Strategies to Extend Battery Life in Embedded Systems

by

Neil Puthuff
Field Applications Engineer
Green Hills Software
Santa Barbara, CA
805-965-6044
neil.puthuff@ghs.com

Introduction

As data, voice, fax, and video communications converge, today's and tomorrow's advanced telecommunications devices are entering the marketplace. With end-users demanding expanded functionality and performance, product designers are forced to incorporate increasingly power-hungry features. Yet, form factors, extended battery life, and other advanced features remain key design parameters designers must balance.

Power efficiency is particularly sensitive in such battery-powered devices as cell phones and PDAs. As these more complex applications require greater CPU resources, power usage can become the critical factor. Effective power usage translates directly into key design issues such as battery life, battery size, and the overall weight of the final product. The formula for success is very straightforward: Increases in power efficiency will allow for extended battery life or reductions in battery size. It's easy to see why embedded system designers are therefore constantly on the lookout for new power-saving tricks.

Traditionally, designers have relied on hardware based strategies to reduce the power consumption in an embedded system. These approaches include reduced voltages, improved processes, and greater control over power use. These approaches have worked, especially when the CPU is in a suspended state. The following graph shows the active and standby power consumption of several popular CPUs.

[Insert Fig 1: CPU Active and Suspended Power Dissipation]

This graph clearly demonstrates that the more often the CPU is in a suspended state, the lower the overall power consumption will be, resulting in longer battery life. This often overlooked fact provides a new opportunity for portable system designers. This new method imposes no per unit cost penalty, requires no changes to circuitry, and can also increase the amount of available CPU processing power. The secret: Use the best optimizing compiler you can buy.

The Compiler: The Low Cost Power Saving Component

Every instruction executed by an embedded microprocessor represents a drain on the battery. If an embedded program can use few instruction cycles for a given function, the CPU will be in a suspended state sooner. This results in decreased battery use, meaning longer battery life.

To achieve a suspended CPU state more frequently, the designer of a portable embedded project can structure the embedded software to periodically execute code, then suspend the CPU until the start of the next execution period. The power consumed by such an application will follow the general pattern shown below in Fig. 2.

[Insert Fig. 2: CPU Active and Suspended State Battery Drain]

In this general application, the CPU executes a set of instructions and goes into a low power suspended state. If the functions employed in an application can be optimized to execute with a reduced set of instructions, then the CPU could suspend operations sooner, thus extending the battery life. The key is to optimize code so it executes more quickly.

One approach is for a skilled software engineer to produce the application in hand-coded assembly language. Unfortunately, coding in assembly language is an inefficient and error-prone software development process. This approach usually results in prohibitively long product development cycles.

High level languages such as C, C++, and Embedded C++ enable much more efficient and maintainable software development processes. With these languages, high-speed execution depends on the compiler and the software engineer's knowledge of the compiler and its tools. Optimizing compilers generate embedded software that runs faster than unoptimized – sometimes twice as fast or more. The software engineer can use a number of different strategies available in the compiler, sometimes at the expense of a larger program size in memory. When a development team selects a compiler, it is therefore important that the compiler offer a large number of optimization strategies and options.

Accessibility is almost as important as the compiler's skill at generating optimized code. If the software engineer can't access and understand the workings of the compiler, even the smartest optimizations are not helpful. Most compilers support a set of command-line switches to implement various optimization strategies. Some provide the software engineer with an easy-to-use graphical interface to set optimization options. For example, the screen in Fig. 3 illustrates an interface the software engineer can easily use to set optimizations at the function, file, subproject, or program level.

[Insert Fig 3: Optimization Options for Green Hills MULTI® IDE Program Builder]

Code Optimization Strategies

Each of the following five code optimization strategies briefly describes the theoretical circumstance for use, possible risks, and the likely benefit.

1. Inlining
Circumstance: In a program, the main body of code makes calls to a set of functions. Each time an outside function is called, CPU cycles are consumed as overhead in saving and restoring registers and passing values to and from the called function. Inlining is a potentially useful optimization strategy when one function is called hundreds of times from the same place in the main body of the code. Every time this function is called, cycles are wasted in the calling overhead. An optimization strategy to solve this problem is to insert the called function into the main body of code. This also replaces the call to the function with a copy of the entire function. This is known as "inlining a function."

The software engineer runs the risk that aggressive inlining will substantially increase the code size. With simple inlining strategies, a copy of the entire function will be placed at every place within a program that it's called. Improvements in compilers now allow the software engineer to selectively inline functions on a file-by-file basis.

2. Loop unrolling
Another popular optimization strategy is loop unrolling. Consider the following C code section:

```
for(iIndex=255 ; iIndex ; iIndex--) {
//
// perform some repetitive loop operation here
//
}
```

Unoptimized, this code will execute something like this:
A. initialize the loop counter to 255
B. test if loop counter = 0 then done
C. perform the loop operation
D. decrement the loop counter
E. jump back to B (above)
This sequence incurs the overhead of decrementing, jumping, and testing 256 times before the loop is complete.
To optimize this circumstance, the compiler deploys the "loop unrolling" optimization strategy. Now the same code executes like this:
A. initialize the loop counter to 255
B. test if loop counter = 0 then done
C. perform the loop operation
D. perform the loop operation
E. perform the loop operation
F. perform the loop operation
G. subtract 4 from the loop counter
H. jump back to B (above)
The overhead of decrementing, jumping, and testing has been reduced to one quarter of its original amount, at the expense of increased code size.

3. Remove loop invariant expression out of loops

In this unoptimized loop operation, the calculation of ((iCoef*6/3) occurs on every pass through the loop – even though the resulting value never changes.

```
int iCoef=10, j=30;
int iArray[30];
while(j)
{
iArray[--j]=((iCoef * 6) / 3);
}
```

Good coding practice (experience gained with unoptimizing compilers) encourages the software engineer to move the invariant value from the inside to the outside of the loop. Modern optimizing compilers will automatically perform this move for the engineer.

4. Register caching invariant memory references over loops

Similar to the above, the main body of the code makes repeated accesses to memory locations that don't change. The compiler can implement an optimization that accesses the memory once at the beginning of the loop and stores its value in a CPU register where it can be quickly accessed.

5. Rotate loop termination tests to the bottom

In the above example for loop unrolling (#2 above), the test for when the loop operation was finished was performed at the top of the loop. When the loop is finished, it will jump to the start of the loop (to test for loop completion), then jump back to the end of the loop to exit. Placing the loop test at the end of the loop can eliminate this set of jump operations, as follows:

A. initialize the loop counter to 255
B. perform the loop operation
C. decrement the loop counter
D. test if loop counter does not equal 0 then jump back to B (above)

Optimization Example on a PowerPC:

The following example code illustrates the advantages a good compiler capable of a wide variety of optimizations. This code segment contains a loop similar to the types of loops many designers would incorporate in their portable or battery-operated designs. [See Fig. 4 for graphic comparison of unoptomized code vs. the two different optimization approaches discussed below.]

```
int main(void)
{
volatile char j[256], l[256], m[256];
int i, k=8;
for(i=255 ; i ; i--)
{
int val = m[i] + i;
if(j[i]>l[i]) {
```

```
val = val * k;
}
m[i] = val;
}
}
```

Compiled, but with no optimization, the code above requires 12,753 clock cycles and 25 instruction words and looks like this:

```
stwu sp, -776(sp)
li r9 <k>, 8                    // int i, k=8;
li r8 <i>, 0xff                 // for(i=255 ; i ; i--)
addi r12, sp, 8                 // int val = m[i] + i;
add r12, r12, r8 <i>
lbz r12, 0(r12)
add r10 <val>, r12, r8 <i>
addi r12, sp, 0x108             // if(j[i]>l[i])
add r11, r12, r8 <i>
addi r12, sp, 0x208
add r12, r12, r8 <i>
lbz r11, 0(r11)
lbz r12, 0(r12)
cmplw r11, r12
ble main+0x40 (0x10110)
mullw r10 <val>, r10 <val>, r9 <k>          // val = val * k; }
addi r12, sp, 8                             // m[i] = val;
add r11, r12, r8 <i>
clrlwi r12, r10 <val>, 24
stb r12, 0(r11)
addi r8 <i>, r8 <i>, -1
cmplwi r8 <i>, 0
bne main+0xc (0x100dc)                      // } }
addi sp, sp, 0x308
blr
```

When the loop and speed optimization options are selected (see Fig. 3) as optimizing strategies, the following compiled code now requires only 6,011 clock cycles and 19 instruction words. The compiler optimizes the code to take advantage of the PowerPC's hardware loop capabilities and also replaces the "multiply" instruction with a faster executing "shift" instruction.

```
li r10 <i>, 0xff                    // int i, k=8;
stwu sp, -776(sp)
addi r6, sp, 0x308
addi r9, sp, 0x208
addi r11, sp, 0x108
```

```
li r5, 0xff                          // for(i=255 ; i ; i--)
mtctr r5
lbzu r12, -1(r6)
lbzu r7, -1(r11)                     // if(j[i]>l[i]) {
lbzu r8, -1(r9)
cmplw r7, r8
add r12, r12, r10 <i>
ble main+0x38 (0x10108)
slwi r12, r12, 3                     // val = val * k; }
stb r12, 0(r6)                       // m[i] = val;
addi r10 <i>, r10 <i>, -1
bdnz main+0x1c (0x100ec)             // } }
addi sp, sp, 0x308
blr
```

The resulting compiled optimized code requires less memory and operates twice as fast as the same compiled, but unoptimized code. When the code is optimized totally for speed, based primarily on loop unrolling (example not shown), the speed improves to 5,744 clock cycles, a 4.6% improvement. However, the code size swells almost 10 times to 185 instruction words.

Conclusion

Designers of portable and battery-operated devices have a new ally, the optimizing compiler, in their on-going struggle to balance increasingly power-hungry feature sets, smaller form factors, and extended battery life. Intelligently applied, an optimizing compiler can often reduce the clock cycles necessary to execute a code segment and/or reduce the memory required. This leaves the processor in standby mode more frequently, reducing the drain on the battery, thus extending battery life.

The examples shown above demonstrate that a good optimizing compiler should include a wide variety of optimization options. The designer should be able to apply the optimizations selectively to different parts of the program as appropriate. Optimization strategies need to be carefully considered because some may achieve advanced speed, but do so at the sacrifice of substantial additional memory. Successful optimization in tightly designed applications requires carefully balanced, selective application. Aggressive optimization should be applied to the parts of the program that are executed most frequently, while the remainder of the program should be optimized with less-aggressive (but still much faster) compiler strategies.

* * * * * * END * * * * *

(Word Count: 1,998)