Iowa's Timesync Component

Ted Herman

November 3, 2007

Abstract

Several appliations developed at Iowa depend on a synchronized clock component to support scheduling. The implementation has numerous commands and some events. This document describes the current state of the T2 (TinyOS version 2) implementation, which was adapted from a previous implementation for T1. The T1 implementation was known to work for Telosb, Tmote, Mica2, and MicaZ platforms; the T2 implementation currently only works for Telosb.

1 Introduction

Clock synchronization algorithms are one of the earliest services proposed and implemented for sensor networks. Synchronization is exploited not only for recording timestamps of sensor readings, but also for coordinated actuation, to control timed behavior of motes.

The implementation described here includes four interfaces, `Wakker`, `Tsync`, `PowCon`, and `OTime`. Another interface, `Neighbor`, is described in another document. Here is typical wiring from "`applicationP`" (representing some application to use Timesync) to the Timesync component(s):

```
components MainC, applicationP, TsyncC, WakkerC, OTimeC;
applicationP.Boot  -> MainC.Boot;     // so application initilizes
appliactionP.Tsync -> TsyncC.Tsync;  // for a "sync" signal
applicationP.Wakker -> WakkerC.Wakker[unique("Wakker")];  // alarm clock
applicationP.OTime -> OTimeC;         // various "get time" functions
```

Purposes of the interfaces (we'll discuss PowCon later):

`Wakker` is similar to the TinyOS Timer service, to be used for waking up at specified times or delays; its behavior is eventually synchronized among all motes in the sensor network, so that if all motes schedule a wakeup event at time 12160, then all motes get the signal simultaneously (agreement should be well under a millisecond's difference). `Wakker` offers a number of commands described later in this document.

`OTime` is the interface with a bunch of commands that query one or another kind of clock.

2 The Wakker Interface and WakkerP Module

The Wakker interface originally had a more sensible name, Alarm, however that name was appropriated by one of the TinyOS distributions, hence the weird name. In the beginning, Wakker had a command schedule and an event wakeup. As Wakker was debugged, improved and embellished, an annoying "feature" of the TinyOS timer interface(s) also became an objective.

> *Suppose you are writing a TinyOS application that needs to periodically wake up for different reasons; perhaps sometimes for sensor reading, perhaps another time do perform some statistical calculations. The solution in TinyOS is achieved by wiring: you can use the Timer interface twice, giving each a different name in your application and then associating that name in the wiring with a different unique counter (part of this has been automated in T2). You need to name the two interfaces and write different event handlers for each; and if there is the possibility of failure and retry, you need to take care of that.*

The philosophy of Wakker is that you should only need one interface for all types of alarm. This is done by associating an *index* with each scheduled alarm time. Each alarm set command specifies an index; each signalled event provides the index associated with the event. I claim this simplifies life for the programmer. Here is the usual idiom for Wakker's wakeup event:

```
event error_t Wakker.wakeup(uint8_t indx, uint32_t wake_time) {
 switch (indx) {
  case 0: r = post my0action(); break;
  case 1: r = post my1action(); break;
  case 2: r = post my2action(); break;
  }
 return SUCCESS;
}
```

Here I've supposed there are tasks for each of the possible indices (to be defensive for TinyOS 1 (where post may fail), the wakeup can return FAIL, which would cause the signalled event to be retried automatically).

The typical "alarm set" command is called soleSet, as in the following example:

```
call Wakker.soleSet(0,3*8);  // wakeup in three seconds
```

This call schedules a wakeup with index 0, to be signalled after three seconds (the nominal units for Wakker are 1/8 seconds). The name soleSet indicates that this setting of index 0 should override (replace) any existing, outstanding alarm setting for index 0. Wakker does allow there to be multiple events of the same index scheduled; these are kept in a wakeup queue (which has a limited size, so watch out!).

Commands

The following list documents the current commands for Wakker. In cases where an alarm-set command returns FAIL, it is probably because of an alarm-queue overflow.

command `error_t schedule (uint8_t indx, uint32_t X)`

Use this command if you need to schedule a wakeup event at absolute time `X` (measured in 1/8 seconds). Since the Timesync service will adjust "absolute time" to some global standard, this interface probably isn't what you want.

command `error_t soleSchedule (uint8_t indx, uint32_t X)`

Like the `schedule` command, but this one replaces any existing entry in the wakeup queue that has the same `indx`.

command `error_t set (uint8_t indx, uint32_t D)`

This command puts an entry in the wakeup queue to be fired after a delay of `D` time units (measured in 1/8 seconds).

command `error_t soleSet (uint8_t indx, uint32_t D)`

As described above, this command is like `set`, but replaces any existing wakeup entries with the same `indx`.

command `error_t clear()`

The `clear` command clears the wakeup queue of all scheduled wakeup alarms – *for the calling component*. The `clear` command *will not* remove wakeup events that have been set by another component that is wired to `WakkerP`. This explains why the wiring is paramaterized, because each component wiring to `Wakker` gets its own "virtual" queue of events, and the `clear` command only refers to the events owned by the calling component. (Note: if for some reason you really want to clear all everyone's queue of wakeup events, you could use `WakkerP`'s `Boot` interface to force reinitializing.)

command `error_t cancel (uint8_t indx)`

The `cancel` command is like `clear`, but restricted to events of the specified `indx`.

command `uint32_t clock()`

This command returns the current "`Wakker` time", which could be used to `set` future alarm wakeup events, for example.

command `void setSync()`

This command is used only by the Timesync component to align the alarm clocks of motes, so that they have the same "`Wakker` time".

command `void rewind()`

The `rewind` command would only be used for some internal restart of the sensor, to restart time back at zero.

command `uint8_t slotsAvailable()`

Returns the number of free slots in the wakeup queue.

Notes

- The `WakkerP` module also uses the `Boot` interface; currently wired to `MainC` to start up `Wakker`.
- Unlike T2, which has the intelligence to reserve enough memory for task scheduling, the `WakkerP` module fixes the queue size by a constant (see `ALRM_slots` in `Wakker.h`). See `Wakker.h` where the enum for `ALRM_slots` estimates that `4*uniqueCount("Wakker")` will be enough slots for indices in the various types for each application wiring to `WakkerP`. Each slot in the queue is now six bytes.
- Although the implementation of `Wakker` depends on the Timesync component for aligning the node clocks, this is not a deep dependency; it wouldn't be hard to use `Wakker` without the Timesync component, provided you don't require synchronized wakeups.

3 The `OTime` Interface

The `OTime` interface offers commands (no events) for reading clocks, converting time values, and operating on time values. These are implemented by the `OTimeP` module. The `OTime.h` header file defines these structures:

```
typedef struct timeSync_t {
  uint16_t ClockH;
  uint32_t ClockL;
  } timeSync_t;
typedef timeSync_t * timeSyncPtr;
typedef union bigClock {
  struct { uint64_t g; } bigInt;
  struct { uint32_t lo; uint32_t hi; } partsInt;
  } bigClock;
```

The first structure defines the basic type of clock variables used internally by the Timesync component: 48 bits of clock, with the lowest-order bit representing about 1.08507 microseconds on Atmel sensors; this means a 48-bit clock rolls over after `(2^48)*1.08507=3.08466e8` seconds, or about 9.7747 years. The figures vary for MSP430 processors, since the lowest-order bit in that case is 0.953674 microseconds. On the one hand, some applications need microsecond precision, on the other hand, most do

not – therefore OTime has many other friendlier ways of looking at clock values. The second structure above is just to give you some idea of how the code manages arithmetic with the 48-bit values (copying ClockH to partsInt.lo, etc).

command uint32_t getGlobalTime32()

Returns the current synchronized clock, but with precision reduced to fit in 32 bits. As an aside, for Atmega processors, the native SysTime component written by Vanderbilt uses a 32-bit counter, which rolls over every 4660 seconds, about 1.3 hours. In fact, the lowest level counter is really only 16 bits, and this rolls over every 71.111 milliseconds. So SysTime keeps track of a high-order 16 bit part, and then OTime in turn keeps track of another 16 bits (thus 48 bits total). 48 bits sure is inconvenient, but to save memory in RAM and messages, this seems like a reasonable idea.

The interface requirement here is for 32 bits, with a desired precision around 1 millisecond, so we drop the 10 low-order (least significant) bits. That means each "jiffy" in the result is 1.11111 milliseconds on Atmega processors.

command uint32_t getLocalTime32()

Like getGlobalTime, but returns an unsynchronized local clock. The advantage of getLocalTime32 is that two calls to getLocalTime32 return values that are comparable for purposes of timing duration of intervals; whereas two calls to getGlobalTime32 might have some clock adjustment by the background Timesync protocol, and then a duration calculation would be incorrect due to the adjustment.

command uint8_t convTimeByte(timeSyncPtr t)

This converts a 48-bit timeSync value to a one-byte value, where low-order bit of the result is about 145.636 seconds (truncated). This one-byte value rolls over every ten hours. It's handy for saving space, and good enough for some crude timing and synchronizing.

command uint8_t getLocalTimeByte()

Read the local clock and convert to a one-byte value (see convTimeByte).

command uint8_t getGlobalTimeByte()

Read the synchronized clock and convert to a one-byte value (see convTimeByte).

command uint8_t subtractTimeByte(uint8_t a, uint8_t b)

Calculate absolute difference between two 1-byte time values (where each jiffie is about 145.636 seconds); result is somewhat ambiguous, due to rollover; we assume here that the first parameter is "larger" than the second parameter (which may not strictly look true, because of rollover, but used wisely, this works anyway).

command `uint8_t convEightTimeByte( uint32_t time )`

Convert a time measured in 1/8 second units to a one-byte time value.

command `uint32_t convTimeEights( timeSyncPtr r )`

Convert a 48-bit `timeSync` value to units measured in 1/8 seconds.

command `uint32_t convTimeSeconds( timeSyncPtr r )`

Convert a 48-bit `timeSync` value to units measured in seconds.

command `uint32_t getGlobalTimeSeconds()`

Read the synchronized clock, rounding to seconds.

command `uint32_t getGlobalTimeEights()`

Read the synchronized clock, rounding to 1/8 seconds.

command `void getGlobalTime( timeSyncPtr t )`

Read the synchronized clock into the provided `timeSync` structure.

command `void getGlobalOffset( timeSyncPtr t )`

Used internally to read the *offset* value that when added to the local clock, gives the synchronized clock.

command `void setGlobalOffset( timeSyncPtr t )`

Used internally to adjust the offset between local and global clocks.

command `void adjGlobalTime( timeSyncPtr t )`

The Timesync protocol uses this to increase the offset between local and global clocks.

command `void getLocalTime( timeSyncPtr t, uint32_t * v )`

Reads the local clock into the provided `timeSync` structure, and also saves the native 32-bit clock into the provided variable. If we use the current Timesync's ability to adjust skew as well as offset (that is, compensate for hardware drifting slow), then `getLocalTime` calculates the current skew factor as a side-effect, but *does not* apply this skew adjustment to the value it returns in the `timeSync` structure. This is so that intervals can be measured without skew and offset adjustments. If the invocation is within `INTERVAL_WO_SKEW` jiffies of a previous call to `getLocalTime`, this extra side-effect is merely interpolated, rather than fully calculated, so as to avoid a case of numerical insignificance (because skew applied to very small numbers would be zero); in such cases the skew adjustment just adds the elapsed number of jiffies between calls (ie, a crude interpolation).

command `pubLocalTime( timeSyncPtr t, uint32_t * v )`

Like `getLocalTime`, but also apply skew to the result. This is only used for "public" exposure of local time, as in beacon messages or by Global Time routines, so they have a basis that includes skew. The logic is simple, consisting of calling getLocalTime and then adding the skew adjustment that getLocalTime has already calculated.

command `void getLastLocalTime( timeSyncPtr t )`

Returns the most recent previous result of `getLocalTime` (intended to be a quick-and-dirty timestamp with very low overhead).

command `void conv1LocalTime( timeSyncPtr t )`

Convert provided local time value to a global time value, by adding offset (but ignoring skew).

command `void conv2LocalTime( timeSyncPtr t )`

Convert provided local time value to a global time value, including both offset and skew factors.

command `void setSkew( float skew )`

Used internally to set a skew value.

command `float getSkew()`

Used internally to read the current skew value.

command `void add( timeSyncPtr a, timeSyncPtr b, timeSyncPtr c )`

Calculate `c = a + b`.

command `void sub( timeSyncPtr a, timeSyncPtr b, timeSyncPtr c )`

Calculate `c = a - b`.

command `bool lesseq( timeSyncPtr a, timeSyncPtr b, timeSyncPtr c )`

Calculate `a <= b`.

Conversions between Telos/Tmote and MicaZ

Commands `Tel2Z`, `Tel2Zs`, `Z2Tel`, and `Z2Tels` convert time values bewtween telos and mica; this is needed because a Mica2/MicaZ clock is based on 921.6e3 jiffies/sec, whereas a TelosB clock is based on 2^{20} jiffies/second. These are only meaningful if you have a mixed telos/micaz network.

> **Interesting Note:** I got the figure 921.6e3 from Vandy's documentation of the Flooding Timesync Protocol implementation; however for my MicaZ motes, my testing indicated that the true figure is 921.778e3, so the current conversions in `Tel2Z`, etc, will need to be changed!

Unresearched is how things might change if one were to increase the default speed of the CPU clock in the Tmote (by enabling the external Rosc by changing the Msp430ClockP.nc file). Timesync now uses the 32kHz off-cpu clock in Telos/Tmote systems, because of its superior stability. But in very short delay periods (if the MAC backoff period is small enough), then I try to use the native CPU clock, which deviates by about half a percent or so according to Cory Sharp. I don't know how things work at higher CPU speeds.

4 Application-Layer Power Control

Note: *because this component/interface has not been tested adequately for T2 (it was working for T2/Boomerang), the radio power-off has been disabled;* see the `#ifdef POWCON` statement disabling power control in `PowConP`.

Once a sensor network has synchronized clocks, application-layer duty-cycling may be feasible (OK, may be possible without synchronized clocks, but possibly simpler with a synchronized clock layer). That's the idea behind the `PowCon` interface. The interface is implemented by `PowConC` with help from `PowCommC`. The `PowConC` configuration supports the `PowCon` interface, and `PowCommC` is a wrapper for `AMSend`, `Receive`, and so on – these are needed to keep track of the state of the radio, so that it can be powered down. Look at `TsyncC` for an example of wiring to `PowConC` and `PowCommC`.

The idea of `PowConC` is to have a parameterized `PowCon` interface, so that module `PowConP` can build a table of all the modules participating in application-layer power control. At some point, likely after all clocks are synchronized, each node would invoke `PowCon.restart()`. In response, `PowConP` will signal each client (that is, each participating application module), to accumulated the projected needs for wakeup intervals. After `PowConP` gets all the projected needs, a schedule can be built.

Note: this is done *at execution time* rather than statically compiled, so that applications could dynamically vary the needs for wakeup intervals. Provided that all the nodes invoke `restart` in unison, there would be no problem switching to a new schedule.

Here, then, are the commands and events for the `PowCon` interface:

command `void restart()`

Causes a recalculation of the schedule for power control, of all participating application modules.

event `void supply(powschedPtr p)`

Signal to each client (participating application module) to solicit wake/sleep data, which the client does by filling in the provided powsched structure. The supply event could be signalled many times as a result of restarts, schedule computation, etc. The `powsched` structure is defined in `PowCon.h`, as

```
typedef struct {
   uint16_t  period;    // period length in 1/8 seconds (max = 2.2 hours)
   uint8_t   livetime;  // alive time per period in 1/8 seconds (max = 32 sec)
   uint8_t   priority;  // priority within interval (for tie-breaks)
   } powsched;
typedef powsched * powschedPtr;
```

Explanation of fields: the `period` is the duration of a periodic interval, in which the application is awake for `livetime` jiffies (counted in eight-second units), then asleep for the remainder of the interval. During the asleep time, the radio could be off, and the application would miss messages, of course, during the sleeping time.

Overall, `PowConP` will assemble the `period` and `livetime` data from all clients, and then determine some suitable periodic schedule for being awake (radio-enabled) and sleeping. Note that application clients could still become awake during a sleeping period, say by Timer or Wakker event, but the radio would be off (but maybe there is some need to read a sensor, so this could make sense). A special value `0xffff` for `period` specifies that the participating application is part of every awake/sleep interval.

A remaining question is: if there is some periodic schedule, such as sleeping for ten minutes, then waking up for three seconds, what is the order in which applications get event signals during the three awake seconds? The `priority` field is used to order multiple clients; `PowConP` takes `0` to be the "highest" priority, and `255` to be the lowest priority. The highest priority client is first during the awake time, then the second highest, and so on — clients are signalled sequentially, in order by priority, spacing them according to their `livetime` specification.

In the Timesync implementation, we have inserted two default clients, in module `MarginP` (named for Marge Innovera, of Car Talk fame). `MarginP` wires twice to `PowConC`, once with priority `0` and once with priority `255` – these are "pre" and "post" applications for every period interval. These pre and post applications are just for padding, in case synchronization could be somewhat inaccurate.

Internally, `PowCon` uses the `Wakker` interface to get synchronized clock wakeups, so that all nodes would presumably have the periods for awake/sleep in unison. Note that if, say, three application

modules have the same wakeup periodic intervals, then the radio will be powered on for all three applications in one, consecutive interval, equal to the sum of the specified `livetime` numbers: so a client could receive a message even outside of its `livetime` in such a case.

event `void wake()`

This is signalled when the radio is on, and it is the client's turn (the `livetime` interval is active for this client).

event `void idle()`

Signalled when the radio is being powered down.

command `command void forceOn()`

Called by a client to force `PowConP` to suspend energy conservation, that is, tell `PowConP` to leave the radio powered on until further notice.

command `command void allowOff()`

This command is the inverse of `forceOn`, informing `PowConP` that it can go back to normal operation.

command `uint8_t randelay()`

Client command: get random delay amount to delay before message send (to decrease contention on MAC).