NIGEL JONES

Minimize Interrupt Service Routine Overhead

With all the automation available today, it's easy for programmers to overlook costly overhead introduced into machine code by the compiler. Interrupt handlers are one key area worthy of a closer inspection.

In the early days of embedded C compilers, interrupt service routines (ISRs) had to be written in assembly language. Today, most compilers let the developer identify a function as an ISR, with the compiler taking care of all of the hassles associated with the ISR. This could include placing the correct entry into the interrupt vector table, stacking and unstacking registers, and terminating the function with a return from interrupt instruction. There are even compilers available that know when an ISR needs to clear a particular flag before the ISR terminates and will insert the proper code.

Although these advances are most welcome, they have come at a price–namely that it's so easy to write an interrupt handler in a high-level language that one can easily lose track of the overhead that the compiler is introducing. Doing so can unwittingly result in a high price for the convenience of using a high-level language.

ISR overhead can be split into two parts, fixed and variable. Fixed overhead is associated with the CPU detecting that an interrupt has occurred, vectoring to the ISR, clearing any required flags to prevent the ISR from being re-entered, and finally executing the return from interrupt instruction. Thus, an ISR that does nothing, such as the one shown in Listing 1, still incurs some CPU overhead every time it occurs. Naturally, there isn't much a compiler can do about the fixed overhead.

```
__interrupt void timer_isr(void)
{
      /* Do nothing */
}
```

Listing 1. Do nothing ISR

ISR variable overhead is the time spent stacking and unstacking the registers used in the ISR. This is a key topic, as it's not uncommon for the time spent stacking and unstacking registers to dwarf the time spent doing useful work. One of the ironies of using modern, register-rich CPUs is that the presence of all those registers encourages the compiler writers to use them!

Normally, this leads to very fast, tight code. However in an ISR, all those registers can work to our detriment.

Consider an ISR that uses 12 registers (and thus incurs 12 push and 12 pop operations). Even though the ISR's body may be faster than code that uses just six registers, the overall execution time of the ISR may increase simply because it must stack and unstack so many registers. For infrequently occurring interrupts, this is irrelevant. However for interrupts that occur frequently, this overhead can rapidly consume a very large portion of the CPU's bandwidth.

For example, consider a full-duplex serial link running at 38,400 baud on a CPU with a 1-MHz instruction cycle. If the CPU is interrupted upon receipt and transmission of every character, then assuming 10 bits per byte, it'll be interrupted every 106 / (38,400 / 10) / 2 = 130 ?s. If each interrupt requires 12 registers to be pushed and popped, then the processor spends 24/130 = 18% of its time doing nothing more than stack operations. A coding change that requires only six registers to be stacked would free up 9% of the CPU time and save stack space.

To determine how much overhead you are incurring in an ISR, have the compiler generate an assembly language listing of your C code, preferably with the C instructions interleaved. An example is shown in Listing 2.

```
160 static   interrupt void timer0_CompareMatchAIsr(void)
\     timer0_CompareMatchAIsr:
  161 {
\     00000000    938A                    ST    -Y,   R24
\     00000002    93FA                    ST    -Y,   R31
\     00000004    93EA                    ST    -Y,   R30
\     00000006    923A                    ST    -Y,   R3
\     00000008    922A                    ST    -Y,   R2
\     0000000A    921A                    ST    -Y,   R1
\     0000000C    920A                    ST    -Y,   R0
\     0000000E    937A                    ST    -Y,   R23
\     00000010    936A                    ST    -Y,   R22
\     00000012    935A                    ST    -Y,   R21
\     00000014    934A                    ST    -Y,   R20
\     00000016    933A                    ST    -Y,   R19
\     00000018    932A                    ST    -Y,   R18
\     0000001A    931A                    ST    -Y,   R17
\     0000001C    930A                    ST    -Y,   R16
\     0000001E    B78F                    IN    R24,  0x3F
  162 TCCR0B = 0;       /* Stop the timer */
\     00000020    E000                    LDI   R16,  0
\     00000022    BF03                    OUT   0x33, R16
  163 fifo_AddEvent(Event);          /* Post the event */
\     00000024    9100....                LDS   R16,  Event
\     00000028    ....                    RCALL fifo_AddEvent
  164 }
\     0000002A    BF8F                    OUT   0x3F, R24
\     0000002C    9109                    LD    R16,  Y+
\     0000002E    9119                    LD    R17,  Y+
\     00000030    9129                    LD    R18,  Y+
\     00000032    9139                    LD    R19,  Y+
\     00000034    9149                    LD    R20,  Y+
\     00000036    9159                    LD    R21,  Y+
\     00000038    9169                    LD    R22,  Y+
\     0000003A    9179                    LD    R23,  Y+
\     0000003C    9009                    LD    R0,   Y+
\     0000003E    9019                    LD    R1,   Y+
\     00000040    9029                    LD    R2,   Y+
\     00000042    9039                    LD    R3,   Y+
\     00000044    91E9                    LD    R30,  Y+
\     00000046    91F9                    LD    R31,  Y+
\     00000048    9189                    LD    R24,  Y+
\     0000004A    9518                    RETI
```

Listing 2. Mixed C and generated assembly language listing of two-line ISR

Even if you aren't familiar with your CPU's instruction set, the push and pop operations are usually easy to spot. Note that this two-line ISR requires 15 registers to be stacked and unstacked.

I'll present various techniques for minimizing ISR overhead. The order they're presented in follows a project's progression. The first suggestions are only applicable at the start of a project, whereas the final suggestions apply to the poor soul at the end of a project who needs to improve performance without changing anything.

CPU selection

If you're a regular reader of *Embedded Systems Design*, you know that the latest CPUs offer amazing capabilities; yet survey after survey shows that huge numbers of us continue to work with ancient architectures such as the 8051. I previously wrote this off as a simple case of inertia until I went through an interesting exercise about a year ago, when I had to select a small CPU core to be embedded in an ASIC. The ASIC was to be used in a portable product where ultra low-power consumption was critical. As is the case for most low-power products, most of the code was to be executed under interrupt, and hence ISR overhead was a major concern.

To determine the best CPU for the application, example pieces of code were compiled using the best compilers I could find for each CPU candidate and the energy consumed by each CPU calculated. The results were quite clear: the older CPUs such as the 8051 and the HC08 from Freescale had a distinct advantage over the newer, register-rich architectures. This advantage was due in part to the low ISR overhead of these devices. In the case of the 8051, this advantage also had a lot to do with the fact that the 8051 allows one to allocate a register bank to a particular block of code–such as an ISR–and thus drastically reduce the number of registers that need to be stacked.

The bottom line is if you know your code will be dominated by interrupt handlers, don't be in too much of a hurry to embrace the latest and greatest register-rich CPU.

Compiler selection

As will be shown shortly, a major factor on ISR performance is your compiler's register allocation strategy. If you aren't familiar with this term, it refers to the algorithm the compiler uses in allocating registers across function calls. In broad terms, a compiler has three options:

- Assume that a called function will trash every register
- Require that the called function preserve every register that it uses
- Use a hybrid approach, whereby certain registers are deemed scratch registers–and hence may be trashed by a called function, and other registers are preserved registers–and thus must be preserved if used by a called function

Now, if your compiler has a sophisticated global register-coloring algorithm, whereby it can track register allocation across inter-module function calls–and so eliminate all unnecessary register preservation instructions, then these three algorithms effectively all collapse down to the same thing, and thus the strategy is of little concern. However, if your compiler doesn't do this, then you need to be aware of its impact on making function calls from within interrupt handlers.

Firmware design

Once you've chosen your CPU and compiler, you'll probably turn your attention to the detailed firmware architecture. A key decision that you'll have to make is what interrupts are needed and what gets done within them. In making these decisions, be aware of the consequences of making function calls from within an ISR.

Consider the example shown in Listing 3, which is the source code for the assembly language listing shown in Listing 2. On the face of it, it meets the criteria of keeping interrupt handlers short and simple. After all it's just two lines of code.

```
void fifo_AddEvent(uint8_t event);

__interrupt void timer_isr(void)
{
      TCCROB = 0;                    /* Stop the timer */
      fifo_AddEvent(Event);          /* Post the event */

}
```

Listing 3. A "simple" ISR

However, as Listing 2 shows, a huge number of registers have been stacked. The reason is quite simple: the compiler that generated this code uses a hybrid register allocation strategy. Thus, without any other information, the compiler has to assume that the called function will use all the scratch registers it's allowed to use–and so stacks all of them. Now, if the called function is a complex function, then stacking all the registers is unfortunate but necessary. However, if the called function is simple, then most of the register stacking is probably unnecessary. In order to avoid this problem, don't make function calls from within an ISR.

This advice falls into the same category as soldiers being advised to avoid dangerous situations for health reasons. It's true, but useless most of the time. Presumably, if you're making a function call, you're doing it for a good reason. In which case, what can you do about it? In a nutshell, you must do something that lets the compiler know what registers the called function is using. There are three ways you can do this, but they all amount to the same thing.

If you are writing in C++, or C99, then make the function that's called from the ISR an inline function. This will result in copies of the function being used throughout the project, but should result in only the required registers being stacked. It's your call as to whether the tradeoff is acceptable.

If you are using C89, you can do something similar by using macros and global variables–with all the problems that this entails.

The third option is to ensure that the ISR and the called function reside in the same module. In this case, a good compiler can see exactly what you're calling and thus stack only the required registers. Furthermore, if it's possible to declare the called function as static, there's a good chance that the compiler will inline the function and also eliminate the overhead of the actual function call.

Occasional function calls

Although the overhead of a function call can be annoying, it can be frustrating when you have code that looks like Listing 4. In this case, most of the time, the ISR simply decrements a counter. However, when the counter reaches zero, a function call is made. Unfortunately, many compilers will look at this ISR, see that a function call is made, and stack all the registers for the function call every time the ISR is invoked. This wastes CPU cycles. The long-term solution to this problem is to pressure the compiler vendors to improve their product such that the registers only get stacked when the function call is made. In the short term, use a software interrupt. The code for such an approach is shown in Listing 5.

```
__interrupt void timer_isr(void)
{
      if (--Cntr == 0)
      {
            post_event(TIMER_TICK);
      }

}
```

Listing 4. An occasional function call

```
__interrupt void timer_isr(void)
{
      if (--Cntr == 0)
      {
            __software_interrupt;
      }
}

__interrupt void software_isr(void)
{
      post_event(TIMER_TICK);
}
```

Listing 5. Use of software interrupt

In place of the function call, we generate a software interrupt. This results in another interrupt being generated. The requisite function call is made from within this interrupt. The advantage of this approach is that the timer ISR no longer needs to stack the registers associated with the function call. Of course the software interrupt handler does need to stack the registers, but now they're only stacked when they're really needed. The main cost of this approach is slightly increased latency from the time the interrupt occurs to the time the function call is made.

Depending on your perspective, this technique is either very elegant or a kludge. However, with adequate documentation, it should be possible to explain clearly what you're doing and why. For me, the choice between doing this and resorting to assembly language is clear cut, but the choice is yours.

Incidentally, you may be unaware of the concept of a software interrupt. Some CPUs include in their instruction set a software interrupt instruction (SWI). Execution of the instruction causes an interrupt flag to be set (just as if a timer had rolled over, or a character had been received in a UART). If all the usual criteria are met, at some point the CPU will vector to the software interrupt handler and execute it just like any normal hardware based interrupt.

Don't be despondent if your CPU doesn't include a SWI because it's possible to fashion a software interrupt in several ways using unused hardware resources. The easiest way is typically to configure an unused port pin as an output, while also configuring it to interrupt on change. Thus, simply toggling the port pin will generate an interrupt.

A second technique is to employ a spare counter. In this case, one can simply load the counter with its maximum value, configure it to interrupt on roll over, then enable the counter to count up. With a bit of ingenuity, you'll find that most of the hardware resources in a typical microcontroller can be configured to generate an interrupt upon software command.

A final warning for those of you whose CPU supports an SWI instruction, is that debuggers often make use of the SWI instruction to set breakpoints. So before using this technique, make sure you aren't going to break your debugger.

Coding constructs

What if your CPU has been selected and your software architecture designed, such that the functionality of your interrupt handlers is set? In this case, it's time to look at some of your coding constructs. Small changes in this area can make surprisingly large changes to the number of registers that need to be stacked. Most of the normal suggestions for making code tighter apply, of course, and won't be reiterated here. However, there are a few areas to watch out for.

Switch statements are a bad idea for two reasons. First, the overhead associated with a switch statement is typically quite large, resulting in a lot of registers being stacked. Second, most compilers will implement switch statements in one of several ways depending on the values that are switched. Thus, a small change to a switch statement can result in a completely different set of instructions being generated (and hence registers stacked). The result is that an innocuous change to the code can make a dramatic difference to the CPU bandwidth consumed. This is not something that makes for a maintainable system.

Watch out for unexpected type promotions. These can have a devastating impact on an ISR in two ways. First, a type promotion can result in more registers being used to hold a variable than is necessary (which of course means that more registers need to be stacked). Second, a type promotion may result in the compiler making a function call to a library function to perform an operation rather than doing it in line. If you're lucky, your compiler knows what registers are used by library functions. If you're unlucky, you'll be hit with the full overhead of a function call.

Interrupts often test and/or set flags. Many processors contain special flag locations that can be operated on without going through a CPU register. These special flag locations can thus have a double benefit within interrupts as one avoids the overhead of stacking the intermediate registers.

If you have done all that you can to minimize overhead, and you are still facing an unacceptable burden, before reaching for the assembler, look at your compiler optimization settings, as the usual rules don't always apply to interrupt handlers.

Most compilers will let you optimize for size or speed. Furthermore, most compiler manuals carry a notice that sometimes optimizing for speed will actually result in a smaller footprint than optimizing for size. This can occur because the act of inlining code, unraveling loops, and so forth, results in the optimizer seeing a different view of the code and making optimizations that weren't previously possible. Interestingly, I've never seen a compiler manual admonish the user that optimizing for size may result in faster executing code. However, this is exactly what can happen in the case of an ISR.

I'm not privy to the way compiler writers optimize interrupt handlers. However, what seems to happen is that the optimizer concentrates on optimizing the C language constructs and then simply adjusts the registers that need to be stacked based on the resulting code, rather than looking at the entire routine. By asking the compiler to optimize for size, you create the possibility of the compiler generating more compact code that uses less registers. Because it uses less registers, the stacking overhead is reduced and the overall execution time of the ISR is potentially reduced. Thus the bottom line is, when asking a compiler to optimize an ISR, try both optimizing for speed and size, as you may find that size-optimized code is actually faster.

This article was published in the January 2007 issue of *Embedded Systems Programming*. If you wish to cite the article in your own work, you may find the following MLA-style information helpful:

Jones, Nigel. "Minimize your ISR overhead" *Embedded Systems Programming*, January 2007.