Pawel Gburzynski

One-bit PDM microphone

[preliminary] driver

September 25, 2017

Introduction

This driver has been written for the SPH0641LM4H-1 one-bit microphone available on the CC1350 SENSORTAG. The digital interface to the microphone is trivial and assumes that the device returns a stream of bits strobed by a clock. There are only two wires, i.e., the clock going from the µC to the microphone and the signal going in the opposite direction. The driver is general in the sense that it will work with any microphone device using this kind of interface. There are no commands going to the device, just a blind and continuous (for as long as the device is logically on) series of pulses on the clock wire.

Summary of the functionality

I don't think we can hope to do nontrivial signal processing, e.g., stuff like voice recognition and so on, but I am not an expert. However, it seems that we can easily do, say, noise detection, perhaps even driven by some patterns. The present driver demonstrates that we can extract the signal from the microphone and detect simple changes in it. Quite likely, we can do considerably more.

When the microphone is turned on, the two pins interfacing it to the µC are configured as a one-way SSI bus. Null data is sent on the TX (unconnected) end of the bus to force clock strobes which cause the reception of a bit series from the microphone. The reception is implemented in an SSI interrupt service routine, in a very tight loop, so the bits can be received even at the maximum rate of the microphone, which is 2.475 MHz. At that rate, the function has about 3 µs to complete its job (without depriving the system of the CPU), so there's no room in it for serious processing.

The microphone appears to the application as a sensor. The interrupt service routine updates two counters that are returned as the sensor value. There are no sensor events. It is assumed that when the microphone is running, the application will periodically read the sensor value looking for patterns.

The counters keep track of:

1. The number of sampled bits in multiples of 8, i.e., in bytes. Data is extracted from the SSI bus in bytes, so one turn of the interrupt service routine takes care of 8 bits of the signal.

2. An aggregate measure of the imbalance between zeros and ones observed in the signal.

The counters are zeroed when the sensor value is read, so together they reflect a normalizable measure of imbalance between zeros and ones collected on a per-byte basis. This is rather naive, but already useful. We can experiment with other ideas when we find a specific goal.

A measure of local imbalance between zeros and ones is meaningful, because with PDM encoding, the zero level of the signal is represented by the alternating sequence of zeros and ones. The locally relative ratio of ones to zeros determines the signal level (in the plus [more ones] or minus [more zeros] direction). Thus a momentary increase of the imbalance between zeros and ones correlates with the signal amplitude. Probably, by changing the sampling frequency and/or the window of imbalance, one can roughly “tune'” the device to different kinds of noise.

At present, the measure of imbalance is simply the sum of imbalances of the consecutive bytes over the measurement period (between two consecutive invocations

of **read_sensor**), where the imbalance of a byte is defined as the squared difference between the number of zeros and ones. For example, the imbalance of 0xAB (10101011) is 4, because the difference is –2. The imbalance of 0x00 is 64. Note that this is not a perfect measure (in fact, quite far from that), because a good measure should be based on some notion of locality in the string of bits viewed continuously. I don't know at present how to do it right. Ideas are appreciated. The present (naive) variant of the measure is easy to calculate very quickly using a precomputed table of 256 values indexed by bytes.

The interface

A program using the microphone should include the header *obmicrophone.h*. The sensor must be turned on to become active. Here is the operation:

```
obmicrophone_on (word rate);
```

where **rate** is the clock frequency which must be between 1000 and 24750 (too low and too high values are properly adjusted). The number is multiplied by 100 to yield the SSI clock rate (from 100 kHz to 2.475 MHz).

The device is stopped with:

```
obmicrophone_off ();
```

which operation also powers the microphone down. On the SENSORTAG, the microphone is powered from a pin, and it should always be, because the device drains about 700 μA when operating at full speed. The current drops for lower strobing frequencies, but it is still of order 100-300 μA.

The strobing frequency can be changed while the device is active by calling **obmicrophone_on** with a different rate (there is no need to switch the device off in between).

The data structure describing the value returned by the sensor (defined in *obmicrophone.h*) looks like this:

```
typedef struct {
      lword nsamples;
      lword amplitude;
} obmicrophone_data_t;
```

The value is read with the standard operation:

```
read_sensor (word st, address val);
```

where **val** should point to a structure like the one above. The state argument is ignored: the function always returns immediately (it just copies the current values of the counters). The original counters are zeroed after the value is read, so they reflect the signal parameters from the last acquisition.

For example, when I start the device at the strobing frequency of 1 MHz (with the **rate** argument of 10000) and read the sensor value at 1 s intervals, I get a sequence like this:

```
…
N: 125840, A: 136476
N: 125844, A: 129640
```


```
N: 125836, A: 135000
N: 125836, A: 131324
…
```

Note that the number of samples is about 1/8 of the frequency (because it refers to bytes). When I make some noise, e.g., clap hands a few times, reasonably close to the device, the numbers become:

```
…
N: 125828, A: 188496
N: 125832, A: 200628
N: 125840, A: 242312
N: 125836, A: 142296
…
```

showing a rather well pronounced increase in the imbalance.