Microcontroller Power Management

TEP:	112
Group:	Core Working Group
Type:	Documentary
Status:	Draft
TinyOS-Version:	2.x
Author:	Robert Szewczyk, Philip Levis, Martin Turon, Lama Nachman, Philip Buonadonna, Vlado Handziski
Draft-Created:	19-Sep-2005
Draft-Version:	1.8
Draft-Modified:	2009-07-07
Draft-Discuss:	TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

Note

This memo documents a part of TinyOS for the TinyOS Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with TEP 1.

Abstract

This memo documents how TinyOS manages the lower power state of a microcontroller.

1. Introduction

Microcontrollers often have several power states, with varying power draws, wakeup latencies, and peripheral support. The microcontroller should always be in the lowest possible power state that can satisfy application requirements. Determining this state accurately requires knowing a great deal about the power state of many subsystems and their peripherals. Additionally, state transitions are common. Every time a microcontroller handles an interrupt, it moves from a low power state to an active state, and whenever the TinyOS scheduler finds the task queue empty it returns the microcontroller to a low power state. TinyOS 2.x uses three mechanisms to decide what low power state it puts a microcontroller into: status and control registers, a dirty bit, and a power state override. This memo documents these mechanisms and how they work, as well as the basics of subsystem power management.

2. Background

The TinyOS scheduler[2] puts a processor into a sleep state when the task queue is empty. However, processors can have a spectrum of power states. For example, the MSP430 has one active mode (issuing instructions) and five low-power modes. The low power modes range from LPM0, which disables only

the CPU and main system clock, to LPM4, which disables the CPU, all clocks, and the oscillator, expecting to be woken by an external interrupt source. The power draws of these low power modes can differ by a factor of 350 or more (75 uA for LPM0 at 3V, 0.2 uA for LPM4). Correctly choosing the right microcontroller low power state can greatly increase system lifetime.

TinyOS 1.x platforms manage MCU power in several different ways, but there are commonalities in the approaches. The mica platforms, for example, have a component named HPLPowerManagement, which has a commands for enabling and disabling low power modes, as well as a command (adjustPower()) to tell it to compute the low power state based on the configuration of its various control and status registers, storing the result in the Atmega128's MCU control register. When TinyOS tells the microcontroller to go to sleep, it uses the control register to decide exactly which power state to go into. In contrast, MSP430 based platforms such as Telos and eyes compute the low power state every time the scheduler tells the system to go to sleep.

Each of the two approaches has benefits and drawbacks. The 1.x mica approach is efficient, in that it only calculates the low power state when told to. However, this leaves the decision of when to calculate the low power state to other components, which is an easy way to introduce bugs. The lack of a well-defined hardware abstraction architecture in 1.x exacerbates this problem. In contrast, the MSP430 approach is simpler, in that the system will always enter the right power state without any external prompting. However, it is correspondingly costly, introducing 40-60 cycles of overhead to every interrupt that wakes the system up, which can be a bottleneck on the rate at which the system can handle interrupts.

Both of these approaches assume that TinyOS can determine the correct low power state by examining control and status registers. For example, the MSP430 defaults to low power mode 3 (LPM3) unless it detects that Timer A, the USARTs, or the ADC is active, in which case it uses low power mode 1 (LPM1). From the perspective of what peripherals and subsystems might wake the node up or must continue operating while the MCU sleeps, this is true. However, power modes introduce wakeup latency, a factor which could be of interest to higher-level components. While wakeup latency is not a significant issue on very low power microcontrollers, such as the Atmega128 and MSP430, more powerful processors, such as the Xscale family (the basis of platforms such as the imote2) can have power states with wakeup latencies as large as 5ms. For some application domains, this latency could be a serious issue. Higher level components therefore need a way to give the TinyOS microcontroller power manager information on their requirements, which it considers when calculating the right low power state.

3. Microcontroller Power Management

TinyOS 2.x uses three basic mechanisms to manage and control microcontroller power states: a dirty bit, a chip-specific low power state calculation function, and a power state override function. The dirty bit tells TinyOS when it needs to calculate a new low power state, the function performs the calculation, and the override allows higher level components to introduce additional requirements, if needed.

These three mechanisms all operate in the TinyOS core scheduling loop, described in TEP 106: Schedulers and Tasks[2]. This loop is called from the boot sequence, which is described in TEP 107: Boot Sequence[3]. The command in question is `Scheduler.taskLoop()`, when microcontroller sleeping is enabled.

If this command is called when the task queue is empty, the TinyOS scheduler puts the microcontroller to sleep. It does so through the `McuSleep` interface:

```
interface McuSleep {
    async command void sleep();
}
```

`McuSleep.sleep()` puts the microcontroller into a low power sleep state, to be woken by an interrupt. This command deprecates the `__nesc_atomic_sleep()` call of TinyOS 1.x. Note that, as the 1.x call suggests, putting the microcontroller to sleep MUST have certain atomicity properties. The command

is called from within an atomic section, and MUST atomically re-enable interrupts and go to sleep. An issue arises if the system handles an interrupt after it re-enables interrupts but before it sleeps: the interrupt may post a task, but the task will not be run until the microcontroller wakes up from sleep.

Microcontrollers generally have hardware mechanisms to support this requirement. For example, on the Atmega128, the `sei` instruction does not re-enable interrupts until two cycles after it is issued (so the sequence `sei sleep` runs atomically).

A component named `McuSleepC` provides the McuSleep interface, and `TinySchedulerC` MUST automatically wire it to the scheduler implementation. McuSleepC is a chip- or platform-specific component, whose signature MUST include the following interfaces:

```
component McuSleepC {
    provides interface McuSleep;
    provides interface PowerState;
    uses interface PowerOverride;
}

interface McuPowerState {
    async command void update();
}

interface McuPowerOverride {
    async command mcu_power_t lowestState();
}
```

McuSleepC MAY have additional interfaces.

3.1 The Dirty Bit

Whenever a Hardware Presentation Layer (HPL, see TEP 2: Hardware Abstraction Architecture[1]) component changes an aspect of hardware configuration that might change the possible low power state of the microcontroller, it MUST call McuPowerState.update(). This is the first power management mechanism, a *dirty bit*. If McuPowerState.update() is called, then McuSleepC MUST recompute the low power state before the next time it goes to sleep as a result of McuSleep.sleep() being called.

3.2 Low Power State Calculation

McuSleepC is responsible for calculating the lowest power state that it can safely put the microcontroller into without disrupting the operation of TinyOS subsystems. McuSleepC SHOULD minimize how often it must perform this calculation: it is an inherently atomic calculation, and so if performed very often (e.g., on every interrupt) can introduce significant overhead and jitter.

MCU power states MUST be represented as an enum in the standard chip implementation header file. This file MUST also define a type `mcu_power_t` and a combine function that given two power state values returns one that provides the union of their functionality.

For example, consider a hypothetical microcontroller with three low power states, (LPM0, LPM1, LPM2) and two hardware resources such as clocks (HR0, HR1). In LPM0, both HR0 and HR1 are active. In LPM1, HR0 is inactive but HR1 is active. In LPM2, both HR0 and HR1 are inactive. The following table describes the results of a proper combine function (essentially a MAX):

	LPM0	LPM1	LPM2

LPM0	LPM0	LPM0	LPM0
LPM1	LPM0	LPM1	LPM1
LPM2	LPM0	LPM1	LPM2

In contrast, if in LPM2, HR0 is active but HR1 is inactive, the combine function would look like this:

	LPM0	LPM1	LPM2
LPM0	LPM0	LPM0	LPM0
LPM1	LPM0	LPM1	LPM0
LPM2	LPM0	LPM0	LPM2

3.3 Power State Override

When McuSleepC computes the best low power state, it MUST call `PowerOverride.lowestState()`. McuSleepC SHOULD have a default implementation of this command, which returns the lowest power state the MCU is capable of. The return value of this command is a `mcu_power_t`. McuSleepC MUST respect the requirements of the return of this call and combine it properly with the low power state it computes.

The PowerOverride functionality exists in case higher-level components have some knowledge or requirements that cannot be captured in hardware status and configuration registers, such as a maximum tolerable wakeup latency. Because it can overrides all of the MCU power conservation mechanisms, it SHOULD be used sparingly, if at all. Because it is called in an atomic section during the core scheduling loop, implementations of PowerOverride.lowestState() SHOULD be an efficient function, and SHOULD NOT be longer than twenty or thirty cycles; implementations SHOULD be a simple return of a cached variable. Wiring arbitrarily to this command is an easy way to cause TinyOS to behave badly. The presence of a combine function for mcu_power_t means that this command can have fan-out calls.

Section 5 describes one example use of McuPowerOverride, in the timer stack for the Atmega128 microcontroller family.

As part of power state override, a platform MUST define the enum TOS_SLEEP_NONE in its hardware.h file. This enum defines the highest power state of the platform's microcontroller in a chip-independent way. If a component wires to McuPowerOverride and returns TOS_SLEEP_NONE, this will cause TinyOS to never put the microcontroller into a power saving state. This enum allows a component to prevent sleep in a platform-independent way.

4. Peripherals and Subsystems

At the HIL level, TinyOS subsystems generally have a simple, imperative power management interface. Depending on the latencies involved, this interface is either `StdControl`, `SplitControl`, or `AsyncStdControl`. These interfaces are imperative in that when any component calls `stop` on another component, it causes the subsystem that component represents to enter an inactive, low-power state.

From the perspective of MCU power management, this transition causes a change in status and control registers (e.g., a clock is disabled). Following the requirements in 3.1, the MCU power management subsystem will be notified of a significant change and act appropriately when the system next goes to sleep. TEP 115[5] describes the power management of non-virtualized devices in greater detail, and TEP 108[4] describes how TinyOS can automatically include power management into shared non-virtualized devices.

5. Implementation

An implementation of McuSleepC can be found in `tinyos-2.x/tos/chips/atm128`, `tinyos-2.x/tos/chips/msp430`, and `tinyos-2.x/tos/chips/px27ax`.

An example use of McuPowerOverride can be found in the atmega128 timer system. Because some low-power states have much longer wakeup latencies than others, the timer system does not allow long latencies if it has a timer that is going to fire soon. The implementation can be found in `tinyos-2.x/tos/chips/atm128/timer/HplAtm128Timer0AsyncP.nc`, and `tinyos-2.x/tos/chips/atm128/timer/HplAtm128T` automatically wires it to McuSleepC if it is included.

For the atmega128 microcontroller, TOS_SLEEP_NONE is the "idle" power state.

A second example use of McuPowerOverride is in the msp430 timer system. By default, the msp430 lowest power state is LPM4, which does not keep clocks enabled. If `tinyos-2.x/tos/chips/msp430/timer/Msp430Clock is included in the component graph, however, this configuration wires the McuPowerOverride of ``tinyos-2.x/tos/chips/msp430/timer/Msp430ClockP.nc` to McuSleepC. This implementation of McuPowerOverride raises the lowest power state to LPM3, which keeps clocks enabled.

For msp430 microcontrollers, TOS_SLEEP_NONE is the "active" power state.

6. Author's Address

Robert Szewczyk
Moteiv Corporation
2168 Shattuck Ave, Floor 2
Berkeley, CA 94704

email - rob@moteiv.com

Philip Levis
358 Gates
Computer Science Laboratory
Stanford University
Stanford, CA 94305

phone - +1 650 725 9046
email - pal@cs.stanford.edu

Martin Turon
PO Box 8525
Berkeley, CA 94707

phone - +1 408 965 3355
email - mturon@xbow.com

Lama Nachman
3600 Juliette Lane, SC12-319
Intel Research

Santa Clara, CA 95052

email - lama.nachman@intel.com

Phil Buonadonna
Arched Rock Corp.
2168 Shattuck Ave. 2nd Floor
Berkeley, CA 94704

phone - +1 510 981 8714
email - pbuonadonna@archedrock.com

Vlado Handziski
Sekr FT5
Einsteinufer 25
10587 Berlin
GERMANY

email - handzisk@tkn.tu-berlin.de

6. Citations

[1] TEP 2: Hardware Abstraction Architecture
[2] TEP 106: Schedulers and Tasks.
[3] TEP 107: TinyOS 2.x Boot Sequence.
[4] TEP 108: Resource Arbitration
[5] TEP 115: Power Management of Non-Virtualised Devices