Powering sensors

March 25, 2004

The general case for powering sensors on a sensor board requires handling $1, \ldots, n$ distinct power states. For example, the LeadTek 9546 GPS "sensor" has 4 power states, which may be succinctly described as Off, Cold, Warm and Hot, which influence the data aquisition rate of the sensor. There are several ways of dealing with this issue, in particular.

Do nothing One way is to ignore the issue and use the StdControl interface Start command to power the sensor into its default power state. Works fine for ON/OFF sensors. Simplicity is the obvious advantage, rigidity the obvious limitation. This is what the current driver code for the LeadTek GPS does. It works, sort of: cold start data aquisition (GPS location) requires 45 seconds, an unpleasantly long time on a device trying to operate in the μsec to msec time frame.

Embed in specialized component Another way is to define an interface corresponding directly to the hardware in question bypassing the definition of a power interface altogether. Using the LeadTek again, this would result in commands

```
interface LeadTek9546 {
   ...
   command result_t powerColdStart();
   command result_t powerWarmStart();
   command result_t powerHotStart();
   command result_t powerOff();
   // Possible power*Done events defined here
   ...
}
```

The advantage to the application programmer is reasonably fine-grained control over the behavior of the sensor, at the cost of learning he intricacies of slightly different APIs for different sensors. A disadvantage of this technique is overspecialization; there isn't any abstraction here, just a definition for a device. (One could probably argue that is also an advantage, since presumably, once drivers are written for some sensor of year/model, they need not be written again. For an analogy, consider CC1000Control)

Power level as argument For certain, now historical reasons, the driver code for the Crossbow, Inc. MTS420CA "fireboard" was decomposed into it's constituent sensor parts, each in their own directory, each with their own header file. This has several advantages, one of which allows a defining a power command as part of a sensor interface abstraction, and enumerating states as arguments to that command.[1] For example:

```
interface Sensor {
   ...
   command result_t power(uint8_t power_state);
```

[1] This has actually worked fairly well. It allows application developers to mess around with data collection algorithms without messing around with the sensor driver code.

```
   event result_t powerDone();
   ...
}
```

In the header file `foo_sensor.h`, for a sensor `FOO`, one finds

```
enum {FOO_SENSOR_POWER_OFF, FOO_SENSOR_COLD_START, ...};
```

An error event can be signaled if something unexpected transpires. The advantage is that the sensor API only needs to be learned once, the differences being only in the arguments to the power command.

Likely, most sensors will have two states, on and off. Note that this technique has not yet been, but probably will be, implemented in the driver code for this board. Just as soon as the proprietary byte string for controlling the LeadTek device is reverse-engineered. See note following on problems with defining the "off" state in this kind of call.

Analogy with CC1000 radio power As stated, the general case requires handling $1, \ldots, n$ distinct power states for each sensor. n is probably 2, could be up to 5 or 6, but possibly very many more. An analogy could be drawn with the CC1000 power states: how could RF power be handled with an interface, given that each radio has multiple, but differing number of power states?

Compromise Given that

1. there will be a standardized Sensor interface defined for the TinyOS core,
2. [Std/Split]Control with semantics of init, start and stop will be adopted, and
3. there is no way to know beforehand how many power states will be needed in the future,

the following compromise exploits the fact that most existing sensors driven by TinyOS code have simply on and off power states:

- Define a power(uint8_t power_state) command in the Sensor interface.
- Define an enum (or preprocessor value) in a header file, perhaps sensorboard.h, perhaps foo_sensor.h, for each power state for each sensor.

Some comments from David Gay

```
>
> tex and pdf file outlining various schemes
> for controlling power to sensors committed
> to cvs: contrib/ucbce/doc/misc/sensor_power.[tex,pdf]

I like the discussion of power levels, current problems, etc. I question the
assumption that we're going to stick with SplitControl - I'm going to propose to
replace it with:

interface PowerControl
{
   /**
    * Initialize the component and its subcomponents.
    * @return Whether initialization was successful.
    */
   command result_t init();

   /**
    * Change the component's power level, the component is responsible for
    *   changing its subcomponent's power levels as appropriate
    * @param state New power state for this component. Possible values are
    *   POWER_ON: switch to default power on mode
    *   <component specific values>: some components may define additional
    *     power levels
    * @return Whether the power change request was accepted.
    */
   command result_t power(uint8_t state);

   /**
    * Notify components that the power level has been changed
    */
   event result_t powerDone();

   /**
    * Switch the component's power off, the component is responsible for
    *   switching its subcomponent's power off as appropriate
    * @return Whether power off request was accepted
    */
   command result_t stop();

   /**
    * Notify components that the power level has been changed
    */
   event result_t stopDone();
}
```

```
Rationale:
- split-phase init is not really useful - calling split-phase commands in other
components in init is not possible as start hasn't happened yet. Better to keep
init for internal state initilisation - start/power can do the hardware/slow
initialisation

- mostly keep the power levels from David Doolin's proposal, define a standard
one (on)

- powerOff (aka stop) is defined separately - this should simplify the logic in
powerDone/stopDone event handlers

David Gay
```

0.1 Response to DG from David Culler

```
The problem is that if the hardware takes a long period to initialize,
what does the processor do in the meantime?  This is a natural case for
one shots.  Kick the hardware initialization process.  Wake me up again
```

```
some time in the future.  Trigger the upper software layer know when
things are all ready.

It is not that you always need split phase, but that split-phase is the
more general case.  It means that if you replace the component with a
very different initialization requirement, the rest of the system will
already be able to deal with it.
```

From Dave Doolin Gay's proposal makes a fair bit of sense, although using "power" and "stop" instead of "powerOn" and "powerOff" needlessly breaks symmetry. That is, "stop" what, exactly? And who will write and maintain the documentation explaining that "stop" means turning off the power. Using "powerOn" and "powerOff" is plain english clear.

Condensing powerOn/Off into a single call has the advantage of being terse and elegant, with a possible very large major drawback: handling completion of powerOn is different than handling completion of powerOff. So rolling in the powerOff mode into a single power function would require switching code to differentiate powering on and off.