access.

The OCG compiler knows exactly which functions call, and are called by, other functions, which variables and registers are required, and which pointers are pointing to which memory banks. This gives the compiler the necessary intelligence to know exactly which registers will be used for every interrupt in the program, so it can generate code accordingly, minimizing both the code size and the cycles required to save and restore the context.



The compiler determines the size of the context for each interrupt dynamically, depending on the state of the code at the time of compilation. An interrupt may require as few as 17 cycles - 10 to save the context and 7 to restore it. The worst case for the OCG compiler is only 25 cycles. Compared to a conventional compiler, an OCG compiler can reduce the number of interrupt-related instruction cycles by 40% to 60%.

Depending on the application, the cycle savings can be substantial. A mouse, for example generates about 300 interrupts a second, taking up less than 1% of the processors total cycles. On the other hand, an interrupt driven serial communication port with a baud rate of 480,600 bps generates 24,000 interrupts per second. Using a conventional compiler with 42 instruction cycles per interrupt (168 clock cycles per interrupt) saving and restoring the context will use up over 4,032,000 CPU cycles per second or 20% of the available cycles on a 20 MHz PIC16. An OCG compiler, averaging 21 instructions cycles per interrupt (84 clock cycles per interrupt), can reduce that number to only 2,016,000 cycles - saving ½ the clock cycles otherwise spent on saving and restoring contexts, and allowing the CPU to be put into sleep mode for 10% of its cycles. Assuming 10 mA active, and 1 µA sleep mode, power consumption, an OCG compiler could reduce total MCU current consumption by nearly 1 mA – about 10%.

Banked Memory Architectures Can Waste Cycles. Many 8- and 16-bit microcontrollers have banked memories that cannot be addressed simultaneously. Switching between the memory banks requires at least one bank selection instructions. Thus, if data in one bank must be copied to another bank, bank selection instructions are always necessary. Obviously, placing all the variables accessed by a function in the same memory bank will reduce the number of bank selection instructions and the total required cycles for the application. However, conventional compilers have no way of knowing which functions call which variables and are unable to optimize their memory assignment. Nor do these compilers have any way of knowing whether or not a particular memory bank will be selected at any point in the code. As a result, these compilers automatically generate bank selection instructions for every memory access, whether or not that bank is already selected.

Some compiler vendors have addressed this issue by providing bank qualifiers, extensions to the C-code that identify the address of the variable. Programmers may manually assign variables to memory banks using this non-standard, non portable code. Using bank qualifiers allows the compiler to see the exact bank an object resides in and reduces the number of bank selection instructions. However, this approach does not guarantee that dependent variables will be placed in the same bank. Every time a variable in one memory bank needs to be written to another memory bank, bank selection instructions will still be required. In addition, trying to track all the memory addresses across multiple code modules and ensuring that all pointers have the correct addresses is a time consuming, tedious process that can itself introduce programming errors.

In contrast, since an OCG compiler knows every register, stack, pointer, object and variable declaration from *all* program modules, it can optimize every variable and register allocation, as well as the size and scope of every pointer and every object stored on the compiled stack. It can allocate memory optimally to minimize or eliminate bank selection instructions, without any intervention from the programmer.

The OCG compiler assigns global and static variables fixed addresses in available unbanked memory, which does not require bank selection instructions. If unbanked memory is not available they are assigned to a specific memory bank based on their size, special qualifiers, and the number of times they are referenced. Variables that are dependent are placed in the same memory bank whenever possible.

Function parameters and auto variables are assigned space on the compiled stack. The linker automatically allows variables to share the same memory if their functions are not active at the same time, thereby reducing RAM usage by as much as 80%.

By placing frequently accessed variables in unbanked memory and by placing any dependent variables in the same memory bank, an OCG compiler can radically reduce the number of cycles and power wasted on bank selection instructions in these MCU architectures. Since the OCG compiler knows which bank is selected at any point in the code, it can also eliminate any unnecessary bank selections instructions when the bank is already selected. Reducing the number of instructions reduces the number of CPU cycles, allowing the CPU to spend more time in sleep mode.

The total number of cycles that can be saved is extremely application dependent, and therefore difficult to quantify. It is possible that the intelligent assignment of objects in memory and knowledge about which memory bank is selected could reduce the total cycles required by 30% to 50%, with a linear effect on MCU power consumption.

Conclusion. Choosing a low power device and exploiting the sleep mode capabilities of the MCU are important means of keeping power consumption to a minimum. However, the way in which the compiler manages interrupts and memory usage can also have a significant impact on power consumption. Newer compilers with omniscient code generation technology can make a substantial contribution to saving cycles and power.

About the Author

Clyde Stubbs is founder and CEO of HI-TECH Software, Queensland, Australia. His university research in compiler technology led to the founding of HI-TECH Software in 1984. In 2006, he developed omniscient code generation (OCG) technology which generates object code based on a comprehensive analysis of all program modules, before compilation, allowing truly global optimization of stacks, registers, pointers and memories for much better code density that can be achieved by conventional compilation techniques.

Publications:

Whole-program Compiler Technology For Increased Code Density, Embedded Control Europe, June,2007

Using Whole-program Compilation to Improve MCU Code Density and Performance, embedded.com, March 15, 2007