Optimizing Code Performance and Size for Stellaris® Microcontrollers

Application Note

AN01265-02

Copyright © 2007–2009 Texas Instruments

Copyright

Copyright © 2007–2009 Texas Instruments, Inc. All rights reserved. Stellaris and StellarisWare are registered trademarks of Texas Instruments. ARM and Thumb are registered trademarks, and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Table of Contents

Introduction 4
Detailed Information on Performance and Size Factors 7
  Compiler Switches 7
  Use of Interrupts 8
    Critical Sections 10
    Spin Locks 10
  Size of Variables 11
  Use of Global Variables 12
    Aliasing and Global Reload 12
    Use of Locals to Avoid Excess Loads and Stores 13
  Const Types 13
  Taking Address of Local Variables 13
  Accessing Peripherals (at Fixed Address) 14
  Use of C Standard Library 16
  Asm() Inserts 16
    Asm with Keil/ARM 16
    Asm with GCC 17
  Floating Point 18
  Volatiles 18
  Back-to-Back Writes to Peripherals 18
  Immediate Use of Loads from Peripherals 19
  Recursion 19
  Many Small Functions 19
  Too Many Function Parameters 20
Conclusion 20
References 20

Introduction

This application note provides a summary of main factors that affect code performance and size for Stellaris® microcontrollers as well as a detailed discussion of how to improve code performance and size including example code where useful. Table 1 provides a summary of the main factors.

Table 1. Main Factors Affecting Performance and Size

Factor	Impact	Hidden Cost	Stellaris and ARM® Cortex™-M3
Compiler switches (see page 7)	• Code performance • Code size	Code and/or data are larger or slower than expected due to not getting the most from the compiler.	Each compiler has its own switches (see page 7). But, minimally, use of –O2 or –O3 along with optimizing for time or space is crucial.
Interrupt usage (see page 8)	• System load • Responsiveness	This includes overhead, worst-case nesting, and latency (especially if there are not enough priorities). Multi-cycle instructions will hold off interrupt. Higher priority interrupt coming in during servicing is held off. "Stubs" and hidden code to handle interrupts in software.	Stellaris parts have very fast interrupt response, will interrupt a multi-cycle instruction (interrupt-continue for LDM and STM, unlike the blocking ARM7/ ARM9 for example), and support 8 levels of fully nested priority.
Variable size (see page 11)	• Algorithm time • Load/store • Computation • Extra code size (often dramatic)	Using variables larger than the processor is comfortable with means extra loads/stores, extra computation (software routines versus hardware), or far slower instructions. This often ends up as calls, which does not add to size, but does significantly affect performance. Using variables smaller than optimal may mean extra instructions to sign or unsign extend (on load and after computations). These may also prevent the use of optimal load and store instructions.	Long long is generally optimized due to special instructions (for example, ADDC and UMULL). Smaller globals and statics are okay, but locals are best as ints and unsigned ints (or longs). If globals/static are used a lot, copy to int locals for duration, and copy back. It is not unusual to have a 40% increase in function size due to use of short locals.
Global variable usage (see page 12)	• Load/store cost • Extra instructions	Compilers have to assume globals have been modified across function calls and so must reload around calls. This is even worse with global pointers (it has to assume both the pointer may have changed and what is pointed to has also changed).	Cortex-M3 has 13 32-bit general-purpose registers, so it is best to use locals whenever possible. Compilers will not put locals on the stack unless necessary.
Const types (see page 13)	• Loads instead of moves • Duplicate memory in flash and RAM	Global const variables in C may be allocated to SRAM and have their init value in flash. In all cases, the value would normally be loaded from memory (unless the compiler can see its initial value). Static const scalar variables are like #define macro constants and will not be stored in memory if not needed (address not taken, value small enough to be an immediate).	Different compilers and optimization levels will affect how global const is treated. Static const is more reliable for all compilers. Enum constants are also a good choice (and can be used with normal ints).

Table 1. Main Factors Affecting Performance and Size (Continued)

Factor	Impact	Hidden Cost	Stellaris and ARM® Cortex™-M3
Taking address of local variables (see page 13)	• Load/Store versus registers	Local variables are only allocated to the stack if they have to be. Keeping them in registers (and sharing a register between different ones not used at the same time) gives big gains in performance (that is, not having to do loads and stores). Taking the address of a local will force it to the stack regardless of optimization level.	All Cortex-M3 compilers support all-in-register locals when optimizations are turned on (size or speed).
Accessing peripherals (at fixed address) (see page 14)	• Extra loads • Loads versus moves	Access to peripherals and system registers can be handled in a number of ways in C. But, most ways generate more code and are much slower. The optimal solution varies from one processor to another, and also one compiler to another. This makes it much harder when porting code. The four most common techniques for accessing peripherals are: • Casted constants, for example, *((short*)0x400000) • Global or static pointers (to scalar or structure) • Local pointer loaded from a constant (structure or scalar) • Global structures (or arrays) positioned by the linker (via section or linker script)	Local pointers to volatile structures (and scalars) and positioned structures (and arrays) are usually the most efficient on many compilers because offsets are generated from a register-based base address pointer. The pointers loaded from a constant are extra work (per routine) but may be the most efficient because the compiler can generate the constant inline in many cases (which avoids a load from a linker fixed up address).
Use of standard C library (see page 16)	• Performance • Size	The C runtime library was developed for hosted computers. It carries a lot of baggage which can rob performance in many places. Each compiler vendor treats the libraries differently, some doing a better job than others. Most library functions should be avoided, although memcpy() is normally optimized.	DriverLib with both peripheral support and debug support (such as a printf) is provided. Some vendors also support a cut-down library (for example, Keil Microlib or Rowley mcu-lib). Gcc does not have a cutdown lib.
Asm() inserts (see page 16)	• Extra instructions	Asm() inserts are a way that compilers allow use of processor special instructions. Sometimes, these are used to get to system instructions (for example, interrupt masking), but they are also used for performance enhancing instructions (for example, fixed point support, special bit handling, and so on). The problem is that many compilers treat these as a function call and so unload all scratch registers (saving back to stack, reloading after) thereby adding cost. A few compilers (for example, gcc) allow you to mark the asm() insert to say what registers it modifies (other than output result), so it is inlined quite well.	For Keil/ARM compiler, use built-in “intrinsics”, such as __clz (for CLZ instruction) when possible. Keil/ARM compiler no longer supports asm() inserts, must use asm functions. But, function with exactly 4 bytes of size (other than return) will be inlined by linker (though still treated as call) with --inline. New DriverLib include will map instructions and system operations as ideally as possible for the different compilers.

Table 1. Main Factors Affecting Performance and Size (Continued)

Factor	Impact	Hidden Cost	Stellaris and ARM® Cortex™-M3
Floating point (see page 18)	• Performance • Code size	Using floating point is expensive on microcontrollers due to lack of hardware support. The software algorithms provided by compilers vary significantly in quality. Some vendors have non-standard but fast floating point libs. Non-standard means less checking (NaN, Infinity, Denormal), so is often quite acceptable if floating is absolutely required.	If you must use floating point, check with your compiler vendor and third parties for highly optimized floating point (such as single precision only, minimal checking). This can have more than a 10x improvement over the default double-precision software libraries.
Volatiles (see page 18)	• Load/store • Broken behavior	It is important to use the volatile keyword for peripherals. The behavior when not used may vary from compiler to compiler and even with the same compiler depending on command line switches, debug versus release builds, and even based on small changes in the code. An example case is: `while (Uart0_status & UART_DATA_WAITING)` This can cause a problem (many compilers optimize that based on it being a global that will not change its value if no function calls within the loop).	The volatile keyword should be used on pointers and globals/ statics positioned over peripheral registers. Note that volatile use prevents most optimizations, including code order and other factors. So, volatile use is best localized in a routine.

Back-to-back writes to peripherals (see page 18)	• Store cost	In spite of the previous statement about localizing volatiles, it is important to understand the impact of back-to-back stores. Many processors can hide some of the cost of a wait-stated store, but this is generally defeated once there is more than one in a row.
		By moving register-based instructions between, the store has a chance to be processed in the background.
			Code of the form:
			`MyPeriph_reg0 = x;`
			`MyPeriph_reg1 = y;`
			`MyPeriph_reg2 = z;`
			Will often be less efficient than:
			`MyPeriph_reg0 = x;`
			`y = some computation`
			`MyPeriph_reg1 = y;`
			`z = some computation`
			`MyPeriph_reg2 = z;`

Immediate use of loads from peripherals (see page 19)	• Load cost	Unlike stores, loads are often best when they are back-to-back. For normal load followed by an ALU instruction, the processor has to wait for the load to complete before it can do anything. For wait-stated loads, this adds extra time. If a set of loads are back-to-back, the processor can often generate the next address while waiting for the preceding wait-stated load, so hiding some of the cost.	Code of the form:
			`// x is local`
			`x = MyPeriph_reg0;`
			`y = MyPeriph_reg1;`
			`z = MyPeriph_reg2;`
			`// now operate on x, y, z`
			Will usually be more efficient and may compress to LDM.

Recursion (see page 19)	• Stack memory • Performance	Recursion is often the simplest way to deal with certain algorithmic problems. But, recursion uses stack space and reduces register optimizations (because the recursive calls force scratch registers to be unloaded, and locals must be re-generated).	One way to improve recursion is to make the recursive function static. The compiler will often be able to optimize the entry and exit when it knows where all the calls are coming from.

Table 1. Main Factors Affecting Performance and Size (Continued)

Factor	Impact	Hidden Cost	Stellaris and ARM® Cortex™-M3
Many small functions (see page 19)	• Performance • Size	For many 8-bit and 16-bit processors, it is more efficient to break applications up into many small functions (sometimes called “factoring”). This is faster due to limits on number of registers and lack of a real stack. This is generally unnecessary and counter-productive on a 32-bit processor.	Larger functions do well due to the number of registers. Extra calls add performance costs due to call overhead, scratch register rules, and other factors impacting optimizations. These also add size per function.
Too many function parameters (see page 20)	• Performance • Memory use	Each processor has different rules about argument passing. For those that can only pass on the stack, the parameter count is not typically a real factor (unless there are two classes of stack). For cases where register passing is used, it is slower to have more than four parameters to a function.	It is true that functions with four or less scalar/pointer arguments will be faster. This is because the first four arguments are passed in registers, and the remaining arguments are passed on the stack.

Detailed Information on Performance and Size Factors

This section describes the factors summarized in Table 1 in more detail and provides example code to improve code performance and size.

Compiler Switches

The most important starting point is the optimizer. When debugging early on, it may be reasonable to use minimal or no optimizations. This ensures that the code behaves as written (for example, steps follow source lines, variables have the expected value at the expected time, and so on).

Once the application appears to be functionally correct, it is important to enable optimizations. The reason for optimizing is faster code and smaller code. Generally, faster code is smaller (or only larger where it adds advantage) and smaller code is generally faster than unoptimized code.

The two main optimization switches are the level of optimization control (-O0, -O1, -O2, -O3) and the optimize for time versus space switch (-Os for gcc to optimize for space (versus time), -Otime for Keil/ARM to optimize for time (versus space)). Note that the different compilers and IDEs default differently, so you need to check that you are using the switch needed for what matters.

The optimization level makes a big difference between none, -O0, and –O1 and –O2. –O3 normally provides more performance optimizations (code rearrangement) than space, and is very hard to use with a debugger (line tracking).

For example, the Stellaris LM3S811 evaluation board game program on a Keil/ARM compiler has the statistics shown in Table 2.

Table 2. Stellaris LM3S811 Evaluation Board Game Statistics

Optimization	Code Size	Delta (Code) from -O0	RO	RW	ZI (C Commons)
-O0 space	9008	–	1812	24	736

Table 2. Stellaris LM3S811 Evaluation Board Game Statistics (Continued)

Optimization	Code Size	Delta (Code) from -O0	RO	RW	ZI (C Commons)
-O1 space	6764	-2244 (-33%)	1812	24	736
-O2 space	6668	-2340 (-35%)	1812	24	736
-O3 space	6644	-2364 (-35.5%)	1812	24	736
-O3 time	9800	+792 (+8%)	1812	24	736

As can be seen, the biggest drop in size is from –O0 to –O1. However, some applications respond far more favorably to the change from –O1 to –O2 and then to –O3. It is important to understand that different optimizations are applied at different levels.

One important consideration also is multi-file optimization. The Keil/ARM uVision make system compiles all the files together on one line. The compiler takes advantage of having multiple source files at the same time, and can perform far more aggressive optimizations when given all/many files at once.

The gcc compiler with –O3 and –Os gives 6664 for the code size, so comparable to –O2. Again, the differences between the compilers depends on application style, such as number of loops.

Use of Interrupts

The biggest issue with use of interrupts is the hidden costs of many processors. The main measure has to be the time from interrupt assertion in hardware to the first line of real user code (not just the code-saving registers; this is not about “code-saving” but rather code (instructions) used to save registers on the stack). That is, how long before any work is done to address the cause of the interrupt.

Further, you have to consider best, worst, and average response. Ideally, these are all close together in time. But, for many processors they are not. This is due to five main factors (shown in Figure 1):

1. Current instruction is blocking the interrupt

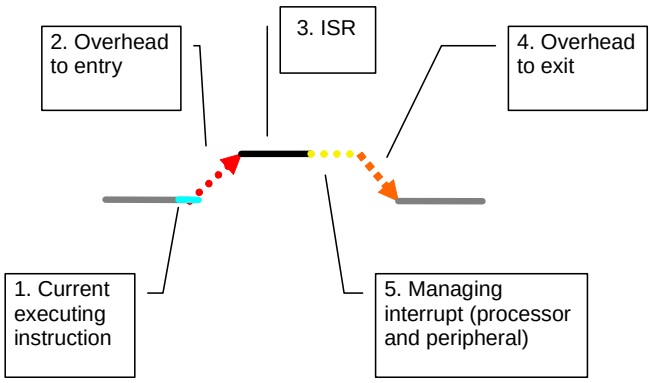
2. Code or processing of interrupt

3. Not enough priorities

4. Code or processing of return

5. Code required by interrupt controller for correct operation

Figure 1. Interrupt Factors that Affect Performance

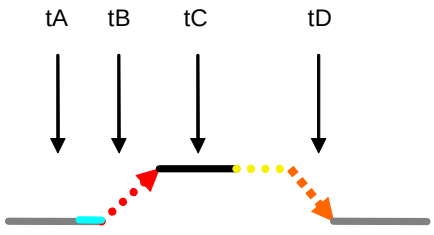


Current instruction blocking the interrupt is based on slowest instruction, unless it allows abandon. For example, on an ARM7 or ARM9, an LDM/STM takes on the order of two cycles per word and is not interruptible. If the memory being accessed has wait states, you have to multiply that cost by the wait states. So, an 8-word transfer across a peripheral bus with four wait states (two for address and two for data) would go from 16 cycles to 48 cycles where the interrupt cannot be activated. Other processors have similar issues. On Stellaris parts, all multi-cycle instructions are abandoned on interrupt. LDM and STM (multi-cycle load and store) save which word they were on, and continue on return, so are volatile safe.

Processing overhead is often misunderstood or misleading. For example, on an ARM7 or ARM9, if you have exactly one interrupt (IRQ), your code only has to push five registers more than a normal function (that is what the _irq keyword does). This would be only an additional 6 to 10 cycles on entry (and 6 to 10 more cycles on exit). But, if you have more than one interrupt, you must perform a number of instructions to change mode and preserve state. This must be done in assembly code for each ISR and adds approximately 48 to 58 cycles of overhead (depending on how many assumptions you are making). On a Stellaris part, the overhead is 12 cycles on preemption and 12 cycles on return from preemption (3 to 6 cycles if transferring to another interrupt that was pending). This 12 cycles involves the processor pushing the scratch registers and link registers, so the ISR is a normal C function. This means that after 12 cycles, the first instruction of user code will be executing.

Priority of interrupts is important at two times: once if more than one interrupt is asserted at the same time (rare) and the interrupt controller has to choose which to run, and once when an interrupt has already been chosen (is in overhead or is in ISR) and a new higher priority interrupt is asserted (see Figure 2).

Figure 2. Priority of Interrupts



The times shown are as follows:

- tA (when executing non-ISR instructions) can use priorities to select if more than one is pended at the same time
- tB is when the interrupt overhead comes into play
- tC is when in the ISR
- tD is when exiting the ISR

For most processors, the higher priority interrupt cannot be serviced during tB or tD. For processors that cannot support nested interrupts, then the interrupt cannot be serviced during tC either. Unless extra work is done in each ISR, an ARM7 and ARM9 cannot service a second interrupt during any of tB, tC, or tD.

Stellaris parts have eight priority levels and can service interrupts in any of these points. If a higher priority interrupt comes in at time tB, it will switch to that one with no time penalty (uses same push of registers). If a higher priority one comes in at tC, it will preempt it. If a higher priority one comes in at tD, it will chain to it and avoid the register pops and pushes. Likewise, when a lower or equal priority interrupt is pended, it will chain on exit from the first, therefore saving the 12 cycles of exit and 12 cycles of entry work.

Critical Sections

One other major factor for interrupt-based applications is critical sections. To avoid data race conditions and for other uses, interrupts are masked off by many applications and all RTOSs. This means both that interrupts are held off for this period of time, and that interrupts tend to stack up against the re-enable point. On Stellaris parts, priority masking may be used instead. This means that instead of disable interrupts, the application only masks off interrupts of a certain priority level and below, thereby allowing the higher priority ones to continue (they must not touch the protected data). To make this even faster, a special processor register is used, called **BASEPRI_MAX**. This allows the application to read the mask level (**BASEPRI**) into a register, set **BASEPRI_MAX** to the desired level, perform the critical section code, then restore the **BASEPRI** with the saved value. This contrasts with reading it, comparing to see if you will make more secure, and only setting if so. That is, **BASEPRI_MAX** will not let you lower the priority, so it is safe to just write with the desired value.

An example of using **BASEPRI** is motor control. The motor control software runs completely as high priority interrupts. An RTOS or other application can run at the base level and lower priority interrupts, and not even critical sections will have any impact on the motor control ISRs.

Spin Locks

One other mechanism used for access to data without critical sections is the spin lock (often a SWAP instruction). Instead of this expensive instruction, Stellaris parts have an “exclusive” instruction. An exclusive instruction allows access to a location in memory (byte, half, or word) such that the hardware prevents a store from being performed if some other code has already accessed it. This works by loading the value first, performing some operation on the value (such as setting a request bit), and then writing back. If any other code has accessed the location between the load and store, the store will indicate it was refused, and the code can try again. This allows for shared access to data without critical sections.

Size of Variables

As mentioned in Table 1, use of variables larger than the processor is naturally comfortable with usually means many extra instructions and/or calls to functions. On 8-bit and 16-bit processors, this often means that anything larger than 16-bits (a short int) will have this effect. On Stellaris, only long long ints can cause this. However, the instruction set fully supports most long long operations inline using only one additional instruction. For example, long long add, uses ADD followed by ADDC. Likewise, subtract uses SUB and SUBB (borrow). Multiply has special operations to support 64-bit multiply. Only divide will require a callout, unless the result is into a 32-bit result.

When the local variables are smaller than the register size, then extra code is usually needed. On a Stellaris part, this means that local variables of size byte and halfword (char and short int respectively) require extra code. Since code ported from an 8-bit or 16-bit microcontroller may have had locals converted to smaller sizes (to avoid the too large problem), this means that such code will run slower and take more code space than is needed. Changing the locals to int/unsigned (or long and unsigned long) often saves 40% or more in code space. SeeTable 3 for a comparison.

Table 3. Comparison of Variable Sizes

Locals of size int	Locals of size short int (half word)

`typedef int BASE;`
 	
`BASE foo(BASE last, BASE x, BASE y)`
`{`
`0:  2300  movs  r3, #0`
`2:  e002  b.n  a <foo+0xa>`
`  BASE  i;`
 	
`  for (i = 0; i < last; i++)`
`    x += (y * x);`
`4:  fb02 1101  mla  r1, r2, r1, r1`
`8:  3301  adds  r3, #1`
`a:  4283  cmp  r3, r0`
`c:  dbfa  blt.n  4 <foo+0x4>`
`e:  ebc2 0001  rsb  r0, r2, r1`
`  return(x-y);`
`}`
`12:  4770  bx  lr`
	`typedef short BASE;`
	
	`BASE foo(BASE last, BASE x, BASE y)`
	`{`
	`0:  f04f 0c00  mov.w  ip, #0  0x0`
	`4:  e004  b.n  10 <foo+0x10>`
	`  BASE  i;`
	
	`  for (i = 0; i < last; i++)`
	`    x += (y * x);`
	`6:  fb02 1301  mla  r3, r2, r1, r1`
	`a:  f10c 0c01  add.w  ip, ip, #1 ; 0x1`
	`e:  b219  sxth  r1, r3`
	`10:  fa0f f38c  sxth.w  r3, ip`
	`14:  4283  cmp  r3, r0`
	`16:  dbf6  blt.n  6 <foo+0x6>`
	`18:  ebc2 0001  rsb  r0, r2, r1`
	`1c:  b200  sxth  r0, r0`
	`  return(x-y);`
	`}`
	`1e:  4770  bx  lr`

The simple algorithm of a loop has added 12 extra bytes to a function of 20 bytes. Worse, it has added two extra cycles to each iteration of the loop (which is a five-cycle loop to start with). In other words, changing existing code to move from use of short and unsigned short locals to int and unsigned saves size and improves performance by large and unexpected amounts.

Note that the same example with ARM7 and ARM9 using Thumb code is 28 bytes with integers (but much slower) and 40 bytes with shorts. The extra 12 bytes for the short ints for Thumb is due to using shift-left and then shift-right to sign or unsign extend, so four extra cycles per loop.

Use of Global Variables

Globals and statics use SRAM, and if initialized, use flash as well (the initial value is stored in flash to be copied in at startup). Obviously, if the data must be persistent, then globals are the only option; but, static variables (static to a module) may be more optimal in terms of code size (less reloading).

Further, be aware that the compiler and linker may or may not group globals of the same size. For example, having declarations: “char c1; int x; char c2;” may result in three bytes of waste after c1 (since x has to be aligned to a word). Also be aware that statics tend to be grouped, and some compilers (for example, Keil/ARM) tend to recognize the offset relationship, so avoid extra “literal loads” to get the address of the variables (when used in the same function) and avoid wasted flash to hold the extra literals. That is, in the example of c1/c2/x, the compiler would group c1 and c2 together (at base+0 and base+1) and then x (at base+4); if all three were used in one function, the address of the lowest would be loaded into a register, and then each would be accessed as offsets. For example, using:

```
LDR  R0,base    ; load base address for c1, c2, and x from “literal”
LDR  R3,[R0,#0] ; load c1
LDR  R4,[R0,#1] ; load c2
LDR  R5,[R0,#4] ; load x
```

Aliasing and Global Reload

Compilers have to assume globals have been modified across function calls, so must reload around calls. This means extra instructions. An example is shown below using the GCC compiler and –O3 optimizations.

C Code	Asm Code	Explanation

extern int x;
extern void bar();
int func(int i) {
  x++;
  if (i)
  x *= 2;
  else
  x *= 3;
  bar();
  return(x);
}
	x++;
	  0: 4908 ldr r1, [pc, #32]
	  2: b500 push {lr}
	  4: 680b ldr r3, [r1, #0]
	  6: b081 sub sp, #4
	  8: 3301 adds r3, #1
	  a: 600b str r3, [r1, #0]
	  c: 005a lsls r2, r3, #1
	if (i) x *= 2; else x *= 3;
	  e: b108 cbz r0, 14
	  10: 600a str r2, [r1, #0]
	  12: e001 b.n 18
	  14: 4413 add r3, r2
	  16: 600b str r3, [r1, #0]
	bar();
	  18: f7ff fffe bl 0 <bar>
	  1c: 4b01 ldr r3, [pc, #4]
	  1e: 6818 ldr r0, [r3, #0]
	  return(x);
	}
	  20: b001 add sp, #4
	  22: bd00 pop {pc}
		R1 = &x
		
		R3 = x
		
		R3 = x++
		x = R3
		R2 = x*2
		
		if (!i) branch
		x = R2 (should share)
		branch around
		R3 = x*3 (x*2 in R2)
		Store (should share)
		
		Call
		Reload &x
		Reload x

As can be seen in the example, the value of “x” is reloaded. In this case, the &x is also reloaded, although that is simply a trade-off of pushing an extra register (for example, R4) to hold its address. Since a push and a pop of R4 is likely the same cost as the LDR to load the address, no advantage.

The important point is that reloading x is needed around a function, unless it knows for sure that the function does not modify x (directly or indirectly through a pointer—called aliasing).

Note: Note: the code shown is less efficient than it should be. The Keil/ARM compiler avoids the two stores (shares them).

Any of the compilers would have generated more efficient and space compacted code if ?: had been used, and in the right way. If the if line had been “x *= i ? 2 : 3;”, all the compilers would use ITE instead of CBZ and B (branch) and would use one STR. This is an area where the coding style can significantly affect the generated code.

The first store is a choice that compilers make due to aliasing. Some compilers have an optimization extension to allow anti-aliasing support. This should only be used with care, as a pointer pointing to x would not see the correct value.

Use of Locals to Avoid Excess Loads and Stores

If a lot of work is being done on a global variable, it will be smaller and faster if a local variable holds its value and stores it back at the end. So, in the example above, if a line was added “int lx = x;” at the top and all the work was on lx, then at the end, “x = lx;” was added, the code would only operate on registers and only perform one load and one store.

Note that local variables initialized with a small constant are often cheaper in space and performance because the compiler can generate a MOV instruction versus a LDR instruction.

Const Types

Most people assume that enum constants (for example, “enum colors {red=1,blue=2,green=3};”), #defines (for example, “#define RED=1”), static consts (for example, “static const int red=1;”), and global consts (for example, “const int red=1;”) all behave the same way.

In fact, enum and define constants are pure constants. The compiler uses a MOV when possible, else load from a Flash “literal” (note that there may be more than one of these, so that may be wasteful of Flash).

Generally, static const is also treated the same way. That is, since the source file is the only scope, the compiler may choose to never allocate space for it, and just use MOV if small enough of a value. Equally, a static const array is always stored in flash.

A global const is more complex. The first problem is that C (unlike C++) allows a global const to be declared const in one source file and not const in another. As a result, it may end up in SRAM (with an init copy from flash if not 0). If the global const is initialized in the source file (defined), then the compiler may treat as a true constant (with the init value), but will likely still have to allocate to memory (since it does not know if used elsewhere).

Taking Address of Local Variables

Local variables are only allocated to the stack if they have to be. Most compilers keep them in registers (or nowhere when not needed), unless you take their address. Keeping them in registers (and sharing a register between different ones not used at the same time) gives big gains in

performance (not having to do loads and stores) and saves memory. Taking the address of a local forces it to the stack. An example of two coding approaches shows this:

```
int   x, y, *pxy;
...
      pxy = test ? &x : &y;
...
variable = *pxy – delta;
```

The above is not uncommon as a technique. But, if pxy was not used and the last line was:

```
variable = (test ? x : y) - delta;
```

the result would usually be much faster code. This is because x and y are likely cached in registers and so this is a conditional MOV. In the previous case, stack is allocated for x and y, even if not needed, and the reference to *pxy requires a load (or two if pxy is not in a register).

Accessing Peripherals (at Fixed Address)

The best way to access peripheral registers in C is unfortunately dependent on the processor and compiler. The traditional approach has been casted constants of the form:

```
#define UART_REG1  *((volatile unsigned*)0x40000100))
```

The advantage, in theory, is that the compiler knows the constant and so can generate it with a MOV instruction. Unfortunately, this rarely works with peripheral addresses. So, the constant 0x4000100 would be stored in flash and loaded (for example, “LDR R0,[PC,#30]”). With many such registers, many such literals get stored and loaded from. So, reading three registers in a row may very well cause six loads (three loads of the constant and three loads from the registers).

Another traditional approach is a global or static pointer. This means you have accesses of the form:

```
status = ptr_peripheral[UART_REG_STATUS];
```

In the above example, ptr_peripheral is a pointer to the peripheral space (or just one peripheral). If ptr_peripheral is a static const pointer to a volatile location (so pointer is const, what it points to is volatile), it should behave the same as the casted constant pointer example above (UART_REG1). If it is not a const, then this will cause three loads to read one register: get the address of the global, read the global, and read the register.

Another approach is a global (volatile) structure or array which is positioned at link time. The main question is whether the compiler of choice allows this. All compilers can support this concept, but not all evaluation versions will. The most common way to do this is with __attribute__, such as:

```
volatile UART_DEF uart0 __attribute__ ((section(“uart0_section”)));
```

It is then necessary to use the linker (for example, scripting) or an assembly file to position section uart0_section.

Note that the positioned structure still requires literals to be used as in the UART_REG1 example.

One other approach is a local pointing to a structure:

```
void UartReadyISR(void) {
   // normally use a define or static const
   volatile UART_DEF *uart = 0x40000100;
   if (uart->status_reg)
```

This approach is instructing the compiler to cache the address base and so will generally generate small and fast code for most compilers.

Table 4 compares the code generation of GCC and Keil/ARM using the same function (see below), but with different methods to access the peripheral registers. Both were compiled with -O3 and space focused optimization (-Os for GCC).

The results are shown in terms of code size; additionally, read-only data size (Flash) and data (RAM) is shown if not 0 additional.

Table 4. Code Generation Comparison

Method	GCC (Q107)	Keil/ARM
#define constants (casted)	0x30 bytes, 3 literal loads	0x2C bytes, uses MOV (faster)
Static const pointer to scalar (e.g. unsigned)	Same	Same
Static const pointer to structure	0x28	Same
Global const pointer to scalar	0x30 bytes + 4 for rodata	Same. but only if whole program analysis
Global const pointer to structure	0x28 bytes + 4 for rodata	Same
Global pointer to scalar or structure	0x2C bytes + 4 for data + 4 for rodata	0x24 bytes + 4 for data + 4 for rodata
Structure mapped over peripherals	0x28	0x24, but not possible with evaluation version
Local pointer to structure (local per function)	Same	Same, but allowed on evaluation version

As stated earlier, local pointers to structures are the smallest for both compilers. But, local structure pointers are more work to retrofit.

Structure overlaid on the peripheral is also the smallest. But, it has some issues:

- It is not possible on the evaluation version of the Keil tools.
- It requires adding a linker command or memory file to position the structure.
- The Keil non-evaluation version would allow use of __at to position, but that is not portable to GCC.

If compiler portability is not an issue, then the Keil use of a MOV instruction with #define and const pointers is faster and on average will not be much larger (however, the small example is larger by 8 bytes).

Function Example	Example of Methods

void MyRegISR(void) {
  SETUP_LOCAL
 	
  if (MY_REG1 != 0)
  {
  MY_REG2 = 0x20;
  while (MY_REG1--)
  MY_REG3 = 0x11;
  }
}
	#ifdef USE_RAW_CONST
	  #define MY_REG1 (*((volatile unsigned*)0x40000100))
	  #define MY_REG2 (*((volatile unsigned*)0x40000104))
	  #define MY_REG3 (*((volatile unsigned*)0x40000120))
	  #define SETUP_LOCAL // do nothing
	#endif
	
	#ifdef USE_STATIC_CONST_PTR
	  static volatile unsigned * const ptr_peripheral =
	  (volatile unsigned*)0x40000100;
	  #define MY_REG1 ptr_peripheral[0x00]
	  #define MY_REG2 ptr_peripheral[0x04/4]
	  #define MY_REG3 ptr_peripheral[0x20/4]
	  #define SETUP_LOCAL // do nothing
	#endif

Use of C Standard Library

As explained in Table 1, the standard C library was not designed for embedded systems. Even with great effort from compiler vendors, the library tends to be much larger than warranted for normal embedded applications. Some vendors have reduced libraries, usually by cutting out features not used by embedded systems.

Some library functions are good to use and have usually been optimized. But, care must be taken. Generally, any functions that do not set errno and do not use the heap and do not imply persistent memory are okay to use. Memcpy() and other memory and string moving operations are usually quite compact for what they do, and are usually much faster than doing the same work in C.

DriverLib allows for peripheral access, but also debug uses, such as a printf() replacement.

RTOS vendors usually provide a subset of the C runtime library, as they provide their own memory and device concepts.

Asm() Inserts

Asm with Keil/ARM

The Keil/ARM compiler does not support asm() inserts anymore. This means that you have to use one of the following:

- Built-in “intrinsics”, which provide ways to insert special instructions and actions into the code directly. Examples such as __rbit (reverse bit), __rev (reverse bytes in a word, such as needed for network use) allow for direct insertion for most optimal code.
- __asm tagged functions. This means you write a function in assembly code and call it from your application.

The built-in intrinsics are optimal (when available) as the compiler does not treat them as calls. It knows what registers it can apply to them and what registers will be affected. So, it can use them without affecting optimizations.

The __asm tagged functions allow for functions to be written and then called from C. The functions must follow the calling conventions to work (for example, the first four parameters in R0, R1, R2, and R3, return in R0). For larger functions, there is no issue, as the overhead of calling and returning is normal and expected.

When trying to insert a single 32-bit instruction or two 16-bit instructions, it is inefficient to pay for a call and return. The linker provides a (partial) solution (using the --inline switch). Any asm function which is 6 bytes long, with the last 2 bytes as “BX LR” is inlined to replace the BL call. That is, the call, which is a 32-bit instruction is replaced with the first 4 bytes of the function. This is not a perfect solution since the compiler will still have assumed the function would modify R0-R4, and R12. But, it is better than paying for a call and return. An example function would be:

```
__asm int my_clz(unsigned word) {
 clz r0,r0
 bx lr
}
```

A call to this:

```
pos = my_clz(pri_bits);
```

Would be replaced with inline code, such as:

```
mov   r0,r7
clz   r0,r0
mov   r4,r0
```

Note that if the built-in intrinsic __clz were used, the code would look like:

```
clz   r4,r7
```

Further, the compiler will likely have had to move some data out of R1, R2, R3, and maybe R12 before calling my_clz.

Asm with GCC

GCC has an advanced asm() insert model. This allows the definition to include details about what registers are input, output, and destroyed. This allows the compiler to insert asm() code without breaking optimizations.

The general form is:

```
asm(instructions : output : input : destroyed);
```

The instructions are one or more instructions (in quotes, separated by \n). The output and input allows specifying registers or memory to feed the instructions. The destroyed list indicates a register that has been modified and/or that memory has been changed (affects aliasing).

An example is from the previous subsection:

```
asm("clz %0,%1": "=r" (pos): "r" (pri_bits));
```

The above example is interpreted as follows:

- The clz instruction's first operand (destination) is replaced with variable pos ("=r" means register to write to).
- The clz instruction's second operand (source) is replaced with variable pri_bits ("r" means register to read from).
- There are no destroyed registers.

From this, the compiler can use any register for the two operands, so would end up with code such as:

```
clz   r4,r7
```

Where r4 is the pos variable and r7 is the pri_bits variable.

A full description of the asm inline rules can be found on the Internet.

Floating Point

No additional information is needed.

Volatiles

It is important to understand where volatile goes in a variable declaration and a pointer:

```
volatile int       x;           // x is in volatile memory
volatile int       *p;          // p points to volatile memory
int   * volatile vp;            // vp is a volatile pointer to non-volatile memory
volatile int* const cpv = &mem; // cpv is a const pointer to volatile memory
```

The vp case is rarely meaningful (peripherals are not normally pointers). The cpv case is useful for peripheral pointers, as explained in "Accessing Peripherals (at Fixed Address)" on page 14.

Back-to-Back Writes to Peripherals

As explained in Table 1 on page 4:

```
MyPeriph_reg0 = x;
MyPeriph_reg1 = y;
MyPeriph_reg2 = z;
```

will often be less efficient than:

```
MyPeriph_reg0 = x;
y = some computation
MyPeriph_reg1 = y;
z = some computation
```

```
MyPeriph_reg2 = z;
```

The reason is that stores can complete in the background on Stellaris. This means that the STR generally takes one cycle or two (depending on what precedes it) even if wait states. Stellaris does not add wait states to peripheral memory; many processors, including most ARM chips, run the peripheral bus slower and so cause wait states. However, APB buses take three cycles to perform a write. So, if two STRs are back-to-back, then the second one must wait for the first to complete. If other activity precedes the STR, it will not stall, as a store buffer will drain the operation out.

Immediate Use of Loads from Peripherals

As explained Table 1 on page 4, code of the form:

```
x = MyPeriph_reg0; // x is local
y = MyPeriph_reg1;
z = MyPeriph_reg2;
// now operate on x, y, and z
```

will usually be more efficient and may compress to LDM. Unlike the STR case, the processor has to wait for the load to complete no matter what. So, back-to-back LDRs (and an LDM) optimize the time by pipelining the address generation. So, it is better to keep loads together when possible.

Recursion

No additional information is needed.

Many Small Functions

A strategy commonly used with 8-bit and 16-bit processors is to break up functions into many small functions. The purpose is three-fold:

- Most 8-bit and 16-bit compilers are not optimal and code generation gets worse as the function size and complexity increases. When small functions, the compiler can often “see” how to generate the smallest code.
- Because these are not usually real stack-based machines, the need for too many local variables causes use of slow memory and slower instructions.
- Some of the functions can be rewritten in assembly language to get smaller size or better performance.

None of these strategies apply to 32-bit processors. For Stellaris parts, moderate to large size functions are more efficient since they provide many opportunities for register reuse, amortize stack push/pop operations, amortize call/return overhead, allow for full use of the register set, and allow for other optimizations.

So, it is best to put functions back to the way they normally should be to get the best size and performance.

Too Many Function Parameters

As explained in Table 1 on page 4, the optimal number of function arguments is four or less scalar/ pointer variables. This allows for pure register passing.

Conclusion

There are many factors that affect performance and size for Stellaris microcontrollers. By looking through the factors discussed in this application note, it is often possible to find quick changes that can yield large improvements in size and/or performance. Further refinements can then be used as the application develops.

References

The following documents and source code are available for download at www.luminarymicro.com:

- *Stellaris LM3Snnn and Stellaris LM3Snnnn microcontroller data sheet*, (where nnn or nnnn is the device number) Publication Number DS-LM3Snnn or DS-LM3Snnnn
- Stellaris Family Peripheral Driver Library
- *Stellaris Family Peripheral Driver Library User's Manual*, publication PDL-LM3S1968

Important Notice

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2009, Texas Instruments Incorporated