Use an MCU's low-power modes in foreground/background

By Miro Samek
President
Quantum Leaps, LLC

In today's world of battery-operated devices, the proper use of the low-power/sleep modes provided in most embedded microcontrollers (MCUs) is critical. At the same time, most high-volume MCU applications, such as home appliances, vending machines, motor controllers, and electronic toys, are organised as foreground/background systems (super-loops or main + ISRs).

The foreground/background architecture consists of two main parts—the foreground comprises the interrupt service routines (ISRs) that handle asynchronous external events in a timely fashion, and the background is an infinite loop that uses all remaining CPU cycles to perform the less time-critical processing.

The foreground typically communicates with the background through shared memory. The background loop protects this memory from potential corruption by disabling interrupts when accessing the shared variables.

To employ a low-power MCU mode, the background loop must first determine that all external and internal events have been processed so that the CPU clock can be stopped until the next external event (an interrupt) will wake the CPU up. This situation is called the idle condition and is illustrated in **Figure 1**.

Because the determination of the idle condition involves testing the variables shared with the foreground (ISRs), the background loop must disable interrupts before detecting the idle condition. Moreover, the idle condition remains valid only as long as interrupts remain disabled. If the interrupts were enabled after the background loop determines that all its work is done for now, but before actually switching to the low-power mode, an interrupt could preempt the background loop at this point and an ISR could produce new work for the background loop, thus invalidating the idle condition.

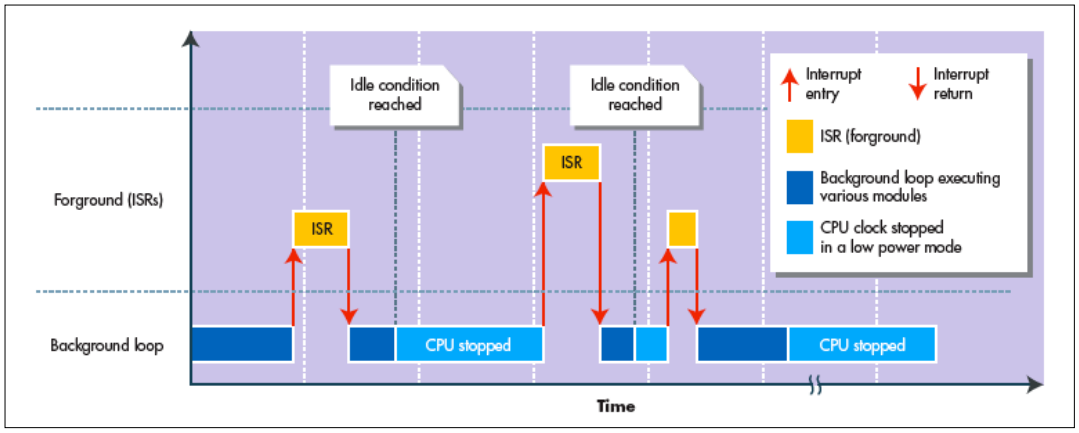

Figure 1: Foreground/background system with a low-power sleep mode.

By the simplistic nature of the foreground/background processing, the background loop always resumes at the point it was interrupted, so the background loop would enter the low-power sleep mode while the MCU would have urgent work to do. The MCU will be stopped for a non-deterministic time period until the next interrupt wakes it up. Thus, enabling interrupts before transitioning to a low-power state opens a time window for a race condition between any enabled interrupt and the transition to the low-power mode.

Entering a sleep mode while interrupts are disabled poses a chicken-and-egg problem for waking the system up, because only an interrupt can terminate the low-power sleep mode. To operate in the foreground/background architecture, the MCU must allow entering the low-power sleep mode and enabling the interrupts at the same time, without creating this race condition.

```
main() {
    . . .                                   /* initialization */
    for (;;) {                              /* for-ever background loop */
        . . .                               /* the main body of the background loop */
        __asm SEI;                          /* Set Interrupt Mask (disable interrupts) */
        if (idle_condition()) {             /* idle condition reached? */
            __asm WAIT;                     /* atomically enter the WAIT mode */
        }
        else {                              /* idle condition not reached yet */
            __asm CLI;                      /* clear Interrupt Mask (enable interrupts) */
        }
    }
}
```

Listing 1: Code for the CodeWarrior HCO8 C-compiler shows a background loop with the atomic trasntions to the WAIT mode.

```
    _disable_interrupts();                  /* _asm("DINT"); */
    if (idle_condition()) {                 /* idle condition true? */
        _bis_SR_register(LPM1_bits | GIE);  /* _asm("BIS.W #0x58,SR"); */
    }
    else {                                  /* idle condition not reached yet */
        _enable_interrupt();                /* _asm("EINT"); */
    }
```

Listing 2: Code for the GNU MSP430 gcc compiler shows how to atomically transition to the LPM1 low-power mode and simultaneously enable the interrupts.

```
    __interrupt void Timer_A(void);          /* prototype for ISR */
    TIMERA0_ISR(Timer_A) __interrupt void Timer_A (void) {
        _bic_SR_register_on_exit(LPM1_bits);  /* clear LPM1 bits in the stacked SR */
        . . . /* process the TIMER0 interrupt */
    }
```

Listing 3: The ISR example for the GNU gcc compiler for MSP430.

Many MCUs indeed allow such an atomic transition to the sleep mode. Other MCUs support multiple levels of disabling interrupts and can accomplish low-power transitions with interrupts disabled at one level. Yet other MCUs don't provide any way to enter the low-power mode with interrupts disabled and require some different approaches.

```
__disable_interrupt();          /* __asm("CLI"); */
if (idle_condition()) {         /* idle condition reached? */
    SMCR = . . .;               /* manage clocks specifically to your project */
    __enable_interrupt();       /* __asm("SEI"); */
    __sleep();                  /* __asm("SLEEP"); */
}
else {                          /* idle condition not reached yet */
    __enable_interrupt();       /* __asm("SEI"); */
}
```

Listing 4: A C example for the IAR AVR compiler shows hos to enter the sleep mode from the background loop.

```
__asm__ __volatile__ ("CLI" "\n\t" :: );
if (idle_condition()) {          /* idle condition reached? */
    SMCR = . . .;                /* manage clocks specifically to your project */
    __asm__ __volatile__ ("SEI" "\n\t" :: );
    __asm__ __volatile__ ("SLEEP" "\n\t" :: );
}
else {                           /* idle condition not reached yet */
    __asm__ __volatile__ ("SEI" "\n\t" :: );
}
```

Listing 5: A C example for the GNU AVR (WinAVR) compiler that shows how to enter the sleep mode from the backgound loop.

```
__disable_interrupt();          /* disable interrupts at the ARM-core level */
if (idle_condition()) {         /* idle condition reached? */
    AT91C_BASE_PMC->PMC_SCDR = 1; /* Power-Management: disable CPU clock */
    __enable_interrupt();       /* enable interrupts at the ARM-core level */
}
else {                          /* idle condition not reached yet */
    __enable_interrupt();       /* enable interrupts at the ARM-core level */
}
```

Listing 6: The general strategy of transitioning to a low-power mode for ARM-based MCUs (IAR ARM compiler, AT-91SAM MCU)4.

HC08

HC08 is an 8-bit MCU family from Freescale Semiconductors. The HC(S)08 instruction set includes two special instructions—WAIT and STOP—for transitioning to the low-power wait and stop modes, respectively.[1] The HC08 documentation states very clearly that both WAIT and STOP instructions atomically enable interrupts as a side effect of entering the sleep mode. Clearly, the HC08 designers anticipated that the transition to the low-power mode must happen with interrupts disabled. The code in **Listing 1** for the CodeWarrior HC08 C-compiler shows a background loop with the atomic transition to the WAIT mode.

MSP430

The MSP430 is an ultra-low-power, 16-bit MCU from Texas Instruments. It allows for a clean atomic transition to any of the five supported low-power modes because the bits controlling the various clock domains and the general interrupt enable (GIE) bit are all located in the same CPU status register (SR).[2]

The code in **Listing 2** for the GNU gcc MSP430 compiler shows how to atomically transition to the LPM1 low-power mode and simultaneously enable the interrupts. In particular, the macro _bis_SR_register (LPM1_bits | GIE) generates a single machine instruction BIS.W #0x58,SR, which atomically sets bits 0x58 in the SR (status register). The bit 0x10 (CPUOFF) turns the CPU clock off, while the bit 0x08 (general interrupt enable) enables interrupts.

While atomic transition to any low-power mode is natural in the MSP430, you have the opposite problem: in each ISR you must explicitly disable the low-power mode in the stacked SR, so that the machine doesn't automatically return to the low-power mode, but rather the background loop can continue after the ISR restores the SR from the stack as part of the return from the interrupt. Luckily, this is quite simple with the intrinsic functions provided by most C compilers for the MSP430. **Listing 3** shows the ISR example for the GNU gcc compiler for MSP430.

AVR

Atmel's AVR low-power 8-bit RISC also provides a method for atomic transition to the sleep mode, but it's less obvious than in the case of HC08 or MSP430. The AVR core provides a SLEEP instruction to stop the CPU clock, but it doesn't enable the interrupts and, in fact, must be executed with interrupts enabled. This would be a problem, if not for the following obscure note in the AVR datasheet:3

"When using the SEI instruction to enable interrupts, the instruction following SEI will be executed before any pending interrupts, as shown in this example:

```
SEI     ;       set Global Interrupt Enable
SLEEP;          enter sleep, waiting for interrupt
        ;       note: will enter sleep
        ;       before any pending interrupt(s)
. . .”
```

In other words, the pair of instructions SEI-SLEEP is guaranteed to execute atomically, most likely due to the AVR pipeline structure. Be careful to always use the SEI-SLEEP pair of instructions together, never separated by any other instruction.

Listing 4 is a C example for the IAR AVR compiler that shows how to enter the sleep mode from the background loop. The same example for the GNU AVR (WinAVR) compiler is shown in **Listing 5**.

ARM

ARM-based MCUs take a different approach to the atomic low-power transition. The ARM silicon vendors, such as Atmel, NXP (formerly Philips), and TI, integrate the standard ARM7 or ARM9 cores with the set of proprietary peripherals, such as the interrupt and power-management controllers. The integration is loose in that the ARM core's internal state doesn't impact the peripherals. In particular, the core can disable interrupts internally by setting the I and F bits in the current program status register (CPSR), but it doesn't effect the external power-management or interrupt controller, which provide another layer for disabling and enabling interrupts.

Because of this design, the ARM-based MCUs allow transitioning to a low-power mode with interrupts disabled at the ARM-core level. Upon such a transition, the power-management controller stops the CPU clock for the ARM core, but any interrupt enabled at the interrupt controller level can start the CPU clock. As soon as the core starts running again, it can enable interrupts to achieve low interrupt latency.

Listing 6 shows the general strategy of transitioning to a low-power mode for ARM-

based MCUs (IAR ARM compiler, AT91SAM MCU).4 The power-management controller stops the CPU clock (AT91C_BASE_PMC->PMC_SCDR = 1) while the interrupts are disabled at the core. The interrupts are enabled only after the CPU wakes up again and executes the __enable_interrupt() intrinsic function. You can see this behaviour if you try to break into a running application with a JTAG-based debugger. Usually, the code will stop at the __enable_interrupt() line.

Speaking of debugging, the sleep mode can interfere with many on-chip debuggers because it stops the CPU clock. Therefore, you must use conditional compilation to include the low-power transition only in the nondebug (production) version of the code.

Cortex-M3

Cortex-M3 is ARM's 32-bit RISC architecture designed for low-cost and low-power mobile applications. It differs from the traditional ARM7 or ARM9 cores in a few ways. The main difference relevant to this discussion is a tighter integration of the MCU core with the system power management and the nested vectored interrupt controller (NVIC).

The Thumb-2 instruction set, used exclusively in the Cortex-M3, provides a special instruction WFI (wait for interrupt) for stopping the CPU clock. Unfortunately, the reference manuals (the ARMv7-M Reference Manual, the Cortex-M3 Technical Reference, or the LM3Sxxx data sheets)5,6,7 don't describe whether the WFI instruction can be used with interrupts disabled.

Given this lack of information, I was forced to experiment with the actual Cortex-M3 MCU. Using the LM3S811 Cortex-M3 MCU from Luminary Micro, I discovered that the WFI instruction can be used while interrupts are locked (the PRIMASK register set to one). As expected, after the WFI instruction, the LM3S811 stops executing code, but any interrupt enabled in the NVIC wakes the CPU up. **Listing 7** shows the atomic transition to the sleep mode for the Cortex-M3 (IAR ARM compiler).

8051

The 8051 architecture supports two low-power levels (idle and power down). These modes are activated by setting the IDL or PD bits in the power control register PCON at the address 0x87. Writing to the IDL or PD bit stops the CPU immediately and must happen with interrupts enabled, otherwise the 8051 locks-up. This means that it's impossible to transition to one of these modes atomically. Any enabled interrupt can preempt the idle processing after the interrupts are enabled, but before the idle mode is entered. Clearly, the 8051 requires a different technique than those discussed so far.

This other technique is to invalidate the idle mode transition in every interrupt. So, if an interrupt preempts the background loop just before the indented transition to idle, the ISR will disable the transition. After the interrupt returns, the idle mode is not entered.

One way of implementing this technique on the 8051 is to shadow the PCON register allocated in the 8051's bit-addressable memory (bdata). Let's call this variable PCON_shadow. **Listing 8** (for the Keil C51 compiler) shows how the background loop uses the PCON_shadow variable.

The background loop sets the IDL bit only in the PCON_shadow variable when the interrupts are still disabled. Then, interrupts get enabled and the register PCON is restored from the shadow. It's important that the PCON register's update occurs in one machine instruction. As it turns out, the simple assignment of a bit-addressable variable to the special register, such as PCON, can be accomplished in one instruction—MOV 87H,20H.

The PCON shadow must be updated in every ISR that can produce work for the background loop, as shown in Listing 9. Note that the 8051 clears the IDL/PD bits in the PCON register before entering any interrupt, so these bits are guaranteed to be cleared in the shadow register when it's updated from PCON in the interrupt context.

With this design, an interrupt can occur at any machine instruction between enabling interrupts until restoring the PCON register from the shadow PCON_shadow. Any such interrupt will clear the IDL/PD bits in the PCON_shadow variable, so the bits won't survive to the point when the background loop actually restores PCON from the shadow. Thus, any interrupt that preempts the idle loop disables the idle mode, which accomplishes the goal of an interrupt-safe transition to idle mode.

M16C

The M16C 16-bit processor from Renesas supports the low-power wait mode, which is entered using a special WAIT instruction. However, the M16C datasheet is very specific that interrupts must be enabled before executing the WAIT instruction, so clearly the M16C doesn't support an atomic transition to the wait mode.8 The M16C datasheet contains no side notes similar to the AVR note about atomic execution of the SLEEP-SEI instruction pair, so I assume that the interrupt-disable instruction (FCLR I) is effective immediately.

Like the 8051, the only option for the M16C is to somehow disarm the transition to the wait mode in the ISRs, to prevent the background loop from entering the wait mode just after an interrupt. In contrast to the 8051, however, the M16C accomplishes the low-power mode transition using a special instruction, not

```
__disable_interrupt();              /* __asm("CPSID i"); */
if (idle_condition()) {             /* idle condition reached? */
    __asm("WFI");                   /* stop the CPU and Wait-For-Interrupt */
    __enable_interrupt();           /* __asm("CPSIE i"); */
}
else {                              /* idle condition not reached yet */
    __enable_interrupt();           /* __asm("CPSIE i"); */
}
```

Listing 7: The atomic transition to the sleep mode for the Cortex-M3 (IAR ARM compiler).

```
unsigned char bdata PCON_shadow;  /* allocate PCON_shadow in bdata area */
. . .
EA = 0;                           /* lock interrups */
if (idle_condition()) {           /* idle condition reached? */
    PCON_shadow = PCON | 0x01;    /* set the IDL bit in the shadow */
    EA = 1;                       /* unlock interrupts */
    PCON = PCON_shadow;           /* go to idle (single instruction: MOV 87H,20H) */
}
else {                            /* idle condition not reached yet */
    EA = 1;                       /* unlock interrupts */
}
```

Listing 8: How the background loop uses the PCON_shadow variable (for the Keil C51 compiler).

```
void my_ISR(void) interrupt xx {
    . . .
    PCON_shadow = PCON;           /* update the PCON shadow */
}
```

Listing 9: The PCON shadow must be updated in every ISR that can produce work for the back-ground loop.

through a write to a register, so the shadow register technique doesn't apply. The idea of disarming the wait-mode transition from ISRs can be made to work in the M16C, but it requires replacing the WAIT instruction with something else (such as NOP or RTS). Yes, I am talking about self-modifying code, but I don't know of any other option for the M16C. Luckily, the M16C is a von Neumann architecture, so it can execute code from the RAM address space.

The piece of self-modifying machine code can be quite small. You define a 4-byte array in RAM, as in **Listing 10**. This machine code represents a tiny C-callable function that executes the WAIT instruction and returns to the caller. In the background loop, you modify this code and call it using a pointer-to-function, as in **Listing 11**.

You must disable the transition in every interrupt to wait mode by replacing the WAIT instruction at Wait_code[0], as in **Listing 12**. With this design, an interrupt can occur at any machine instruction between FSET I (enabling interrupts) and executing the instruction at Wait_code[0]. Any such interrupt will replace the code in Wait_code[0] with the RTS,NOP instruction pair that immediately returns to the background loop, so the WAIT instruction won't survive to the point when the background loop actually comes around to execute it. Thus, any interrupt that preempts the idle loop disables the wait mode, which accomplishes the goal of an interrupt-safe transition to idle mode.

```
unsigned int volatile Wait_code[] = {
    0xF37D,                          /* WAIT instruction (M16C is little endian) */
    0x04F3                           /* RTS,NOP instruction pair (M16C is little endian) */
};
```

Listing 10: Define a 4-byte array in RAM. This machine code represents a tiny C-callable function that executes the WAIT instruction and returns to the caller.

```
    _asm("FCLR I");                  /* lock interrupts */
    if (idle_condition()) {          /* idle condition reached? */
        Wait_code[0] = 0xF37D;       /* insert the WAIT instruction */
        _asm("FSET I");              /* unlock interrupts */
        (*(void (*)(void))Wait_code)();/* call the wait code */
    }
    else {                           /* idle condition not reached yet */
        _asm("FSET I");              /* unlock interrupts */
    }
```

Listing 11: A pointer-to-function that calls code in Listing 10.

```
#pragma INTERRUPT my_ISR
void my_ISR(void) {                  /* M16C enters ISR with interrupts locked */
    . . .
    Wait_code[0] = 0x04F3;           /* insert the RTS,NOP instruction pair */
}
```

Listing 12: Disable the transition in every interrupt to wait mode by replacing the WAIT instruction at Mait_code(0).

Atomic low-power states

Running the MCU at full-speed all the time will never lead to a truly low-power design, even if you use the lowest-power MCU available. The biggest power savings are only possible by frequently switching the MCU to a low-power sleep state under the software control.

The simplest foreground/background software design requires that the transition to a low-power state be atomic, or at least interrupt-safe. This requirement does not apply when you use a more sophisticated architecture, such as a preemptive kernel or a real-time operating system. A preemptive kernel executes a special idle task when no other tasks are ready to run because all are blocked waiting for events.

Most kernels provide a way to customise the idle task (using callback functions or macros), so that you can conveniently implement the transition to a low-power state inside the idle task. The main difference between a preemptive kernel and a foreground/background system is that as long as tasks are ready to run, the kernel doesn't switch the context back to the idle task. Consequently the transition to a low-power mode is much simpler, because it doesn't need to occur with interrupts disabled.9 Unfortunately, a preemptive RTOS isn't always an option for a low-end MCU, which simply might not have enough RAM to accommodate a preemptive RTOS.

Endnotes:

1. Freescale Semiconductors, MC68HC908QY4/D datasheet, 2003.
2. Texas Instruments, MSP430x1xx Family User's Guide, 2006.
3. Atmel, ATmega169 Datasheet, 2005.
4. Atmel, AT91SAM7S32 Datasheet, 2005.
5. ARM Ltd., ARM v7-M Architecture Application Level Reference Manual, 2006.
6. ARM Ltd., Cortex-M3 Technical Reference Manual, 2006.
7. Luminary Micro, LM3S811 Microcontroller datasheet, 2006.
8. Renesas, M16C/62 Group (M16C/62P) Hardware Manual, 2003.
9. Samek, Miro and Robert Ward, "Build a Super Simple Tasker," Embedded Systems Design, July 2006, p. 18. http//:www.embedded.com/columns/technicalinsights/190302110.